VM Supported AspectJ

by

Ryan M. Golbeck

BMath, University of Waterloo, 2005

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF GRADUATE STUDIES

(Computer Science)

The University of British Columbia

(Vancouver)

November 2010

© Ryan M. Golbeck, 2010
Abstract

Traditional AspectJ implementations use a VM-external implementation approach based on JBC instrumentation. These approaches pose an inherent penalty in both steady state performance as well as startup or recompilation performance. We present a map of the design space for AspectJ implementations with potential advantages and disadvantages to evaluate the coverage of the space by related work to identify unexplored possibilities.

We propose an architecture for AspectJ implementations which supports both VM-internal and VM-external implementations. VM-internal implementations of our architecture fit naturally into existing JVMs, are straightforward to implement and support all known AspectJ optimizations as well as maintaining efficient startup performance. However, because our architecture also supports VM-external implementations, it is both backward and forward compatible.

Additionally, we present a suite of benchmarks designed to evaluate AspectJ implementations. This suite contains macro and micro benchmarks designed to measure the steady state, startup, and overall performance of an implementation.

To demonstrate the viability of our architecture, we present one primary implementation, ajvm, in the Jikes Research Virtual Machine (RVM), and several secondary partial implementations in the Jikes RVM and a second JVM, J9.

We use our proposed benchmark suite to produce an analysis that confirms our claims that VM-integrated AspectJ implementations can support known and proposed steady state and startup optimizations. Furthermore, they overcome inherent performance penalties associated with VM-external implementations.
Preface

Parts of this work involved other collaborators. In particular Sam Davis, Peter Selby, and Igor Ostrovsky. These collaborators all participated in parts of the implementation of the AspectJ benchmark suite we present. Igor implemented the initial version of the microbenchmark generator. Sam continued the implementation of the microbenchmarks and helped direct the statistical analysis portion of the early benchmarks. Finally, Peter worked on the final version of the benchmark suite which included both work on the benchmark framework and implementation of individual benchmarks.

The design of the benchmark suite was directed by me. I also implemented many of the individual benchmarks, as well as the initial versions of the benchmark framework which ties together all of the aspects of the benchmark suite: the benchmarks themselves, the benchmark harnesses, as well as the measured virtual machines and compilers. Work on the framework was continued under my direction by Peter.

The remainder of the work in this dissertation was done by me. This includes the identification and overall design of the research work and experiment; the analysis of the design space and design and implementations of our proposed architecture; and the overall design of the benchmark analysis suite and the analysis itself.

The research in this dissertation has been published in three articles [34], [35] and [36].
# Table of Contents

Abstract ................................................................. ii
Preface ................................................................. iii
Table of Contents ......................................................... iv
List of Tables .......................................................... viii
List of Figures ............................................................ ix
Glossary ................................................................. xi
Acknowledgments ......................................................... xiii
Dedication ................................................................. xiv

1 Introduction ............................................................ 1
  1.1 Thesis Statement .................................................. 4
  1.2 Claims and Contributions ....................................... 4
  1.3 Organization ....................................................... 5

2 Background ........................................................... 7
  2.1 Program Structuring and Modularity ............................ 7
  2.2 Aspect-oriented Programming .................................. 12
  2.3 AspectJ ................................................................. 14
    2.3.1 Example ....................................................... 15
    2.3.2 Pointcuts ..................................................... 16
List of Tables

2.1 Summary of AspectJ Pointcuts ........................................ 17
3.1 Related Work Coverage of Design Space ............................. 43
5.1 Implementation Components Summary ............................... 72
6.1 Startup Workload Summary of Eclipse, Tomcat, and DaCapo .... 104
6.2 Micro Benchmark Generation Control Parameters ............... 111
6.3 Hand Coded Micro Benchmarks ....................................... 113
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Part Simple Figure Editor</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>Display Updating Aspect</td>
<td>14</td>
</tr>
<tr>
<td>2.3</td>
<td>Display Updating Aspect with Cflow</td>
<td>18</td>
</tr>
<tr>
<td>2.4</td>
<td>Java Virtual Machine Architecture</td>
<td>22</td>
</tr>
<tr>
<td>2.5</td>
<td>Example Main Method</td>
<td>25</td>
</tr>
<tr>
<td>2.6</td>
<td>Example Method Bytecode produced from Figure 2.5</td>
<td>26</td>
</tr>
<tr>
<td>4.1</td>
<td>AJVM Architecture</td>
<td>57</td>
</tr>
<tr>
<td>4.2</td>
<td>Example Aspect Configuration File</td>
<td>61</td>
</tr>
<tr>
<td>5.1</td>
<td>Java VM Architecture Annotated with Implemented Sub-components Implementing AspectJ</td>
<td>71</td>
</tr>
<tr>
<td>5.2</td>
<td>Java bytecode (JBC) Generated for a Simple Before Call Advice</td>
<td>76</td>
</tr>
<tr>
<td>5.3</td>
<td>HIR for Simple Before Call Advice and Join Point Generated without Optimizations</td>
<td>77</td>
</tr>
<tr>
<td>5.4</td>
<td>HIR for Simple Before Call Advice and Join Point Generated with Optimizations</td>
<td>78</td>
</tr>
<tr>
<td>5.5</td>
<td>Outer Matching and Munging Loops</td>
<td>85</td>
</tr>
<tr>
<td>5.6</td>
<td>Example Main Method</td>
<td>86</td>
</tr>
<tr>
<td>5.7</td>
<td>Example Aspect and Advice</td>
<td>86</td>
</tr>
<tr>
<td>5.8</td>
<td>Example Method Bytecode produced from Figure 5.6</td>
<td>87</td>
</tr>
<tr>
<td>5.9</td>
<td>Bytecode List Representation of Example Method</td>
<td>87</td>
</tr>
<tr>
<td>5.10</td>
<td>Bytecode List Representation of Example Method After Munging</td>
<td>88</td>
</tr>
<tr>
<td>5.11</td>
<td>Example Method Bytecode after Munging</td>
<td>89</td>
</tr>
<tr>
<td>6.1</td>
<td>Micro benchmark Class Hierarchy</td>
<td>106</td>
</tr>
</tbody>
</table>
## Glossary

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AO</td>
<td>aspect-oriented</td>
</tr>
<tr>
<td>AOP</td>
<td>aspect-oriented programming</td>
</tr>
<tr>
<td>AOS</td>
<td>adaptive optimization system</td>
</tr>
<tr>
<td>API</td>
<td>application programming interface</td>
</tr>
<tr>
<td>APIS</td>
<td>application programming interfaces</td>
</tr>
<tr>
<td>AST</td>
<td>abstract syntax tree</td>
</tr>
<tr>
<td>CCC</td>
<td>crosscutting concern</td>
</tr>
<tr>
<td>CCCS</td>
<td>crosscutting concerns</td>
</tr>
<tr>
<td>COV</td>
<td>coefficient of variation</td>
</tr>
<tr>
<td>DJP</td>
<td>dynamic joinpoint</td>
</tr>
<tr>
<td>DJPS</td>
<td>dynamic joinpoints</td>
</tr>
<tr>
<td>GC</td>
<td>garbage collector</td>
</tr>
<tr>
<td>GUI</td>
<td>graphical user interface</td>
</tr>
<tr>
<td>HIR</td>
<td>high level intermediate representation</td>
</tr>
<tr>
<td>IR</td>
<td>intermediate representation</td>
</tr>
<tr>
<td>ITD</td>
<td>inter-type declaration</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>ITDS</td>
<td>inter-type declarations</td>
</tr>
<tr>
<td>J9</td>
<td>IBM J9 Virtual Machine</td>
</tr>
<tr>
<td>JBC</td>
<td>Java bytecode</td>
</tr>
<tr>
<td>JIT</td>
<td>just-in-time</td>
</tr>
<tr>
<td>JPM</td>
<td>join point model</td>
</tr>
<tr>
<td>JPMS</td>
<td>join point models</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>JVMS</td>
<td>Java Virtual Machines</td>
</tr>
<tr>
<td>LIR</td>
<td>low level intermediate representation</td>
</tr>
<tr>
<td>MIR</td>
<td>machine level intermediate representation</td>
</tr>
<tr>
<td>MMTK</td>
<td>memory management toolkit</td>
</tr>
<tr>
<td>OO</td>
<td>object oriented</td>
</tr>
<tr>
<td>OS</td>
<td>operating system</td>
</tr>
<tr>
<td>OSR</td>
<td>on-stack replacement</td>
</tr>
<tr>
<td>RVM</td>
<td>Research Virtual Machine</td>
</tr>
<tr>
<td>TPT</td>
<td>type pattern table</td>
</tr>
<tr>
<td>TPTS</td>
<td>type pattern tables</td>
</tr>
<tr>
<td>VM</td>
<td>virtual machine</td>
</tr>
<tr>
<td>VMS</td>
<td>virtual machines</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
</tbody>
</table>
Acknowledgments

I would like to thank my supervisor, Gregor Kiczales, from whom I learned much more than is represented here, and my supervisory committee, Norm Hutchin-son and Andrew Warfield for their feedback and approachability. They definitely helped to make the process easier.

