Ribbons: A Partially Shared Memory Programming Model

Kevin John Hoffman
Purdue University

Follow this and additional works at: https://docs.lib.purdue.edu/open_access_dissertations

Part of the Computer Sciences Commons

Recommended Citation
https://docs.lib.purdue.edu/open_access_dissertations/159

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.
This is to certify that the thesis/dissertation prepared

By  Kevin J. Hoffman

Entitled
Ribbons: A Partially Shared Memory Programming Model

For the degree of  Doctor of Philosophy

Is approved by the final examining committee:

Dr. Patrick Eugster
Chair

Dr. Tony Hosking

Dr. Suresh Jagannathan

Dr. Dongyan Xu

To the best of my knowledge and as understood by the student in the Research Integrity and Copyright Disclaimer (Graduate School Form 20), this thesis/dissertation adheres to the provisions of Purdue University’s “Policy on Integrity in Research” and the use of copyrighted material.

Approved by Major Professor(s):  Dr. Patrick Eugster

Approved by:  Dr. Sunil Prabhakar  07/06/2013
Head of the Graduate Program  Date
RIBBONS: A PARTIALLY SHARED MEMORY PROGRAMMING MODEL

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Kevin J. Hoffman

In Partial Fulfillment of the
Requirements for the Degree

of

Doctor of Philosophy

December 2013

Purdue University

West Lafayette, Indiana
ACKNOWLEDGMENTS

First, I acknowledge my advisor, Professor Patrick Eugster, who challenged me to reach for goals I never would have thought possible and for helping me to achieve them, who gave me the freedom to explore my diverse research interests yet still remain productive, who taught me both the fun and the rigor of academic research, and whose merciless editing standards forced refinement of my writing skills.

Second, I acknowledge my colleagues and mentors that I have had the privilege of working with over the years, including Professor Tony Hosking, Professor Suresh Jagannathan, Harrison Metzger, Professor Cristina Nita-Rotaru, Murali Krishna Ramanathan, and David Zage, as well as fellow members of the S3 research lab. The fruitful technical discussions and brainstorming sessions gave root to many ideas exhibited within this dissertation.

Third, I acknowledge my parents, who have always believed in me, who gave me the foundation I needed to achieve my dreams, and who set me down the path of excellence in all aspects of life by their own quiet examples.

Finally, I acknowledge my infinitely patient wife, Mindy, who tirelessly supported me throughout all the years and long nights, who listened with real interest to the technical challenges that were at hand, who never said no when asked to proofread final paper submissions after midnight, and who has wholeheartedly backed all my dreams and aspirations no matter how outlandish.

I’m also grateful for the financial support I received that funded my research efforts: the Graduate Assistance in Areas of National Need (GAANN) Fellowship (2005–2008), NSF grants 0834619 and 1117065, and the Maurice H. Halstead Memorial Award.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF TABLES</th>
<th>vii</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF FIGURES</td>
<td>viii</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>xi</td>
</tr>
<tr>
<td>1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Processes</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Virtual Memory</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Threads</td>
<td>3</td>
</tr>
<tr>
<td>1.4 Multi-core Computing and Plugin Architectures</td>
<td>3</td>
</tr>
<tr>
<td>1.5 Memory Isolation Mechanisms</td>
<td>4</td>
</tr>
<tr>
<td>1.5.1 Hardware-based Fine-grained Memory Access Control</td>
<td>5</td>
</tr>
<tr>
<td>1.5.2 Guarding Against Heap Corruption and Memory Bugs</td>
<td>5</td>
</tr>
<tr>
<td>1.5.3 Region-based Memory Allocation</td>
<td>6</td>
</tr>
<tr>
<td>1.5.4 Ownership Types</td>
<td>6</td>
</tr>
<tr>
<td>1.5.5 Code Rewriting, Inline Monitors, and Language-based Techniques</td>
<td>7</td>
</tr>
<tr>
<td>1.5.6 Software Fault Isolation</td>
<td>8</td>
</tr>
<tr>
<td>1.5.7 Specialized or Enhanced Operating Systems</td>
<td>12</td>
</tr>
<tr>
<td>1.5.8 Virtualization</td>
<td>13</td>
</tr>
<tr>
<td>1.6 Component Isolation in Apache Tomcat</td>
<td>15</td>
</tr>
<tr>
<td>1.7 Thesis Statement</td>
<td>16</td>
</tr>
<tr>
<td>1.8 Contributions</td>
<td>17</td>
</tr>
<tr>
<td>2 PROTECTION DOMAINS, RIBBONS, AND RIBBONJ</td>
<td>18</td>
</tr>
<tr>
<td>2.1 Ribbons Memory Programming Model</td>
<td>18</td>
</tr>
</tbody>
</table>
## 2.2 Privileges Between Domains vs. Privileges Between a Ribbon and a Domain .................................. 19

## 2.3 Master–Worker Example .................................. 20

## 2.4 Web Browser Example .................................. 22

## 2.5 RIBBONJ ........................................ 24

## 2.6 RIBBONJ Example ..................................... 26

## 3 RIBBONJ-LITE ........................................ 29

### 3.1 Syntax and Definitions .................................. 29

### 3.2 Typing ................................................. 32

### 3.3 Dynamic Semantics ..................................... 34

### 3.4 Type Preservation ....................................... 38

### 3.5 Progress ............................................... 44

### 3.6 Isolation ................................................. 51

#### 3.6.1 Cooperative Isolation ................................ 51

#### 3.6.2 Full Isolation ....................................... 52

#### 3.6.3 Adaptable Isolation Levels .......................... 53

### 3.7 Narrowing Extension .................................... 53

## 4 IMPLEMENTATION ........................................ 55

### 4.1 Language Frontend: JastAdd .......................... 56

#### 4.1.1 Compiler Frontend ................................... 56

#### 4.1.2 Compiler Backend ................................... 61

### 4.2 Language VM Implementation: JikesRVM .............. 63

#### 4.2.1 JikesRVM Overview .................................. 64

#### 4.2.2 Virtual Memory Address Space Changes ............. 66

#### 4.2.3 Support for Memory Domains ......................... 67

#### 4.2.4 Allocation Path Changes ............................. 69

#### 4.2.5 GC Enhancements .................................... 71

#### 4.2.6 Signal Handling Enhancements ....................... 71

#### 4.2.7 Support for Ribbons ................................ 73

#### 4.2.8 JikesRVM Boot Loading Changes ................. 74
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2.9</td>
<td>JikesRVM C Runtime Changes</td>
<td>75</td>
</tr>
<tr>
<td>4.3</td>
<td>Operating System Support on Linux</td>
<td>76</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Implementation Overview</td>
<td>76</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Growing and Shrinking Memory Domains</td>
<td>78</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Isolation Properties</td>
<td>80</td>
</tr>
<tr>
<td>4.3.4</td>
<td>Bootstrapping the First Ribbon</td>
<td>80</td>
</tr>
<tr>
<td>5</td>
<td>EVALUATION</td>
<td>84</td>
</tr>
<tr>
<td>5.1</td>
<td>Performance Evaluation</td>
<td>84</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Microbenchmark Design</td>
<td>84</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Microbenchmark Results</td>
<td>87</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Methods and Benchmarks</td>
<td>87</td>
</tr>
<tr>
<td>5.1.4</td>
<td>Benchmark Results</td>
<td>88</td>
</tr>
<tr>
<td>5.2</td>
<td>Small-scale Refactoring Experiences</td>
<td>94</td>
</tr>
<tr>
<td>5.3</td>
<td>Apache Tomcat Ribbonization</td>
<td>95</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Apache Tomcat Overview</td>
<td>95</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Architecture and Coding Overview</td>
<td>97</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Assessment of Refactoring Efforts</td>
<td>100</td>
</tr>
<tr>
<td>5.3.4</td>
<td>Performance Evaluation</td>
<td>100</td>
</tr>
<tr>
<td>5.4</td>
<td>Apache httpd Ribbonization</td>
<td>108</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Apache httpd Overview</td>
<td>109</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Ribbonization Overview</td>
<td>111</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Assessment of Refactoring Efforts</td>
<td>113</td>
</tr>
<tr>
<td>5.4.4</td>
<td>Performance Evaluation</td>
<td>114</td>
</tr>
<tr>
<td>5.4.5</td>
<td>Benchmark Results for Short Pages</td>
<td>116</td>
</tr>
<tr>
<td>5.4.6</td>
<td>Benchmark Results for Average-sized Pages</td>
<td>123</td>
</tr>
<tr>
<td>5.4.7</td>
<td>Benchmark Results for Large Pages</td>
<td>123</td>
</tr>
<tr>
<td>5.4.8</td>
<td>ApacheBench Results Summary</td>
<td>127</td>
</tr>
<tr>
<td>6</td>
<td>CONCLUSIONS AND FUTURE WORK</td>
<td>128</td>
</tr>
<tr>
<td>6.1</td>
<td>Conclusions</td>
<td>128</td>
</tr>
<tr>
<td>6.2</td>
<td>Future Work</td>
<td>130</td>
</tr>
</tbody>
</table>
LIST OF REFERENCES ........................................ 132
VITA .............................................................. 142
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Summary of data from the evaluation of Tomcat</td>
<td>104</td>
</tr>
<tr>
<td>5.2</td>
<td>Summary of additional latency in request processing due to ribbonization</td>
<td>105</td>
</tr>
<tr>
<td></td>
<td>in TomcatRJ</td>
<td></td>
</tr>
<tr>
<td>5.3</td>
<td>Summary of code changes required to ribbonize httpd (worker MPM)</td>
<td>113</td>
</tr>
<tr>
<td>5.4</td>
<td>Summary of latency differences for ribbonized httpd under the apachebench</td>
<td>116</td>
</tr>
<tr>
<td></td>
<td>workload for small [45 byte] web pages.</td>
<td></td>
</tr>
<tr>
<td>5.5</td>
<td>Summary of latency differences for ribbonized httpd under the apachebench</td>
<td>120</td>
</tr>
<tr>
<td>5.6</td>
<td>Summary of latency differences for ribbonized httpd under the apachebench</td>
<td>125</td>
</tr>
<tr>
<td></td>
<td>workload for large [1MB] web pages.</td>
<td></td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Modeling a master–worker relationship in ribbons.</td>
<td>21</td>
</tr>
<tr>
<td>2.2</td>
<td>How ribbons could provide heap isolation for components within a web browser.</td>
<td>23</td>
</tr>
<tr>
<td>2.3</td>
<td>RIBBONJ code defining protection domain types for the Master–Worker example.</td>
<td>28</td>
</tr>
<tr>
<td>2.4</td>
<td>RIBBONJ code defining ribbon types for the Master–Worker example.</td>
<td>28</td>
</tr>
<tr>
<td>3.1</td>
<td>RIBBONJ-LITE language syntax.</td>
<td>30</td>
</tr>
<tr>
<td>3.2</td>
<td>Auxiliary definitions for RIBBONJ-LITE, providing relations for lookup of fields, types, and privileges.</td>
<td>31</td>
</tr>
<tr>
<td>3.3</td>
<td>Term typing rules for RIBBONJ-LITE.</td>
<td>33</td>
</tr>
<tr>
<td>3.4</td>
<td>Method, class, protection domain, and ribbon typing rules for RIBBONJ-LITE.</td>
<td>33</td>
</tr>
<tr>
<td>3.5</td>
<td>Global evaluation in the form $\langle Q, E, P, \mathcal{R} \rangle \rightarrow \langle Q', E', P', \mathcal{R}' \rangle$.</td>
<td>35</td>
</tr>
<tr>
<td>3.6</td>
<td>Evaluation contexts, $E$, for RIBBONJ-LITE.</td>
<td>35</td>
</tr>
<tr>
<td>3.7</td>
<td>Definition of the $\text{buildnewprivs}$ helper relation, which is used to define the set of privileges that a newly created protection domain acquires.</td>
<td>36</td>
</tr>
<tr>
<td>3.8</td>
<td>Local evaluation in the form $\langle t, E, P, \mathcal{R} \rangle \rightarrow \langle t', E', P', \mathcal{R} \rangle$.</td>
<td>37</td>
</tr>
<tr>
<td>4.1</td>
<td>AST extensions for RibbonJ compiler.</td>
<td>57</td>
</tr>
<tr>
<td>4.2</td>
<td>Example of a parsing rule in the RIBBONJ compiler that extends the JastAddJ parser for Java.</td>
<td>59</td>
</tr>
<tr>
<td>5.1</td>
<td>Throughput in items processed per second for the non-ribbonized and ribbonized (lower-layer only) versions of JikesRVM.</td>
<td>86</td>
</tr>
<tr>
<td>Figure</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>5.2 Throughput in MB processed per second for the non-ribbonized and ribbonized (lower-layer only) versions of JikesRVM.</td>
<td>86</td>
<td></td>
</tr>
<tr>
<td>5.3 Wall-clock running times for the benchmarks. Times are normalized to the running time of the original version of JikesRVM.</td>
<td>89</td>
<td></td>
</tr>
<tr>
<td>5.4 Normalized wall-clock running times for the mutator portion of execution for the first half of the benchmarks.</td>
<td>92</td>
<td></td>
</tr>
<tr>
<td>5.5 Normalized wall-clock running times for the garbage collection portion of execution for the first half of the benchmarks.</td>
<td>92</td>
<td></td>
</tr>
<tr>
<td>5.6 Normalized wall-clock running times for the mutator portion of execution for the second half of the benchmarks.</td>
<td>93</td>
<td></td>
</tr>
<tr>
<td>5.7 Normalized wall-clock running times for the garbage collection portion of execution for the second half of the benchmarks.</td>
<td>93</td>
<td></td>
</tr>
<tr>
<td>5.8 Overview of Tomcat ribbonization.</td>
<td>96</td>
<td></td>
</tr>
<tr>
<td>5.9 Visualization of the observed distributions of request latency within Tomcat under the SPECweb2009 Banking workload as a boxplot.</td>
<td>101</td>
<td></td>
</tr>
<tr>
<td>5.10 Visualization of the CDF of request latency within Tomcat under the SPECweb2009 Banking workload.</td>
<td>102</td>
<td></td>
</tr>
<tr>
<td>5.11 The distribution of the additional time required to process requests in Tomcat due to ribbonization during SPECweb2009 Banking.</td>
<td>106</td>
<td></td>
</tr>
<tr>
<td>5.12 Throughput for each 10-second interval during SPECweb2009 Banking.</td>
<td>107</td>
<td></td>
</tr>
<tr>
<td>5.13 Throughput for httpd under the apachebench workload for small [45 byte] web pages.</td>
<td>116</td>
<td></td>
</tr>
<tr>
<td>5.14 Visualization of the observed distributions of request latency within httpd under the apachebench workload for small [45 byte] web pages.</td>
<td>117</td>
<td></td>
</tr>
<tr>
<td>5.15 Visualization of the CDF of request latency within httpd under the apachebench workload for small [45 byte] web pages.</td>
<td>118</td>
<td></td>
</tr>
<tr>
<td>5.16 Throughput for httpd under the apachebench workload for average-sized [11.2KB] web pages.</td>
<td>121</td>
<td></td>
</tr>
</tbody>
</table>
5.17 Visualization of the observed distributions of request latency within httpd under the apachebench workload for average-sized [11.2KB] web pages. 121

5.18 Visualization of the CDF of request latency within httpd under the apachebench workload for average-sized [11.2KB] web pages. 122

5.19 Throughput for httpd under the apachebench workload for large [1MB] web pages. 124

5.20 Visualization of the observed distributions of request latency within httpd under the apachebench workload for large [1MB] web pages. 124

5.21 Visualization of the CDF of request latency within httpd under the apachebench workload for large [1MB] web pages. 125
ABSTRACT


The need for programs to execute subcomponents in isolation from each other or with lower privileges is prevalent among today’s systems. While modern operating systems provide mechanisms for fine-grained isolation of OS resources, only coarse-grained mechanisms exist for efficient isolation of heap memory. This dissertation develops a new memory programming model known as ribbons to enable fine-grained isolation of heap memory. Ribbons is a shared memory programming model that allows for more implicit sharing of memory than multiprocessing yet is more restrictive than multithreading. The ribbons model hierarchically structures the heap into protection domains. Privileges between these protection domains are carefully controlled in order to “sandbox” computation as needed.

To allow practical use of the model in new and existing programs, this dissertation defines a backwards-compatible extension of Java, termed RIBBONJ. The progress and isolation properties of RIBBONJ are analyzed within a simplified formal model. Additionally, ribbons is implemented within JikesRVM, leveraging existing hardware memory protection mechanisms to avoid the overhead of inline security checks and read or write barriers. Runtime efficiency is evaluated through microbenchmarks and the DaCapo benchmarks, exhibiting minor overhead. This dissertation also studies ribbons further by refactoring Apache Tomcat and Apache httpd. Apache Tomcat is refactored to use RIBBONJ for application isolation, the resulting design and complexity are detailed, and performance is evaluated using the SPECweb2009 benchmark. Finally, Apache httpd is enhanced with a ribbonized multiprocessing module, and performance is analyzed via the Apache HTTP benchmarking tool, ab.
1 INTRODUCTION

The need for programs to execute subcomponents in isolation from each other or with lower privileges is prevalent among today’s systems. While modern operating systems provide mechanisms for fine-grained isolation of OS resources, only course-grained mechanisms exist for efficient isolation of heap memory. This dissertation develops and evaluates a new memory programming model known as ribbons to enable fine-grained isolation of heap memory. Ribbons is a shared memory programming model that allows for more implicit sharing of memory than multiprocessing yet is more restrictive than multithreading. The ribbons model hierarchically structures the heap into protection domains. Privileges between these protection domains are carefully controlled in order to “sandbox” computation as needed.

1.1 Processes

Processes and threads are foundational elements of computing. A process in a modern operating system (OS) models an instance of an executing program and all of the computing resources in use by the executing program. Processes allow multiple programs and users to share common computing resources, such as memory, processors (e.g., CPUs), peripherals, and storage, subject to the constraints setup by the operating system and its administrators.

The process is typically the smallest unit whereby other processes and users manage what is running inside the operating system. One process is not (typically) directly aware of the internal specifics of another process, such as memory segment locations, specific threads of execution, or file handle values. Operating systems provide explicit interprocess communication (IPC) mechanisms, such as shared memory
regions, FIFO pipes, and sockets, to allow processes to coordinate execution with each other and with processes on remote systems through structured, well-defined mechanisms.

Processes also provide fault isolation whereby the erroneous actions of one process cannot directly cause another process to fail (provided that one process is not directly modifying the internal details of another process, which is normally the case). This provides an important mechanism for system stability in the presence of multiple (potentially untrusted) users.

1.2 Virtual Memory

Virtual memory is a key element in isolating processes from each other by providing each process the illusion that it has full control over the entire addressable memory address space of the computer, whereas in reality there is only a finite amount of memory and one physical address for each addressable word of memory. Virtual memory addresses are translated on demand into physical addresses through careful cooperation between the operating system and the underlying CPUs. Modern CPU architectures enable efficient implementation of virtual memory mechanisms through hardware support for hierarchical page tables, translation look-aside buffers (TLBs), multi-level caches, cache coherency protocols, and “traps” that enable the OS to programatically respond to exceptional conditions.

Such memory isolation features of VMM work in tandem with protection ring features of CPUs, which allow programs to execute in different security contexts such that changes to key OS and CPU data structures, such as VMM isolation and sharing information, can be structured and controlled. The “kernel” is the part of the operating system that operates with the highest security privileges and is responsible for ensuring secure and efficient execution of all processes while mediating access to shared resources according to the desired policies. Recently hardware virtualization mechanisms such as Intel VT-x and AMD-V provide even another layer of abstraction,
allowing hypervisors to coordinate, control, and mediate the execution of multiple 
operating system kernels and the overlying processes they support.

Beyond providing isolation, virtual memory management (VMM) supports many 
features in modern operating systems, such as shared memory segments, memory-
mapped file I/O, and zero-copy inter-layer communication functions. With shared 
memory segments, two (or more) processes can share the same logical segment of 
memory, even if each process uses a different addressing scheme in accessing the same 
logical (or physical) segment of memory.

1.3 Threads

Threads are additional instances of computation within a process that share the 
same virtual memory address space and other OS resources as their parent process. 
Threads enable a single program to concurrently implement its desired logic in a 
shared-everything model, unlike other concurrency mechanisms such as message pass-
ing (shared-nothing). Although shared-memory multithreading comes with its own 
set of challenges, multithreading has become a popular method for implementing 
concurrent programs.

1.4 Multi-core Computing and Plugin Architectures

Furthermore, advances in computing power have shifted from increasing serial 
execution speed via frequency scaling to increasing parallelism. This is for instance 
demonstrated in an impressive manner by Intel’s recent 80-core Teraflops Research 
Chip [57], which fits an entire supercomputer onto a single chip.

While the multi-core computing paradigm is here to stay, the effects of the un-
derlying shift from serial to concurrent programming and the resulting challenges are 
just beginning to be experienced by both consumers and programmers alike. In order
to capitalize on the ever-increasing raw computing capacities, serious challenges to more programmable, scalable, reliable, and secure computing must be overcome.

Additionally, programmers at large are finding the transition to parallel computing difficult. Parallelism introduces several new classes of bugs, including deadlocks, livelocks, and data races, which are often subtle and hard to reproduce. These problems are magnified further by the varying underlying memory models and the ambiguities that exist therein [70]. Although many viable proposals exist for more elegant and safe parallel programming (e.g. languages that tightly restrict shared mutable state [8]), these often require programmers to tackle problems in radically different ways, and are not well-suited for the parallelization of existing codebases. New methodologies are needed that simplify the inherent complexity within parallel systems without forcing programmers to radically abandon existing models they know and trust.

Another trend is that many applications are composed of a “mash-up” of components from various untrusted sources. Such applications need the ability to execute subcomponents with lower privileges or in (partial) isolation from each other to enforce security requirements. Web servers, application servers, and web browsers allow for embedded execution of third-party subcomponents such as plugins, which are often targets of exploitation.

While modern operating systems provide a rich set of features for fine-grained protection of various OS resources (e.g., access control mechanisms on files and SELinux policies), even at the thread level, no mechanisms are provided for fine-grained intra-process memory isolation.

1.5 Memory Isolation Mechanisms

The need for modern applications to isolate sub-components has become prevalent, and over the last two decades this need has led to diverse strategies to provide such
isolation. This section reviews the major classes of isolation techniques and the foundational and recent work within each class of technique.

1.5.1 Hardware-based Fine-grained Memory Access Control

Many proposals for fine- and course-grained memory protection with hardware modifications have recently appeared, including Mondrian [114,115], Legba [113], InfoShield [97], and others [27,68,94,121]. These systems only provide protection mechanisms at the very lowest level, and do not address how to easily program within such a model. Additionally, they require hardware modifications and can currently only be used and evaluated within simulations. Capability-based access-control mechanisms, such as [31,88,111], are not suitable for protecting memory regions as they can be shared in an unrestricted fashion across threads.

The implementation strategy for lightweight protection domains in Nooks [103] and others [29,46,68,104,112] utilize page table manipulation and/or hardware segmentation already implemented in existing commodity hardware. Nooks focused on error detection and recovery of device drivers, and had a static structure of protection domains designed for separating kernel modules. While effective for its targeted purpose, the protection domains and corresponding privileges are fixed in these systems, and so are not easily extensible into a more general model that is dynamically adaptable at runtime.

1.5.2 Guarding Against Heap Corruption and Memory Bugs

Hardware-based [38,87,96,108,123] and software-based [2,4,5,9,28,61–63,89,92] techniques have been proposed for guarding against heap corruption and memory bugs. These techniques target specific types of bugs and attacks and do not provide a general-purpose protection model. Also, they either do not distinguish between
threads in a process [38, 62, 87, 89, 96] or are based solely on temporal constraints [28, 108, 123]. All hardware-based techniques except [87] require hardware modifications.

1.5.3 Region-based Memory Allocation

Region-based memory management has been investigated in various programming paradigms, e.g., functional [105], imperative [53]. The main driving force is to avoid costly heap allocation and garbage collection overheads, which is achieved by introducing some structure into the memory such as to deallocate entire regions in single steps. However, region-based memory management approaches force rigid hierarchical organizational patterns on the heap whose lifetime is tied to code scope, and provide no protection features – only improved performance.

1.5.4 Ownership Types

Ownership types [15, 16, 30] and derivates are a corresponding attempt to control the effects of aliasing in object settings, by limiting accesses to objects to their owners. These are objects themselves, which are given certain access rights on the former objects, while these rights are denied to others, leading to a rigid hierarchical structure of the heap. Recent research has focused on greater flexibility by separating read and write accesses [39, 69] or through multiple ownership [20] or ownership transfer [78]. However, ownership types require annotating every object and are thus burdensome. Also, the static enforcement of access rules is quite restrictive because communication patterns are fixed statically by type – different threads acting in different roles may not access the same type differently. Furthermore, ownership types cannot guard against malicious compilers and cannot protect low-level system components.
1.5.5 Code Rewriting, Inline Monitors, and Language-based Techniques

Pure software techniques based on code rewriting [23, 43, 46, 71, 73, 110, 117], inline monitors [2, 4, 5, 43, 61], and static enforcement through typing or other language features [3, 11, 19, 54, 60, 66, 74, 77, 101, 109] have been proposed and show promise for efficient enforcement of fine-grained protection mechanisms. Code rewriting and inline monitors rely on either a trusted rewriter or a trusted static verifier to ensure correctness. Additionally, these techniques either have fixed protection domains [23, 73, 110, 117] or require slow paths with excessive overheads (2x to 8x slowdown as observed in [43]), and may be highly architecture specific, as was the case with [110]. Language-based techniques cannot typically protect the entire operating system, as protection is provided via safe language semantics, and certain system components must be written in unsafe languages. Recent work has made progress towards entire operating systems being written in type safe languages with isolation being enforced by type safety and other language semantics, for example Singularity [58] or Verve [116].