I would also like to thank my friends and family, especially Colleen, for putting up with being married to a PhD student and everything that goes with that, and, the Mighty bike shop guys and club for all the riding and team meetings.

I am also thankful for the of IBM Centers for Advanced Study and The Natural Sciences and Engineering Research Council of Canada.
Dedication

This thesis is dedicated to my family. Both my parents, Joni and Robert, my sister Sarah, and wife Colleen, all of whom without I would never have made it this far. I wish you were here to see it, dad.
Chapter 1

Introduction

Modern programming languages have evolved significantly since their inception in the 50s. Starting with complicated direct assembly programming, to the first appearances of the Algol and lisp-based languages like FORTRAN and Common Lisp. Over this period of time, much research has guided the creation and adoption of new programming techniques, idioms and languages. Starting with modularizing programs procedurally, functionally [48, 49] and with object-oriented [64] based methods, and with more modern modularization techniques such as feature-oriented programming [11, 24, 59], context-oriented programming [23, 51, 77], and aspect-oriented programming [52, 54, 55, 62].

Modern programming languages are supported by robust runtime environments. These environments range from reasonably complete runtime libraries for functionality, such as the standard C library, to managed runtime support including complex memory management with garbage collection [9, 13, 66], and dynamic optimizing compilers [21, 47, 71], such as with high performance Java Virtual Machines (JVMS) [39, 57].

High level programming language runtime environments as rich as modern JVMS have to balance a number of considerations of the programs that they execute. Programs written in these languages range from short running script-like programs, to long running server programs, to CPU-intensive programs using scientific and numerical computations. One such performance trade-off is the balance between startup execution performance (typically required by short running or interactive
programs) versus long running performance (typically required by long running server programs or CPU-intensive applications).

A large body of research has explored issues in both startup performance and long running performance resulting in Java Virtual Machine (JVM) architectures containing interpreters, or baseline quick compilers, together with more costly runtime dynamic optimizing compilers. [14, 31, 33].

This dissertation focuses on aspect-oriented programming (AOP) [52], and more specifically the AspectJ [55] AOP programming language, and how this programming language can be efficiently and naturally implemented with the support of a virtual machine (VM) runtime environment.

Typical AspectJ implementations have used unmodified JVMs as a target architecture. Although AspectJ can be implemented using Java bytecode (JBC), in practice it constrains programmers by imposing a performance limitation during development, it limits the flexibility of dynamic configuration of already deployed systems, and it is unable to exploit some possible optimizations that require the support of the runtime system.

For example, Wiese et al. [78] describe the successful use of AspectJ in a large industry project developing a hospital information system which provides access to patient data like medical results, medication and prescriptions. However, in their case study they noted a build speed decrease of around 12% and a recompilation pause of 2-3 seconds in Eclipse whenever a modification was made to some large modules using a traditional offline AspectJ implementation. These speed decreases posed unacceptable wait times for developers during the edit-compile-debug cycle, and so they constrained AspectJ usage to a secondary part of the build process which is tested separately so that the main unit test suites can be run during development.

Furthermore, a critical piece of their application is the ability to perform optional live performance monitoring of the system. This functionality is modularly captured using AspectJ constructs, but its use requires execution-time configuration to control whether monitoring should be enabled or not on an already deployed system.

Additionally, since traditional approaches target unmodified JVMs, they are unable to perform some optimizations because of the semantic constraints of the
Many of these issues stem from AspectJ’s *pointcut and advice* mechanism which allows a programmer to specify well-defined points in a program’s execution at which additional blocks of code should be executed. Performing or arranging for this execution to happen is a dispatch style operation with both lookup and invocation phases typically called pointcut matching and advice munging respectively. This process differs from virtual method lookup and invocation in one very important way: it is non-local in nature. Because it is non-local, small code changes to a module in build environments using traditional AspectJ implementations can result in the recompilation of large portions of a program which can cause, as noticed by Wiese *et al.* above, long pauses in IDEs like Eclipse that continuously rebuild programs during editing.

We present an architecture for a VM-integrated AspectJ implementation that addresses all of these concerns. A VM-integrated approach can free the build environment from the non-local nature of pointcuts and advice by integrating their implementation into the runtime system. Our startup performance analysis of an implementation of our architecture show that this integration removes onerous recompilation requirements that cause delays during the edit-compile-debug cycle, like those described by Wiese *et al.*, without negatively impacting the startup performance of the program. This same integration allows the efficient implementation of late configuration of systems by allowing the deployment of AspectJ’s pointcuts and advice to be changed on a per-execution basis.

Our architecture and implementation also support all known AspectJ specific optimizations, some of which are not possible to implement in traditional AspectJ implementations that target unmodified JVMs. Our steady state performance analysis of our implementation demonstrates that these optimizations have a real measurable impact on the performance of long running programs.

Finally, our architecture is designed in a way that it is both forward and backward compatible with existing AspectJ programs and JVMs and that implementations of the architecture are lightweight additions to modern VMs.
1.1 Thesis Statement

We show a VM-integrated AspectJ implementation that can support all known AspectJ optimizations, thereby reducing the problem of optimizing AspectJ execution to that of optimizing standard Java execution. Furthermore, the architecture is such that these optimizations are straightforward to implement; it fits naturally into existing staged JVM architectures; and it maintains forward and backward compatibility with existing AspectJ compilers and JVMs.

1.2 Claims and Contributions

We support our thesis statement by making the following claims:

- our proposed architecture enables optimizations that reduce the problem of optimizing AspectJ to one of optimizing plain Java because:
  - the optimizations largely exploit existing JVM infrastructure;
  - startup time is reduced to standard Java in most cases, and near standard Java in degenerate cases;
  - additional memory consumption is linear: no expensive data structures are required; and
  - major areas of our implementation are macro optimized: class loading, advice matching and munging, and steady state code quality.

- it is both forward and backward compatible;

- it is a lightweight addition to modern VMs;

In our work we produce the following contributions:

- an architecture of a proposed VM-integrated AspectJ implementation;

- an implementation of the proposed architecture;

- a design of an AspectJ benchmark suite including startup and steady state measurement methodology, and an implementation of the benchmark suite with a combination of existing and novel benchmarks (joint work\(^1\)); and

\(^1\)See Preface.
• an analysis of the data produced by the benchmark suite with conclusions
drawn regarding the AspectJ design space and defense of claims.

• a mapping of the AspectJ implementation design space and characterization
our work and related work in terms of this mapping;

• several partial implementations in the Jikes RVM and J9;

1.3 Organization

We map out the complete design space for AspectJ implementations described in
Chapter 3. Since AspectJ is an extension to the Java programming language and
traditional approaches for implementation have targeted the JVM, our design space
analysis includes AspectJ-specific modifications to the JVM. We characterize re-
lated work in terms of our mapping so as to place our work in context with other
work in this area, as well as to highlight areas of the design space that have been
previously unexplored.

We use this mapping to propose an architecture for an AspectJ implementation
that addresses the shortfalls identified in other approaches while still operating un-
der design constraints imposed by modern JVMs. This architecture is described
in detail in Chapter 4. Our proposed architecture still allows for implementa-
tion-specific design decisions, so we implement our architecture, with several different
design decisions, in the Jikes RVM, and partially implement it in IBM’s production
IBM J9 Virtual Machine (J9). These implementations are used with another sup-
plemental implementation and related work to evaluate the AspectJ design space.
All three of our implementations are described in detail in Chapter 5.