Cloneable JVM [60] attempts to mitigate the cost of isolating subcomponents in Java programs by allowing a running JVM to be cloned in a post-initialized state. However, applications running within cloned JVMs are fully isolated and must communicate with its parent JVM only through slower IPC mechanisms.

JavaSeal [19] allowed code, data, and a security policy to be bundled together into a mobile agent, which could then execute on any Java platform and environment with guaranteed enforcement of the bundled security policy and corresponding behavior. In this model, a mobile agent must be isolated from the application invoking the mobile agent (as well as from other mobile agents), and the application must only interact through the mobile agent through the mobile agent’s well defined interface. All interaction occurs through a well defined message passing mechanism and not via sharing references to objects.
Object Spaces [18] places Java objects into distinct object spaces. Rather than attempting to control access to object references themselves, enforcement of a security policy is made when an object belonging to one object space calls a method on an object belonging to a separate object space. The security model is enforced at runtime via generated proxy objects termed “bridge objects” that wrap each real object instance.

1.5.6 Software Fault Isolation

Software fault isolation (SFI) [29,99,110] isolates different software modules within a shared address space by loading the code and data for different modules into their own distinct fault domains and then rewriting loaded code to prevent the code from modifying data outside of the associated fault domain or jumping to code outside of the fault domain. While some code instructions can be statically verified to not violate the constraints (e.g., absolute jump instructions), others must be verified at runtime through checks inserted prior to any “unsafe” instructions. Alternatively, instead of check operations, a technique termed “address sandboxing” can be used where instructions can be inserted prior to unsafe instructions that force the relevant bits of the addresses used by the following unsafe instruction to be within the address range for the associated fault domain. On some instruction set architectures address sandboxing requires fewer instructions with smaller encodings as compared to check instructions, thus resulting in smaller pressure on the CPU’s instruction cache.

Software fault isolation also guards access to OS resources by transforming code that makes system calls to instead make calls through a trusted mediator. Calls to the trusted mediator are made by placing trusted “trampoline” code in a dedicated read-only section of the address space, and then allowing control flow operations to escape the current fault domain only by calling or jumping to code in this trusted address space. The trampoline code has full privileges to make actual system calls – it will perform (or transfer to code that performs) any desired security checks on
the requested operation based on the calling fault domain and then completes the operation and returns control flow to the calling fault domain.

The granularity of software fault isolation has been improved to even the byte-level [23], providing a logical association of an access control list and a type for each byte in memory. Such features were provided through compiler enhancements that adjust data layouts and efficiently enforce ACL checks inline (less than 15% overhead for the scenarios tested). However, such techniques require custom compilation at the source code, or requires binary code that has full debugging symbols. Additionally, the techniques that make it efficient are specific to the context of device drivers where there is a limited number of fault domains and where usually each byte of memory only has one entry in its corresponding ACL.

Pure software fault isolation typically only sandboxes data writes and control flow transfer, it does not prevent code from reading data outside of the current fault domain due to the overhead of guarding all read operations instead of just write and control flow operations. Certain modern software fault isolation designs [29, 46, 117] leverage a combination of code verification, code rewriting, and hardware isolation features such as segmentation or paging to deliver efficient read isolation as well. Palladium [29] leverages ring levels, segmentation, and paging features of the x86 architecture to isolate kernel modules or different user modules from each other. However, in Palladium fault isolation boundaries are fixed at the module level into a single privileged vs unprivileged fault domain hierarchy.

Native client [117] combines software fault isolation techniques with x86 hardware segmentation to achieve efficient read, write, control flow, and resource sandboxing of components. On the x86 architecture, specific CISC features such as variable-length instruction coding, self-modifying code, and indirect jumps prevent reliable disassembly. For example, if code jumps to a location that is in the middle of a multi-byte instruction, this would result in an alternate sequence of instructions that may be different than what a normal sequential disassembly would have produced.
Native client addresses these challenges through compiler adjustments that ensure any control flow transfer are made only onto aligned 32-byte boundaries and then through a small, trusted code validator that verifies these and other constraints on all loaded code. Self-modifying code is also restricted by restricting writes to any code segment via OS page-level protection mechanisms.

Performance is improved compared to pure software isolation because isolation on data operations is achieved through hardware segmentation rather than inline checks, with 5% observed overhead being typical. Control flow integrity is also simplified by relying on segmentation to constrain jumps only to locations within the same code segment (within the same fault domain). All indirect jumps are verified to only jump to aligned 32-byte boundaries, and thus only jumping to a location whose disassembly sequence is proven to already have been verified by the code validator. Direct jumps can be validated by inspection of the target location by the code validator.

Native client has been enhanced to also support other CPU architectures, including amd64 and ARM [93]. These architectures do not support the segmentation features required by the x86 native client design, and thus for these architectures native client implements data (stores only) and control flow isolation through address sandboxing, as is used in SFI originally. Although average observed overhead was only 7%, on some specific benchmarks observed overhead exceeded 25%.

The security of the trusted portion of the native client design, the code validator, has been improved substantially by having the implementation be automatically generated from a declarative model formally proven against a significant subset of the x86 instruction set via the Coq proof assistant [76]. The technique reduces the size of the trusted code base from around 600 C statements that directly disassemble the x86 instruction stream to just 10 C statements that implement table-based transition logic for a finite state machine.

Recent work has leveraged the native client SFI design to provide software fault isolation features for the Java JVM: Robusta [98] uses native client to isolate execution
of JNI native code. Robusta also enforces several additional isolation constraints specific to the JVM, including mediating access to OS resources through the Java Security Manager and enforcing the Java object reachability graph (e.g., preventing reads of private fields). Robusta enhances native client to support dynamic loading of modules (which is how JNI code is loaded within the JVM). However, Robusta only provides two fault domains – one domain for the JVM and one domain for JNI native code. Additionally, the observed overhead of switching between these fault domains was quite high, causing a 7x slowdown in specific cases where code was rapidly switching between Java code and JNI native code.

Arabica [102] builds on the Robusta design to improve JVM portability by implementing JNI native code isolation through a combination of a custom JVMTI agent and custom dynamic library stubs that intercept all inbound calls to loaded native JNI libraries. Arabica also allows for isolating a subset of the native code that implements a JVM’s support for the Java standard class library. With a significant portion of the standard class library using native code isolation, observed overheads on the SPECjvm 2008 benchmark were often less than 10%, but was as high as 112%. As with Robusta, only two fault domains are provided.

The primary limitation of almost all software fault isolation systems, including [7, 23, 43, 73, 93, 98, 99, 102, 106, 110, 117], is that fault domains are statically fixed and bound to particular sections of code. The same region of code (e.g., a library) is not able to execute in the context of different fault domains. A recent SFI implementation by Mao et al. [71] allows multiple fault domains to be instantiated at runtime; however, these fault domains are still fixed to fault domain classes defined at compile time, and the observed slowdown was 30%. Additionally, this technique does not provide read sandboxing.

Erlingsson et al. [7] implement support for SFI in language runtimes that employ techniques that modify code at runtime, including just-in-time (JIT) compilation and on-stack code replacement (OSR). The work enhances the native client design to
allow code to safely be added, modified, or deleted at runtime from fault domains, even in the presence of concurrency. The design was implemented within two modern language runtimes: Mono (CLR (.NET)) and V8 (JavaScript). Typical observed overheads on benchmarks ranged from 1% (x86 platform within Mono) to 60% (x86-64 within Mono and all platforms within V8), but were as high as 196%. Although code regions can dynamically be added, modified, or deleted, this work does not solve the limitation that a particular section of code is always associated with a particular fault domain. Additionally, the x86-64 implementation does not provide sandboxing of reads.

1.5.7 Specialized or Enhanced Operating Systems

Opal [24] is an OS providing features for processes with partially-shared memory address spaces. However, Opal requires a single system-wide shared virtual address space, and also does not model the memory allocation privilege, which is important both for security and for allocation optimization. Opal’s dynamic grant and deny segmentation model is potentially less secure (via accidental capability leakage via bugs or manipulation) and prevents analysis of programs for automatic parallelization. Others [34,36,37,90] have similar limitations.

Fluke [47] is a specialized OS designed to provide “recursive virtual machines” – Fluke redefines the services provided by a kernel so that all provided resources to a child kernel are relative to the containing parent kernel. These abstractions are then mapped down to “flat” structures that a typical OS uses to provide such functionality. For example, instead of allowing kernels full access over the physical address space of a computer’s memory, a kernel has access to a “view” of memory relative to the view of its parent kernel. For a particular process living within a particular recursive virtual machine, Fluke will then flatten all of the relative memory views of that process into an actual page table mapping virtual memory to physical memory. This provides an efficient nested isolation model for both OS kernels and processes.
Wedge [12] is an enhanced Linux OS that provides sthreads, which are program threads that execute with no privileges on heap memory by default, and can then be granted fine-grained access to a program’s memory. Memory resources are assigned a memory tag at allocation. All blocks of memory with the same memory tag are considered to be within the same protection domain. An access policy indicates the set of privileges (read or read/write) an sthread should have over memory associated with certain memory tags. Callgates control the transfer of control between sthreads, allowing escalation and de-escalation or privileges. An sthread cannot create a more privileged callgate, but an sthread can authorize a more restricted sthread to use a callgate that it has created. Wedge implements a strict sandboxing model where an sthread can only spawn child sthreads with the same or fewer privileges.

In Wedge, enforcement is implemented within the OS kernel by treating each sthread as a separate process and modifying the page table entries of an sthread so that it only has the appropriate privileges over a subset of its parent process as determined by its access policy. Despite this efficient enforcement mechanism, the overhead introduced by callgates caused the sthread-enhanced Apache server to have 30% to 45% reduced throughput in their benchmark. Observed overhead for the sthread-enhanced OpenSSH server was negligible (2%).

Specialized operating systems that provide data flow isolation policies have emerged, including Asbestos [107], HiStar [120], and Flume [67]. Data flow isolation focuses on protecting the confidentiality or privacy of data, and is complementary to systems that enforce memory access privileges based on code location or other code-contextual information (such as the running thread).

1.5.8 Virtualization

While virtualizing entire operating systems is too heavyweight a mechanism to virtualize different sub-components within a single application, many of the same prin-
ciples and implementation techniques have been leveraged to provide lighter weight isolation semantics [10, 17, 32, 40, 47, 83, 84].

Tahoma [32] focuses on isolating web applications from each other and from the web browser by leveraging hardware virtualization to run distinct web browser instances as separate hardware VMs. All communication of a browser instance within a VM with the network and with other browser instances is mediated by the trusted browser operating system, which enforces the desired security policies.

Dune [10] leverages CPU hardware virtualization features to allow user-level OS processes to run separate code at different CPU protection ring levels. This allows user-space processes to employ ring-based isolation mechanisms to enforce access policies over memory or OS resources (as all system calls of code running in a less privileged ring will be intercepted by the user-level code running at virtualized ring level 0).

Apiary [84] provides application-level isolation containers that fully virtualizes the display and filesystem resources of an application. Applications run within a fully virtualized process container, such that to the application it appears that it is the only application running and that it has full control over OS resources. Apiary’s virtualized filesystem implementation provides copy-on-write semantics, which allows even large applications with complex data requirements to be quickly virtualized without heavy startup costs. Interaction between isolated application containers is carefully controlled according to the defined system security policy, mitigating the impact of any hacked application.

Xax [40] is similar in goals to Apiary in that it allows native code to run in a fully isolated environment with carefully defined interaction boundaries with the host OS. However, in this case the isolation is provided by defining a custom application binary interface (ABI) that Xax applications are compiled against. Xax then implements this ABI so that the application appears to the outside world as a web server running an interactive HTML/JavaScript application. Thus, all interaction between the isolated
application and the host OS occurs through the well defined interface of a web browser communicating with the Xax web server, which then communicates through the Xax ABI with the isolated application.

1.6 Component Isolation in Apache Tomcat

To further motivate the need for *intra*-process isolation, consider the case of how component isolation is implemented within Apache Tomcat, an open-source implementation of the Java Servlet and Java Server Pages (JSP) technologies. Tomcat allows multiple web applications and related components to run in isolation from each other within the same Java Virtual Machine (JVM). Executing components (servlets, JSPs, tag libraries, etc.) are isolated so that they can not interfere either with the operation of the main Tomcat server nor with other running components. Tomcat enforces isolation through two mechanisms: (a) a separate class loader for each web application, and (b) the Java Security Manager [35], an access monitor enforcing isolation boundaries upon JVM properties, files and directories, networking, and reflection. Using separate class loaders prevents a web application from accessing Tomcat’s internals or other applications’ state, only allowing applications to interact with the host server code through certain classes loaded by the global class loader. Class loaders achieve this isolation by copying loaded classes and giving them names which are logically distinct across loaders. The normal type-safety mechanisms in the JVM thus enforce isolation between uses of a class loaded by different class loaders.

This mechanism, however, is subject to vulnerabilities through bugs in classes loaded by Tomcat’s global class loader (CVE-2009-0783\textsuperscript{1}). The bug allows a web application to escape isolation by replacing Tomcat’s XML parser at runtime, allowing a malicious application to read sensitive files of other applications (e.g., containing passwords). The complexity of Tomcat’s ad-hoc isolation mechanisms makes it hard to verify that any given version is 100% correct and remains so as the code evolves.

\textsuperscript{1}http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-0783
In contrast, another approach is to run each isolated component in a separate language virtual machine or process. This strategy is used by the Chromium and Firefox web browsers, which execute tabs and plugins in distinct processes. These techniques induce runtime overhead due to increased memory usage (e.g., extra copies of objects) and IPC between tabs and the main process (e.g., remote method invocations if using something like Isolates [33,55]). IPC also increases code complexity. Specific solutions such as browser isolation mechanisms [25] cannot easily be applied to other cases such as application servers, or even related plugin architectures such as integrated development environments (e.g., Eclipse).

1.7 Thesis Statement

The thesis of this dissertation is that existing hardware virtual memory and operating system mechanisms can be leveraged to provide an efficient programming model wherein multiple instances of computation within a process only partially share memory resources, as structured by fine-grained access control privileges defined by the application.

This work will develop and evaluate the ribbons programming model, which enables the heaps of subcomponents within a process to be fully or partially isolated in well-structured ways. The heap is divided into an arbitrary and dynamic number of protection domains. Access privileges are tracked pair-wise between protection domains, and threads are grouped into ribbons. Ribbons subsume threads, and help protect against inadvertent or malicious access to data and mitigate unbounded heap corruption in unsafe runtime environments. Enforcement during execution relies on standard hardware-level virtual memory and memory protection mechanisms, avoiding per-instruction overheads.
1.8 Contributions

This dissertation contributes the following:

- A definition of the ribbons programming model.
- **RIBBONJ**, a Java language extension, for writing programs using the ribbons model.
- **RIBBONJ-LITE**, a formalism for RIBBONJ’s core, to study progress and isolation properties.
- An evaluation of an implementation of RIBBONJ in JikesRVM, using microbenchmarks and the DaCapo benchmarks [14].
- A case study wherein Apache Tomcat is refactored to isolate web applications using ribbons.
- An evaluation of a ribbonized version of Apache httpd using the ab benchmarking tool.

The remainder of this dissertation is organized as follows: Chapter 2 details the ribbon model and its concepts in RIBBONJ. A subset of RIBBONJ is formalized in Chapter 3 and analyzes isolation and progress properties. Chapter 4 presents an implementation of RIBBONJ in JikesRVM. Chapter 5 evaluates performance characteristics through benchmarks, and also presents a ribbonization of Tomcat, discusses required code changes, and evaluates performance differences; this chapter also presents a new version of Apache httpd with a ribbonized multiprocessing module and studies performance differences through benchmarks. Chapter 6 presents conclusions and discusses future work.
2 PROTECTION DOMAINS, RIBBONS, AND RIBBONJ

In this section the ribbons memory programming model is defined, as well as RibbonJ, a backwards-compatible extension of Java that provides the features of the ribbons model.

2.1 Ribbons Memory Programming Model

The ribbons model defines three abstractions: protection domains, domain sharing privileges, and actual ribbons. Protection domains are the basic unit from which memory sharing constraints are formed. All memory allocations are associated with some protection domain (implicitly or explicitly), such that each area of memory allocated (and thus each object in object-oriented languages) is placed “within” that domain.

Every thread in a program is associated with one or more protection domains, which determines its set of privileges as follows: Domain sharing privileges specify the access that threads executing within the context of one protection domain (termed the source domain) have on memory contained within some other protection domain (target domain). A privilege is composed of a source domain identifier $SD$, a target domain identifier $TD$, and any number of the following permissions:

- **read**: Domain $SD$ can read memory in domain $TD$.
- **write**: Domain $SD$ can write memory in domain $TD$.
- **inject**: Domain $SD$ can allocate memory in domain $TD$. 
Finally, a ribbon is a set of protection domains that models the assignment of threads to sets of protection domains. Creating a new ribbon also creates a new thread that is associated with a new set of existing protection domains (programmatically specified). This thread and all threads that it spawns execute within the context of these protection domains. Once created, the set of domain sharing privileges for a ribbon cannot be changed. The privileges acquired by a ribbon upon creation are upper bounded by the privileges of the ribbon of the creating thread. This ensures that once sharing constraints are placed upon a newly created ribbon, all future computation resulting directly or indirectly from that ribbon will never violate the sharing constraints.

In summary, the model of execution is modified so that a process contains one or more ribbons, and a ribbon contains one or more threads. Threads spawned within a ribbon share their enclosing ribbon’s set of protection domains. Upon process creation, a new ribbon is created, and thereby a new thread. Programs that are unaware of ribbons operate as before, as they are contained within a single ribbon, and all threads share the same protection domain.

2.2 Privileges Between Domains vs. Privileges Between a Ribbon and a Domain

An explicit design choice was made to model a privilege as a relationship between a source domain and a target domain rather than between a source ribbon and a target domain. Data structures model both (a) the information contained in the structure itself, and (b) the relationships between different data structures. In the same way, protection domains model both (a) the kind of information contained in the protection domain (as determined by the way in which a program allocates information into a protection domain), and (b) the allowed security relationships between different protection domains. The protection domain concept is thus a higher-level counterpart to lower-level data structures that programmers are already familiar with. If privileges were between a ribbon and a domain, this would rather force the programmer to think
in terms of both execution context (code/logic) and heap design (data structures) at
the same time.

The two approaches to defining privileges can be shown to be equivalent to each
other, and thus the choice was made to go with the design that has fewer dependencies
between program concepts during the design of the program. The approach where
privileges are defined as a relationship between a source ribbon and a target domain
can be emulated in the model where privileges are between a source domain and a
target domain by having each ribbon be associated with a unique protection domain
and where the privileges of this domain are equivalent to the privileges of the ribbon.
Likewise, the approach where privileges are defined as a relationship between a source
domain and a target domain can be emulated in the model where privileges are
between a source ribbon and a target domain by requiring that a ribbon be associated
with one or more protection domains upon construction, and the privileges of this
created ribbon are equivalent to the privileges of the protection domains it is being
associated with.

2.3 Master–Worker Example

The Master–Worker pattern is common in parallel processing, and consists of a
master delegating tasks to one or more workers through a shared queue. Workers
complete tasks in parallel and report completed tasks back to the master. Figure 2.1
demonstrates how the ribbons model could be used to enforce heap isolation between
workers within the same process. The owner ribbon represents the main program.
The owner ribbon contains its own protection domain, containing the main state of
the program.

The owner ribbon creates a master ribbon, which contains the master protection
domain containing the state needed by the master to create new tasks and post-
process completed tasks. The owner domain (and thus the owner ribbon) has full
control (inject, read, and write privileges) over the master protection domain. In
Figure 2.1.: Modeling a master–worker relationship in ribbons.
this example, the master domain is only given read privileges on the owner domain, allowing it to read state information needed to create new tasks. A variation of this design would be to use an intermediary protection domain between the owner domain and the master domain, so that neither domain has any direct permissions on the other domain.

The master ribbon creates the queue domain, which is used to contain task information, as well as the shared queue for delegation of tasks to Workers. The master is given full control over the queue domain, enabling the master to create and modify tasks as needed.

The master ribbon creates one or more worker ribbons. Each worker ribbon has its own worker domain that is fully isolated from the other worker domains of other ribbons. Additionally, while worker domains have read and write privileges on the queue domain (so they can add and remove from the queue of tasks to be processed), they do not have the inject privilege, so they are unable to directly create additional new tasks to be processed. The master ribbon does not need full control over the private worker domains, it only needs read access to easily determine the current processing status of all workers.

This example demonstrates that not all protection domains need to be bound to a particular ribbon. Each ribbon is bound to one or more protection domains, and the immediate privileges of these bound domains determine the ribbon’s heap access privileges. Note that the heap access privileges are not transitive—the fact that the owner domain (and thus the owner ribbon) has full control over the master domain does not also mean that the owner domain has full control over the queue domain.

2.4 Web Browser Example

For illustrative purposes, Figure 2.2 visualizes how these concepts could be applied to provide heap isolation for components within a web browser. A significant security
Figure 2.2.: How ribbons could provide heap isolation for components within a web browser. Note that ribbons for tab #2 and scripts #2 and #3 are not shown. “DOM” represents the document object model, which is a set of data structures that model the web page contents.
risk that must be mitigated within web browsers is the leakage of information—allowing scripts loaded from one (malicious) website to read the contents of a page loaded from an unrelated website. For example, a malicious web page trying to read information downloaded from a bank website, or a malicious web page reading username and password information submitted to other websites. Guarding against this threat is made more complicated by the fact that websites might each load the same type of “plugins” and thus the state between the separate instances of these same plugins also need to be strongly isolated.

In this example, protection domains are created for each “type” of information that can be contained within the web browser. Additionally, a hierarchical relationship is setup between protection domains such that any information from one website cannot be directly accessed by a domain containing information from a different website. Each protection domain is also associated with a ribbon that executes code relating to managing the information in its associated protection domain. The only ribbons that have access to information from multiple websites are small sections of code dedicated to managing multi-page state and can be carefully audited—they do not contain any code or logic downloaded from remote websites. Certain plugins may also have the need to provide global shared state between all instances of its plugin, and again this global state can be carefully managed by a dedicated plugin global state ribbon.

This example is only concerned with the conceptual design of heap isolation within a web browser and does not address issues such as plugin crash isolation and recovery [25].

2.5 RIBBONJ

RIBBONJ is an an extension of Java that implements the ribbon memory programming model. Modeling ribbon concepts at the language level offers improved programmability through type system support and syntactic sugar.
The **protectiondomain** keyword is used to define new protection domain types. These protection domain types represent entire “classes” of protection domains that may be instantiated at runtime. The type optionally accepts as domain arguments other typed protection domains. The **protectiondomain** definition contains a list of rules that affect privileges when a new domain of that type is created: **grant** adds a new privilege where the source domain is explicitly indicated, and the target domain is the new protection domain; **require** adds a new privilege where the target domain is explicitly indicated, and the source domain is the new protection domain. The list of rules is abstract and is just a policy until an actual protection domain is created at runtime and concrete protection domains (instead of abstract types) are bound to the domain arguments.

Note that **require** is needed in addition to **grant** because privilege specifications are used only when a new protection domain is created, and a new protection domain cannot grant a privilege to a protection domain that has not yet been created. In the web browser example, the browser global shared state domain cannot grant privileges to domains for tabs, because the tab domains do not exist when the global shared state domain is created; rather, the tab domains must require certain privileges from the global shared state domain when a tab domain is created.

Ribbons are modeled in the language by special classes that implicitly inherit from **java.lang.ThreadGroup** and implement the **Ribbon** interface. The **ribbon** keyword declares new ribbon types. The declaration includes a list of named protection domains, termed the domain arguments, which represent the protection domains with which threads within the ribbon will be associated. The first protection domain in this list is called the primary protection domain. The actual domain argument values are not specified until the ribbon is instantiated. The **getdomain** operator is provided to retrieve the protection domain value of a given name for a given ribbon object. **ribbon** classes may optionally extend other **ribbon** classes, and are instantiated using the normal **new** keyword and constructor call, followed by a **where** clause to specify the domain argument values for the new ribbon. Once instantiated,
a ribbon begins execution via the start method exposed through the Ribbon interface, allowing the application to indicate the `java.lang.Thread` to use as the first thread.

The `new` operator is also extended so that a new object can be allocated within a specific protection domain. If the extended new operator is not used, then the new object is allocated within the primary protection domain of the ribbon associated with the currently executing thread. Finally, the `thisribbon` keyword is added for convenience to get the object representing the ribbon associated with the currently executing thread to execute within the context of the new ribbon.

2.6 RIBBONJ Example

We give the specifics of the RIBBONJ syntax through the Master–Worker example, as conceptually visualized in Figure 2.1. Each master may spawn multiple workers, yet:

- Masters may not read/write/inject other masters.
- Masters may read data in workers that are spawned by that master but not by other masters.
- Workers may not read/write/inject other workers.
- Workers may only receive work from their respective masters.

Three “classes” of protection domains are required to model this: `MasterDomain` (stores information needed to produce items), `WorkerDomain` (stores information needed to process items), and `QueueDomain` (stores produced items ready for consumption). Note that multiple masters may be created (e.g., one master for HTTP requests and another for HTTPS). One must distinguish between the protection domains of these masters. This is effectively modeled within RIBBONJ since protection
domains are modeled as types that are used as “templates” and not fully instantiated until runtime. Protection domain types structure runtime protection domains, providing rich information for static analyses and compiler optimizations. Figure 2.3 gives the RIBBONJ code for defining the protection domain types.