Our evaluation uses a proposed suite of AspectJ benchmarks designed to evalu-
ate varying language features, as well as evaluate different properties of an AspectJ
implementation. This benchmark suite design and implementation is discussed in
Chapter 6. Chapter 7 presents a whole picture analysis of our implementations and
related work where appropriate using the benchmark suite. This analysis demon-
strates the traditional trade-offs made in the different approaches, and compares
these approaches to suggest a definitive ideal AspectJ implementation strategy that
achieves efficient performance given the design constraints of modern JVMs. We conclude in Chapter 8.
Chapter 2

Background

This chapter covers important background information and work that this dissertation builds on. The first three sections, Section 2.1 - Section 2.3 broadly describe program modularity and the AspectJ programming language, which is a focus of this work. The next two sections, Section 2.4 and Section 2.5 cover important details about Java and Java virtual machines in general. Finally, the last section, Section 2.6, covers important details about the Jikes RVM, the JVM that most of our work uses.

2.1 Program Structuring and Modularity

Program structuring plays an increasingly important role in development of large programs with many developers. A programming language that allows the structure of a program’s source code to reflect its conceptual design allows programmers to more easily understand and manipulate the program. In large programs with a programming team of varying experience, this becomes especially important for maintaining the program and enhancing its readability.

Although small programs written in machine code can be very elegantly designed, as they became larger with more developers, they become much more difficult to maintain. The problem being that machine code has few built-in constructs for structuring related parts of a program.

Typical machine code languages allow programmers to structure a program
into logical units by allowing the source code to be divided up between files. Within each file the programmer can specify labels as entry points or as locations allocated memory to share data that allows a compilation and linking program to join these files together using the labels as cross references. These pieces can then be assembled into larger and larger programs and libraries.

Idiomatic use of exported labels in files this way naturally fits into a procedure-calling based organization. By following specified stack and calling conventions, the contract specifying where to find parameters and return values between a procedure caller and callee, programmers gain a reliable way to separate and structure these units.

Using exported labels in this way, together with other idiomatic use of private labels to encode local variables and looping constructs, this type of programming has a natural correspondence with early block structured ALGOL based languages.

For example, in C-like languages, there is a direct correspondence between the block structure of the C program and the machine code that a compiler would produce while compiling this block structure. These languages are typically compiled straight into the separate-file-plus-public-and-private-label format that a machine code program would be written in directly.

Block organization of programs in this way also allowed nesting of the blocks such as in Simula [64]. Nesting creates a hierarchy of block scopes that both helps to break up the program into logical units, but also provides an inherent scoping of blocks for visibility and hiding.

Simula, although created before the term was coined, is widely considered the first object oriented (OO) programming language and heavily influenced modern OO languages. Designed primarily for creating discrete event simulations, it supports constructs to create process-like objects which conceptually correspond to real world objects in the simulation.

More modern OO languages include Smalltalk [27, 38], C++, and Java. Although not all of these languages are “purely” OO: Java, for example, has both primitive values and objects, whereas in Smalltalk, a pure OO language, even integers and strings are treated as objects where addition and concatenation operations are treated as standard message passing.

The key insight in OO languages is that language features allow the program-
Figure 2.1: Part of a simple figure editor from [55]. This example shows two figure elements in the program, Points and Lines. The display in this program is represented by an object stored in the global variable Display.
mer to structure the program in a way that is very similar to how the programmer would think about the program abstractly. Many programs, especially those that correspond closely with the real world, are naturally broken down into classes of objects that correspond to real world things. It is natural to think of these objects as their own entities in the program that communicate among one another via messages. Each object represents a concrete unit of data and operations on that data that naturally fit together. Most OO languages also allow similar classes of objects to be grouped together and share implementation using either interfaces or inheritance.

Visual languages \[60, 69\] allow the structuring of the program graphically which can be especially helpful to first time programmers by visually reinforcing the block and hierarchical structure of OO programs that can be less apparent textually. Experienced programmers tend to have an intuitive notion of this structure already in their head.

Each program is said to have a dominant decomposition: the primary way in which that program is structured. In an OO language as above, the dominant decomposition is into a hierarchy of objects. This decomposition concisely captures many concerns into modular units that are represented textually close in the code \[53\].

There are, however, many other concerns which conceptually, and in design documents, are thought of as one unit but whose implementations end up spread throughout many of classes which are concisely modularized in design documents. These concerns crosscut the primary decomposition.

Consider the program snippet in Figure 2.1. In this program, we can move an entire figure element by a delta x and delta y on the screen using the incrXY(x,y) method on FigureElements. However, this movement only updates the model of the figures in the program, it does not refresh the display to show the updated figure positions.

One way of ensuring the display is updated is by relying on programmers to ensure that a call to Display.update() is made every time code they write makes a change to any FigureElements. But this is error prone, especially if some FigureElements are updated as side-effects deep in a call graph.

Another approach is to modify each FigureElement class to call
Display.update() as the last thing it does in each method in that class that moves the figure. This would result in many code blocks similar to:

```java
public int setX(int newX) {
    x = newX;
    Display.update();
}

public int setY(int newY) {
    y = newY;
    Display.update();
}
```

Note that in our small example, higher level methods like Point.incrXY don’t need an explicit call to Display.update() because they update the x and y position using helper methods. A more complicated figure object may not be able to rely on helper methods updating the display, however, and so even higher level objects may need their own calls to Display.update() to refresh the display.

Furthermore, it is interesting to note that a call to Point.incrXY now results in two calls to Display.update() for each point movement. This could produce erroneous results as the Point moves quickly along the x-axis before appearing at its final location on the screen. But, we ignore this effect for now.

So, abstractly, we want the display to be updated every time a figure is moved in our model. Even though we can describe this functionality or concern in our program concisely and modularly in English, the implementation of display updating is spread across at least two methods in our program and potentially many more in the whole program. More concisely, display updating should really occur if any of the four methods Point.setX, Point.setY, Line.setP1, or Line.setP2 is called. And we end up with calls to Display.update() spread throughout the program.

So we see that the display updating concern in this program is not easy to modularize into its own class. Much of the implementation of the concern must be spread throughout the other classes of the program. Although in some cases, the addition of features, such as display updating, to a program cannot be modularized concisely because of a poor original design, there is an inherent conflict between some pairwise features that makes them impossible to modularize simultaneously.
in a single method of decomposition. We call these pairs of features, or concerns, *crosscutting* because one of them will necessarily crosscut the modularization of the other in some representation.

In [50], Janzen et al. present an editor which fluidly allows the programmer to change the view of the dominant decomposition of their program to manipulate it with respect to different decompositions. This editor would allow one view of the program to present the decomposition of the program by class hierarchy in the traditional OO sense. But, another view of the program could present the program decomposed by crosscutting modules, such as the display updating “module.”

An alternative solution is to add programming language constructs that help modularize crosscutting concerns. We present this area of research, aspect-oriented programming, in the following section.

### 2.2 Aspect-oriented Programming

There have always been multiple ways of breaking down our programs into views or effective ways to think about how programs are composed. The primary way of viewing program composition is through their actual textual dominant decomposition, typically procedure-based or OO, but during debugging a program’s composition is often thought of in terms of its data flow or control flow. Furthermore, design documents often view programs in this way in terms of use-cases or program features which do not typically map one-to-one onto a set of classes in the program text.

Recently, alternate ways of structuring a program text have risen to better handle textual modularization of programs in light of these different composition views. These include techniques such as featured-oriented programming [11, 24, 59], context-oriented programming [23, 51, 77] and aspect-oriented programming [52, 54, 55, 62, 73]. In this work we focus specifically on AOP approaches to modularizing naturally crosscutting concerns in a program.

Aspect-oriented programming gives us another way of modularizing these types of concerns. It has proven especially useful at modularizing non-functional features such as security, tracing, error handling [58], object persistence [68], synchronization [26], and design patterns [42]. These types of features tend to interact
with the traditional, functional part of a program’s object hierarchy in a crosscutting way. For example, object persistence would require code to be scattered throughout disparate objects which either participate in persistence or are the targets of persistence. This scattering of code makes it hard to reason about the singular concern of persistence in a program, and makes it especially hard to ensure that object persistence is handled in a consistent way in a program.

Aspect-oriented techniques allow such concerns to be expressed in textually modular fashions. An aspect-oriented (AO) compiler or runtime environment would then be used to join these concerns appropriately with the concerns represented by the dominant decomposition of the program.

More formally, many AOP language constructs can be described in terms of a join point model (JPM). A JPM defines three things:

1. A set of join points. These are the points at which aspect and non-aspect functionality will join. These points can range from statically identifiable program elements like classes, method declarations, and if statements, to dynamic events such as the execution of a method, the access of a member variable, or the catching of an exception.