Note how the MasterDomain requires read access to some generic owner domain. Also, note that MasterDomain needs inject access to the QueueDomain to allocate new objects inside that domain, while WorkerDomain only requires read/write access to remove objects from the worker queue for processing. Also of interest is that a Master needs to be able to read data in Worker domains that it creates. This cannot be specified in the MasterDomain, because the Worker domains do not exist when the Master is created. The grant statement allows a Worker, when it is created, to add to the privileges of the MasterDomain passed to the Worker instantiation. The protection domain types defined in Figure 2.3 can then be used to instantiate ribbons that operate within them, as shown in Figure 2.4.

When a new work item is produced, the Master must allocate it within the QueueDomain so that Worker objects can modify it. This can be done by specifying a concrete protection domain (some object deriving from ProtectionDomain) within angle brackets after the new operator and before the type name.

Also, when a new Worker ribbon is instantiated, first a new WorkerDomain protection domain is created. The constructor for the WorkerDomain requires two domain arguments. Here, they are the protection domain named mdom for the current ribbon, and the qdom protection domain value created earlier. In this way the protection domain “template” is applied to effect the runtime privileges for the new domain. The new WorkerDomain value is used to create a new Worker ribbon. The newly instantiated ribbon is then told to begin execution, using a new WorkerThread thread class as the first thread of execution within the newly executing ribbon. Note that the WorkerThread and MasterThread classes need not have been declared as inner classes of their respective ribbons, but were done so for convenience.
**Figure 2.3.** RIBBONJ code defining protection domain types for the Master–Worker example.

```java
protectiondomain QueueDomain ()
protectiondomain MasterDomain(ProtectionDomain owner, QueueDomain q) {
    grant {read,write,inject} to {owner};
    require {read} from {owner};
    require {read,write,inject} from {q};
}
protectiondomain WorkerDomain(MasterDomain m, QueueDomain q) {
    require {read,write} from {q};
    grant {read} to {m};
}
```

**Figure 2.4.** RIBBONJ code defining ribbon types for the Master–Worker example.

```java
ribbon Worker(WorkerDomain wdom) {
    static class WorkerThread extends Thread {
        public void run() { ... }
    }
}
ribbon Master(MasterDomain mdom) {
    QueueDomain qdom = mdom.q;
    Queue q = new<qdom> Queue();

    public void createWorker() {
        WorkerDomain wd = new WorkerDomain(
            getdomain(thisribbon, mdom), qdom);
        Worker w = new Worker() where (wd);
        w.start(new Worker.WorkerThread());
    }
}
class MasterThread extends java.lang.Thread {
    public void produce() {
        q.add(new<qdom> WorkItem(...));
    }
    public void run() {
        for (int i=0; i<5; i++) {
            createWorker();
            ...
        }
    }
}
```
3 RIBBONJ-LITE

To study the isolation properties of RIBBONJ we formalize its core. The resulting syntax and semantics follow in the spirit of other object calculi, e.g., Featherweight Java (FJ) [59] or Classic Java [45].

3.1 Syntax and Definitions

The syntax of RIBBONJ-LITE is presented in Figure 3.1. RIBBONJ-LITE can be viewed as FJ without subclasses (and thus casts) for simplicity, but augmented with: protection domain types \( \pi \) and protection domain values \( \tau \), ribbon types \( R \) and ribbon values \( r(R) \), references via location values \( l(\pi,C) \), field assignments \( t.f = t \), sequences of terms \( t; \), threads \( \text{newthread} <t> \{ t; \} \), and ribbons \( \text{newribbon} R(\tilde{t};) \).

Protection domain values model first class heaps, and consist of a typed identifier denoting a concrete instance of the protection domain (rather than actually containing the values within the protection domain). Protection domain values are typed, and corresponding declarations are instantiated to yield new protection domains (heaps) at runtime. The protection domain types indicate which permissions to grant to or require from existing protection domains when creating a new domain. A global protection domain that always exists is identified as \( \pi_\bot \), which has a protection domain type of \( \text{PDomain} \). Ribbon values model the set of protection domains that should be “active” when executing terms in threads belonging to the ribbon. Ribbon values consist of a sequence storing the values of the named protection domains for that ribbon. Location values \( l(\pi,C) \) uniquely identify how to retrieve a value from the object store. Locations may only point to object values, and not to
program $PR ::= \text{newribbon } R (\pi \bot) \{ \tilde{t}_i \}$

protection dom. $PD ::= \text{protectiondomain } D (\overline{D} \overline{\pi}) \{ \overline{P} \}$

privilege $P ::= Z (A, x)$

privilege type $Z ::= \textit{grant} | \textit{require}$

access type $A ::= \textit{read} | \textit{write} | \textit{inject}$

ribbon $RB ::= \text{ribbon } R (\overline{D} \overline{\pi})$

class $CL ::= \text{class } C \{ C f ; K M \}$

construction $K ::= C (\overline{C} \overline{f} ; f = \overline{f};)$

method $M ::= T m (\overline{T} \overline{\pi}) \{ \overline{t}_i \}$

type $T ::= C | D | R | \textit{Ref} T$

term $t ::= x | v | t.f | t.f = t | t.m (\overline{t})$

\quad | \texttt{new<t> } C (\overline{t}) | \texttt{newdomain } D (\overline{t})$

\quad | \texttt{newribbon } R (\overline{t}) (\overline{t}_i)$

\quad | \texttt{newthread}<t> (\overline{t}_i) | \texttt{getdomain} (t, x)$

value $v ::= \pi (D) | r (R) | l (\pi, C)$

Figure 3.1.: RIBBONJ-LITE language syntax.

protection domain values, ribbon values, or other locations. As field values are stored by reference via locations, fields may thus only store references to object values. This simplifies the typing and evaluation rules and related theorems, without the loss of any interesting points from the proofs. $C$ denotes the class of the object, while $\pi$ denotes the protection domain value within which the object is contained. Locations may simply be written $l$ for brevity when the containing protection domain and type $C$ are not germane to the context.

A class table $CT$ is a mapping from class names $C$ to class declarations $CL$. Similarly, there is a protection domain table $PDT$ and ribbon table $RT$ that map protection domain and ribbon names to their declarations. For simplicity, it is assumed that the $CT$, $PDT$, and $RT$ tables are fixed. A complete program is a tuple $(CT, PDT, RT, PR)$ consisting of the definitions of the fixed tables, as well as a statement $(PR)$ that creates an initial ribbon, passing the global protection domain value $\pi \bot$ to the ribbon constructor (the specified type of the initial ribbon must therefore have a ribbon constructor that accepts a single argument of type $PDomain$).

In RIBBONJ-LITE we chose not to overload the $\texttt{new}$ operator for readability, but instead introduced different operators to create new ribbons, protection domains, and objects. The $\texttt{newdomain}$ operator creates new protection domains and accepts
**Field lookup**

\[
\text{fields}(\text{Thread}) = \emptyset \\
\Rightarrow CT(C) = \text{class } C \{ C \text{ f}; K \text{ M}\}
\]

**Method type lookup**

\[
CT(C) = \text{class } C \{ C \text{ f}; K \text{ M}\} \\
T \text{ m}(\overline{T} \overline{x}) \{ \overline{t}_i \} \in \overline{M} \\
\Rightarrow \text{mtype}(m, C) = \overline{T} \rightarrow \overline{T}
\]

**Method body lookup**

\[
CT(C) = \text{class } C \{ C \text{ f}; K \text{ M}\} \\
T \text{ m}(\overline{T} \overline{x}) \{ \overline{t}_i \} \in \overline{M} \\
\Rightarrow \text{mbody}(m, C) = (\overline{x}, \overline{t})
\]

**Protection domain field lookup**

\[
d\text{fields}(\text{PDomain}) = \emptyset \\
\Rightarrow PDT(D) = \text{protectiondomain } D(\overline{D} \overline{x})... \\
d\text{fields}(D) = \overline{D} \overline{x}
\]

**Protection domain privilege lookup**

\[
PDT(D) = \text{protectiondomain } D(\_\_ \{ \overline{P} \}) \\
Z(A, x); \in \overline{P} \\
\Rightarrow (Z, A, x) \in \text{domprivs}(D)
\]

**Ribbon field lookup**

\[
RT(R) = \text{ribbon } R(\overline{D} \overline{x}) \\
r\text{fields}(R) = \overline{D} \overline{x}
\]

**Concrete protection domain values referenced within the relevant protection domain type. newribbon creates new ribbons and accepts the protection domain values within which the new ribbon should operate.**

The first protection domain value passed to the ribbon constructor is also defined as the default protection domain for the new ribbon. In addition to creating a new memory execution context the new ribbon also implicitly creates a new thread within the new ribbon’s memory context. **getdomain allows retrieval of one of the named protection domain values associated with a given ribbon value (or the default protection domain value).** Thread creation is augmented to indicate the ribbon value of the ribbon in which the new thread should execute.
Object creation is augmented to accept the protection domain value in which the new object should be created.

In RibbonJ-lite the `thisribbon` keyword provides access to the ribbon value of the ribbon that owns the currently executing thread; however, in RibbonJ-lite the `thisribbon` keyword is not allowed to be used within method body terms. Supporting `thisribbon` within method body terms while still having typing be syntax directed and modular would require modeling subtyping between ribbon types and downcasts. Rather than unnecessarily complicate the model and distract from its main purpose (modeling isolation properties), the choice was made to limit `thisribbon` support only to program terms. The `thisribbon` keyword in method body terms can be emulated in RibbonJ-lite by instead requiring the ribbon value to be passed as a parameter to the method. Similar to how the `this` keyword is implemented, the `thisribbon` keyword is implemented as a special variable, which allows typing to remain strictly syntax directed. Figure 3.2 has auxiliary definitions for the language.

3.2 Typing

Typing rules follow straightforwardly from the syntax and the above descriptions and are given in Figure 3.3 and Figure 3.4. Term typings are similar to those for FJ, with the exception that the target of field accesses and method invocations expect references to objects instead of object themselves. Object construction typing is likewise modified to expect a reference to the new object to be returned. Additional term typing rules are present for protection domain instantiation, ribbon instantiation, thread instantiation, and a few other new terms. Method and class well formedness is simplified compared to FJ due to the lack of subtyping. Syntactically correct protection domain declarations and ribbon declarations are always well typed. Despite the presence of locations, a store typing is not necessary, because location values explicitly contain the type of the value being reference by the location.
Figure 3.3.: Term typing rules for RIBBONJ-LITE.

Figure 3.4.: Method, class, protection domain, and ribbon typing rules for RIBBONJ-LITE.
3.3 Dynamic Semantics

Figures 3.5–3.8 define an operational semantics for RIBBONJ-LITE. Global evaluation is of the form \( \langle Q, E, P, R \rangle \rightarrow \langle Q', E', P', R' \rangle \) where \( Q \) is a parallel composition of executing threads \((T \{ \ldots \})\)

\[
Q ::= \emptyset \mid Q \cdot T \{ t \}
\]

and where \( E \) is an object store, \( P \) is the privilege set, and \( R \) is the *ribbon thread context* – an ordered set of ribbon values storing the current ribbon value for each active thread. The privilege set tracks permissions for source protection domains \( \pi_{SD} \) on target domains \( \pi_{TD} \) using tuples \( \langle \pi_{SD}, \pi_{TD}, \text{privilege} \rangle \). The object store \( E \) is explicitly typed, which simplifies the proofs. The initial \( Q \) value is initialized to be a thread that contains the `newribbon` term from the starting statement of the program \((PR)\).

Figure 3.5 defines global evaluation rules. Rule CONGR-E relates local and global evaluation. Congruence on global evaluation (CONGR-E) relies on *evaluation contexts*, \( E \), defined in Figure 3.6. Rule FORK-E creates a new thread of control operating within the context of the specified ribbon. The ribbon thread context \( R = r \langle R \rangle \) is appended with the given ribbon value, remembering for each thread the ribbon within which the thread is operating. The `newthread` expression on the creating thread of execution is replaced with an expression to create a new `Thread` object representing the newly created thread. Rule END-E ends a thread that has finished computing its terms into values.

The evaluation contexts in Figure 3.6 implicitly define local congruence evaluation rules with the name given by the evaluation context. For example, the evaluation context \( v . m \langle v, E, t \rangle \) implicitly defines the local evaluation rule METH-ARG-CTX-E:

\[
\frac{\langle t, E, P, R \rangle \rightarrow \langle t', E', P', R' \rangle}{\langle v . m \langle v, t, t' \rangle, E, P, R \rangle \rightarrow \langle v . m \langle v, t', t \rangle, E', P', R' \rangle}
\]
\[
\langle t, \mathcal{E}, \mathcal{P}, \mathcal{R}_j \rangle \rightarrow \langle t', \mathcal{E}', \mathcal{P}', \mathcal{R}_j \rangle
generate (Congr-E)
\]
\[
\langle \overline{T(\_)} \cdot T'_j (E[t]) \cdot \overline{T(\_)}''', \mathcal{E}, \mathcal{P}, \overline{\mathcal{R}} \cdot \mathcal{R}_j \cdot \overline{\mathcal{R}} \rangle \rightarrow
\langle \overline{T(\_)} \cdot T'_j (E[t']) \cdot \overline{T(\_)}''', \mathcal{E}', \mathcal{P}', \overline{\mathcal{R}} \cdot \mathcal{R}_j \cdot \overline{\mathcal{R}}' \rangle
\] (Fork-E)
\[
\langle \overline{T(\_)} \cdot T'_j (E[\text{newthread} <E > \{ \overline{t}; \}]) \cdot \overline{T(\_)}''', \mathcal{E}, \mathcal{P}, \overline{\mathcal{R}} \rangle \rightarrow
\langle \overline{T(\_)} \cdot T'_j (E[\text{new Thread}()]) \cdot \overline{T(\_)}''', \mathcal{E}, \mathcal{P}, \overline{\mathcal{R}} \cdot \mathcal{R}(R) \rangle
\]
\[
\overline{\mathcal{R}} = \overline{\mathcal{R}}'' \cdot \mathcal{R}_j \cdot \overline{\mathcal{R}}''' \quad \overline{\mathcal{R}} = \overline{\mathcal{R}}'' \cdot \overline{\mathcal{R}}'''
\] (End-E)
\[
\langle \overline{T(\_)} \cdot T'_j (\overline{v};) \cdot \overline{T(\_)}''', \mathcal{E}, \mathcal{P}, \overline{\mathcal{R}} \rangle \rightarrow \langle \overline{T(\_)} \cdot T'_j (\overline{v};)''', \mathcal{E}, \mathcal{P}, \overline{\mathcal{R}} \rangle
\]

Figure 3.5.: Global evaluation in the form \( \langle Q, \mathcal{E}, \mathcal{P}, \overline{\mathcal{R}} \rangle \rightarrow \langle Q', \mathcal{E}', \mathcal{P}', \overline{\mathcal{R}}' \rangle \).

\[
E ::= \\
\bullet \quad \text{(hole)}
\]
\[
\overline{v}; E; \overline{t} \quad \text{(Sequence-CTX-E)}
\]
\[
E.f \quad \text{=} t \quad \text{(Field-Ass-CTX-E)}
\]
\[
E.f \quad \text{=} E \quad \text{=} \quad \text{(Field-Ass-Tgt-CTX-E)}
\]
\[
v.f \quad \text{=} E \quad \text{(Field-Ass-Val-CTX-E)}
\]
\[
E.m (\overline{t}) \quad \text{(Meth-Tgt-CTX-E)}
\]
\[
v.m (\overline{v}, E, \overline{t}) \quad \text{(Meth-Arg-CTX-E)}
\]
\[
\text{new} <E > C (\overline{t}) \quad \text{(Cons-DomArg-CTX-E)}
\]
\[
\text{new} <\pi > C (\overline{v}, E, \overline{t}) \quad \text{(Cons-Arg-CTX-E)}
\]
\[
\text{newribbon} \quad R (\overline{v}, E, \overline{t}) \{ \overline{\overline{v}}; \} \quad \text{(New-Ribbon-DomArg-CTX-E)}
\]
\[
\text{newthread} \quad \text{<E> \{ \overline{t}; \}} \quad \text{(New-Thread-RibArg-CTX-E)}
\]
\[
\text{newdomain} \quad D (\overline{v}, E, \overline{t}) \quad \text{(New-Domain-DomArg-CTX-E)}
\]
\[
\text{getdomain} (E, x) \quad \text{(Get-Domain-DomArg-CTX-E)}
\]

Figure 3.6.: Evaluation contexts, \( E \), for RIBBONJ-LITE.
\[(Z, A, x) \in \text{domprivs}(D) \quad Z = \text{grant} \quad (x: \pi_x) \in \Pi \quad \frac{}{(\pi, \pi, A) \in \text{buildnewprivs}(P, R, \Pi, \pi(D))} \quad \text{(Build-Grant-Privs)}\]

\[(Z, A, x) \in \text{domprivs}(D) \quad Z = \text{require} \quad (x: \pi_x) \in \Pi \quad \frac{}{(\pi, \pi, A) \in \text{buildnewprivs}(P, R, \Pi, \pi(D))} \quad \text{(Build-Require-Privs)}\]

Figure 3.7.: Definition of the buildnewprivs helper relation, which is used to define the set of privileges that a newly created protection domain acquires. The relation is contextual to \(P\), the current privilege set, \(R\), the ribbon thread context, \(\Pi\), the mapping from protection domain parameter name to protection domain value, and \(\pi(D)\), the protection domain value representing the newly created protection domain.

The pattern for the other implicitly defined congruence rules is the same. Naming these implicitly defined rules allows for greater precision in the progress proof.

Figure 3.8 defines local evaluation rules. Rules for local evaluation \(\langle t, E, P, R \rangle \rightarrow \langle t', E', P', R \rangle\) only include one ribbon value, \(R\), corresponding to the ribbon value associated with the thread of the single thread term being reduced. Note that local evaluation never changes \(R\). For simplicity it is assumed that only RIBBONJ-LITE programs that produce the same result regardless of thread evaluation order need to be considered. Adding mechanisms to the formal model to properly deal with programs that are not deterministic with respect to thread evaluation order is not worth the extra complexity, as it would detract from the main purpose of the formal model (to model heap isolation properties). Using formal models to understand determinism within concurrent programs is explored by Ziarek [124].

Rule \text{SEQUENCE-NEXT-E} reduces a sequence to the last value after all terms in the sequence have been fully evaluated. Rule \text{Cons-E} creates a new object within a specified protection domain, verifying that the current ribbon has the \text{inject} privilege on the target domain. The access check occurring within the second line of the rule’s antecedent ensures that at least one of the protection domains associated with the current thread’s ribbon has the \text{inject} privilege on the protection domain within which the new object is being created. Rules \text{FIELD-ACC-E} and \text{FIELD-Ass-E} allow field access and assignment and checks for the \text{read} or \text{write} privilege as appropriate.
\[
\langle v_1; \ldots; v_n, \mathcal{E}, \mathcal{P}, \mathcal{R} \rangle \longrightarrow \langle v_n, \mathcal{E}, \mathcal{P}, \mathcal{R} \rangle \quad \text{(SEQUENCE-NEXT-E)}
\]

\[
\mathcal{E}(l) = \left[\ldots, \text{Ref } C_i : f_i : v, \ldots \right] \quad \exists (x_i : \pi_i) \in r(R) | \langle \pi_i, \pi, \text{read} \rangle \in \mathcal{P} \\
\langle l(\pi, C).f_i, \mathcal{E}, \mathcal{P}, r(R) \rangle \longrightarrow \langle v, \mathcal{E}, \mathcal{P}, r(R) \rangle 
\] (FIELD-ACC-E)

\[
\mathcal{E}(l) = \left[\ldots, \text{Ref } C_i : f_i : v, \ldots \right] \quad \exists (x_i : \pi_i) \in r(R) | \langle \pi_i, \pi, \text{write} \rangle \in \mathcal{P} \\
\langle l(\pi, C).f_i=v', \mathcal{E}, \mathcal{P}, r(R) \rangle \longrightarrow \langle v', \{l\rightarrow \left[\ldots, \text{Ref } C_i : f_i : v', \ldots \right]\} \mathcal{E}, \mathcal{P}, r(R) \rangle 
\] (FIELD-ASS-E)

\[
\text{mbody}(m, C) = (\bar{\pi}, \bar{\alpha}) \quad \exists (x_i : \pi_i) \in r(R) | \langle \pi_i, \pi, \text{read} \rangle \in \mathcal{P} \\
\langle l(\pi, C).m(\bar{\nu}), \mathcal{E}, \mathcal{P}, r(R) \rangle \longrightarrow \langle \{\langle \text{this}, \bar{\pi} \rangle \bar{\alpha}, \mathcal{E}, \mathcal{P}, r(R) \rangle \quad \text{(METH-E)}
\]

\[
l \not\in \text{dom}(\mathcal{E}) \quad \mathcal{E}' = \{l\rightarrow \left[\text{Ref } C_1 : f_1 : v_1, \ldots, \text{Ref } C_n : f_n : v_n \right] \mathcal{E} \} \\
\text{fields}(C) = \mathcal{C} \quad \bar{\alpha} \]

\[
\langle \text{new}<\pi> C(\bar{\nu}), \mathcal{E}, \mathcal{P}, r(R) \rangle \longrightarrow \langle l(\pi, C), \mathcal{E}', \mathcal{P}, r(R) \rangle 
\] (CONS-E)

\[
r(R) = [\text{default}: v_1, x_1 : v_1, \ldots, x_n : v_n] \\
r \text{fields}(R) = \bar{D} \quad \bar{\alpha} 
\]

\[
\langle \text{newribbon } R(\bar{\nu}) (\bar{\pi}), \mathcal{E}, \mathcal{P}, \mathcal{R} \rangle \longrightarrow \langle \text{newthread}<r(R) >\{\{r(R) \} \text{/thisribbon}\} \bar{\pi}; r(R), \mathcal{E}, \mathcal{P}, \mathcal{R} \rangle 
\] (NEW-RIBBON-E)

\[
\ldots (\pi(D), \ldots) \not\in \text{dom}(\mathcal{E}) \\
\Pi = [x_1 : v_1, \ldots, x_n : v_n] \quad \text{dfields}(D) = \bar{D} \quad \bar{\alpha} \\
\mathcal{P}' = \text{buildnewprivs}(\mathcal{P}, \mathcal{R}, \Pi, \pi(D)) \cup \mathcal{P} 
\]

\[
\langle \text{newdomain } D(\bar{\nu}), \mathcal{E}, \mathcal{P}, \mathcal{R} \rangle \longrightarrow \langle \pi(D), \mathcal{E}, \mathcal{P}', \mathcal{R} \rangle 
\] (NEW-DOMAIN-E)

\[
\text{v} = r(R) \quad \text{rfields}(R) = \bar{D} \quad \bar{\alpha} 
\]

\[
\langle \text{getdomain } (v, f_i), \mathcal{E}, \mathcal{P}, \mathcal{R} \rangle \longrightarrow \langle \pi(D_i), \mathcal{E}, \mathcal{P}, \mathcal{R} \rangle 
\] (GET-DOMAIN-E)

Figure 3.8.: Local evaluation in the form \( \langle t, \mathcal{E}, \mathcal{P}, \mathcal{R} \rangle \longrightarrow \langle t', \mathcal{E}', \mathcal{P}', \mathcal{R} \rangle \).

The rule for method calls, METH-E, verifies that the current ribbon has the read privilege on the target object.

Rule NEW-RIBBON-E creates a new ribbon value and forks a thread using this value. The / operator is a special variable substitution operator that is used to substitute the thisribbon special variable with the value representing the new ribbon; however, it differs from normal variable substitution in that it does not recursively substitute inside of any newribbon terms (doing so would not be correct, because the value of thisribbon inside of these newribbon terms would be different). The new
ribbon value’s named protection domain values are initialized using the protection
domain values passed as arguments. Note that the value to which the \textit{newribbon}
term evaluates is the new ribbon value, not the value that the \textit{newthread} expression
evaluates to.

Rule \textit{New-Domain-E} creates a new protection domain value using the given pro-
tection domain type as a template. This template is used to add to the privilege set
to both (a) grant other protection domains privileges to the new domain (through \textit{grant}), and (b) grant the new protection domain privileges to other domains (through \textit{require}). These effects are modeled through \textit{buildnewprives}. Rule \textit{Get-Domain-E}
retrieves a named protection domain value from a given ribbon value.

3.4 Type Preservation

Type preservation is proved by following the proof strategy used in FJ [59]. The
lack of subtyping simplifies the proof’s complexity. Of all of the additional features
in \textsc{RibbonJ-lite} vs FJ, locations add the most complexity to the proof. However, a
store typing is not needed (removing the need for supporting properties and lemmas
required to show that a store is well typed with respect to a typing context) because
location values explicitly encode the types of the values they refer to.

First, a few straightforward lemmas are developed.

\textbf{Lemma 3.4.1 (Term Substitution Operator / Preserves Typing)} \textit{If }\Gamma, \vec{x} : T  \vdash 
t : T \textit{ and } \Gamma  \vdash \vec{t} : T \textit{, then } \Gamma  \vdash \{x/\vec{t}\} t : T

\textbf{Proof} Straightforward induction on the derivation of \(\Gamma, \vec{x} : T \vdash t : T\). The proof
follows the same strategy as Lemma A.1.2 in FJ [59], except that it is much simpler
because of the lack of subtyping. Intuitively, a substituted term is always replaced
with a term that has an identical type, and therefore the type of the overall expression
does not change. \hfill \blacksquare
Lemma 3.4.2 (Term Special Substitution Operator // Preserves Typing) If
\[ \Gamma, \bar{x} : T \vdash t : T \text{ and } \Gamma \vdash \overline{t} : \overline{T}, \text{ then } \Gamma \vdash \{\overline{\gamma/\bar{x}}\}t : T \]

**Proof** The proof is nearly identical to the proof for Lemma 3.4.1, except for the case of the \textsc{T-NewRibbon} typing rule. In this case, substitution is not applied into the sequence of terms executed in the context of the new ribbon. It follows that in this special case the type of the expression does not change, because no changes are made to the expression.