2. A means of quantifying join points. This is the way that join points can be identified, selected and grouped so that they can be referred to.

3. A means of affecting a join point. That is, a way of causing the execution of a subset of identified join points, using the quantification mechanism, to be altered. For example, causing some body of code to be executed before executing an identified join point.

These three language constructs work together to improve the modularity of the program in question by enabling the programmer to modularize concerns which crosscut each other in the dominant decomposition of the program. In Java programs, the dominant decomposition is a hierarchical decomposition in the form of classes, sub-classes, interfaces, and inheritance relationships. Aspects in AspectJ, an AOP language built upon Java, give the programmer an additional mechanism for modularizing concerns which do not modularize well with this hierarchical organization.
aspect DisplayUpdating {
    pointcut move():
        call(void Line.setP1(Point)) ||
        call(void Line.setP2(Point)) ||
        call(void Point.setX(int)) ||
        call(void Point.setY(int)) ||
        call(void FigureElement.incrXY());
    after(): move() {
        Display.update();
    }
}

Figure 2.2: Display Updating Aspect. This aspect modularizes the display updating concern in Figure 2.1 and Section 2.1.

2.3 AspectJ

AspectJ has been the defacto standard implementation of an AOP language based upon Java. AspectJ has two different join point models (JPMS) which can work together to modularize many different crosscutting concerns (CCCS).

It has a static join point model in which the join points are Java language types, the quantification mechanism is a declarative pattern language over these types, and the effects are the insertion of members into matched types. This JPM is static in the sense that the join points it refers to are static parts of the programming language. This JPM is typically referred to as inter-type declarations (ITDS).

The second is a dynamic JPM. Join points in the dynamic model are well defined points in a program's execution called dynamic join points. These are events such as the execution of a method, the setting of a field, or the construction of an object. Like the static JPM, the quantification mechanism is a declarative pattern language over these program events called the pointcut language. Advice, the means of effect, are bodies of code that are associated with pointcuts that are to be executed at identified join points. Finally, aspects are the means in which pointcuts, advice, and other state associated with them are encapsulated and modularized in the programming language.
2.3.1 Example

Figure 2.2\textsuperscript{1} shows an example of these three constructs in an aspect in AspectJ. This aspect modularizes the display updating crosscutting concern (CCC) from Section 2.1. It refers to classes shown in Figure 2.1. Here, as declared on line 1, aspects are similar to standard classes and act in similar ways. They encapsulate shared state between advice and pointcuts the same way a class encapsulates shared state between methods.

This aspect defines one pointcut, move, on line 2 which picks out call join points to various set and increment methods in the code base. This pointcut is an aggregate of primitive pointcuts; the primitive pointcuts are designators like the above exampled call pointcuts, which take a method signature with optional wildcards, as their parameter. Each of these primitive pointcuts are tied together by the ||, or, operator. Since this pointcut is named, it can be used by other aspects and multiple pieces of advice. In this way, the pointcut becomes abstracted by its functionality: it is designed to capture all join points in the program which correspond to a move of GUI elements.

Line 9 shows the definition of an advice using the pointcut defined above it. This definition pairs the pointcut and advice together and is all that is required for the advice to be effective in the program. This means that the execution of any join point matched by the pointcut move will cause this advice body to be executed. This advice is further defined to be an after advice which causes the advice body to be executed after the matched join point executes. Other options include before and around. AspectJ provides functionality and syntax for capturing contextual state from the joint point and making it available in the advice body by passing parameters from the pointcut into the advice body.

Finally, the advice body simply makes a call to Display.update(). This aspect captures the abstract functionality (concern) of updating the GUI display after any of the GUI elements have moved in the display. Instead of calls to Display.update() being littered throughout the code base in various GUI elements, AspectJ allows them to be modularized into a single aspect capturing the functionality.

\textsuperscript{1}This example is a modified version of the example used in [55].
2.3.2 Pointcuts

AspectJ supports many different primitive pointcuts. These are summarized briefly in Table 2.1.

<table>
<thead>
<tr>
<th>Pointcut</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>call(MethodPattern)</td>
<td>Matches call join points to matching method or constructor patterns.</td>
</tr>
<tr>
<td>execution(MethodPattern)</td>
<td>Matches execution join points to matching method or constructor patterns.</td>
</tr>
<tr>
<td>get(FieldPattern)</td>
<td>Matches references join points to matching field patterns.</td>
</tr>
<tr>
<td>set(FieldPattern)</td>
<td>Matches assignment join points to matching field patterns.</td>
</tr>
<tr>
<td>initialization(ConPattern)</td>
<td>Matches the initialization join point of an object based on the constructor pattern.</td>
</tr>
<tr>
<td>preinitialization(ConPattern)</td>
<td>Matches the pre-initialization join point of an object based on the constructor pattern.</td>
</tr>
<tr>
<td>staticinitialization(TypePattern)</td>
<td>Matches static initialization join points of a class based on the type pattern.</td>
</tr>
<tr>
<td>handler(TypePattern)</td>
<td>Matches exception handler join points based on the type pattern.</td>
</tr>
<tr>
<td>adviceexecution()</td>
<td>Matches all advice execution join points.</td>
</tr>
<tr>
<td>within(TypePattern)</td>
<td>Matches all join points given rise to by executing code within a type matching the type pattern.</td>
</tr>
<tr>
<td>withincode(MethodPattern)</td>
<td>Matches all join points given rise to by executing code within a method matching the method pattern.</td>
</tr>
<tr>
<td>this(TypePattern)</td>
<td>Matches all join points whose “this” object matches the type pattern.</td>
</tr>
<tr>
<td></td>
<td>This pointcut can also be used to bind the “this” object to a name in the advice body.</td>
</tr>
</tbody>
</table>
target(TypePattern)  Matches all join points whose “target” object matches the type pattern. This pointcut can also be used to bind the target object to a name for use in the advice body.

args(TypePattern, ...) Matches all join points whose argument types match the type pattern. This pointcut can also be used to bind any of the argument values to names in the advice body.

cflow(Pointcut)  Matches every join point in the control flow (dynamic extent) of join points matched by Pointcut, including that join point itself.

cflowbelow(Pointcut)  Matches every join point in the control flow (dynamic extent) of join points matched by Pointcut, excluding that join point itself.

if(BooleanExpression)  Matches every join point for which Boolean-Expression evaluates to true.

Table 2.1: Summary of AspectJ Pointcuts. MethodPattern, FieldPattern, ConPattern (Constructor Pattern), and TypePattern are all a simple pattern language which match against Methods, Fields, Constructors and Types respectively.

Pointcuts are combined using the logical operators &&, ||, and !.

The most frequently used pointcuts are for matching method calls or executions and field references. Call and execution pointcuts differ in their context of the executing object bound to this. Call pointcuts have this bound to the calling object, whereas execution pointcuts have this bound to the callee object.

Other less common specialized pointcuts (initialization, preinitialization, and staticinitialization) allow the programmer to pick out subtle specialized parts of an object’s and class’ initialization. preinitialization and initialization correspond to the execution of the matching type’s super class initialization code, and that type’s own initialization code respectively. While staticinitialization join points cover the execution of the static initializers of a matched class.
The `within` and `withincode` pointcuts allow the programmer to lexically scope the applicability of the advice. For example, the pointcut `within(Line)` captures all join points which were given rise to by code lexically scoped within the `Line` class. `withincode` pointcuts allow further scoping based on individual methods. In this sense, these two pointcuts don’t quite fit in with the rest because they constrain which dynamic joinpoints (DJPS) match based on the lexical location of the source code that gives rise to those DJPS.

The `this`, `target`, and `args` are pointcuts which allow the programmer to constrain join point matching based on dynamic type information, as well allow the programmer to bind those objects to names at runtime. For example, the following pointcut:

```java
call(void Line.setP1(Point)) && target(Line l) && args(Point p)
```

captures calls to `Line.setP1(Point)`, but also binds `l` to the `Line` object participating in the call, and `p` to the argument of the call in the advice body.

Lastly, one more notable pointcut which has been the focus of some optimization strategies is `cflow`. This pointcut allows the programmer to use dynamic extent information during the running of the program in selecting applicable join points. For example, it allows the programmer to identify executions of a given method that are in the control flow of another method. This pointcut has typically been expensive to implement because it requires either stack walking or control flow entry and exit accounting to determine if a method is within the control flow of another.

---

**Figure 2.3: Display Updating Aspect with Cflow**
Figure 2.3 shows the aspect from the previous example modified to take advantage of the `cflow` pointcut. In the previous example, the display would be updated every time any GUI element was modified. So, if a `Line` object was moved, the advice would be executed several times, once after the line has moved and also several times after the two end points in the line object move as a result. Note, however, that the movement of the two end points happens within the control flow of the movement of the line: `Line.incrXY()` is on the stack below calls to `Point.incrXY()` during line moves.

The after advice in Figure 2.3 takes advantage of an idiomatic use of the `cflowbelow` pointcut, `move() && !cflowbelow(move())`, to avoid these multiple advice executions for a single conceptual GUI element move. Abstractly we can read this pointcut as saying that we want to capture all top-level GUI element moves, excluding any GUI element moves within the control flow of the the top-level one, after which we update the display. This usage of `cflowbelow` ensures that when we move a compound GUI element, such as a line (which is built from two points), that the update happens only after the entire line move has happened.

There are several terms associated with the `cflow` pointcut which talk about different parts of the pointcut. The pointcut inside the `cflow` clause is typically called the `cflow boundary`. In Figure 2.3, the `cflow` boundaries are the join points matched by the `move()` pointcut. This, target, and args pointcuts can capture and bind values at `cflow` boundaries. These bindings are typically said to be recorded in a `cflow` stack, even if the actual implementation does not explicitly represent the stack.

Finally, the `move()` pointcut appearing outside the `cflowbelow` pointcut is called the `dependent` pointcut because join points only match this pointcut when in the control flow of the other: they depend on it. When testing these join points for matches they require a match against the dependent pointcut, but must also be checked to ensure that they are within the control flow of the boundary pointcut. This second check is typically called `cflow` testing.
2.3.3 Aspect Instantiation

Aspects in AspectJ have a runtime presence that is similar to Java objects. They may have member fields and methods, just like Java objects, however their instantiation is dictated by a selected aspect instantiation model rather than directly by the programmer. Aspect instances can be retrieved by calling the static `aspectOf(..)` method on the aspect.

The primary model for instantiation is that the aspect has a single instance for the entire execution of the program. This is the `issingleton` model. This aspect instance is guaranteed to exist for the entire execution of the program, so that the call to `aspectOf()` will always return the same valid aspect instance.

Other models include the `perthis` and `pertarget` models which allocate one aspect instance per this or target objects at specified dynamic joinpoint (DJP) executions. This allocation model is used when an aspect’s state is more naturally associated in a one-to-one correspondence with either the source (this) or target of a DJP. These aspect instances can be retrieved by calling the static `aspectOf(Object)` method on that aspect where the `Object` parameter specifies which instance of the aspect should be returned.

Finally, there is the `percflow` instantiation model. This model associates one aspect instance for a specified control flow in the program. Every time the specified control flow is entered, a new aspect instance will be created, and will then be accessible using the `aspectOf()` static method.

All of the details of the AspectJ language are covered in the AspectJ Programming Guide [5].

2.4 Java Virtual Machines

Since virtual machines (VMS) comprise a large portion of our work, it is necessary to have an understanding of their architecture. This understanding better allows us to identify the VM system’s which could be used to implement or optimize AspectJ.

There are two broad definitions of VMS, one from the systems perspective and one from the programming language perspective. Although they overlap in a few of their properties, they are two largely distinct types of programs.

From the programming language perspective, VMS are an abstract hardware
machine that are designed to execute the hosted programming language efficiently by providing and handling a range of runtime features such as memory management, thread management, and code optimization. This type of VM provides an abstract machine interface which client programs can target, and for many, running more than one program at a time is not a consideration.

From a system’s perspective, VMS are hardware or platform abstraction programs which are designed to provide an interface to hosted software that mimics the interface that would have been provided by the hosting platform [10, 67]. However, it does it in a way that it can run multiple occurrences of hosted programs without them knowing that they are running on an abstraction layer. In this way, the VM multiplexes the hardware by virtualizing the hardware interfaces presented to hosted software systems.

The first type of VM is what we are primarily concerned with, and they have been studied in various incarnations since the 1960s with the Pascal p-code machine and variants of the Lisp programming language that were implemented in runtime systems such as the Interlisp-D [22] implementation of Interlisp. Other languages and implementations of special note are Smalltalk [27] and Self [46], which laid the foundational work for modern VMS such as the JVM [57].

These implementations demonstrate that reasonable performance can be achieved in a runtime environment that provides automatic memory management, dynamic optimizing compilers, and allow programming languages, like Java, to be compiled once and run on any hardware architecture which has an implementation the VM specification.

Figure 2.4 shows a simplified diagram of the architecture of a modern JVM. This diagram focuses on the information flow of the program (via arrows) through the various subsystems (boxes) of the VM. The dotted lines and boxes refer to subsystems of the architecture which are necessary, but do not directly affect our focus.

Java source files are compiled into Java class files, a standard binary representation of the JBC model. These formats are further discussed in Section 2.5. The output of the offline compiler defines the interface between the offline portion of the architecture and the runtime portion. Output from this compiler conforms to the JVM specification so that it can be loaded by any standard VM.
When the program is run, these class files are loaded by the VM through a multi-stage class loading infrastructure. The specification of this infrastructure allows user programs to extend the class loader to provide additional functionality, although custom class loaders must eventually use the VM default class loader to ultimately load the class. This is described in the architecture diagram as the external class loader. If the user does not provide their own class loader, the standard Java runtime library defines a default class loader.

class files are then processed by the VM’s internal class loading mechanisms. These mechanisms perform several important steps that verify the incoming class file for consistency. This verification ensures that the VM can rely on specific guarantees about the safety of the bytecode and other information found in the class file. For example, method bytecode is checked for rudimentary type safety and its adherence to stack and local variable constraints. Additionally, the internal class loader translates the class file format into the VM’s own internal, implementation dependent class model.

Member methods of a loaded class are typically executed for the first number of executions by an interpreter. The interpreter directly processes and dispatches
the bytecode of the method being executed, rather than compiling it to native code. The primary use of an interpreter in this way is to improve the startup performance of the VM as it begins to execute a program. Since most methods are executed an infrequent number of times [56], it is quicker to interpret them than to compile them and execute the more efficient compiled version of the method.

Frequently executed methods are identified via dynamic profiling in the interpreter. These are then compiled by a dynamic optimizing compiler, called the just-in-time (JIT) compiler, inside the VM. This compiler translates the Java bytecode directly into the native code of the host computer in a multistage process and typically uses optimization plans based on the profiling data stored from previous executions of the method. Note that in most VMS a method that has already been compiled by the JIT compiler can be recompiled with higher, more costly optimization plans if new profiling data determines that this would be a beneficial operation.

Both the interpreter and JIT compilers in the VM must conform to and share some runtime invariants imposed by the various other subsystems present in the VM. For example, a garbage collector (GC), necessary to the Java programming language, imposes some constraints on the layout of the stack and objects in memory. Both the interpreter and JIT must conform to these models if the GC is going to be correct and efficient. Additionally, the scheduler imposes constraints on execution for atomicity and processor sharing. Finally, since the JVM architecture was designed to run one program at a time, some VMS also use areas of read-only memory to store data that can be shared across multiple VM instances. Static data that does not change across VM invocations is typically stored in these read-only segments. Commonly, this data tends to be VM-internal class representations and common runtime libraries. The specific constraints tend to be implementation dependent, but they do exist in each implementation.

Furthermore, Java does not require compile-time linking of references between classes before a program is run. Instead, the VM is relied upon to dynamically locate and load classes by reference when they are needed by an executing program. This means that when a program first begins executing only its main class will have been loaded (unless some other class references have been made in the static initialization of the main class). The rest of the program is dynamically loaded on
demand as more classes of the program are used. An implementation which causes static initialization of a class which hasn’t been dynamically required by the program isn’t in compliance with the Java specification and could cause side effects in an executing program which do not occur on compliant JVMs.

This part of the specification results in a potentially beneficial side-effect whereby the JVM does not incur the overhead of loading classes which are never used by a particular run of the program.

2.5 Java Bytecode

The Java Bytecode specification is described in detail in [57]. However, a moderate understanding of the semantics and operation of JBC, and the corresponding stack machine, are important to understanding the AspectJ implementation space. Below we cover the necessary details of this specification.

The bytecode accepted by the JVM must be stored in the Java class format. This format stores a single class per file, and separates methods, fields and string constants into their own sections of the file. The part containing string constants is called the constant pool. All string data, including references to methods, fields and other classes, even if the field is in the enclosing class, are stored in this constant pool and referenced by index.