Lemma 3.4.3 (Weakening) If \( \Gamma \vdash t : T \), then \( \Gamma, x : T' \vdash t : T \).

**Proof** Straightforward induction, following the same strategy as Lemma A.1.3 in FJ [59].

Theorem 3.4.4 (Preservation) If \( \Gamma \vdash t : T \) and \( \langle t, \mathcal{E}, \mathcal{P}, \mathcal{R} \rangle \rightarrow \langle t', \mathcal{E}', \mathcal{P}', \mathcal{R} \rangle \), then \( \Gamma \vdash t' : T \).

**Proof** By induction on derivation of \( t \rightarrow t' \), with a case analysis on the reduction rule used. Note that \( \mathcal{E}, \mathcal{P}, \) and \( \mathcal{R} \) are not relevant to the proof and so are elided when writing out evaluation rules in this proof to keep the notation easier to read.

**Case** \textsc{Sequence-Next-E}:
\[
\begin{align*}
\Gamma & \vdash v_1 : T_1 \quad \ldots \quad \Gamma \vdash v_n : T_n \\
t & = v_1 ; \ldots ; v_n \\
t' & = v_n
\end{align*}
\]

By rule \textsc{T-Sequence},
\[ \Gamma \vdash v_1 ; \ldots ; v_n : T_n \]

It immediately follows that setting \( T = T_n \) completes this case.
Case **FIELD-ACC-E:**

\[ \mathcal{E}(l) = [\_, \text{Ref } C_i ; f_i ; v_i, \_] \]

\( \text{fields}(C) = \overline{C \ f} \)

\( t = l(\pi, C) \cdot f_i \)

\( t' = v_i \)

By rule **T-FIELDGET,**

\[ \Gamma \vdash l(\pi, C) \cdot f_i : \text{Ref } C_i \]

From the explicit typing information in \( \mathcal{E}(l) \) for \( f_i \) we have

\[ \Gamma \vdash v_i : \text{Ref } C_i \]

Setting \( T = \text{Ref } C_i \) completes this case.

Case **FIELD-ASS-E:**

\[ \mathcal{E}(l) = [\_, \text{Ref } C_i ; f_i ; v_i, \_] \]

\( \text{fields}(C) = \overline{C \ f} \)

\( \Gamma \vdash v' : \text{Ref } C_i \)

\( t = l(\pi, C) \cdot f_i = v' \)

\( t' = v' \)

By rule **T-FIELDSET,**

\[ \Gamma \vdash l(\pi, C) \cdot f_i = v' : \text{Ref } C_i \]

It immediately follows that setting \( T = \text{Ref } C_i \) completes this case.

Case **METH-E:**

\[ \text{mbody}(m, C) = (\pi, t_n ; \_ ; t_0) \]

\( t = l(\pi, C) \cdot m(\overline{v}) \)

\( t' = \{v\}_{\text{this} / \pi} t_n ; \_ ; t_0 \)
By rule T-Invoke,

\[ \text{mtype}(m, C) = \overline{T} \rightarrow T'_0 \]

\[ \Gamma \vdash \overline{v} : \overline{T} \]

\[ \Gamma \vdash l(\pi, C) . m(\overline{v}) : T'_0 \]

Also we know that \( m \) OK in \( C \), so it follows that

\[ \bar{v} : \overline{T}, \text{this} : \text{Ref} C \vdash t_0 : T'_0 \]

By Lemma 3.4.3,

\[ \Gamma, \bar{v} : \overline{T}, \text{this} : \text{Ref} C \vdash t_0 : T'_0 \]

Then by rule T-LVALUE and by Lemma 3.4.1,

\[ \Gamma \vdash l(\pi, C) : \text{Ref} C \]

\[ \Gamma \vdash \{/\text{this}:/\bar{v}\}t_0 : T'_0 \]

Finally, by rule T-SEQUENCE,

\[ \Gamma \vdash \{/\text{this}:/\bar{v}\}t_0; \ldots; t_0 : T'_0 \]

Setting \( T = T'_0 \) finishes this case.

**Case** CONS-E:

\[ \Gamma \vdash \pi(D) : D \]

\[ \text{fields}(C) = \overline{C} \overline{f} \]

\[ \Gamma \vdash \overline{v} : \text{Ref} \overline{C} \]

\[ t = \text{new}<\pi(D) > C(\overline{v}) \]

\[ t' = l(\pi(D), C) \]
By rule T-New,

\[ \Gamma \vdash \mathtt{new<\pi (D) > C (\overline{v}) : Ref \ C} \]

By rule T-LVALUE,

\[ \Gamma \vdash l (\pi (D), C) : \mathtt{Ref \ C} \]

Setting \( T = \mathtt{Ref \ C} \) therefore completes this case.

**Case** NEW-RIBBON-E:

\[
\begin{align*}
    r (R) &= [\mathtt{default:v}_1, x_1:v_1, \ldots, x_n:v_n] \\
    rfields (R) &= D \pi \\
    t &= \mathtt{newribbon \ R (\overline{v}) \{\overline{t}_i\}} \\
    t' &= \mathtt{newthread<r (R) >\{\{r (R) /\text{thisribbon}\}\overline{t}_i\}; r (R)}
\end{align*}
\]

By rule T-NewRibbon,

\[ \Gamma \vdash \overline{v} : D \]

\[ \Gamma, \text{thisribbon} : R \vdash \overline{t} : T \]

\[ \Gamma \vdash \mathtt{newribbon \ R (\overline{v}) \{\overline{t}_i\} : R} \]

Then by rule T-RVALUE and by Lemma 3.4.2,

\[ \Gamma \vdash r (R) : R \]

\[ \ldots \]

\[ \Gamma \vdash \{r (R) /\text{thisribbon}\} t_i : T_i \]

\[ \ldots \]

By rule T-NewThread,

\[ \Gamma \vdash \mathtt{newthread<r (R) >\{\{r (R) /\text{thisribbon}\}\overline{t}_i\}; : \mathtt{Ref \ Thread}} \]
Finally, by rule T-SEQUENCE,

\[ \Gamma \vdash \text{newthread} < r(R) > \{ \{ r(R) \} / \text{thisribbon} \} \overline{t_i}; r(R) : R \]

Setting \( T = R \) completes this case.

**Case NEW-DOMAIN-E:**

\[ dfield(D) = \overline{D} \overline{x} \]

\[ t = \text{newdomain } D(\overline{v}) \]

\[ t' = \pi(D) \]

By rule T-NEWDOMAIN,

\[ \Gamma \vdash \text{newdomain } D(\overline{v}) : D \]

By rule T-DVALUE,

\[ \Gamma \vdash \pi(D) : D \]

Setting \( T = D \) completes this case.

**Case GET-DOMAIN-E:**

\[ r(R) = [\ldots, f_i \cdot \pi(D_i), \ldots] \]

\[ rfield(R) = \overline{D} \overline{f} \]

\[ t = \text{getdomain}(r(R), f_i) \]

\[ t' = \pi(D_i) \]

By rule T-GETDOMAIN,

\[ \Gamma \vdash \text{getdomain}(r(R), f_i) : D_i \]

By rule T-DVALUE,

\[ \Gamma \vdash \pi(D_i) : D_i \]
Setting $T = D_i$ completes this case.

**Case** for all local congruence rules (*-CTX-E):

All of these implicitly defined local congruence evaluation rules involve the evaluation of some subterm $t_0$ to $t'_0$ with the enclosing expression remaining otherwise unchanged. By the induction hypothesis, if $\Gamma \vdash t_0 : T_0$, then $\Gamma \vdash t'_0 : T_0$. By Lemma 3.4.1, if the original term $t$ has type $T$, then substituting $t_0$ (of type $T_0$) with $t'_0$ (of the same type) within $t$ does not change the type of $t$. Thus, after evaluation, $t$ still has type $T$, completing this case.

3.5 Progress

In the absence of casts, the original progress theorem of FJ is simplified, notwithstanding our additions of locations (since locations explicitly include type information). We define stuck terms to account for the cases where a protection domain privilege check fails.

First, we prove a lemma that shows that for every well typed term that is able to be evaluated, there is a unique decomposition of such a term into an evaluation context and a redex, and the redex is able to be reduced by one of the “base case” evaluation rules. As before, if $E$, $P$, and $\mathcal{R}$ are not changed by local evaluation they are elided from the notation.

**Lemma 3.5.1 (Unique Decomposition)** If $\langle t, E, P, \mathcal{R} \rangle \rightarrow \langle t', E', P', \mathcal{R} \rangle$ then there are unique $E$, $e$, and $e'$ such that (1) $t = E[e]$, (2) $t' = E[e']$, and (3) $e \rightarrow e'$ by one of the “redex reduction rules”: Cons-E, Field-Acc-E, Field-Ass-E, Meth-E, Sequence-Next-E, New-Domain-E, New-Ribbon-E, or Get-Domain-E.
Proof. By induction on the derivation of \( \langle t, \mathcal{E}, \mathcal{P}, \mathcal{R} \rangle \rightarrow \langle t', \mathcal{E}', \mathcal{P}', \mathcal{R} \rangle \), with a case analysis on the last derivation rule used.

For each of the cases of Cons-E, Field-Acc-E, Field-Ass-E, Meth-E, Sequence-Next-E, New-Domain-E, New-Ribbon-E, or Get-Domain-E, setting \( E = \bullet, e = t \), and \( e' = t' \) completes the case.

Case Meth-Arg-CTX-E:

\[
t_i \rightarrow t'_i
\]

\[
t = v.m(\overline{v}, t_i, \overline{I})
\]

\[
t' = v.m(\overline{v}, t'_i, \overline{I})
\]

By the induction hypothesis, there are unique \( E_0, e_0 \), and \( e'_0 \) such that (1) \( t_0 = E_0[e_0] \), (2) \( t'_0 = E_0[e'_0] \), and (3) \( e_0 \rightarrow e'_0 \) by one of the redex reduction rules. Setting \( E = v.m(\overline{v}, E_0, \overline{I}) \), \( e = e_0 \), and \( e' = e'_0 \) completes this case.

All of the other cases for *-CTX-E rules are similar. The key point in proving each case is that for a given expression to be evaluated, and a given particular sub-term that can be evaluated under one of the redex reduction rules, there is only one possible evaluation context that can be used to place the hole where the given sub-term is being evaluated. By inspection of the evaluation contexts in Figure 3.6 it can be seen that for each type of syntactic expression for a particular location in that expression where is sub-term is being evaluated by a redex rule there is exactly one evaluation context that is applicable. Combining this with a one-to-one correspondence between each evaluation context and an *-CTX-E rule, we conclude that this Lemma holds for all of the remaining cases.

\[\blacksquare\]

**Theorem 3.5.2 (Local Progress)** Suppose \( t \) is a closed, well-typed term and \( t \) is within \( \langle t, \mathcal{E}, \mathcal{P}, \mathcal{R} \rangle \). Then either

1. \( \langle t, \mathcal{E}, \mathcal{P}, \mathcal{R} \rangle \rightarrow \langle t', \mathcal{E}', \mathcal{P}', \mathcal{R} \rangle \) for some \( t', \mathcal{E}' \), and \( \mathcal{P}' \)
2. \( t \) is a value

3. \( t \) is a “stuck term” in that \( \exists E \) such that any of the following hold:

\begin{enumerate}
\item \( t = E[\text{newthread} < r (R) > \{ \overline{t}; \overline{t} \}] \)
\item \( t = E[\text{new} < \pi > C (\overline{x})] \land \not\exists (x_i: \pi_i) \in R | (\pi_i, \pi, \text{inject}) \in \mathcal{P} \)
\item \( t = E[l (\pi, C) . f_i] \land \not\exists (x_i: \pi_i) \in R | (\pi_i, \pi, \text{read}) \in \mathcal{P} \)
\item \( t = E[l (\pi, C) . f_i=v'] \land \not\exists (x_i: \pi_i) \in R | (\pi_i, \pi, \text{write}) \in \mathcal{P} \)
\item \( t = E[l (\pi, C) . m (\overline{x})] \land \not\exists (x_i: \pi_i) \in R | (\pi_i, \pi, \text{read}) \in \mathcal{P} \)
\end{enumerate}

**Proof** By induction on the typing derivation of \( t \), with a case analysis on the last typing rule used:

**Case T-SEQUENCE:**
\[
t = t_n; \ldots; t_0
\]

If \( t_n; \ldots; t_{i+1} \) are values and \( t_i \) is not a value, then by the induction hypothesis on \( t_i \), the following are possible: (1) \( t_i \rightarrow t_i' \) and then \text{SEQUENCE-CTX-E} applies, or (2) \( t_i \) is a value, but for this sub-case we already know \( t_i \) is not a value so this sub-case does not apply, or (3) \( t_i = E_i[\text{stuck}] \), and then \( t = E[\text{stuck}] \) for \( E = t_n; \ldots; t_{i+1}; E_i; \ldots; t_0 \).

If \( t_n; \ldots; t_0 \) are all values, then \text{SEQUENCE-NEXT-E} applies and progress is made.

**Case T-FIELDGET:**
\[
t = t_0 . f_i
\]
\[
\mathcal{R} = r (R)
\]

By the induction hypothesis on \( t_0 \), the following are possible: (1) \( t_0 \rightarrow t_0' \) and then \text{FIELD-Acc-CTX-E} applies, or (2) \( t_0 = E_0[\text{stuck}] \), and then \( t = E[\text{stuck}] \) for \( E = E_0 . f \), or (3) \( t_0 \) is a value. If \( t_0 \) is a value \( (t_0 = l (\pi, C)) \) there are two cases: if \( \exists (x_i: \pi_i) \in r (R) | (\pi_i, \pi, \text{read}) \in \mathcal{P} \), then \text{FIELD-Acc-E} applies and progress is
made; otherwise, we are stuck. However, applying Lemma 3.5.1 to the last step of
the evaluation derivation of $t$ implies that $t = E[e]$ for some unique $E$ and $e$. By
inspection, this unique $e$ must be $l(\pi, C).f_i$, and thus stuck term $(c)$ of this theorem
is fully applicable.

**Case T-FieldSet:**

$t = t_0.f_i=t_d$

$\mathcal{R} = r(R)$

If $t_0$ is not a value, then by the induction hypothesis on $t_0$, the following are
possible: (1) $t_0 \rightarrow t'_0$ and then Field-Ass-Tgt-CTX-E applies, or (2) $t_0$ is a value,
but for this sub-case we already know $t_0$ is not a value so this sub-case does not apply,
or (3) $t_0 = E_0[\text{stuck}]$, and then $t = E[\text{stuck}]$ for $E = E_0 . f = t_d$.

If $t_0$ is a value and $t_d$ is not a value, then by the induction hypothesis on $t_d$, the
following are possible: (1) $t_d \rightarrow t'_d$ and then Field-Ass-Val-CTX-E applies, or (2)
t_d is a value, but for this sub-case we already know $t_d$ is not a value so this sub-case does not apply,
or (3) $t_d = E_d[\text{stuck}]$, and then $t = E[\text{stuck}]$ for $E = t_0 . f = E_d$.

If both $t_0$ and $t_d$ are values ($t_0 = l(\pi, C)$), then there are two cases: if $\exists \langle x_i: \pi_i \rangle \in r(R) |\langle \pi_i, \pi, \text{write} \rangle \in \mathcal{P}$, then Field-Ass-E applies and progress is made; otherwise,
we are stuck. However, applying Lemma 3.5.1 to the last step of the evaluation
derivation of $t$ implies that $t = E[e]$ for some unique $E$ and $e$. By inspection, this
unique $e$ must be $l(\pi, C).f_i=t_d$, and thus stuck term $(d)$ of this theorem is fully
applicable.

**Case T-Invoke:**

$t = t_0.m(T)$

$\mathcal{R} = r(R)$

If $t_0$ is not a value, then by the induction hypothesis on $t_0$, the following are
possible: (1) $t_0 \rightarrow t'_0$ and then Meth-Tgt-CTX-E applies, or (2) $t_0$ is a value, but
for this sub-case we already know \( t_0 \) is not a value so this sub-case does not apply, or (3) \( t_0 = E_0[stuck] \), and then \( t = E[stuck] \) for \( E = E_0.m(t) \).

If \( t_0 \) is a value and \( t_1..t_{i-1} \) are values and \( t_i \) is not a value, then by the induction hypothesis on \( t_i \), the following are possible: (1) \( t_i \rightarrow t'_i \) and then Meth-Arg-CTX-E applies, or (2) \( t_i \) is a value, but for this sub-case we already know \( t_i \) is not a value so this sub-case does not apply, or (3) \( t_i = E_i[stuck] \), and then \( t = E[stuck] \) for \( E = t_0.m(t_1,..,E_i,..,t_n) \).

If both \( t_0 \) and \( t_1..t_n \) are all values (\( t_0 = l(\pi,C) \)), then there are two cases: if \( \exists (x_i: \pi_i) \in r(R) | (\pi_i, \pi, \text{read}) \in \mathcal{P} \), then Meth-E applies and progress is made; otherwise, we are stuck. However, applying Lemma 3.5.1 to the last step of the evaluation derivation of \( t \) implies that \( t = E[e] \) for some unique \( E \) and \( e \). By inspection, this unique \( e \) must be \( l(\pi,C).m(t) \), and thus stuck term \( (e) \) of this theorem is fully applicable.

**Case T-New:**

\[
\begin{align*}
    t &= \text{new}<t_0> C(t) \\
    \mathcal{R} &= r(R)
\end{align*}
\]

If \( t_0 \) is not a value, then by the induction hypothesis on \( t_0 \), the following are possible: (1) \( t_0 \rightarrow t'_0 \) and then Cons-DomArg-CTX-E applies, or (2) \( t_0 \) is a value, but for this sub-case we already know \( t_0 \) is not a value so this sub-case does not apply, or (3) \( t_0 = E_0[stuck] \), and then \( t = E[stuck] \) for \( E = \text{new}<E_0> C(t) \).

If \( t_0 \) is a value and \( t_1..t_{i-1} \) are values and \( t_i \) is not a value, then by the induction hypothesis on \( t_i \), the following are possible: (1) \( t_i \rightarrow t'_i \) and then Cons-Arg-CTX-E applies, or (2) \( t_i \) is a value, but for this sub-case we already know \( t_i \) is not a value so this sub-case does not apply, or (3) \( t_i = E_i[stuck] \), and then \( t = E[stuck] \) for \( E = \text{new}<t_0> C(t_1,..,E_i,..,t_n) \).

If both \( t_0 \) and \( t_1..t_n \) are all values (\( t_0 = \pi \)), then there are two cases: if \( \exists (x_i: \pi_i) \in r(R) | (\pi_i, \pi, \text{inject}) \in \mathcal{P} \), then Cons-E applies and progress is made; otherwise, we
are stuck. However, applying Lemma 3.5.1 to the last step of the evaluation derivation of \( t \) implies that \( t = E[e] \) for some unique \( E \) and \( e \). By inspection, this unique \( e \) must be \textbf{new} \( <\pi> C(\tilde{t}) \), and thus stuck term (b) of this theorem is fully applicable.

**Case** T-NewDomain:

\[
t = \text{newdomain} \ D(\tilde{t})
\]

If \( t_1..t_{i-1} \) are values and \( t_i \) is not a value, then by the induction hypothesis, the following are possible: (1) \( t_i \rightarrow t'_i \) and then NEW-DOMAIN-DomArg-CTX-E applies, or (2) \( t_i \) is a value, but for this sub-case we already know \( t_i \) is not a value so this sub-case does not apply, or (3) \( t_i = E_i[\text{stuck}] \), and then \( t = E[\text{stuck}] \) for \( E = \text{newdomain} \ D(\ t_1,...,E_i,...,t_n) \). If \( t_1..t_n \) are all values, then NEW-DOMAIN-E applies and progress is made.

**Case** T-NewRibbon:

\[
t = \text{newribbon} \ R(\tilde{t}) \ (\tilde{\nu};)
\]

If \( t_1..t_{i-1} \) are values and \( t_i \) is not a value, then by the induction hypothesis, the following are possible: (1) \( t_i \rightarrow t'_i \) and then NEW-RIBBON-DomArg-CTX-E applies, or (2) \( t_i \) is a value, but for this sub-case we already know \( t_i \) is not a value so this sub-case does not apply, or (3) \( t_i = E_i[\text{stuck}] \), and then \( t = E[\text{stuck}] \) for \( E = \text{newribbon} \ R(\ t_1,...,E_i,...,t_n) \ (\tilde{\nu};) \). If \( t_1..t_n \) are all values, then NEW-RIBBON-E applies and progress is made.

**Case** T-NewThread:

\[
t = \text{newthread} <t_0>(\tilde{t} \ ;)
\]

By the induction hypothesis on \( t_0 \), the following are possible: (1) \( t_0 \rightarrow t'_0 \) and then NEW-THREAD-RibArg-CTX-E applies, or (2) \( t_0 = E_0[\text{stuck}] \), and then \( t = E[\text{stuck}] \) for \( E = \text{newthread} <E_0>(\tilde{t} \ ;) \), or (3) \( t_0 \) is a value. If \( t_0 \) is a value, we are stuck, because
there is no local evaluation rule that is applicable. However, applying Lemma 3.5.1 to
the last step of the evaluation derivation of \( t \) implies that \( t = E[e] \) for some unique \( E \)
and \( e \). By inspection, this unique \( e \) must be \texttt{newthread <r (R) > (t_i)} \), and thus stuck
term \( (a) \) of this theorem is fully applicable.

**Case** T-
GetDomain:

\[
    t = \text{getdomain}(t_0, f_i)
\]

By the induction hypothesis on \( t_0 \), the following are possible: (1) \( t_0 \rightarrow t'_0 \) and then
Get-Domain-DomArg-CTX-E applies, or (2) \( t_0 \) is a value and then Get-Domain-E applies, or (3) \( t_0 = E_0[stuck] \), and then \( t = E[stuck] \) for \( E = \text{getdomain } (E_0, x) \).

**Case** T-VAR: This case does not apply, as \( t \) is a closed term.

For the cases for T-DVALUE, T-RVALUE, and T-LVALUE, \( t \) is a value.

\[\square\]

**Theorem 3.5.3 (Global Progress)** Suppose \( Q \) is a closed, well-typed normal form
under \( \rightarrow \). Then either of the following holds:

1. \( Q \) is a parallel composition of non-value terms where each term matches one of
   the second, third, fourth, or fifth non-value normal forms in Theorem 3.5.2.

2. \( Q \) is a parallel composition of zero executing threads.

**Proof** Proof by contradiction. There are two cases to consider:

1. Suppose \( Q \) is in normal form and is a parallel composition of thread terms,
   and at least one of the terms is not a value and does not match one of the
   second, third, fourth, or fifth stuck terms listed in Theorem 3.5.2. If this term
   contains a \texttt{newthread} term, then because \( Q \) is well typed and because of type
preservation, the rule Fork-E is applicable and progress is made. If this term does not contain a \textit{newthread} term then by Theorem 3.5.2 this non-value term can be further evaluated under $\rightarrow$, and thus global progress can be made under Congr-E. Thus, in either case, $Q$ cannot be in normal form.

2. Suppose $Q$ is in normal form and is a parallel composition of thread terms, and at least one of the terms is a value. For such a term, the rule End-E is applicable and progress is made. Thus, $Q$ cannot be in normal form.

3.6 Isolation

With the RibbonJ-lite semantics above, a new protection domain can grant or require privileges to or from any other domain. While this structures the heap and prevents inadvertent accesses, it is not strict enough to fully isolate or sandbox code executing in ribbons. We provide two extensions for different levels of isolation, ending with an isolation model matching that of RibbonJ.

3.6.1 Cooperative Isolation

Under this first model, a new protection domain $P$ cannot \textit{require} a privilege from another protection domain $Q$ unless at least one of the protection domains in the creator’s ribbon already has the privilege on domain $Q$. Thus, the new protection domain is unable to increase the scope of what the creator’s ribbon is able to access, but the new domain is able to grant privileges to other domains which these did not have on the creator’s ribbon. The creating ribbon thereby controls which protection domains have access to the new domain via the protection domain values it passes to the domain being constructed. While not suitable for sandboxing potentially malicious or insecure components, this isolation level does allow for more flexibility in
that a ribbon can create a new protection domain in order to communicate, indirectly, with domains which with it currently does not have privileges to communicate.

The semantics of Figures 3.5–3.8 are retained, but we replace the **Build-Require-Privs** rule of Figure 3.7 as follows:

\[
\begin{align*}
\langle Z, A, x \rangle &\in \text{domprivs}(D) \quad Z=\text{require} \quad (x:\pi_x) \in \Pi \\
\mathcal{R}=r(\mathcal{R}) \quad \exists (x_i:\pi_i) \in r(\mathcal{R}) | (\pi_i, \pi_x, A) \in \mathcal{P} \\
\langle \pi, \pi_x, A \rangle &\in \text{buildnewprivs}(\mathcal{P}, \mathcal{R}, \Pi, \pi(D))
\end{align*}
\]  

(Build-Require-Privs)

This rule has an additional check in the antecedent, ensuring that the creating ribbon already has the privilege that the new protection domain is requiring. Observe that rather than creating a new kind of stuck term, progress is still made if a new protection domain tries to require a privilege it is not allowed to receive. Instead of becoming stuck, such a privilege is not added to the new set. Local evaluation could later get stuck if it tries to access a location in a protection domain that it was not allowed to require a privilege to.

3.6.2 Full Isolation

In this level the same restrictions apply as in cooperative isolation. Additionally, a new protection domain \( P \) cannot *grant* a privilege to another protection domain \( Q \) unless domain \( Q \) already has this privilege on one of the domains of the creator’s ribbon. Under this level of isolation, a new protection domain is fully sandboxed within its creating ribbon. This matches the isolation behavior of RibbonJ. The dynamic semantics of this isolation level are the same as with cooperative isolation, except that **Build-Grant-Privs** is substituted with:

\[
\begin{align*}
\langle Z, A, x \rangle &\in \text{domprivs}(D) \quad Z=\text{grant} \quad (x:\pi_x) \in \Pi \\
\mathcal{R}=r(\mathcal{R}) \quad \exists (x_i:\pi_i) \in r(\mathcal{R}) | (\pi_x, \pi_i, A) \in \mathcal{P} \\
\langle \pi_x, \pi, A \rangle &\in \text{buildnewprivs}(\mathcal{P}, \mathcal{R}, \Pi, \pi(D))
\end{align*}
\]  

(Build-Grant-Privs)
A privilege check is added to the antecedent of the rule to ensure that a privilege can only be granted when appropriate, as defined above. The effect on progress is similar to that of cooperative isolation—no new stuck terms are added, but rather local evaluation may become stuck if it tries to use a privilege that was not granted when the protection domain was created.

3.6.3 Adaptable Isolation Levels

The isolation level need not be decided once for the whole program, but can be indicated when a new ribbon is created. Only a same or more strict isolation level than the current ribbon’s level would be allowed, and the rules for constructing the modified privilege set would depend on the isolation level.

We outline the required changes to the dynamic semantics to support adaptable isolation levels. First, ribbon values would be augmented to store the isolation levels associated with them. Ribbon creation would be augmented to allow the new isolation level to be specified. A check would be added to the evaluation rule for `newribbon` such that the isolation level of a new ribbon \( r \) could not be less restrictive than the isolation level of the ribbon value associated with the thread that is creating \( r \). Finally, the different variants of the `Build-Grant-Privs` and `Build-Require-Privs` rules would become applicable based on the isolation level of the new ribbon.

3.7 Narrowing Extension

As defined above the granularity of the sandboxing is at the thread-level—all code executes within a certain thread with the same privileges. It can be useful to provide a finer granularity where certain segments of code executed within a thread to execute with less privileges than the rest of the thread. Imagine an application that needs to call code within some untrusted plugin. In its present form, a new ribbon (and thereby new thread) would have to be created, and the plugin code executed within
this separate thread. If the trusted code and untrusted code are interwoven, this can create complexities as interaction has to be marshalled and synchronized across threads. The concept of narrowing is thus introduced to allow a thread to lower its privileges for the duration of a method call.

Narrowing is supported by optionally allowing a new ribbon value to be specified when calling a method as the new “ribbon context” in which the thread should run. A check is made if a new ribbon context is provided to ensure that the privileges are indeed the same or are being narrowed, rather than being expanded.

For brevity, the complete details of the modified semantics are elided, but rather a sketch of the required changes is given. First, the ribbon thread context \( \mathcal{R} \) would not store, for each thread, the ribbon values, but a stack of ribbon values. All inference rules that referred to the ribbon value for the currently executing thread would be changed to instead refer to the ribbon value at the top of the thread’s ribbon value stack. The method call expression would be augmented to provide the new ribbon value context in which the method executes. A check would be added to ensure that the set of protection domains for the ribbon value passed to a method call is a subset of the set of protection domains for the ribbon value at the top of the current thread’s ribbon value stack. A method return expression would be added so that the ribbon value stack for a thread could be properly maintained. Finally, the congruence rule would be adjusted to allow local evaluation to change \( \mathcal{R} \).
4 IMPLEMENTATION

The implementation of RIBBONJ is three-fold, comprised of the compiler modifications for RIBBONJ within JastAdd [41], the modifications to JikesRVM [6], and the system level support. All code and evaluation data is available for download [56].

The implementation separates support for RIBBONJ concepts into two layers – a higher-level layer implementing the semantics of protection domains and ribbons as provided within RIBBONJ, and a more flexible low-level layer that provides the basic runtime primitives necessary to support the higher-level layer. The lower layer implementation separates the notion of memory domains and protection domains. Memory domains divide the heap into different regions, just as with protection domains; however, access privileges are modeled between concrete ribbons and concrete memory domains (instead of between classes of protection domains, as it is modeled in the higher layer), and isolation properties are only enforced if it is explicitly told to do so. Additionally, ribbons are allowed to dynamically join and leave memory domains (if they have the privileges to do so).

This underlying runtime model is more flexible than the protection domain and ribbon runtime model in RIBBONJ as domain membership and privileges are dynamic rather than static. The higher layer implementation builds upon and constrains these lower-level concepts to provide the stricter semantics of RIBBONJ. This separation allows the lower layer implementation to focus strictly on the runtime aspects of the core RIBBONJ concepts, enabling higher-level layers to be built that provide semantics specific to a higher level language. Other language extensions (not in Java) could thus be implemented that are more appropriate to the specific language being extended, without having to modify the functionality provided by the lower layer.
4.1 Language Frontend: JastAdd

The parser and bytecode compiler for RIBBONJ are implemented with JastAdd [41]. JastAdd provides declarative mechanisms for modular and composable compiler extensions.

Modules were implemented for extending the lexer and parser as well as for RIBBONJ-specific type checking, which were then composed with the unmodified modules for the JastAddJ Java 5 compiler. Most of the RIBBONJ features were implemented through straightforward AST rewriting mechanisms to translate features to a Java API implemented by the VM. A few features required modifications in the compiler backend as there was no equivalent syntactic Java expression that the RIBBONJ expression could be translated into.

4.1.1 Compiler Frontend

The compiler frontend consists of the extended lexer, new AST nodes, extended parser, type checking logic, and AST rewriting logic.

The lexer was extended to handle the new protectiondomain, grant, require, ribbon, where, getdomain, and thisribbon keywords.

The AST was extended as shown in Figure 4.1. A protection domain was modeled in the AST as a class declaration that also specifies protection domain parameters for protection domain construction. Type checking rules enforces that ProtectionDomainPrivDecl is the only type of class body declaration statement allowed within a protection domain declarations. Likewise, type checking also ensures that ProtectionDomainPrivDecl statements are only used within protection domain declarations. This strategy of having a protection domain declaration be treated as a sub-class of a class declaration by the compiler simplified the implementation, as special typing rules did not have to be added to variables that were protection domain types. Additionally, this strategy is in harmony with the runtime
ProtectionDomainDecl: ClassDecl ::= 
   DomainParm:ProtectionDomainParmDecl*;

ProtectionDomainParmDecl: ParameterDeclaration;

ProtectionDomainPrivDecl: BodyDecl ::= 
   PrivType:ProtectionDomainPrivType Privs:ProtectionDomainPriv* 
   <PrivTargetType: String> PrivTargets:Access*;

ProtectionDomainPrivType ::= <ID: String>;

ProtectionDomainPriv ::= <ID: String>;

ProtectionDomainInstanceInitializer: InstanceInitializer;

RibbonDecl: ClassDecl ::= 
   WithinProtectionDomains:RibbonParameterDecl*;

RibbonParameterDecl: ParameterDeclaration;

RibbonClassInstanceExpr: ClassInstanceExpr ::= 

ProtectionDomainArrayCreationExpr: ArrayCreationExpr ::= 
   [ProtectionDomainAccess:Expr];

ThisRibbonAccess: Access ::= <ID: String>;

GetDomainAccess: Access ::= 
   <RibbonInstance:Expr> <DomainName:Access>;

Figure 4.1.: AST extensions for RibbonJ compiler. The new AST nodes declaratively sub-class Java 5 AST nodes. AST definitions follow the form *TypeName*: *SuperTypeName* ::= *list of child nodes*. Unbracketed terms denote a child AST node of the given AST node type. Straight brackets denote optional child nodes. Angle brackets denote typed tokens. A star after a term indicates a list of child nodes of that type.
implementation, where protection domains are modeled as objects that implement a well defined protection domain interface.

The strategy was similar for ribbon declarations. They are modeled in the AST as a class declaration that also specifies the protection domain parameters for ribbon construction. Unlike protection domains, ribbon declarations are equivalent to class declarations in that they are specifically declaration a new java class and can contain any class declaration body term that a normal class declaration can. The two important differences are that the supertype of all ribbons is implicitly defined and also ribbon declarations must specify their protection domain construction parameters.

The RibbonClassInstanceExpr AST node is used to model both (a) the use of the extended new operator that allows specification of the protection domain that the new object should be allocated within, and (b) the use of the new operator to instantiate a new ribbon, which requires the use of the where clause and specification of the values for the protection domain parameters. For cases where the extended new operator is used to instantiate a normal class (without use of the where clause), the parser initializes the WithinList child node to be empty.

Additionally, new AST nodes are defined to represent the use of the new expressions thisribbon and getdomain.

The JastAddJ compiler uses the Beaver [1] LALR parser generator framework. Parser extensions are given through additional BNF-style rules with attached Java code enclosed in special brackets { and }. The Java code attached to each BNF expression is responsible for creating the AST nodes from the parsed input. BNF rules previously defined can be extended with additional alternate forms by declaring rules with the same rule name (on the left hand side). This allows modular specification of the additional parsing rules for Ribbon.J separately from the parsing rules for Java itself. The parsing logic required to construct the AST for Ribbon.J features was straightforward and directly followed from the grammar and the AST. Figure 4.2 gives an example showing the parsing logic required to parse ribbon declarations.
ClassDecl class_declaration =
    modifiers.m? RIBBON IDENTIFIER.id
    LPAREN ribbon_within_type_list.l? RPAREN class_body.b
{
    return new RibbonDecl(
        new Modifiers(m), id, new Opt(), new List(), b, l);
}
;

List ribbon_within_type_list =
    ribbon_within_type.i
{: return new List().add(i); :}

| ribbon_within_type_list.l COMMA ribbon_within_type.i
{: return l.add(i); :}
;

RibbonParameterDecl ribbon_within_type =
    modifiers.m? type.t IDENTIFIER.id
{: return new RibbonParameterDecl(
        new Modifiers(m), t, id);
};

Figure 4.2.: Example of a parsing rule in the RIBBONJ compiler that extends the JastAddJ parser for Java. This example shows parsing logic needed to parse ribbon declarations. Note that it is extending the class_declaration BNF rule, allowing precise specification of where this additional syntax should be allowed in the extended language.
Explicit typechecking rules were used to ensure that extended RibbonJ features (such as privilege declarations and the use of the `where` keyword) only occur in appropriate places. Most of the typechecking for the additional features of RibbonJ is accomplished by the unmodified Java compiler as it type checks the program after our AST rewriting phases have completed.

AST rewriting is used to implement most of the RibbonJ features in the compiler frontend. The AST for protection domain declarations is rewritten into a standard class declaration as follows:

- The class implicitly inherits from `jikesrvm.runtime.ribbonj.ProtectionDomain`.
- A field is added for each protection domain parameter.
- The protection domain parameters become the formal arguments of an implicitly defined constructor.
- Code is added to the implicitly defined constructor to initialize the fields representing the protection domain parameters to be the values passed as arguments to the constructor. At the end of the constructor, a call is added to the `initDomain` method to allow the implementation an opportunity to actually create the protection domain.

Similarly, all privilege declarations inside a protection domain declaration are transformed into a static initializer block on the class that adds a runtime representation of the declared `grant` or `require` privilege to an internal list of privilege declarations. This list of privilege declarations is used by the runtime implementation inside the `initDomain` call when actually creating the protection domain and setting up actual privileges.

Ribbon declarations also follow a similar strategy and they are rewritten into a standard class declaration as follows:
- The class implicitly inherits from \texttt{jikesrvm.runtime.ribbonj.Ribbon}.

- A field is added for each protection domain parameter.

- The protection domain parameters become the formal arguments of an implicitly defined method named \texttt{initWithinDomains\_TYPE\_NAME} (where \texttt{TYPE\_NAME} is the name of the ribbon type being declared).

- Code is added to the implicitly defined \texttt{initWithinDomains\_TYPE\_NAME} method to initialize the fields representing the protection domain parameters with the values from the formal arguments passed to the method. Additionally, the internal field \texttt{internal\_DefaultProtectionDomain} is assigned to the value of the first protection domain passed to the method. Each protection domain parameter value is also added to a list stored in an internal field named \texttt{livingWithinDomains}, which is used by the runtime implementation. At the end of the method a call to the method \texttt{bootstrapRibbonAfterConstruct} in the superclass is inserted. This allows the runtime implementation to actually create the ribbon once all parameter values and internal fields have been properly populated after object construction.

Finally, the \texttt{this\_ribbon} keyword is rewritten into a call to the static method \texttt{Ribbon\_getCurrent\_Ribbon()}. \texttt{get\_domain} is rewritten into a standard Java field access where the first argument is the target of the field access and the second argument is the field name. This works properly because the protection domain arguments of a ribbon are stored in public fields with the same name as the protection domain argument.

4.1.2 Compiler Backend

Two features of \texttt{RIBBONJ} could not be implemented via a direct AST transformation into a normal Java program: (1) ribbon initialization after construction
(implementation of the \texttt{where} clause), and (2) the extended \texttt{new} operator where the protection domain that the new object should be allocated from is explicitly specified. While these features could not be implemented through AST rewriting, they could be implemented at the bytecode level in the compiler backend using standard Java bytecodes. Thus, compiled \textsc{RibbonJ} programs are standard Java bytecode and do not require any customizations to the Java bytecode standard itself.

The implementation of the \texttt{where} clause for ribbon construction is implemented by inserting bytecodes to implement the logic for the \texttt{where} clause after the new ribbon object is constructed. These implementation bytecodes evaluate each of the arguments in the \texttt{where} clause, followed by bytecode for a method call to the \texttt{initWithinDomains\textit{TYPENAME}} method.

Java bytecode has a stack-based evaluation model where expressions become values on an operand stack. After the construction of the new ribbon object, a reference to the new object is on the top of the operand stack. As each argument in the \texttt{where} clause is evaluated, it results in the final resulting value for the argument’s expression to be placed on the top of the operand stack. Finally, the call to the \texttt{initWithinDomains\textit{TYPENAME}} method consumes all of these argument values from the operand stack. As the \texttt{initWithinDomains\textit{TYPENAME}} method has no return value, the operand stack is returned to the state where it was immediately after the new ribbon object was constructed. This allows the value of the entire \texttt{new} expression for ribbon construction to still evaluate to the new ribbon object itself, while still allowing the information in the \texttt{where} clause to contain complex expressions and then be passed to the internal method that receives these values and finishes constructing the ribbon.

The implementation of the extended \texttt{new} operator requires a different approach. In this case, the runtime implementation will need to know the value of the protection domain that the new object should be allocated within. The \texttt{new} Java bytecode only passes the type of the object that should be allocated and nothing else. The construc-
tors could be modified to receive the protection domain value, but by this point the object is already allocated, and additionally it would require changing the type signatures of all objects, also breaking compatibility. Instead, the compiler inserts bytecode before the `new` operation that evaluates the protection domain value and then passes the result to the static method `ProtectionDomain.setNextAllocDomain`. The runtime implementation of this method stores the value passed to it in a thread-local variable, and the next allocation on that thread then knows to use the specified protection domain. Each allocation resets this internal thread-local field to null, ensuring that if the default `new` operator is used, the runtime implementation will know to allocate from the default protection domain associated with the currently executing ribbon.

4.2 Language VM Implementation: JikesRVM

The ribbonization of JikesRVM involves:

- Enhancements of virtual memory address space abstractions to distinguish between memory managed by ribbons vs not managed by ribbons.
- Support for memory domains within the memory model of JikesRVM.
- Allocation path enhancements to support explicitly allocating within a specific memory domain.
- Enhancements to the GC so it is aware of and performs GC within all memory domains.
- Signal handling enhancements to support lazy mapping of the address space of memory domains as well as a new trap for protection domain violations.
- Support for grouping threads into ribbons and the initiation of a new ribbon.
- Enhancements to the JikesRVM boot loading process to ensure proper sharing of data not directly managed by the ribbons C library.

- Other JikesRVM C runtime enhancements within the bootloader.

First, a brief background of the main concepts in JikesRVM is given.

4.2.1 JikesRVM Overview

JikesRVM [6] is a meta-circular Java virtual machine (JVM): the JVM itself is written primarily in the Java language. It relies on a bootstrap Java VM to save to disk an in-memory representation of the initial set of objects and the machine code necessary for the VM to begin execution. The relevant machine code is generated by the JikesRVM compiler classes as executed by the bootstrap JVM. A small C bootloader program loads the code and data from disk and then transitions execution to JikesRVM. From that point onwards, the JikesRVM class loader, compiler, and supporting classes (whose code is pre-compiled into the JVM boot image) can then load and compile any additional classes not included in the boot image. JikesRVM includes a baseline compiler as well as an adaptive optimized compiler that recompiles frequently executed methods on demand.

Memory management in JikesRVM is performed through a set of precise garbage collection algorithms implemented in the MMTk toolkit [13]. MMTk models low level memory concepts with high-level, extensible classes. The Mmapper class tracks virtual address space allocation and OS mapping state. The Map class actually implements the logic to map and unmap virtual memory regions. The Space base class represents an abstraction that models page allocation, deallocation, and reservation. A Space may represent a contiguous or discontiguous region of virtual memory, and is used to divide the heap based on the different ways memory should be considered by garbage collection algorithms and other VM subsystems. For example, a copying garbage collector might have a “to” space and a “from” space that each represent contiguous
regions of memory, which can be freed all at once after certain phases of garbage collection. Sub-classes of Space can be defined to track additional properties of memory regions as needed by the GC or the VM.

When a Space is constructed, it is given a VMRequest object, which models information necessary when the space needs to allocate or free pages within the underlying OS. Spaces acquire and release memory in 4MB chunks through the Mmapper and its VMRequest instance. Spaces work in coordination with page resource objects (derived from the PageResource base class), which are responsible for resource accounting and distributing allocated pages within allocated chunks.

One Space used by all different configurations of JikesRVM is ImmortalSpace, which contains objects that never need to be freed once allocated (certain kinds of JVM metadata). Another commonly used sub-class of Space is LargeObjectSpace, which contains only objects whose size is greater than a certain threshold, enabling the JVM to optimize memory allocation and GC algorithms for large objects separately. Two different instances of the same Space sub-class may model distinct types of memory, such as compiled machine code vs heap space for Java objects.

The Allocator base class models logic required to fulfill thread-local allocation and free requests of arbitrary sizes and alignments. An Allocator allocates and frees memory in at least page-sized units from the Space that it is associated with. Commonly used allocators include BumpPointer and SegregatedFreeList. JikesRVM also has a framework that allows for the specification of an allocation type “hint” based on the type of the object being allocated or the code context of the allocation (containing class or method). This allows for example the straightforward separation of heap objects necessary for the internal operation of the JVM itself from objects allocated by the application being executed.

The MutatorContext base class models behavior that is local to each mutator thread. A mutator can represent either a single logical processor that multiplexes the non-parallel execution of many logical threads (i.e., “green threads”), or more
typical for modern uses of JikesRVM, represents a single native OS thread of execution that represents a single logical thread of execution within the Java program. This class models any thread-local state, as well as JVM logic that is unsynchronized (e.g., the “fast path” of allocation and barriers). Similarly, the CollectorContext models the state and unsynchronized logic for each GC collector thread running within the JVM. Global state and operations requiring synchronization are modeled by the Plan base class. The specific instance of the Plan class also determines the concrete instances of all other connected classes, including the actual concrete class instances for MutatorContext, CollectorContext, Allocator, and Space, as well as how these objects are structured and relate to each other. JikesRVM allows different build configurations to be specified in a configuration file, including the name of the concrete Plan sub-class that should be used to define the memory management strategy for the JVM being compiled.

Both JikesRVM and MMTk rely heavily on the vmmagic [48] framework, which allows high level languages to efficiently manipulate low level system constructs, such as direct memory access. vmmagic provides classes in the org.vmmagic package with (a) special types that represent underlying low level primitive words of different sizes or types (e.g., 4-byte integer, 2-byte short, pointers, etc.), and (b) intrinsic methods that have empty bodies whose signatures define low level operations, such as storing or loading a word, or performing an atomic compare and swap. vmmagic intrinsic methods are implemented as special cases inside the baseline and optimizing compilers such that vmmagic methods are implemented not as function calls but as inlined machine instructions. Use of vmmagic is critical for good performance within a meta-circular VM written in a high level language such as Java.

4.2.2 Virtual Memory Address Space Changes

Ribbons support in JikesRVM is built on top of a lower-level C library, as detailed in Section 4.3. This library only manages a specific portion of the virtual address
space for a process. Thus, JikesRVM needed to track the sections of the virtual address space being managed by ribbons, and only acquire or release ribbonized sections of the virtual address space through the lower-level library. The Mmapper class was modified to track whether or not each virtual memory address space region is being managed by the ribbons library or not. The Map class and corresponding PageResource classes were changed to use the ribbons library when appropriate, as well as to optionally specify a specific “preferred address” for newly allocated regions of the virtual address space. JikesRVM was also modified so that the entire virtual address space of the process was not entirely mapped in during VM boot, but rather a large section of the address space is reserved for management by the ribbons library, and this section of the virtual address space is left unmapped initially.

4.2.3 Support for Memory Domains

The org.mmtk.vm.Memdom interface models the low-level ribbons library API calls for memory domains. This interface is implemented by the internal Java class org.jikesrvm.runtime.Memdom. Methods available in the Memdom interface include methods for adding, removing, and changing privileges, allocating and freeing pages, and retrieving the opaque low-level API handle associated with the the Memdom object.

The org.jikesrvm.runtime.ribbonj.ProtectionDomain class implements the higher level protection domain semantics assumed by the RIBBONJ compiler. This class models the protection domain argument list, a list of require and grant privileges, the internal Memdom it is associated with, and a list of ribbons that live within this domain. This class is responsible for implementing the higher-level semantics of protection domains in RIBBONJ using the lower-level semantics provided by memory domains in the lower layer, as follows:
• When a new privilege is added to a protection domain, the corresponding privilege is added between the corresponding memory domain and all ribbons living within the protection domain.

• When a new protection domain is initialized (through the compiler-generated call to initDomain) the list of require and grant privileges are enumerated and added (if allowed by the isolation model) to the appropriate protection domains.

• When a new ribbon is instantiated that should live within a specific protection domain ("home ribbon"), if allowed, the underlying low-level ribbon is granted all appropriate privileges to the memory domains associated with the protection domains that the home ribbon has privileges onto.

Previously, JikesRVM assumed that a static number of spaces would be instantiated at JVM startup, with the number and type depending upon the selected Plan (and thus the selected GC algorithm). JikesRVM was modified to allow for an arbitrary number of spaces to be created at runtime, up to a certain statically defined limit per MutatorContext. This allows each thread in each ribbon to associate with different memory domains, as determined by the ribbon it is operating within. In RibbonJ each ribbon is associated with a fixed number of protection domains (as determined by ribbon types), so this approach was sufficient in order to implement RibbonJ and simplified memory domain lookup code in the allocation path.

Each memory domain is modeled as one or more distinct Space objects. How many Space objects are constructed for each memory domain is determined by the GC plan selected (e.g., a generational GC would need at least two Space objects for each memory domain). The current implementation extends the standard mark-sweep GC, which only requires one concrete Space object for each memory domain.

To enable Space objects to be associated with a specific runtime memory domain, a new type of vmRequest was added that allows specification of a specific org.mmtk.vm.Memdom that the Space should use for allocation and free requests.
4.2.4 Allocation Path Changes

The allocation path in JikesRVM was enhanced to support allocation of objects within different memory domains, including the ability to choose which memory domain to allocate within when constructing a new Java object. As described in Section 4.1.2, the extended new operator that allows explicit specification of the memory domain to allocate within is compiled into a call to the `setNextAllocDomain` static method in the `ProtectionDomain` class. This call modifies thread-local data that was added to the `MutatorContext` to track if the next object allocation was for a specific protection domain. This thread-local field is reset to null after each allocation. Additionally, each thread explicitly remembers the “default” protection domain to use for allocation, which allows for simpler code in the allocation fast path where an object is being allocated from the default protection domain. Keeping the code size of the allocation fast path small is critical to ensure good performance because object allocation in Java is so common, and increasing the code size of the fast allocation path causes code size bloat (reducing the amount of a program that fits into a CPU’s L1 code cache), hampers inlining (both directly and indirectly), and increases register pressure.

A new allocation type “hint” was added, `ALLOC_ALLOW_MEMDOM`, that indicates the allocation is being made from within a context where allocating from a specific memory domain is possible (as determined by object type or lexical context). This allocation hint is only used for allocations made from within the context of actual user applications. This ensures that allocation of VM internal objects are actually allocated within VM internal spaces (rather than within a space within a specific application-specific memory domain), and also enables the use of an unmodified fast allocation path for allocation of internal VM objects.

In JikesRVM the `MutatorContext` class models each running application thread wherein there is one thread-local segregated free-list allocator (`MarkSweepLocal`) associated with the application’s main space. This scheme was modified so that each
thread can support up to $N$ allocators (e.g., $N = 16$) – one for each \texttt{Space} associated with the memory domains used by that thread. For speed this is implemented by a fixed-size table mapping memory domains to allocator instances. The table is initially populated with \texttt{MarkSweepLocal} allocators that are unused and not associated with any \texttt{Space}. These pre-allocated allocator instances will later be bound in the allocation slow path to an appropriate \texttt{Space} object before being used.