Since Java is a dynamically linked language, all method and field references are stored as string designators in the constant pool. When bytecode needs to refer to one of these designators, it does so using an index into the constant pool of the containing class which holds the designator. So, a method call with the invokevirtual bytecode instruction will take as a parameter the index into the constant pool of the string designating the method to be called.

The constant pool uses a number of techniques to eliminate duplicate strings, and otherwise compress the size of the pool, however it is sufficient to know that the constant pool holds a representation of much of the information in each Java class.

The constant pool is an important way of maintaining string data and at the same time reducing the size of the Java class on disk. However, the constant pool represents a problem for programs which manipulate bytecode directly by adding
public static void main(String[] args) {
    MainClass c = new MainClass();

    for (int i = 0; i < 10000000; i++) {
        c.test1();
        c.test2();
    }
}

Figure 2.5: Example Main Method

references because they must also properly manage the constant pool to add these
references. There is no facility in the bytecode to embed references directly.

Of primary concern to our work is the encoding of methods and their byte-
code specifically, as well as the encoding of additional meta-data in the class files.
Method bytecodes are stored in a byte array in the class file. Java has a comprehen-
sive set of bytecode instructions. Each of the bytecodes is of variable length,
and so the byte array is only understandable as bytecode when it is read as a stream
because the beginning of the next bytecode is only known by decoding the current
one and determining its length.

Bytecode jumps, conditional or unconditional, are represented as offsets in
bytes and stored as part of the jump bytecode. This means that bytecodes can-
not be inserted in the middle of an array without recomputing offsets in the rest of
the array. Similarly, the exception table for each method is stored as offsets into the
byte array representing the beginning and end of catch blocks, as well as an offset
specifying the beginning of the exception handler.

Many bytecodes, including invokevirtual for invoking methods, and
getfield for retrieving the value in a field, must encode references to the method
or field that they refer to. These references are embeded in the bytecode as a 2-byte
index into the enclosing class’ constant pool.

Furthermore, the Java bytecode language is a stack language. So arguments to
methods and operators must be pushed onto the Java stack with push instructions
before invoking that method or operator. For optimization purposes, the maximum
stack height and number of required local variables are pre-computed by the static
compiler and stored with the method information.

An example listing of bytecode generated from Figure 2.5 is shown in Fig-


**Figure 2.6: Example Method Bytecode produced from Figure 2.5**

The numbers in the left column indicate the bytecode index of the bytecode on that line. These are followed by the name of the bytecode and any references that bytecode uses. Numbers of the form “#xx” indicate offsets into the constant pool, whereas the number in “goto 24” refers to an absolute bytecode offset from the beginning of the method.

A bytecode vector like this is executed from start to end with most of the bytecodes manipulating entries on the top of the stack. For example, the first bytecode, `new`, creates a `MainClass` object and pushes the created instance onto the Java stack.

### 2.6 Jikes RVM

Much of our work is based upon the Jikes Research Virtual Machine (Jikes RVM) which is an open source JVM implemented almost entirely in Java [19]. The only portions of the VM that are not implemented in Java are the portions that bootstrap the startup of the VM which follow the protocol for binary execution of the host platform.

The architecture of the Jikes RVM differs slightly from the architecture of other modern Java VMs like Hotspot (Section 3.2.12) in that it does not include an inter-
preter for the initial execution of program methods. Instead, it uses a JIT compiler called the baseline compiler, which simulates a Java interpreter with bytecode dispatch compiled out. It also has an optimizing JIT compiler. In this work we refer to the optimizing compiler simply as the JIT compiler because of traditional terminology, and we refer to the baseline compiler explicitly when necessary.

Of particular importance to our work in the Jikes RVM are both of the JIT compilers, which comprise the execution layer of the VM, and the internal representation of the program also called the class model. We discuss each of these parts in turn.

2.6.1 JIT Compilers

Both of the JIT compilers in the Jikes RVM work at a method-level of granularity. Methods are compiled by the baseline compiler first, just before they are executed for the first time, and later they may be optionally recompiled one or more times by the optimizing compiler.

The baseline compiler is designed to compile code quickly so that it can play the same role as the interpreter in traditional implementations like Hotspot. To achieve fast compilation speeds, the baseline compiler produces poor quality code.

The code produced by the baseline is native machine code which simulates the Java bytecode stack machine. The code produced actually simulates a Java interpreter that is implemented directly in machine code, except that the bytecode dispatch has been compiled out. In a traditional interpreter, each bytecode must be dispatched to a machine code vector (typically) which implements that bytecode, but in the baseline compiler approach this dispatch is removed by just appending the machine code vectors of each bytecode one after another.

This type of implementation has two opposing primary consequences: compiling the methods this way leads to a significant amount of bloat due to the code size blow up from bytecode, a concise representation, to machine code, but the resulting code runs more quickly than a pure interpreter would run because there is virtually no dispatch overhead.

The optimizing JIT compiler in the Jikes RVM is part of a larger adaptive optimization system (AOS) [3] which includes profiling, generating optimization
plans, multiple levels of optimization, and on-stack replacement (OSR) compilation and specialization. Because profiling of running methods is expensive, it is usually accomplished by sampling the runtime at specific intervals rather than counting every event or execution. This method has the effect of making the AOS, and hence the optimizing JIT compiler, non-deterministic because optimization plans will vary between executions because of timing and sampling differences during execution. Furthermore, single methods can be optimized multiple times during the execution of a program as the measured profile of that method changes. However, re-compilation is driven primarily by time spent executing rather than more detailed profiling such as if-branch prediction; so a method is not likely to be re-compiled just because the code path taken during its execution has changed slightly. AOS non-determinism can present problems for measuring execution time during benchmarking. We deal with this issue in Chapter 6.

The optimizing JIT compiler [21] itself works in several stages of translation and optimization while compiling a method. It uses three levels of intermediate representation (IR): high level intermediate representation (HIR), low level intermediate representation (LIR), and machine level intermediate representation (MIR). Each of these representations has an optimization stage, and then a translation stage to the next level of IR, and each stage progressively lowers the level of abstraction of the representation closer to that of the native machine code of the host platform. Furthermore, the optimization phases of each level of IR implement optimizations that are most suited to that level of representation. MIR is the last representation before the translation to native code. The result of this final translation is the optimized native code emitted for the method being compiled.

The higher level IR used in the optimizing compiler assumes a virtual register machine model for execution. What this means is that they assume an infinite number of local registers that they can use for intermediate data value storage during calculations. So, many virtual registers are created and used only once, and the IR translator does not need to worry about re-using virtual registers because of finite resources. It is only at a later stage in the optimization process that virtual registers are mapped to actual machine registers.

The initial step of the optimizer is to translate the bytecode of the method into the HIR used by the optimizing compiler. This is the most involved step because the
semantics of the bytecode assume that it is running on a stack machine, whereas all of the optimizing compiler intermediate representations assume that they are running on a virtual register machine. However, it is during this phase of translation where new code generation or IR modifications can be most effective because any code produced during the translation from bytecode to HIR will also be amenable to later optimization by all optimization passes in the compiler, whereas code generation between, say, LIR and MIR will have skipped both optimizations passes in HIR and LIR. So, it is beneficial for modifications to compiler to be implemented at as high level as possible to benefit from as much of the optimizing framework as possible.

2.6.2 Internal Representations

The VM-internal representation of Java data structures is important both to startup performance, memory efficiency, and steady state performance. It is also important to understand how this representation effects and constrains VM aware AspectJ implementations.

Because Java does not require compile-time linking of references between classes, and classes which have not been explicitly required by a running program should not be initialized, the internal representation of the VM has two separate but related ways of referring to classes, methods and fields in Java. Specifically, for each reference to one of these there is a concrete object representation of it, and a reference object representation. So, for example, there are both RVMMethod objects, denoting concrete method representations in the VM, but also MethodReference objects which denote a reference to the former.

Reference objects are used to create a reference to a particular entity without forcing dynamic loading of that entity. They can be used at any time to force the resolution of the referenced entity to retrieve the concrete representation of that entity.

Classes are represented straightforwardly as objects of the RVMClass which contain fields containing that class’ attributes. These fields include arrays of RVMMethod and RVMField objects which represent methods and fields respectively. The constant pool of the class is not decoded in the internal representation
of the class, instead it is stored as an int array in the RVMClass objects. The
constant pool is only decoded when required while processing other parts of the
class such as attributes or method bytecode.