Upon allocation, in the fast path where the default memory domain is being used for allocation, the allocation code remains unchanged and allocates from the \texttt{MutatorContext}'s main allocator object that is associated with the default memory domain of the ribbon. In the slow path for allocating from a specific memory domain, the memory domain to allocator instance mapping table is linearly scanned to find the allocator associated with memory domain of the allocation request. If it is not found, a check is made to ensure the thread’s ribbon has the \texttt{inject} privilege on the target memory domain. If the privilege check passes, one of the pre-allocated allocator instances is then attached to the space associated with the requested memory domain, and the allocation mapping table for the thread is updated. Certain JVM runtime features also require being able to map a \texttt{Space} back to its corresponding thread-local \texttt{Allocator} for a given \texttt{MutatorContext}, so the \texttt{getAllocatorFromSpace} method was enhanced to understand and search the memory domain to allocator instance mapping table.

This strategy removes the need to check for the appropriate \texttt{inject} privilege on every single object allocation and is key to good performance. Under the semantics of \texttt{RIBBONJ}, once a ribbon has the \texttt{inject} privileges to allocate within a given protection domain, the privilege is never revoked. Thus, the entries in the thread-local mapping tables always remain valid once created. Also note that the pre-allocation of allocator objects avoids a chicken-and-egg type problem that would otherwise arise in the allocation slow path when the slow path needs a new allocator instance in order to satisfy an allocation within a memory domain that the thread has never used before. The allocation path in JikesRVM is marked as “Uninterruptible”, which
enables certain compiler optimizations, and is thus not allowed to allocate any new Java objects using the `new` operator, including VM internal objects.

4.2.5 GC Enhancements

The current implementation extends the non-compacting mark-sweep GC plan. Using a non-moving GC algorithm allows objects to properly remain within the memory domain they were originally allocated within without having to execute individual compacting phases for each memory domain.

The modular design of JikesRVM made extending the GC to support dynamically created memory domains (and thus dynamically created spaces) straightforward. JikesRVM divides the GC process into several phases, and allows the participation of the relevant memory management objects in each phase of the GC, including spaces, allocators, collection contexts, and mutator contexts. The mutator context is responsible for notifying each space of a GC phase, so this was enhanced to notify all dynamically created spaces associated with memory domains. Likewise, logic to trace a specific object instance is sometimes delegated to the space that contains the object, depending on the type of the space. Tracing was thus enhanced to properly discover the space that contains the object even for objects in dynamically created spaces.

The GC itself operates within a ribbon that has privileges to all memory domains within the JVM. Free pages identified by the GC are freed from their corresponding memory domain as appropriate.

4.2.6 Signal Handling Enhancements

Each ribbon within the process maintains its own page table and page directories (threads within the same ribbon share the same page table). When one ribbon performs an allocation that causes a particular memory domain to capture another portion of the virtual address space of the process (which is shared by all ribbons),
either (a) all other ribbons that have privileges on that memory domain must immediately map in that additional section of the virtual address space, or (b) the mapping can be performed on demand ("lazy"), only after an attempted access by a ribbon to that area of the address space. Lazy mapping enables memory allocation to remain more loosely synchronized: the virtual address space itself is a shared resource and must still be protected by a lock when address range reservations are made; however, lazy mapping does avoid logic having to be executed in every ribbon every time a new reservation is made. The virtual address space reservation lock itself is rarely contended, as new reservations are made in at least chunks of 4MB regions.

In order to support lazy mapping, the JikesRVM SIGSEGV signal handler was enhanced to (a) check if a particular address is managed by the lower-level ribbons library, (b) if so, perform a privilege check to determine if the current ribbon has privileges on the memory domain that contains the faulting virtual address, and (c) if the current ribbon does have privileges, to map in that page (and opportunistically map in surrounding pages in the same memory domain) and restart the faulting instruction. Additionally, if the current ribbon does not have privileges on the relevant memory domain, then the signal handler generates a new TRAP_MEMDOM_VIOLATION trap. This trap is handled within the RuntimeEntrypoints class and is converted into an exception being thrown of type Memdom.ViolationException.

As an example of what happens during a privilege violation, the Master/Worker example was modified to reference memory that it should not have privileges for. A modification was made so that the Worker attempted to write to the queue data domain, which the Worker only had read access to. This produced the following RuntimeException:

```java
org.mmtk.vm.Memdom$ViolationException
    at MasterWorker$QueueDomain.take(MasterWorker.java:68)
    at MasterWorker$WorkerRibbonThread.run(MasterWorker.java:138)
```
4.2.7 Support for Ribbons

The org.mmtk.vm.Ribbon interface models the low-level ribbons library API calls for managing actual runtime ribbons. This interface is implemented by the Java class org.jikesrvm.runtime.Ribbon. Methods in the Ribbon interface include methods to join or leave a memory domain, to start a new ribbon given an initial thread, and to get the Ribbon object representing the currently executing ribbon. The implementation also includes a hashmap for quickly mapping a low-level ribbons API opaque handle representing a ribbon into its corresponding Ribbon object (and vice versa), which is required to efficiently implement certain callbacks from the low-level ribbons API (e.g., during ribbon startup).

The org.jikesrvm.runtime.ribbonj.Ribbon Java class implements the RibbonJ semantics for a ribbon using support provided by the lower-level Java class org.mmtk.vm.Ribbon. The higher level Ribbon class tracks which protection domains the ribbon is living within and which low-level memory domains (Memdom instances) the ribbon has joined. The Ribbon class also supports ribbon initialization by implementing the method bootstrapRibbonAfterConstruct, which is called by the backend compiler to implement support for the where clause in RIBBONJ. This method notifies the relevant protection domains that the new ribbon is attempting to join onto it (if it has the proper privileges).

Additionally, thread startup and bootstrap were modified within the JVMThread class and the underlying C JikesRVM library to understand the ribbon concept by adding a new “ribbon initialization” mode for thread creation that uses the ribbon C library instead of pthreads when starting a ribbon. Care must be taken at ribbon startup to avoid issues: a new ribbon begins running with almost none of its virtual address space mapped in (new ribbons often have much fewer privileges than its parent ribbon, so this ensures it will not have access to memory that it should not). A new ribbon therefore immediately joins the JikesRVM global memory domain, re-installs all of the JikesRVM signal handlers (the OS does not automatically clone these for a
new ribbon), and then immediately sets a temporary stack for the SIGSEGV handler that is based on a region on its own stack. This prevents hard crashes that would be caused otherwise if the SIGSEGV handler is triggered while the ribbon is still bootstrapping to lazily map in a region of the virtual address space in the JikesRVM global memory domain and the stack itself for the signal handler is not yet mapped in.

With this in place, a call is made to the ribbons C library to immediately map in all pages belonging to the JikesRVM global memory domain, which is required because the JikesRVM SIGSEGV handler can actually attempt to allocate memory on some code paths, which can trigger additional accesses to the global memory domain, and thereby cause a recursive trap and thus a hard crash. Once all pages from the JikesRVM global memory domain are mapped in, it is safe to reset the stack of the SIGSEGV signal handler to its normal value and then pass control to sysThreadStartup, allowing the first thread in the new ribbon to begin execution.

4.2.8 JikesRVM Boot Loading Changes

JikesRVM uses a relatively small C program to load the JikesRVM boot image (containing the bootstrapped compiled code and object representations of the objects needed for the VM to begin execution) into memory and then jump to the Java boot method entrypoint. This C bootloader was enhanced to initialize the ribbons C library, explicitly specifying the address range that should be managed by the ribbons C library for the use of memory domains. To avoid conflicts with the C memory allocation library, the bootloader requests using the C malloc call a very large contiguous block of memory (as memory is actually allocated only upon first access in Linux, this does not have negative ramifications, but only reserves a section of the virtual address space). Thus, the C malloc library believes that the returned region of memory has been allocated, and will avoid the region for future allocations,
and thereby will not return an address that is being managed by the ribbons C library. The C bootloader also creates the JikesRVM global memory domain.

Once execution control passes into the Java layer of JikesRVM, housekeeping tasks are performed during the subsequent VM boot process, including the allocation and setup of initial Memdom, ProtectionDomain, and Ribbon objects that correspond to the JikesRVM global memory domain and the initial ribbon.

4.2.9 JikesRVM C Runtime Changes

Once the JikesRVM C bootstrap program has initialized ribbons, all JikesRVM C code avoids the direct use of malloc, free, and other C memory management library functions. Instead, these are replaced by alternate functions (e.g., jikesrvm_malloc) that instead redirect allocation and other memory management functions so that they are using the JikesRVM global memory domain instead of the heap managed by the C library. The stacks for new threads are explicitly allocated from the global memory domain to avoid pthread relying on the C heap for allocation. Likewise, the JikesRVM syscalls that allocated additional pages for the JVM use allocation from the global memory domain (unless a memory domain is explicitly specified) instead of direct OS syscalls for mapping and unmapping pages.

When the ribbons C library is initialized, it remaps all code and data segments in memory loaded by the OS process loader so that they are on shared memory pages instead of private memory pages. From that point forward, JikesRVM is careful to always use the global memory domain, to ensure that global state is accessible and is the same across all ribbons. JikesRVM support for dynamically loading shared libraries (through dlopen) was modified so that after the new shared library was opened by the OS, all code and data segments of the newly loaded shared library are remapped onto shared pages. Additionally, support for fork and exec within the Classpath runtime library was patched to return an error, as the userspace implementation does not support forking a process with multiple ribbons.
The BootRecord and Syscall classes were modified to support the additional C-level JikesRVM “syscalls” that then call into the lower-level ribbons C library.

The C syscalls that create pthread mutexes and condition variables were modified to create the pthread constructs with the PTHREAD_PROCESS_SHARED attribute. From the perspective of the kernel, in the userspace implementation of ribbons, differing ribbons appear as different processes (as they do not share a page table). Thus, supporting libraries such as pthreads need to avoid futex optimizations that only work within the kernel when the primitives are used within the same process (as seen by the kernel). Kernel level support for ribbons would remove this limitation.

4.3 Operating System Support on Linux

The higher-level functionality of RibbonJ implemented in JikesRVM depends on a layer-level implementation written as a set of C libraries. While the system level support ultimately belongs in the kernel, the functionality of POSIX and Linux APIs make an initial user-space implementation possible.

4.3.1 Implementation Overview

The implementation strategy relies on POSIX shared memory along with the 
\texttt{mmap}, \texttt{mprotect}, and \texttt{clone}\textsuperscript{1} system calls. Conceptually, the lower-level implementation consists of three concepts:

- A \textit{memory domain}, which corresponds to an instantiated protection domain, is a set of virtual memory pages.

- A \textit{ribbon} is a container of threads.

- A \textit{privilege} is a mapping between a memory domain and ribbon to a set of actions (read, write, inject).

\textsuperscript{1}A generalization of the fork system call used to spawn new threads.
At process startup, a large section of the virtual address space is reserved for management by the lower-level C library.\(^2\) A memory domain is a discontiguous collection of pages within this reserved area. Allocation within each memory domain is performed by a custom multi-pool allocator that is motivated by the Linux kernel slab allocator in its design. The metadata describing memory domains, privileges, and ribbons are kept in a special “control area” shared memory segment that is protected by locks, ensuring that all ribbons within a process have the same view of the set of memory domains, privileges, and ribbons.

A ribbon represents a separate and distinct set of privileges over the virtual address space of the process it belongs to. In the user-space implementation, from the perspective of the OS, each ribbon has a completely separate virtual address space and set of page tables (as if each ribbon was its own process). The ribbons coordinate presenting a single logical virtual address space (partitioned into memory domains) to the upper-level application through synchronized use of the shared control area.

A ribbon is created (along with its first thread) by a system call to Linux’s `clone`. The calling thread is cloned; however, a special set of flags are passed to indicate that the new clone should not share the same virtual address space mappings but will share all other properties (e.g., shared file table) with the creating thread. In this way, a ribbon appears as another thread within the same process except that it can control its own virtual memory mappings and permissions independently.

Privileges dictate specific actions that a ribbon can perform on the pages belonging to each memory domain within the unified virtual address space. A page is termed to be “unmapped” in a ribbon if that particular ribbon cannot currently access it, even if the ribbon’s privileges permits the access. Likewise, a page is termed to be “mapped” if a ribbon can currently access it. A ribbon uses the `mprotect` system call to “map” and “unmap” pages. Throughout program execution, pages are mapped and

\(^2\)The size of this area is configurable and may be larger than the amount of physical RAM in the system. However, because of Linux’s on-demand paging and memory over-commit, each page in this area is only actually allocated (from the perspective of the OS) on the first attempted write to the page.
unmapped from ribbons as needed to enforce the privileges and isolation properties of given ribbon instance. Since modern hardware enforces memory permissions, the overhead of privilege enforcement under normal usage (mappings already established and page tables stable) is non-existent. Overhead is also minimal during the mapping, unmapping, creation, and destruction of ribbons, memory domains, and privileges, as a single syscall can update enforced privileges at the granularity of a page, or more frequently at the granularity of large chunks of pages (e.g., 4MB chunks).

4.3.2 Growing and Shrinking Memory Domains

As a memory domain grows (or shrinks) it allocates (or releases) pages from the reserved area of the virtual address space claimed at startup. Accesses to this area are synchronized such that no two memory domains are allowed to allocate the same section of virtual address space. This enforces the semantics that all ribbons within the same process share the same logical view of the virtual address space of a process, even if each ribbon has a different set of privileges over different sections of the address space.

During execution, a memory domain can grow (by claiming pages from the reserved area) when a ribbon attempts to allocate a block of memory from a domain and there is no room on prior pages reserved for that memory domain. These additional pages will subsequently become a part of the address space dedicated to a particular memory domain, and then will be immediately mapped into the address space for the currently executing ribbon that is performing the allocation.

However, another ribbon may also have access privileges to this memory domain but will be unaware of the change (it would see the updated metadata in the control area if it checked; however, the ribbon may be executing within the application’s code and control will not be within the ribbon library in order for it to check all the time). The pages in this other ribbon would still be unmapped. Rather than rely on signals or some other form of forced interruption or notification, the new pages
within an extended memory domain remain unmapped (except within the ribbon that caused the memory domain to be extended). When a ribbon tries to access an unmapped page the hardware generates a fault causing the OS to issue a \texttt{SIGSEGV} signal. Normally, this signal results in the termination of the process, but a custom signal handler is installed instead. When invoked, special code checks to see if a ribbon has access privileges to this area. If so, this page and opportunistically its surrounding pages are mapped into the current ribbon. If the ribbon does not have privileges on the memory domain that contains the faulting address, an error is raised instead.

This type of “on-demand” mapping technique is prevalent in system implementations and optimizes for the common case where the overhead of extending the size of a memory domain in one ribbon only needs to be paid in other ribbons if and when the other ribbons actually access the newly allocated areas of the memory domain. In this way, the overhead depends on the locality of data access patterns within a single ribbon only, rather than within the entire process.

Likewise, should a ribbon’s actions cause a memory domain to shrink by returning pages to the reserved area, these pages may still be mapped in another ribbon. In the present implementation, the ability to shrink the address space of memory domains is not directly addressed, as a kernel level implementation would be better equipped to handle this efficiently. A user-space implementation could implement support for shrinking the address space of memory domains either through forced signaling (would cause several context switches in all relevant ribbons when a memory domain shrunk) or through a “soft handshake” mechanism wherein ribbons periodically check a shared free queue, and address space regions in the freed queue can only be reclaimed by an expanding memory domain after all relevant ribbons have had an opportunity to process the shared free queue to that point in time (an epoch based system is common for efficiently handling this kind of loose synchronization). Throughout the evaluation of the user-space implementation the lack of support for shrinking memory domains did not cause an issue.
4.3.3 Isolation Properties

A limitation of implementing ribbon support in user-space is that isolation only strongly applies to the “JIT-ed” Java bytecode, which herein is termed as untrusted code. It is assumed that all native code is inherently trusted. Thus, any untrusted code must not be allowed to invoke native (JNI) method calls that are not known to be trusted (e.g., part of the class library implementation). This can be enforced by either the JIT compiler or by the class loader. Further care is taken to protect the shared control area that contains the metadata for ribbons, memory domains, and privileges. The pages where this metadata exist are unmapped from all ribbons during normal execution, and only mapped in during certain critical sections of the library code. This provides some protection against untrusted native code that is benign yet erroneous and attempts to inadvertently modify the shared control area. In order for isolation to be guaranteed for native code in addition to higher-level Java code, there must be support for ribbons natively in the kernel, which is outside the scope of this dissertation.

4.3.4 Bootstrapping the First Ribbon

In the implementation strategy, POSIX shared memory is explicitly used for all memory managed by memory domains. However, multithreaded applications expect that global variables (stored in the data segment of a process) and other global sections of the virtual address space of a process are shared between all threads. The ELF process loader in Linux maps in these areas using private mappings, which allows efficient memory mapped IO on demand (rather than having to read all pages from a program’s file on disk into memory immediately) to speed the initial loading time of a program. However, these private mappings are not compatible with the approach where each ribbon is a clone of its parent ribbon so that it can manage its own virtual address space and page tables (any privately mapped pages get copy on
write semantics in a clone call).³ Read-only text (code) segments do not need to be remapped, only writeable pages that are expected to be shared across the process.

To overcome this, when a ribbonized process is first initializing the ribbon library first analyzes the address space of the process to identify all writable segments so that these pages of memory can be remapped as shared memory segments. The pages that need to be re-mapped are first identified using the special C variables etext (address of end of text segment), edata (address of end of data segment), and end (address of end of uninitialized data segment).

It is not possible to ask the operating system to re-map the private memory mapped IO pages as shared memory pages – doing so requires unmapping the pages first and thus would crash the program. However, it is also not possible to simply copy the appropriate data segments from the old location to a new location in memory – the loader performs address fixups in the code segment specific to the exact virtual addresses where the code segment was loaded, and performing relocation logic again would introduce large amounts of complexity. Instead, the simpler strategy used in the implementation is to first copy all of the relevant data into a specially allocated “trampoline” area of memory, remap the original areas of memory as shared, and then copy data from the trampoline area back into the original areas. However, there are a number of subtleties in the implementation that are required in order for this to work:

- The code itself that is performing the copy and remap operations must not be in the region of the virtual address space that is being unmapped. Otherwise, the program immediately crashes as soon as the original address space is unmapped. This is worked around by (a) buffering the bootstrapping code with enough read-only code segments on either side to ensure that the page exclusively contains read-only code segments and is not a candidate for re-mapping.

³Note that with kernel-level support for ribbons, these pages would not have to be re-mapped at all, because when a new ribbon was created by the OS it would know that the originally loaded data/text segments should be part of the global memory domain of the process.
under most circumstances, and (b) putting the ribbon initialization code into a shared library. Shared library memory segments are remapped in a later phase of the bootstrapping process, and

- Certain C runtime library calls use a relocation table the first time a function is called to lazily resolve references to dynamically loaded functions. Any of the C functions used in the remapping code function (e.g., `mmap`) must be called at least once before the operation begins. This ensures that the program will still be able to use the relocation data in the data segments of a loaded process to resolve the functions required for the remapping operation to succeed.

- In some cases, different process sections are not loaded contiguously next to other but have unmapped pages separating different sections. The information provided by the `etext`, `edata`, and `end` variables do not provide sufficient information to identify these “holes” in the valid address space. Thus, a custom `SIGSEGV` signal handler is installed during the bootstrapping process to ignore access attempts to any of these invalid address regions.

Once the main section of the process has been correctly prepared for ribbonization, any writeable data pages that belong to shared libraries must also be re-mapped onto shared pages. The same challenges exist in that the virtual address space of the data pages must be the same at the end of the remapping operation in order to retain full functionality of shared libraries without requiring them to be modified. It is especially critical for the data segments of the `pthread` shared library to be re-mapped – otherwise, thread information will begin to diverge in different ribbons, breaking the semantics that ribbons act as if they are within the same process.

The problem is addressed by having a second-phase remapping process that executes (a) after the first-phase of bootstrapping, and (b) after any `dlopen` call is made to load additional shared libraries at runtime. The second-phase process parses the information that the Linux kernel exposes in the `procfs` filesystem for the current
process in the /proc/(PID)/maps pseudo-file. This file exposes detailed information about all memory segments of a process created during the loading of shared libraries. The parsed information is then used to perform a copy–remap–copy-back operation as is performed in the first phase for the main program segments.
5 EVALUATION

This chapter analyzes the performance characteristics of the ribbon C library and the ribbonized version of JikesRVM through standard and synthetic benchmarks. Additionally, two widely used applications of notable size, Apache Tomcat and Apache httpd, are ribbonized and changes in performance are analyzed. Qualitative insights are also given into the complexity of the effort required to ribbonize these applications.

5.1 Performance Evaluation

In this section the performance of the ribbonized version of JikesRVM is evaluated by comparing the performance of the version of JikesRVM that we started with (SVN version 15779) to the ribbonized version.

5.1.1 Microbenchmark Design

A synthetic microbenchmark was designed in order to study the performance overhead of the underlying ribbons C library that supports the full ribbonized JikesRVM without encountering the additional overhead caused by the modified allocation path and ribbonized garbage collector.

The original version of JikesRVM was modified for this benchmark so that it was only changed to support new JikesRVM syscalls to the underlying ribbons C library and to bootstrap support for ribbons at startup (it did not have any allocation path or GC changes). For the synthetic benchmark the master/worker example of Chapter 2 was implemented using direct JikesRVM syscalls to the ribbons C library.
Queue throughput was evaluated for both the ribbonized version of JikesRVM and the unmodified version of JikesRVM. In the benchmark the queue items consist of raw byte arrays of varying sizes. The queue is prepopulated with enough worker items to fill 32 MB of memory. The master produces an item by copying the array data into the appropriate worker item in the queue. A worker consumes an item by copying the array data out of the queue item and into a work buffer, and then sums up all of the bytes in the worker item. As the purpose of the benchmark is to measure the overhead for ribbonized memory access, the benchmark was designed to be memory-bound as opposed to being CPU-bound, hence worker items do not perform significant computational work when consuming each item—only enough to ensure all bytes are accessed from memory.

Synchronization over the queue itself is handled through the built in Java object synchronization language feature as implemented by JikesRVM. Waiting for new work within worker threads is implemented through busy waiting rather than through wait and notify calls—it was discovered that busy waiting was approximately twice as fast as wait/notify for this benchmark for both versions of JikesRVM (at least on this particular system for this type of CPU). This is not unexpected as the common case for this benchmark is that each worker will always have work waiting for it to process, so very little CPU cycles are wasted actually doing busy waiting.

A separate ribbon was created for the master thread and each of the worker threads. Two protection domains were used – one for the queue control area, which both the master and workers had read/write access to, and another for the queue data, which the master had read/write access to but the workers only had read access to. To run the benchmark within the unmodified version of JikesRVM, an identical copy of the benchmark was used except that (a) new threads were spawned instead of new ribbons and (b) memory was acquired through the JikesRVM memory map syscall instead of through the ribbons C library.
Figure 5.1.: Throughput in items processed per second for the non-ribbonized and ribbonized (lower-layer only) versions of JikesRVM.

Figure 5.2.: Throughput in MB processed per second for the non-ribbonized and ribbonized (lower-layer only) versions of JikesRVM.
The microbenchmarks were executed on a 2.8 GHz Quad-Core Intel Xeon E5462 with 2 GB of RAM, running Linux kernel Ubuntu 2.6.28-16-generic SMP x86_64. Each data point represents the mean over 5 runs. Confidence intervals are given at 99%. One master ribbon (or thread) and three worker ribbons (or threads) were created. This gave maximum performance on this hardware platform vs 2 worker ribbons or 4 worker ribbons (the same was also true for the number of worker threads for the variant of the benchmark run by the unmodified version of JikesRVM).

5.1.2 Microbenchmark Results

The queue item size was varied from 4 bytes to 1 MB to trade off between higher and lower synchronization overheads. Performance data was collected for the unmodified version of JikesRVM vs the ribbonized (lower-layer only) JikesRVM using lazy mapping and with metadata protection enabled (the recommended and default configuration). Figures 5.1 and 5.2 present the results of the benchmark.

In all cases the mean performance of the each system is within the confidence interval of the other system. The performance of the ribbonized version (lower-layer only) of JikesRVM when lazy mapping was disabled was also statistically identical to these results. This is not unexpected as all of the memory pages are soon accessed at the start of the benchmark and the lazy mapping overhead on first access to a page is amortized to near zero among the millions of memory accesses over the course of the benchmark.

5.1.3 Methods and Benchmarks

Overhead was evaluated using the DaCapo benchmark suite [14] (version 2006-10-MR2\(^1\)). Additionally, lusearch was ribbonized to use 32 ribbons instead of 32

\(^1\)The unmodified version of JikesRVM does not yet run well on DaCapo version 9.12, see http://dacapo.anu.edu.au/regression/sanity/2010-11-19-Fri-02-46/jikesrvm-svn.html
threads (lusearch-rib). The synthetic master/worker microbenchmark of the prior section was also reimplemented using pure Java objects (rather than directly using JikesRVM syscalls to the lower layer ribbons C library and using raw byte buffers). This allows for comparison of the additional overheads added for this microbenchmark from the higher layer of the ribbonization of JikesRVM.

The benchmarks in this section were run on a dedicated system with a 6-core AMD Phenom II 1090T CPU with 8 GB DDR3 RAM, running Gentoo Linux 2.6.36 SMP. JikesRVM was configured with the mark-sweep GC with the optimizing compiler. It was also configured to run with a fixed heap size and to use 6 threads for parallel GC (one per core). To avoid variations introduced by the adaptive optimization system (AOS), AOS was disabled and loaded classes were immediately compiled by the fully optimizing compiler on first access. (Note that with AOS turned on the observed results were also similar.)