Methods are represented primarily as a byte array of bytecode using the same
representation of bytecode as the JBC on-disk format. This byte array is accom-
panied by other supporting fields in the RVMMethod class for representing the
exception table of the method, the method’s attributes, and maximum stack height
and local variable count. Each method is additionally assigned a unique integer
identification number as it is created in the VM. This integer is unique across all
methods in all classes in the executing VM so that references to loaded methods
can be encoded efficiently in the VM.

Useful attributes from loaded Java class files are also preserved in the internal
representation of the classes, methods, and fields.

Although the Jikes RVM differs from other VMs in some cases, it is represen-
tative of modern JVMS and closely follows a modern VM architecture. Because it is
written in Java, and is designed to be used for research endeavours, it is a suitable
choice for our experiments.

2.7 Summary

In this chapter we introduced AOP, a programming technique that gives program-
mers a way of structuring their programs that orthogonal and complimentary to
traditional program decompositions such as procedural or OO. One instance of
AOP is pointcuts and advice type of language extensions found in languages like
AspectJ. These two constructs together allows the modularization of concerns that
may not have been well modularized using only a traditional decomposition.

Furthermore, we introduced the architecture of modern JVMS. We covered
detailed information about the Jikes RVM which is used in most of the implemen-
tation work discussed in this dissertation.
Chapter 3

Design Space

In the first part of this chapter we cover the design space available to AspectJ implementations. Each design is detailed for its possible advantages and disadvantages, as well as including the practical consequences of adoption of that design if possible.

The second half this chapter discusses the related work in this area, and where possible characterizes the related work in terms of the AspectJ design space presented. This characterization is important in identifying AspectJ implementation possibilities that have yet to be explored.

3.1 Implementation of AspectJ

Advice weaving is the term broadly used to describe the coordination of program join points and the execution of advice bodies that may apply to those join points. In this section we discuss the different approaches to advice weaving, how they may be implemented, and their potential trade-offs.

Advice weaving can be broken up into three broad phases: source compilation, advice lookup and advice invocation. Source compilation is the phase of offline compilation which performs type checking, name binding, and translation from the source language, AspectJ, into the target language, the Java class file format.

The other two parts of advice weaving, lookup and invocation, are conceptually similar to method dispatch in a polymorphic object-oriented language: lookup is
required to identify which, if any, method body or advice body must be executed, and invocation is the mechanism for bringing about that execution. These terms are less clear when considering that parts of both phases are amenable to partial evaluation in a compiler, or implemented entirely dynamically in an interpreter.

For example, it is typical to compile out much of the virtual method dispatch in C++ by looking up target methods in tables called vtables [25, 28]. This reduces method lookup costs to a single indirection. Method invocation in C++ then is implemented the same as standard method invocation once the target method has been looked up.

But, in AspectJ, advice lookup is a more complicated operation than C++ virtual method lookup because AspectJ has a declarative pointcut language which describes which advice apply to a specific join point, and many advice can apply to one join point. Furthermore, join points are runtime constructs that are not accessible statically. This means that in some cases, there will be residual advice lookup at runtime to determine applicable advice even with aggressive partial evaluation [8].

Invocation in AspectJ is the execution of the advice body. In simple advice types, such as before advice, this execution is as straightforward in both a static and dynamic approaches as virtual method dispatch. However, around advice is significantly more complex because it generally requires that the enclosed join point be encapsulated as a closure, with the capability of being executed one or more times or even not executed at all.

Most implementations use some sort of partial evaluation to implement advice weaving to avoid unnecessary overhead at runtime, however it is not possible to implement all cases of AspectJ advice weaving with no residual lookup and invocation code because of the possible necessity of runtime type checking. We use the terms “lookup” and “invocation” to refer to both the points where static partial evaluation occurs, as well as the runtime residual code completing the dispatch. Unfortunately, there is no good term that captures both static and dynamic dispatch cleanly.

Note also that static advice weaving does not necessarily need to occur offline. Static advice weaving and partial evaluation can occur at runtime in the VM’s JIT compilers or by instrumenting the internal representation of the program code di-
rectly. Static weaving can occur anytime there is code generation.

Furthermore, these phases are largely independent in that they do not need to be performed at the same time. The main restriction being that lookup must be performed before invocation to identify which advice are to be invoked, and source compilation must occur in the frontend compiler because JVMs are unable to load programs directly from the source language. Although in practice, lookup and invocation are tightly coupled to share work between the two phases.

The design space of AspectJ implementations is a partial product between these three phases of advice weaving and the times during the staged JVM architecture that each can occur. We can use the VM architecture diagram shown in Figure 2.4 to identify the subsystems of the VM where advice weaving can reasonably take place. These subsystems can also be broken down into two basic categories: VM-external systems, where the implementation of advice weaving requires no VM modification or support; and VM-internal systems, where advice weaving is integrated in some way into the VM itself.

We go through each of these subsystems in turn in the next few sections and describe how each of the three phases of advice weaving may be implemented in those subsystems. We further outline the major advantages and disadvantages of using each subsystem for advice weaving, and the impacts a particular design would have.

3.1.1 Frontend (Offline) Compiler

Frontend compiler based weaving performs all three stages of advice weaving either during standard Java compilation or in a post compilation phase that operates directly on generated JBC. Because output of this stage must conform to the runtime interface of the JVM, advice weaving is constrained by the semantics of the JBC and the application programming interfaces (APIS) provided by the JVM. Some specialized operations typical to AspectJ programs do not have a concise semantic representation in JBC, so their implementation in bytecode can be inefficient.

Source compilation in this approach compiles aspects to standard Java classes, and advice bodies to standard Java methods named appropriately\(^1\).

\(^1\)Advice bodies are unnamed in AspectJ.
Advice lookup and invocation is handled either during source compilation or by a post-compilation step that modifies the Java class files emitted by a standard Java compiler. These files are instrumented with additional residual code to effect advice dispatch at runtime.

Because AspectJ’s dynamic join point model specifies join points as a runtime construct, offline weavers do not have direct access to join points themselves. However, each join point has a corresponding location in the source code which gives rise to that join point during program execution. For example, the invokevirtual bytecode gives rise to method call join points. These static locations in the source code of the program are called join point shadows.

So, advice lookup, in offline weavers, operates over the program’s join point shadows. And, because a join point shadow has only a subset of the properties of the join point it gives rise to, offline advice lookup can produce three possible answers for a particular shadow and pointcut: static match, possible match (requiring dynamic information), and static non-match. A static match can munge the code around the join point shadow to invoke the advice body unconditionally at runtime (using appropriate JBC, such as invokevirtual); a possible match will require that the call to the advice body be guarded by dynamic checks that complete the dynamic portion of the pointcut match; and static non-match requires no mutation of the code.

Additionally, since all advice execute in the context of an instance of an aspect, each call to an advice body must be proceeded by code which retrieves the proper aspect instance. In the issingleton aspect instantiation model (the common case), this results in a static method call or field reference to retrieve the aspect instance. However, perthis, pertarget and percflow have more complicated semantics and hence require more work.

Finally, because some of the AspectJ constructs have complicated semantics, it is useful for offline weavers to target a small runtime support library. This library can contain routines for cflow pointcut accounting, such as stack or counter based tracking, or reflection APIs which give the programmer access to dynamic join point information directly in the body of advice.

All aspects to be applied in a program must be known to the weaver before it begins to weave any class files. This requirement comes from the fact that advice
create a reverse dependency between the advice and the class which contains a matching join point shadow. So, when compiling a class, it is not explicit in that class which aspects could affect any of the join point shadows it contains because it is the aspect which specifies this dependency, so we must know of all configured aspects before weaving it.

The primary advantage of doing offline weaving comes from partially evaluating all advice lookup and invocation before the program is run. The overhead caused by weaving is only spent once, and doesn’t incur a penalty on the running program. So, more time can be spent offline to perform static analysis that improves the efficiency of the woven JBC such as around and cflow optimizations [1].

The primary disadvantage of offline weaving is that, as mentioned, the implementation is constrained to the semantics of the target language: JBC. So, without additional meta data, the VM cannot fully optimize code fragments that correspond to specialized AspectJ language constructs. These could be optimized more effectively if the semantics of those constructs were known rather than being lost in the translation to JBC. Optimizations of this type have been discovered as in [8, 16, 34, 35].