Three JVMs were evaluated: (1) unmodified JikesRVM (2) ribbonized JikesRVM with lazy mapping (termed RibbonRVM-lazy), and (3) ribbonized JikesRVM where all pages are immediately mapped in upon allocation (termed RibbonRVM-nolazy). Ten benchmark iterations were performed. Each iteration executed all benchmarks, running all 3 JVMs for the same benchmark back to back. Alternating JVMs and benchmarks between iterations minimizes bias due to systematic disturbance. In each benchmark run, two warmup runs of the benchmark were taken (without the JVM restarting) to allow all methods to be fully compiled and for the system to reach a stable state before the timed run began. All data points represent the mean over 10 iterations and are given with 99% t-test confidence intervals.

5.1.4 Benchmark Results

Figure 5.3 presents the wall-clock running times for the benchmarks. No significant overhead is observed for antlr, hsqldb, and xalan. The overhead of most other benchmarks is between 2% and 6%. Interestingly, the degree of parallelism does not
Figure 5.3.: Wall-clock running times for the benchmarks. Times are normalized to the running time of the original version of JikesRVM.
seem to correlate with observed overhead. The overhead for the parallel benchmarks (mw, hsqldb, lusearch, xalan) varies widely. Additionally, the outlier is jython (single threaded benchmark), with an overhead of 10%.

The overhead on the DaCapo benchmarks (except for lusearch-rib) cannot be due to additional memory mapping, page faults, or context switching caused by ribbonization, because these benchmarks are still running within the context of a single ribbon. For DaCapo performance overheads are caused more by the additional complexities introduced into the fast allocation path. Currently the RIBBONJ compiler uses a thread-local field (in the MutatorContext instances) to store which protection domain a new object should be allocated within (note that this field is only set if a non-default domain is used for allocation). The fast allocation path must retrieve this additional field and also reset it after the allocation. There is also an additional conditional to test if a non-default protection domain needs to be used for allocation. To further substantiate the above theory, we removed the thread-local field accesses from the fast allocation path and the overhead on jython lowered to 6%. However, when we instead removed the extra conditional, performance on jython did not improve.

Comparing results for the mw benchmark in the completely ribbonized version of JikesRVM to the ribbonized version of JikesRVM that only has lower-layer ribbons support produces some interesting insights. Recall that in the microbenchmark results when only the lower-layer was implemented, no statistically significant overhead was observed. In the fully ribbonized version of the microbenchmark for the case of the 512-byte item size, a slowdown of 3% was observed. This overhead is caused by the increased complexity of the allocation path for objects being allocated as part of the loop that retrieves the next work item from the queue. For this case, this cost is relatively significant compared to the work performed processing the item. This additional overhead becomes negligible for the other two item size cases.

For the medium item size (4KB), results were not statistically significantly different. Notably, for the 1MB item size case performance actually consistently improves.
We re-ran this case (another 15 iterations), and the results were consistent (5% improvement). One theory to explain this performance increase is that because ribbons are treated as processes instead of as threads the scheduling (by the kernel) is more favorable in this case. Another possibility is that ribbons slightly changes the virtual address memory layout of the JVM, which could affect caching behaviors (as cachegrind and other tools do not support our special use of the clone system call, it is difficult to better substantiate this theory). This benchmark on a machine with a different configuration (Dual 2.8 GHz Quad Core Intel Xeon) produced an overhead of 2% to 3% instead. This supports the theory that virtual address space layout differences and how these differences interact with the cache hierarchy of the underlying CPU can affect the results of this particular benchmark within a few percentage points in either direction.

The lusearch-rib benchmark runs with 32 ribbons and measures overhead from all factors, since it performs significant object allocation and is highly concurrent. lusearch had the most overhead of any parallel benchmark in the threaded version, and it is also the most concurrent benchmark. Thus, overhead of ribbonization is likely to be the highest of any of the parallel benchmarks. Overhead for lusearch was just over 12%, highlighting potential for future performance optimizations in tandem with kernel-level support for ribbons.

Figures 5.4–5.7 present a breakdown of the running time overheads for the benchmarks separately by the time spent in mutator threads/ribbons vs the time spent during garbage collection. This allows for separate analysis of the overheads caused by the more complex allocation path vs the increased complexity of object tracing (determining the spaces that belong to traced objects) during garbage collection.

Some benchmarks such as mw, antlr, and fop were able to execute within the heap size without any garbage collection taking place during the actual benchmark run (excluding startup work such as work done by the optimizing compiler). The allocation path overhead dominates the overall runtime overhead for the bloat, chart,
Figure 5.4.: Normalized wall-clock running times for the mutator portion of execution for the first half of the benchmarks.

Figure 5.5.: Normalized wall-clock running times for the garbage collection portion of execution for the first half of the benchmarks.
Figure 5.6.: Normalized wall-clock running times for the mutator portion of execution for the second half of the benchmarks.

Figure 5.7.: Normalized wall-clock running times for the garbage collection portion of execution for the second half of the benchmarks.
eclipse, jython, luindex, lusearch, and xalan benchmarks. For these benchmarks, the normalized overhead within garbage collection is further away from the mean overall overhead than mutator overhead, yet does not significantly change the overall mean overhead. This is especially pronounced for lusearch and xalan, where GC overheads were 16% to 22%, but overall overhead was only 6% (lusearch) or 0% (xalan). For hsqldb, there is no runtime overhead in GC—deviations are only seen within the mutator (and the deviations are not statistically significant).

This indicates that even though finding which space a traced object belongs to during GC adds non-trivial overhead, GC runs are relatively infrequent compared to the frequency of execution of the more complex allocation path. Future work should thus first focus on optimizing the allocation path before optimizing object tracing.

5.2 Small-scale Refactoring Experiences

Certain smaller-scale applications were refactored in order to gain experience using RibbonJ in practice before attempting to refactor larger more complex applications. Notable findings from these early refactorings are presented in this section.

The master/worker example was implemented in 291 lines of Java. Ribbonizing this program to isolate each worker required changing 23 lines of code (13 lines for protection domains, 2 lines for ribbons, 3 lines for changed allocation sites, and 5 for changes to threads). The lusearch DaCapo benchmark was also ribbonized, but in this case not to add any isolation properties but rather solely to introduce ribbons into the benchmark to understand performance changes. For this benchmark, only 3 lines of code had to be changed (1 line for the ribbon declaration, 2 to instantiate the ribbon instead of the thread).

jIRCd\(^2\) is an open source Java IRC daemon, consisting of 9,556 lines of code and 121 classes. A two-tier isolation model was applied to the application wherein each

\(^2\)http://j-ircd.sourceforge.net/
connection operates within its own protection domain, and there is a global protection domain for shared state. Ribbonization required 19 lines of code. Most of the client-specific state was allocated in just a few places, so using the appropriate default protection domain for each ribbon allowed us to keep modification of allocation sites to a minimum.

Note that despite the significant difference in size between the master/worker example (291 lines) and jIRCd (9,556), the code changes required to ribbonize the application is about the same (23 lines vs 19 lines). For these relatively smaller applications, it is clear that the complexity of ribbonization is not determined by program size but rather determined by how “cleanly” the heap can be segmented into “classes” of objects (in this case, isolation classes).

To understand whether this principle extends onto more industrial-strength applications, two well known and complex applications were ribbonized: Apache Tomcat and Apache httpd.

5.3 Apache Tomcat Ribbonization

This section presents the results of ribbonizing the Apache Tomcat web server. The required minor program extensions are discussed, and the performance of the ribbonized version compared to the original one is analyzed.

5.3.1 Apache Tomcat Overview

Tomcat is a web server implementing the Java Servlet and JSP standards, and is used in many enterprise applications. Tomcat was first released in 1999 and has grown to become a mature open-source project, comprising over 410,000 lines of Java code.
(a) The class loader hierarchy under the non-ribbonized (left) and ribbonized (right) Tomcat.

(b) Using ribbons to isolate heaps of web applications instead of using private class loaders.

Figure 5.8.: Overview of Tomcat ribbonization.
To use Tomcat, developers package servlets and JSPs as applications into WAR files, which can then be dynamically deployed or undeployed to or from a running Tomcat server. Applications deployed as WAR files typically run in complete isolation from each other. This isolation is enforced in Tomcat through private class loaders (one per application context) and the Java Security Manager.

Using a private class loader for each context provides strong isolation, but also results in memory overhead from redundant class metadata for identical classes shared by distinct web applications. Additionally, private class loaders cause increased startup costs, due to having to decode, verify, and compile several copies of the same method instead of just one copy of each unique method.

While memory savings per instance may be relatively minor (e.g., in the MVM system [33] they found a savings of 5MB-7MB per additional deployed application for the libraries they studied), these savings can add up significantly considering that there may be hundreds (or perhaps thousands) of deployed applications on a single Tomcat server. As individual servers grow increasingly powerful (and can therefore accommodate more users and deployed apps), the memory and CPU overhead from duplicate metadata and class compilation becomes increasingly relevant.

To reduce these inefficiencies Tomcat version 5.5.31 was refactored with RibbonJ to use protection domains to isolate the heaps of deployed web applications and to use ribbons to enforce this isolation while an application processes a request. The prototype, TomcatRJ, uses the ribbonized version of JikesRVM for its runtime, and supports the same features and deployment configurations as the original version of Tomcat.

5.3.2 Architecture and Coding Overview

The left side of Figure 5.8(a) shows the class loader hierarchy for the unmodified version of Tomcat. While there are several class loaders in the hierarchy, most
used only to isolate sensitive parts of the Tomcat runtime from any shared state and
private application state, and once their initial set of classes are loaded they do not
load many additional classes. The exceptions are the application private class loaders,
of which there can be an arbitrary number, and so the focus of the refactoring efforts
were placed on eliminating these class loaders through ribbonization.

The right side of Figure 5.8(a) shows the straightforward division of the Tomcat
heap into protection domains, wherein each application-private class loader is replaced
by an application-private protection domain. Note how only one global protection
domain is used for all of the “fixed” class loaders. Within this global protection
domain the class loaders provide the isolation, and so we do not need to isolate further
using protection domains. Additionally, in the new architecture, classes loaded by
applications are loaded by the Shared class loader, ensuring there is only one copy of
relevant class metadata.

Figure 5.8(b) shows how ribbons are mapped onto the protection domains and also
the domain sharing privileges. All of the threads in the original design of Tomcat
continue to run in the main Tomcat ribbon, termed the *daemon ribbon*. During the
processing of a request mechanisms must be in place to ensure that the request is
processed within an isolated context. To do this, an additional processing thread
pool is maintained for each deployed application, the request is handed off through a
shared queue for processing on the application-private ribbon, and then the original
request thread waits for the result. Hand-off and waiting is accomplished through
normal Java synchronization primitives.

Tomcat follows the *chain-of-responsibility* software design pattern for request pro-
cessing. A *Pipeline* models a sequence of actions required to process a request.
Actions are modeled by *Valve* objects. This design lends itself quite nicely to rib-
onization. A new type of *Valve* was created that offloaded requests to a private
processing ribbon and waited for the response. *Valve* objects are aware of lifecycle,
and so during startup and shutdown the appropriate protection domain, ribbon, and
thread pool could be constructed or cleaned up, as appropriate. Then if the application is configured to use ribbonization for isolation instead of a private class loader, a ribbonizing Valve would be inserted (at runtime) into the processing Pipeline for that application.

Note that in this design the architecture of Tomcat’s original request processing queue did not have to be modified—Tomcat will accept a new connection and perform application dispatching from within the daemon ribbon, and only once it has been dispatched to the proper application will it then be redirected to the appropriate application-specific ribbon. The main challenge in ribbonizing the original thread pool itself is that we do not know which application needs to process a request until the request has been partially decoded. Because a thread cannot “switch” to a different ribbon, the request must be passed off to a different thread in the desired ribbon; we thus require two extra context switches (one to pass off the request, and one to receive the response once processed). The overhead of this technique is measured in detail below.

Finally, certain class metadata must not be shared, such as the values of static fields and the locks of objects, in order to provide the same level of isolation as private class loaders. In this implementation, bytecode is rewritten to mediate accesses to static fields and redirect them to ribbon-local values instead of the corresponding class global field values.

The prototype does not address the redirection of object locking; however, this could likely be addressed at the VM level by redirecting to a ribbon-local lock for the object when an object’s lock is promoted to a heavyweight lock. Additionally, the benchmark that was used to evaluate the performance of the prototype (SPECweb2009) does not make use of locks on isolated objects, so the performance numbers would not change significantly if locks were fully isolated. The benchmark was designed to be representative of the designs of modern web applications. In Tomcat web applications most shared state is contained in either the session object (which
is not managed by the application-private class loader), or in a database (which does not rely on Java object synchronization for concurrency control). Thus, this dissertation does not study the impact of fully isolating locks within this refactoring of Tomcat.

5.3.3 Assessment of Refactoring Efforts

The refactoring required relatively little coding and was exhibited purely as new lines of code (no existing lines of code needed to be changed). In total there were 680 new lines of code, most of which were for the implementation of the ribbonizing \texttt{Valve}. AspectJ was used to write a short yet powerful aspect to redirect static field accesses to a ribbon-local value cache. Only two of the original Tomcat Java files had to be changed – 21 lines of code were inserted to check if an application should be ribbonized, and if so, to insert the ribbonizing \texttt{Valve} into the appropriate \texttt{Pipeline}.

This confirms the results found in Section 5.2 that the complexity required to ribbonize a program does not depend on program size, but rather how “cleanly” the heap can be coarsely partitioned at the source level. Tomcat’s heap already had very clear boundaries of division, which combined with its use of design patterns allowed for a highly modular implementation of ribbonization.

5.3.4 Performance Evaluation

The Banking workload of the SPECweb2009\textsuperscript{3} benchmark was used to evaluate the performance of TomcatRJ under realistic workloads. The Banking workload is modeled after the actual workload and design of a major banking system in Texas. The benchmark is divided into three tiers: backend, application logic, and static content. The backend tier is simulated by an Apache FastCGI module written in C. The application logic is written in JSP and Java, which is hosted on Tomcat.

\textsuperscript{3}http://www.spec.org/web2009/
Figure 5.9: Visualization of the observed distributions of request latency within Tomcat under the SPECweb2009 Banking workload as a boxplot (includes data from 3 runs). The whiskers on the boxplot are plotted at 1.5 IQR, with the diamond marking the mean.
Figure 5.10.: Visualization of the CDF of request latency within Tomcat under the SPECweb2009 Banking workload (includes data from 3 runs).
Finally, static content such as bank check images are pre-generated and stored on the filesystem. Tomcat was also used to serve static content.

The benchmark is driven by one or more multithreaded clients, which make requests to the static web and application server, simulating the behavior of actual users. The benchmark was configured to eliminate any simulated “user think time,” such that once a response to a request was returned the client would immediately make another request; this puts a higher than normal load on the web and application server as all clients are making requests as fast as possible. The benchmark was configured to deploy one client running on its own system with a 2.8 GHz Quad-Core Intel Xeon processor and 8GB of RAM, running OSX 10.5. The client was configured to spawn 40 concurrent request generating threads. During the benchmark the CPU on the client system remained under 50%, so the client was not a bottleneck. The client system was connected to the server system through a single Gigabit switch.

The server system had a 2.8 GHz Quad-Core Intel Xeon processor and 4GB of RAM, running x86 Linux kernel 2.6.32-22-generic #36-Ubuntu SMP. Both Tomcat and the backend simulator were run on the same system. Throughout the benchmark we observed the backend simulator using less than 5% of the CPU, so it was not a bottleneck.

Three iterations of the banking workload were executed, each interleaving between two configurations: first, the unmodified version of JikesRVM (fully optimized build) running the original Tomcat, and second, the ribbonized version of JikesRVM (fully optimized build) running TomcatRJ. The SPECweb2009 benchmark strives to maintain a constant level of throughput and then measures request latency to determine if requests were processed within a “good” amount of time, “tolerable” amount of time, or “unacceptable” amount of time. In all runs all requests were processed in a “good” amount of time. The benchmark slowly ramps up throughput to the target throughput, then warms up the server by maintaining the target throughput, and finally performs the actual benchmark run. Every 10 seconds data is recorded on
Table 5.1: Summary of data from the evaluation of Tomcat

<table>
<thead>
<tr>
<th></th>
<th>Tomcat</th>
<th>TomcatRJ</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines of Code</td>
<td>413,972</td>
<td>414,477</td>
<td>+680 lines</td>
</tr>
<tr>
<td>New+Modified LOC</td>
<td>N/A</td>
<td>680</td>
<td>0.164% of total</td>
</tr>
<tr>
<td>Total # of Requests</td>
<td>5,537</td>
<td>5,562</td>
<td>+25</td>
</tr>
<tr>
<td>Median Latency</td>
<td>2.97ms</td>
<td>3.60ms</td>
<td>+0.63ms</td>
</tr>
<tr>
<td>Mean Latency</td>
<td>4.42ms</td>
<td>6.11ms</td>
<td>+1.69ms</td>
</tr>
<tr>
<td>Latency 99th Percentile</td>
<td>23.5ms</td>
<td>31.5ms</td>
<td>+8ms</td>
</tr>
<tr>
<td>Latency 99.9th Percentile</td>
<td>51.4ms</td>
<td>259.2ms</td>
<td>+207.8ms</td>
</tr>
</tbody>
</table>

the aggregate throughput and average request latency. Additionally, statistics were collected on the individual processing times of requests within Tomcat. In TomcatRJ time required to actually process the request itself was recorded as well as the processing time including the time to offload the request to another ribbon and wait for the response to be returned to the calling ribbon. In this way the additional latency caused by ribbonization could accurately be measured.

Table 5.1 summarizes the key results of the evaluation. Figure 5.9 offers insight into the differences in request processing latency. From the boxplot, we see that TomcatRJ median latency is less than 1ms more than for Tomcat (0.63ms, see Table 5.1). In both cases, the mean is above the upper quartile (75th percentile), indicating there are very large outliers on the upper end. The upper quartile stretches farther for TomcatRJ, indicating the upper half of the distribution is more evenly distributed, and thus interruptions in processing time are more likely. Undoubtedly, this extra source of variation is caused by the extra context switches. In most cases, the extra context switches happen relatively quickly, but in the TomcatRJ case small delays happen more frequently, and the duration of rare delays is larger.

From the CDF in Figure 5.10 we can see that for approximately 80% to 85% of all requests the difference in latency is less than 1ms. However, past this point, the outliers become more dramatic in TomcatRJ. The 99th and 99.9th percentile for latency in Table 5.1 provide insights into the differences in these extreme outliers. At the 99th percentile, the difference is still not so dramatic (8ms longer for TomcatRJ),
Table 5.2: Summary of additional latency in request processing due to ribbonization in TomcatRJ

<table>
<thead>
<tr>
<th>Latency Percentile</th>
<th>TomcatRJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Median Add. Latency</td>
<td>0.559ms</td>
</tr>
<tr>
<td>Mean Add. Latency</td>
<td>0.988ms</td>
</tr>
<tr>
<td>Add. Latency 90th Percentile</td>
<td>1.005ms</td>
</tr>
<tr>
<td>Add. Latency 99th Percentile</td>
<td>3.658ms</td>
</tr>
<tr>
<td>Add. Latency 99.9th Percentile</td>
<td>47.070ms</td>
</tr>
<tr>
<td>Add. Latency 99.99th Percentile</td>
<td>443.7ms</td>
</tr>
</tbody>
</table>

but at the 99.9th percentile, the outlier for TomcatRJ is more than 4 times the outlier for Tomcat. When the system is under heavy load (having 40 clients making requests as fast as possible to the server application), it appears that in very rare cases one or both of the threads processing a request is starved while waiting to be scheduled to receive the processed result.

Table 5.2 and Figure 5.11 present the observed additional latency caused by ribbonization. The additional latency is calculated by taking the request processing time as observed in the main daemon pool for a thread and subtracting the time it took to actually process the request on the isolated thread. This can be obtained by having the final Valve in the Pipeline (which executes on the isolated ribbon) record the time it takes to process the request just on that thread. The difference thus includes both the extra time required to pass off a request and the extra time waiting to be scheduled to receive and process the response.

The interesting observation here is that at the 99th and 99.9th percentile, the observed additional time required for ribbonization is significantly smaller than the observed differences in total request processing time. It appears that the increased OS scheduling queue lengths and increased use of locking when passing objects between ribbons is causing context switches in general to take longer on average, sometimes significantly longer due to starvation, and thus increasing the latency of work that happens during a request (e.g., network I/O) in addition to the dispatching of a request.
Figure 5.11.: Visualization of the distribution for the additional time required to process requests in Tomcat due to ribbonization during SPECweb2009 Banking. The vertical line at 1ms on the CDF marks the 90th percentile.
Figure 5.12.: Throughput for each 10-second interval during SPECweb2009 Banking, averaged over 3 runs. Excludes warmup and rampdown.
Measured throughput over the life of the workload is shown in Figure 5.12. It is important to note that despite the increase in latency in TomcatRJ, there is no statistically significant difference in total throughput. This indicates that the additional context switch is not causing significant overhead in terms of CPU cycles. This is not surprising considering that a modern CPU on a modern OS can perform millions of context switches per second. This result also indicates that the additional logic in the allocation fast path in the ribbonized version of JikesRVM is not causing significant overhead for this workload of this benchmark.

Surprisingly, even the same general shape of the throughput graph is similar over time. We postulate that each benchmark client thread is seeded deterministically, and that because the throughput of the benchmark is carefully regulated, similar patterns of throughput over time can be observed even when averaging data points across several distinct runs.

In conclusion, it was observed that for almost all requests, the cost of ribbonization was less than 1ms and that total throughput is maintained at previous levels. However, the additional context switch introduces variation into the total processing time of a request, such that 1 out of every 1000 requests is expected to experience an additional delay between 200ms and 550ms and 1 out of every 100 requests is expected to experience an additional delay of 8ms.

5.4 Apache httpd Ribbonization

This section presents the results of ribbonizing the Apache httpd web server. The effort required to ribbonize the application is discussed, and performance differences of the ribbonized version are analyzed.
5.4.1 Apache httpd Overview

Apache httpd\(^4\) is an open source web server that is widely used across the Internet. The Netcraft May 2012 web server survey found that Apache httpd is the web server for approximately 65% of the 662 million sites surveyed.\(^5\) The popularity of Apache httpd stems from its flexibility, modularity, simplicity of configuration, and performance.

Apache httpd version 2.0 and onward follow a fully modular design where nearly any functionality can be modified or extended or new features added. Modules are compiled into Apache statically, or they are dynamically loaded at server startup as described within configuration files. Modules register hooks to extend or modify well-defined points of “advisement” within the Apache core program. As with Apache Tomcat, Apache httpd also follows the *chain-of-responsibility* design pattern, and allows modules to add input or output “filters” to the IO filter chain, for example to compress or encrypt data.

Apache httpd can serve both static and dynamic content. Dynamic content can be served by any script supporting the CGI interface, or through Apache modules dedicated to generating dynamic content through a specific scripting language (e.g., PHP or python).

Apache httpd is designed to be highly portable to nearly any operating system. It thus abstracts away any OS-specific functionality required to process requests concurrently into “Multi-Processing Modules” (MPMs). This abstraction layer also enables the use of different multiprocessing techniques best suited for the task at hand, depending on the characteristics a given application and hardware platform. MPMs implement an interface to (a) create, manage, resize, and destroy a “task pool” of workers, (b) queue management logic to accept a request and delegate to a ready worker in the task pool for processing, and (c) worker logic to process a request and

\(^4\)http://httpd.apache.org/
\(^5\)See http://news.netcraft.com/archives/2012/05/02/
cleanup before re-entering the task pool. The two most commonly used MPMs are **prefork** and **worker**.

The **prefork** MPM uses the **fork** call to create child processes that enter the task pool and are each ready to process a request. Each worker is either processing exactly one active request or is idle waiting for a new request to be received. Performance is enhanced in that the **fork** of a process is performed ahead of time (the master attempts to keep a non-zero number of idle workers available at all times) and thus the overhead of the **fork** call does not affect the latency of processing an incoming request. Shared memory is used to coordinate resources with the master process (such as the “scoreboard” information), and a shared mutex is used to synchronize which idle child accepts a new request.

The **worker** MPM builds on the functionality in the **prefork** MPM, but uses multiple worker threads per child process. The main thread of the child process will listen for and accept new requests just as with the **prefork** MPM, but rather than processing the request directly, it delegates a newly received request onto a child-private job queue. Within each child process a fixed number of worker threads wait for new requests within the child-private job queue. This MPM thus has the overhead of an additional queue (and thus point of synchronization) to be traversed in order to process a request; the motivation of the MPM is that on some platforms or workloads the benefits of multithreading (e.g., potentially faster context switching) will outweigh this extra cost. The MPM is configurable and allows constraints over both the number of child processes as well as the number of threads used within each child.

Apache **httpd** uses a “pool” system for tracking system resources, including memory, sockets, and other OS resources. A pool consists of one or more regions of memory that are dedicated to a particular purpose within Apache. Almost all memory allocation and deallocation within Apache **httpd** are performed using pool allocation and deallocation functions rather than those provided by the standard C library.
Additionally, OS resources can be attached to pools so that these OS resources are
guaranteed to be freed when the pool is freed. Pools are hierarchical and can contain
zero or more child pools. When a pool is freed, any child pools (and recursively and
grandchildren pools and so forth) are also freed.

Pools provide resiliency against memory and resource leaks. Although pools do
support explicit deallocation requests for small sections of memory allocated from
within a pool, more typically (especially for the relatively short-lived pools that ded-
icated to request and connection processing) a carefully tested code path frees the
entire pool at the end of its useful lifetime, thus freeing up any associated memory and
OS resources all at once. For example, a pool that is dedicated to resources required
to process a given connection will be freed when all requests on that connection have
been fully processed. This more coarse granularity of resource management results
in fewer code paths potentially leaking data and thus reduces the amount of testing
required to confirm proper functionality.

Gröne et al. [52] provide further information on the software architecture of Apache
httpd, including information on the modules interface and API, the available multi-
tasking server architectures, the request–response loop, and pool-based resource and
memory management.

5.4.2 Ribbonization Overview

Apache httpd is written in the C language. Thus, the ribbonization of Apache
httpd relies solely upon the lower-level ribbons C library. Apache httpd was already
designed so different logical segments of code allocate from different types of pools,
which have a hierarchical relationship to each other. This fits well with the protection
domain and ribbons model, allowing different pools to be logically mapped to differ-
ent memory domains, and different modules of code to run within different ribbons
(having different levels of privileges across different pools/memory domains) without
significant modification. The ribbonization process involved five main changes:
First, a shared library was implemented that replaces the standard `malloc` and `free` (and related) functions. The new implementations allocate (and free) memory from an explicitly specified global memory domain rather than the C runtime library’s managed heap. When a new program starts, this shared library can be loaded first prior to any other shared library (including the C runtime library) via the use of the `LD_PRELOAD` environment variable. In this way, this shared library can override a program’s use of `malloc` and `free` without having to modify the program itself.

Second, logic was added to the `main` function to initialize the ribbons C library. Initialization is straightforward as all of the default initialization parameters of the ribbons C library are used. A global memory domain is then explicitly created and the customized `malloc` and `free` implementations are instructed to use this global memory domain for all future allocations. Thus, this initialization logic takes place at the very beginning of the `main` function before any other code will attempt to call `malloc` or `free`.

Third, the Apache httpd pool allocation and resource tracking system was enhanced so that each pool could optionally be associated with a specific memory domain. Modifications were limited to the construction of pools (where a memory domain is optionally specified) and where pools allocate and free memory from the underlying operating system. Pool allocation was changed to instead rely on the ribbons C library to allocate or free memory from the memory domain associated with a pool. If a memory domain was not specified during pool construction, the pool uses the memory domain of its parent pool (or the global memory domain if there is no parent pool). This design allows specification of isolation boundaries (placing different pools into different memory domains) solely by having to modify pool construction and avoiding modifications to other sections of code that rely on pools.

Fourth, the underlying runtime library that abstracts OS operations that Apache httpd relies on (the “APR” library) was enhanced to support the creation of ribbons in addition to just threads. The worker MPM was then modified so that each worker
Table 5.3: Summary of code changes required to ribbonize httpd (worker MPM).

<table>
<thead>
<tr>
<th></th>
<th>httpd-worker</th>
<th>httpd-ribbons</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines of Code</td>
<td>360,875</td>
<td>361,145</td>
<td>+270 lines</td>
</tr>
<tr>
<td>New+Modified LOC</td>
<td>N/A</td>
<td>283</td>
<td>0.078% of total</td>
</tr>
<tr>
<td>New+Modified Files</td>
<td>N/A</td>
<td>9</td>
<td>1.020% of total</td>
</tr>
</tbody>
</table>

processing thread is created within the context of its own (isolated) ribbon. The memory pool associated with each worker processing thread was thus also associated with the private memory domain associated with the worker thread’s private ribbon. This ensures that each worker thread is isolated from the actions of other worker threads through the ribbons mechanism.

Finally, all synchronization primitives (semaphores, condition variables) were modified to use the appropriate kernel flags so that the Linux kernel implements these primitives correctly when synchronization is being performed across different processes. This is required because the user-space implementation of the Ribbons C library implements each ribbon as a separate kernel-level process (while still sharing the same file resource table), which is the only way to allow each ribbon to independently control its virtual address space mapping (without explicit kernel-level support for ribbons). The Linux kernel uses a slightly slower type of futex when implementing inter-process synchronization primitives vs. intra-process synchronization primitives, and this contributes to the runtime overhead observed in the ribbonized version of Apache httpd.

5.4.3 Assessment of Refactoring Efforts

The refactoring started with Apache httpd 2.4.2, which has 360,875 lines of code and 882 source code files. The changes required by ribbonization are summarized in Table 5.3. The changes were targeted and did not cause “ripple effects” into unrelated components—only 9 out of 882 files had to be changed, and within these changed files,
only 10 existing lines of code had to be changed and 3 lines moved. The ribbonization logic was primarily implemented by 270 lines of additional code added within these 9 changed files. The required changes were quite small because the Apache httpd software architecture already coarsely partitioned the heap with its pool-based allocation design. This confirms the findings of Section 5.2 and Section 5.3.3 that the complexity required to ribbonize a program depends on how “cleanly” the heap can be coarsely partitioned at the source level rather than the program’s overall size or complexity.

5.4.4 Performance Evaluation

ApacheBench\(^6\) is a benchmarking program included with Apache httpd that measures the request processing latency and throughput of a web server. It initiates a fixed number of requests (with a configurable level of concurrency) to a given URL at a specified web server, and then measures the time required to connect, start, and fully process a request. Apachebench reports timing information only to the nearest whole millisecond. It also measures throughput in the average number of requests processed per second.

ApacheBench is initiated from a client system targeting a server system under test. In these benchmarks, the client was a system with a 2.8 GHz Quad-Core Intel Xeon processor and 2GB of RAM, running Linux kernel Ubuntu 2.6.28-16-generic SMP x86 64. The server system had a 2.8 GHz Quad-Core Intel Xeon processor and 4GB of RAM, running x86 Linux kernel 2.6.32-22-generic #36-Ubuntu SMP. The client and servers systems were connected by a 100 Mbit/sec network (network throughput was limited to 100 Mbit/sec so that network-bound workloads could be simulated more effectively, and this is a closer representation of a WAN connection than gigabit ethernet).

\(^6\)http://httpd.apache.org/docs/2.2/programs/ab.html
The level of concurrency was varied over the range (1, 2, 4, 6, 8) so as to exercise the cases when (a) there were fewer concurrent requests active than the number of CPU cores on the client and server (no CPU contention), (b) there was the same number of concurrent requests active as the number of CPU cores (all CPU cores should be close to busy all the time, unless load is network-bound instead of being CPU bound), and (c) there were more concurrent requests active than the number of CPU cores (CPU contention for CPU bound workloads).

The latency and throughput were measured for three different configurations: (a) unmodified httpd using the `prefork` MPM, (b) unmodified httpd using the `worker` MPM (configured with only one process and the desired number of worker threads), (c) ribbonized httpd (modified `worker` MPM configured with only one process and the desired number of worker ribbons). This allows comparison of the case where there is only one point of inter-process synchronization (`prefork` MPM) to a case where there is one point of inter-process synchronization and one point of intra-process synchronization (`worker` MPM) to a case where there are two points of inter-process synchronization (`worker+ribbons` MPM).

Additionally, to understand performance under CPU-bound and network-bound workloads the benchmark was run for three different sizes of pages: small (45 bytes, representing a nearly empty html page), average-sized (11.19 KB), and large (1.0 MB, representing a streaming file download workload). The average page size was computed using data from Google\(^7\) that comprises a study of page sizes for the top 380 million websites. The average size was derived by taking the average total document size for a top website (477.26 KB) and dividing by the average number of GET requests required to fully load a top website (42.63). The small and average page size workloads are expected to be CPU-bound and the large page size workload is expected to be network-bound.

\(^7\)https://developers.google.com/speed/articles/web-metrics
Figure 5.13.: Throughput for httpd under the apachebench workload for small [45 byte] web pages (includes data from 20 runs).

Table 5.4: Summary of latency differences for ribbonized httpd under the apachebench workload for small [45 byte] web pages.

<table>
<thead>
<tr>
<th></th>
<th>httpd-worker</th>
<th>httpd-ribbons</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Median Latency</td>
<td>7ms</td>
<td>7ms</td>
<td>0ms</td>
</tr>
<tr>
<td>Mean Latency</td>
<td>7.74ms</td>
<td>8.96ms</td>
<td>+1.22ms</td>
</tr>
<tr>
<td>Latency 90th Percentile</td>
<td>12ms</td>
<td>16ms</td>
<td>+4ms</td>
</tr>
<tr>
<td>Latency 99th Percentile</td>
<td>20ms</td>
<td>25ms</td>
<td>+5ms</td>
</tr>
<tr>
<td>Latency 99.9th Percentile</td>
<td>74ms</td>
<td>64ms</td>
<td>-10ms</td>
</tr>
</tbody>
</table>

To ensure statistically meaningful data, 20 runs of data were collected for each unique configuration (type of httpd, level of concurrency, and page size). Confidence intervals are given at 99%.

5.4.5 Benchmark Results for Short Pages

Figures 5.13–5.15 and Table 5.4 present the benchmark results for short (45 bytes) web pages.
Figure 5.14.: Visualization of the observed distributions of request latency within httpd under the apachebench workload for small [45 byte] web pages as a boxplot (includes data from 20 runs each with 5 different levels of concurrency). The whiskers on the boxplot are plotted at 1.5 IQR, with the diamond marking the mean.
Figure 5.15.: Visualization of the CDF of request latency within httpd under the apachebench workload for small [45 byte] web pages (includes data from 20 runs each with 5 different levels of concurrency).
Throughput results are presented in Figure 5.13. There is no statistically significant difference in throughput until the point is reached where there are the same number of active concurrent requests as the number of CPU cores (4). After this point the performance of both the worker MPM and the ribbonized worker MPM degrade, with the ribbonized worker MPM showing more significant performance degradation.

One key difference between the prefork MPM and the worker MPM relevant to this performance behavior is that the prefork MPM does not require any synchronization primitives (e.g., mutexes) to coordinate the accepting of new connections between the different processes. Instead, the prefork MPM relies on the operating system’s implementation of the `select` system call to only wake one of the worker processes when a new connection is ready to be accepted. In this way, only a single context switch is required in order to accept a new connection and begin processing.

In contrast, both the unmodified worker MPM and the ribbonized worker MPM rely on a shared queue and synchronization primitives to coordinate the accepting of a new connection and then the distribution of this newly accepted connection to a worker thread (or ribbon) for processing. Thus, the worker MPM requires two context switches (one for the listener thread to accept the new connection and to put the connection into the shared worker queue, and another context switch for one of the worker threads or ribbons to take the new connection out of the shared worker queue for processing) instead of just the one that the prefork MPM requires. Additionally, the ribbonized worker MPM requires the use of `inter`-process synchronization primitives rather than `intra`-process synchronization primitives, which require the use of a slower type of `futex` in the Linux kernel (the user-space implementation of ribbons maps each ribbon onto a separate process–this would not be required by a kernel-level implementation).

The cost of this extra context switch is hidden when there are more CPU cores than the number of active concurrent requests, and thus throughput is identical for these cases. As the level of concurrency matches and then exceeds the number of CPU
Table 5.5: Summary of latency differences for ribbonized httpd under the apachebench workload for average-sized [11.2KB] web pages.

<table>
<thead>
<tr>
<th></th>
<th>httpd-worker</th>
<th>httpd-ribbons</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Median Latency</td>
<td>12ms</td>
<td>15ms</td>
<td>+3ms</td>
</tr>
<tr>
<td>Mean Latency</td>
<td>14.13ms</td>
<td>16.05ms</td>
<td>+1.92ms</td>
</tr>
<tr>
<td>Latency 90th Percentile</td>
<td>22ms</td>
<td>26ms</td>
<td>+4ms</td>
</tr>
<tr>
<td>Latency 99th Percentile</td>
<td>33ms</td>
<td>36ms</td>
<td>+3ms</td>
</tr>
<tr>
<td>Latency 99.9th Percentile</td>
<td>139ms</td>
<td>72ms</td>
<td>-67ms</td>
</tr>
</tbody>
</table>

cores, the cost of this additional context switch becomes more and more apparent. Additionally, in the case of ribbons the extra cost of the inter-process futex causes additional degradation. The additional overheads in accepting a new connection are the most significant in the case of small web pages because the time required to actually process the request is much smaller than with the other two cases.

Table 5.4 presents the mean latency and latency at different percentiles for the different versions of httpd. Figure 5.14 presents a boxplot of the distribution of latency for each version of httpd, and Figure 5.15 plots the corresponding CDF of the distributions. Notably, median latency is identical (within the granularity of 1ms) and the latency distributions are nearly identical until the 50th percentile. After this, 1-2ms of additional delay can be seen between prefork and worker. The ribbonized worker adds an additional 1-5ms of latency beyond that of worker. However, for extreme outliers (the 99.9th percentile) the ribbonized worker is 10ms less than that of the unmodified worker. It is postulated that the inter-process synchronization futex has slightly better fair scheduling properties than the intra-process synchronization futex, thus causing extreme outliers to be less severe for the ribbonized worker case.
Figure 5.16.: Throughput for httpd under the apachebench workload for average-sized [11.2KB] web pages (includes data from 20 runs).

Figure 5.17.: Visualization of the observed distributions of request latency within httpd under the apachebench workload for average-sized [11.2KB] web pages as a boxplot (includes data from 20 runs each with 5 different levels of concurrency). The whiskers on the boxplot are plotted at 1.5 IQR, with the diamond marking the mean.
Figure 5.18.: Visualization of the CDF of request latency within httpd under the apachebench workload for average-sized [11.2KB] web pages (includes data from 20 runs each with 5 different levels of concurrency).
5.4.6 Benchmark Results for Average-sized Pages

Figures 5.16–5.18 and Table 5.5 present the benchmark results for average-sized (11.2KB) web pages. The data generally follows the same pattern as in the case for small web pages. Throughput is statistically equivalent for the cases where there are more CPU cores than the number of active concurrent requests. Beyond this point, throughput for worker is slightly worse than for prefork, and even more so for ribbonized worker. However, in this case overall throughput for ribbonized worker actually improves when the number of active requests exceeds the number of CPU cores (the opposite was seen for small web pages). Additionally, the overall decrease in throughput at the highest concurrency level for ribbonized worker vs unmodified worker is less for this case (14%) than for small web pages (12.1%).

The general shape of the latency distributions are also quite similar. The latency distribution for ribbonized worker diverged at an earlier point in this case (37th percentile vs the 50th percentile) but retains the same relative shape. The data in this case also shows lower latency at the 99.9th percentile for the ribbonized worker vs. the unmodified worker.

5.4.7 Benchmark Results for Large Pages

Figures 5.19–5.21 and Table 5.6 present the benchmark results for large (1MB) web pages. In this case throughput is constrained by the bandwidth of the 100 Mbit/sec interface connecting the client and server machines. This can be seen in the throughput graph by observing that the speedup at concurrency levels of 6 and 8 requests does not increase compared to the concurrency level of 4 requests (in contrast to continued increases past 4 concurrent requests for other benchmarks). Additionally, the theoretical maximum transmission speed of a 100 Mbit/sec interface using TCP/IPv4 is approximately 11.7 MB/sec (after factoring in protocol overhead

---

Data from all runs for all concurrency levels is combined into one data set for each version of httpd when computing the mean latency, latency at different percentiles, and the CDFs of latency.
Figure 5.19.: Throughput for httpd under the apachebench workload for large [1MB] web pages (includes data from 20 runs).

Figure 5.20.: Visualization of the observed distributions of request latency within httpd under the apachebench workload for large [1MB] web pages as a boxplot (includes data from 20 runs each with 5 different levels of concurrency). The whiskers on the boxplot are plotted at 1.5 IQR, with the diamond marking the mean.
Figure 5.21.: Visualization of the CDF of request latency within httpd under the apachebench workload for large [1MB] web pages (includes data from 20 runs each with 5 different levels of concurrency).

Table 5.6: Summary of latency differences for ribbonized httpd under the apachebench workload for large [1MB] web pages.

<table>
<thead>
<tr>
<th></th>
<th>httpd-worker</th>
<th>httpd-ribbons</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Median Latency</td>
<td>365ms</td>
<td>368ms</td>
<td>+3ms</td>
</tr>
<tr>
<td>Mean Latency</td>
<td>373.71ms</td>
<td>378.42ms</td>
<td>+4.71ms</td>
</tr>
<tr>
<td>Latency 90th Percentile</td>
<td>729ms</td>
<td>741ms</td>
<td>+12ms</td>
</tr>
<tr>
<td>Latency 99th Percentile</td>
<td>767ms</td>
<td>766ms</td>
<td>-1ms</td>
</tr>
<tr>
<td>Latency 99.9th Percentile</td>
<td>788ms</td>
<td>772ms</td>
<td>-16ms</td>
</tr>
</tbody>
</table>
for the Ethernet, IP, and TCP layers). Other factors (e.g., line noise, physical interface design, and driver efficiency) can further reduce maximum throughput. In this case, throughput peaks at 10.66 MB/sec.

The latency of requests in this case is dominated by the time required to transmit 1MB of data, such that the additional latency required by ribbonized worker (a few ms) is insignificant compared to the time required to actually transmit the data (at least 150ms, often much longer for higher concurrency cases). Overall throughput is thus statistically equivalent for all levels of concurrency and all MPM versions.

The shape of the distributions of latency for the different MPM versions is dramatically different in this case as well, again because request processing latency is dominated by the time required to transmit the response data. In the latency CDF graph in Figure 5.21 the most frequent latency required to transmit response data at that level of concurrency is clearly visible in the five nearly vertical “jumps” in the CDF. This demonstrates that the TCP connection for each response is utilizing approximately the same amount of bandwidth (there is fair sharing of bandwidth), such that at a given concurrency level (and thus a given amount of bandwidth for each active request/response) the latency is approximately the same. The amount of variation for each request for a given level of concurrency increases as the number of concurrent request increases.

The shape of the CDF is mostly the same for the prefork MPM and the worker MPM, although the prefork MPM actually lags behind in some but not all level of concurrency. Again we see the pattern where the shape of the CDF for the ribbonized worker consistently lags behind the shape of the CDF for worker, due to the increase connection acceptance latency. The increase in median latency is the same as with the case for average-sized web pages–3ms. The pattern also continues where the extreme outlier at the 99.9th percentile has less latency for ribbonized worker than for worker. This behavior is also observed to a lesser degree at the 99th percentile as well. Given that requests are dominated by the transmit time in this case, it could
be that this behavior at the 99th percentile is similar because request latency is more
tightly uniform and the 99th percentile is “similarly extreme” as the 99.9th percentile.
Latency at the 90th percentile (which is near the middle of the observed latency when
the number of concurrent requests is 8) is still increased for the ribbonized worker as
it was for the cases for the small and average-sized web pages.

5.4.8 ApacheBench Results Summary

In all cases, request processing latency between the lower and upper quartiles of the
distribution of latency increased between 1ms–5ms for the ribbonized worker MPM vs
the unmodified worker MPM. In cases where the connection acceptance latency was a
significant portion of the overall request processing latency, this additional overhead
could cause overall throughput to decrease when the benchmark was CPU bound
(i.e., when the number of concurrent requests matched or exceeded the number of
CPU cores and when the data transmit time did not dominate the overall request
processing time). In cases of severe CPU contention (e.g., small web page size and
6 or 8 concurrent requests) overall throughput could decrease significantly (18.9%–
21.7%). In cases with less severe CPU contention the reduction in throughput was
less significant (5.8%–16.5%). In cases without CPU contention in all cases there was
no observed statistically significant difference in overall throughput between worker
and ribbonized worker.
6 CONCLUSIONS AND FUTURE WORK

This dissertation presented Ribbons, a new programming model that provides full or partial heap isolation. RIBBONJ was defined and is a realization of this model in Java. The formal properties of RIBBONJ were explored in RIBBONJ-lite. This dissertation also presented the design and implementation of the RibbonJ compiler, the enhanced JVM that supports RibbonJ at runtime, and the Linux user-space run-time that supports the enhanced JVM. This dissertation also evaluated the practical use of ribbons by “ribbonizing” both large and small applications, by studying the complexity of these refactorings, and by evaluating the performance characteristics of the ribbonized versions vs. their corresponding unmodified versions.

6.1 Conclusions

The applicability of ribbons was explored by first refactoring four small applications to use RibbonJ and then by refactoring two large-scale open-source applications, Apache Tomcat and Apache httpd. In all of these applications the desired isolation could be implemented using ribbons without complex or intrusive code changes. It was found that the complexity required to refactor these applications was related not to the overall size or complexity of the application itself, but rather to how easily the existing software architecture could be adapted to coarsely partition the heap as required in order to achieve the desired isolation properties. In all refactored applications the new isolation properties could be implemented using ribbons in less than 1000 lines of code (representing a change of less than 0.2% of the code for the large Apache applications). While these results do not generalize to all applications,
it provides evidence that in certain cases large-scale, complex applications can make use of ribbons without significantly re-engineering the application.

Performance characteristics of ribbons were evaluated through microbenchmarks, the DaCapo benchmark suite, the SPECweb2009 benchmark, and the ApacheBench benchmarking tool. The microbenchmark results demonstrated that there was no additional overhead in memory access operations for a ribbonized heap vs a non-ribbonized heap. The DaCapo benchmark suite results exhibited mixed results for the complete RibbonJ runtime stack, with some benchmarks showing no difference in performance, some showing minor performance decreases (2% to 6%), and a few cases with more significant overhead (12%). More detailed analysis of the benchmark results demonstrated that the likely primary cause of the reduction in performance in these cases was due to the significantly more complex object allocation code path.

For the webserver benchmarks (SPECweb2009 and ApacheBench), in cases where there was no CPU contention (SPECweb2009 benchmark results and ApacheBench results where the number of concurrent requests was fewer than the number of CPU cores) no statistically significant differences in throughput between the ribbonized and non-ribbonized applications were observed. In cases with CPU contention, throughput decreased, sometimes significantly (a reduction in throughput of 5.8%–21.7% was observed). In all webserver benchmarking cases, with or without CPU contention, ribbonization commonly added between 1ms–5ms to each request: in between the lower and upper quartiles of the distribution of request processing latency it was observed that ribbonized applications had an additional overhead of 1ms–5ms. Behavior for extreme outliers (99.9th percentile in the distribution of latency) was application dependent, where sometimes latency was significantly increased (as was the case with Apache Tomcat) or where latency was sometimes decreased (as was the case with Apache httpd).
6.2 Future Work

The most immediate extension of this work would be to implement support for ribbons within the OS kernel itself, so that ribbons are a first-class construct in the OS just as are processes and threads. Providing support for ribbons within the OS kernel would eliminate the need to use slower inter-process synchronization primitives for inter-ribbon synchronization (instead allowing the use of the intra-process variant of the futex for inter-ribbon synchronization).

A kernel implementation would allow exploration of how to more efficiently implement context switching between ribbons within the same process (an “inter-ribbon context switch”). For example, instead of swapping out the root page table pointer register during an inter-ribbon context switch, the kernel could instead modify the existing page table entries (or page directories) to reflect the new allowed view of the heap. This could be especially efficient when switching between ribbons that have mostly similar views of the heap. If both inter-ribbon context-switching strategies were implemented, then new scheduling algorithms could be explored that scheduled ribbons with similar heap views and memory access patterns onto the same physical CPU core or socket. This could potentially lead to advancements in scheduling efficiency for systems with large numbers of CPU cores, as well as for NUMA systems that have large numbers of CPU sockets.

Some applications may require more than just heap isolation for safeguarding the execution of sub-components (e.g., if sub-components need to execute with different OS-level privileges or have access to a different subset of files or other OS resources). For example, web browsers may desire to execute plugins with highly restricted access to the local filesystem. Additionally, using distinct processes for isolation can provide “kill-safety” [44], if designed correctly, and the concept could be extended so that ribbons could have the “kill-safety” property under certain conditions.

This dissertation focuses solely on isolation of the heap and does not claim to provide complete isolation of sub-components or kill-safety. Full isolation and kill-
safety would require additional support for ribbons within the OS kernel that had
defined recovery semantics when an individual ribbon within a process failed or was
killed. This dissertation lays a foundation for such future work by defining the ribbons
programming model and language (including a formal model), and by implementing
and evaluating the mechanisms necessary for heap isolation.

Another avenue of future work is the ability for a thread to dynamically switch
which ribbon it is running within, termed a “security context switch,” and thus dy-
namically switch its permissions to view and modify the heap. If ribbons provided
isolation for other OS resources, then this would allow a thread to completely switch
its “security context.” Such a mechanism could allow for single-threaded applications
to easily employ ribbons isolation mechanisms without having to implement the abil-
ity for isolated components to run within their own dedicated threads. Open questions
on this line of research include how to properly constrain security context switches
to properly enforce security (especially for “upcalls” where a thread switches to a
ribbon with greater privileges than that of its current ribbon), how to model such
constraints within a language that provides ease of use, modularity, and security,
and how to securely implement support for such constraints and the security context
switch itself within the OS kernel.

Many open questions surround how to best model the isolation of non-heap re-
sources using ribbons, how to model and implement security context switches, and
how to allow for the “kill-safety” of individual ribbons running within a process. The
ribbons programming model is envisioned as providing a comprehensive framework
for the efficient isolation of software components at a language, language runtime,
and systems level, and this dissertation is a step in this direction.
LIST OF REFERENCES
LIST OF REFERENCES


[59] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.


VITA
VITA

Kevin J. Hoffman was born in Provo, Utah. Shortly thereafter he moved to Austin, TX, and later moved to Woodstock, GA. He received his B.S. degree in computer science from Brigham Young University in December 2004. He married his wife, Mindy, in summer 2005. He began graduate school at Purdue University in fall 2005. He received his Ph.D. in computer science in 2013 from Purdue University. During his time at Purdue, he was a member of the Secure Software Systems lab. His research focused on enabling fast, secure, language-based and systems-based solutions for full or partial isolation of application components. He also conducted research in reputation systems, software modularity, aspect-oriented programming methodologies, automated debugging and regression root cause analysis, dynamic program analysis, and scalable yet efficient automated memory management techniques. He is currently the CTO and CEO of eFolder, a cloud storage services company focused on delivering cloud-enabled backup, disaster recovery, and business continuity solutions.