Offline weaving has other disadvantages also. It slows down the compile phase of the edit-compile-debug cycle because even simple changes to an aspect can have wide-ranging affects on the program causing large portions of the program to be recompiled. Additionally, because the weaver needs access to the class files for the program, it is unable to affect join points whose shadow’s are contained in classes that are not available to the programmer. These files tend to be system libraries, or digitally signed libraries that cannot be changed and then used.

3.1.2 Class Loader (Load Time)

Load time weaving in the class loader is similar to offline weaving, except that all of the weaving operations are performed as classes are loaded into the VM.

A load time weaver can be external or internal to the VM itself. External load time weaving is accomplished in the JVM by extending the class loading interfaces provided by the JVM, and using a Java agent to manage how classes are loaded.
In this way, the load time weaver intercepts classes before they are loaded into the VM and performs the same weaving operations an offline weaver performs on those classes to transform them. So, the load time weaver takes an unmodified class file as input, and produces a woven class file as output which is then loaded directly by the JVM’s class loading infrastructure.

This extension is possible because of the APIs provided by the JVM specification. However, this application programming interface (API) does not give access to the internal class structure of the JVM, and so an external load time weaver must replicate the entire class hierarchy itself to perform advice lookup because pointcut matching requires access to the type hierarchy. This replication costs both in memory and CPU usage during startup and class loading of the program.

An internal class loader works in a similar manner, but it is integrated into the JVM itself, bypassing the APIs discussed above. This tighter integration allows the weaver full access to the JVM’s internal structures, and hence it does not need to replicate the class hierarchy to perform advice lookup. Furthermore, an internal class loader has more opportunities for optimizing the load time weaving process because it has more control over class loading, and more knowledge of internal data structures, than an external load time weaver has using the available interfaces.

The use of a load time weaver avoids some of the problems inherent to offline weavers. Namely, weaving is postponed to load time, which means that small changes to aspects require only those specific aspects to be recompiled rather than large portions of the program which those aspects affect. This reduces the length of time spent in the compile phase of the edit-compile-debug cycle, so that only parts of the program that are being used to debug it are woven. Note that this saves time only during the edit-compile-debug cycle because incremental compilation of the program does not require the whole program to be recompiled, and as long as the debug part of the cycle does not require the whole program to be loaded.

Furthermore, the late-binding of aspects allows the same program to be deployed with different aspects each time it is run. Since one use-case for aspects is in the configuration management of large programs [76, 78], load time binding of aspects allows these programs to be configured at startup time rather than at compile time. This ability allows for more flexible deployment options for programmers and users.
Note, however, that a load time weaver still needs some support from the front end static compiler. Since the static compiler will be compiling the aspects themselves, there needs to be a method of communicating aspects, pointcut and advice information from the front end compiler to the load time weaver.

The primary down side of load time weaving is that it pushes some of the cost of weaving into the startup time of the program: Load time weaving performs essentially the same weaving operations as an offline compiler, but the offline compiler performs these pre-runtime. A main difference is that load time weaving only weaves classes that are used (and hence loaded) during a particular run of the program; the offline compiler must weave every class in the program.

This trade-off which contrasts offline and load time weavers is one in which the ideal choice changes depending on the situation. In a development setting, it is more useful to use a load time weaver to address the inefficient compilation times imposed by an offline compiler. A user-configurable system would also benefit from a load time weaving approach to allow the aspect configuration of the system to change between program invocations. Whereas a production build of a statically configured system would be more efficient if it were compiled and woven entirely offline. Ideally, this choice could be made by the developer depending on the design and use of the program.

3.1.3 VM Class Model

Both the lookup and invocation phases of advice weaving can also be done during the running of the program using techniques based on JBC weaving. This type of weaving is done by operating on the VM’s internal representation of the Java classes. Because VMs typically perform multiple stages of interpretation and compilation, the VM maintains the bytecode of the Java classes in its internal class model. During the running of the VM, the bytecode can be examined to find join point shadows that match any pointcuts in the system, and the bytecode can then be munged to effect invocation similar to how a front end compiler munges it.

It is interesting to note that after advice weaving, this method, as well as the above two systems of weaving have the same final result after lookup and invocation have been performed. That is, they all result in the VM’s internal representation
of the Java classes containing instrumented bytecode. This effect occurs because all three of these operate on the bytecode of the program, and weaving is implemented outside the execution systems (interpreter and JIT compiler) of the VM. Furthermore, any local optimizations available to one implementation are also available in another: there is no inherent reason why all three cannot produce exactly the same bytecode, and hence exhibit the same generated code quality, or steady state performance.

However, because class model weaving is performed during runtime of the program, it has several properties that distinguish it from the previous implementations. Firstly, runtime weaving is not a stage in the class loading and execution pipeline. So, there needs to be a trigger in another system of the VM to cause advice weaving to occur. One way of triggering advice weaving is the first execution of a method: just before the method is executed initially, advice lookup and invocation is performed on the entire method. This approach also allows the implementation of AOP languages that support dynamic deployment and undeployment of aspects, such as in CaesarJ [62]. In this case, the trigger that causes advice weaving is programmatic and controlled by the programmer.

Secondly, when weaving offline or during load time it is not possible, except through idiomatic programming, to perform dynamic deployment of aspects. So it is not possible to deploy aspects that were unavailable at the beginning of either of those stages. This restriction is because offline compilation and load time occur only once during the compile and execution pipeline of the Java architecture, while runtime class model based weaving can occur at anytime during the program’s execution.

This ability of a runtime weaver allows a new range of AOP dynamic language semantics. However, in this work we are focused on the implementation of the well-defined non-dynamic AspectJ AOP language.

Additionally, since a runtime weaver is integrated with the VM, it doesn’t suffer from the disadvantages of being constrained by advice invocation in JBC semantics because it represents and manipulates AspectJ constructs as runtime objects. This allows the implementation to optimize AspectJ-specific constructs when appropriate.

However, runtime bytecode weaving has significant drawbacks, both inherent
and with respect to implementation in existing modern Java VMs.

Bytecode munging is an expensive way to provide runtime addition of logic that performs advice dispatch. Since the JBC specification and JVMS were designed with the assumption that bytecode wouldn’t change during runtime, some information regarding each method is pre-computed and stored in supporting tables that the bytecode is closely coupled to. This type of information ranges from pre-computing the maximum Java stack depth that a given method uses, to storing the method and field names of referenced members in an indexed lookup table. Because both the interpreter and JIT compiler depend on this information being correct, they must be modified to be able to deal with modified bytecode instructions that do not conform to the expected protocol, or these additional lookup structures have to be modified in parallel to the bytecode to maintain consistency. The second approach results in a higher runtime overhead.

Furthermore, ITDS cannot be implemented at runtime without significant overhead or changes to many VM systems. ITDS can require changes to the class schema of already created objects, and changing the schema at runtime would violate stack and memory model invariants which are depended upon by the GC.

However, aside from these differences, the main conceptual difference between weaving in the class loader versus weaving in the class model is whether weaving occurs eagerly (an entire class at a time, during load time), or lazily (one method at a time, when triggered, in the class model).

This difference can also be viewed as the granularity that the weaving occurs at: any specific granularity must have the entire unit woven before its first use. So, front end compilation weaves at program level granularity, where the entire program must be woven before it is used; load time weaving weaves at a class level of granularity, where an entire class must be woven before it is used; and finally, class model weaving weaves at a method level of granularity.

### 3.1.4 Interpreter

Most modern JVMS contain an interpreter which processes the bytecode of the Java program and dispatches execution of the program following these bytecodes. Interpreters are typically included to improve the startup performance of the program
This balance is achieved because an interpreter can execute many different, infrequently executed methods quickly during startup, while relying on a dynamic optimizing compiler to ensure that steady state, or long running performance is still achieved [27, 46].

Both the lookup and invocation phases of advice weaving can be done during interpretation of the program. Lookup at this stage requires no additional information to determine matches, because the interpreter has reified representations of the join point being executed, rather than its join point shadow in bytecode. So, all necessary information is available directly at the point in time the lookup takes place to definitely match or fail to match any pointcut.

If a pointcut is determined to match the currently executing join point, the interpreter arranges to weave the execution of the advice appropriately before, after, or around the DJP execution itself.

This type of advice weaving has similar properties to traditional Java interpretation: startup time of the program is quick, because there is no eager advice lookup and invocation steps performed as these steps are performed during execution. However, the same drawback applies, in that frequently executed methods will run more slowly than they would have if they had been bytecode-woven and compiled because of the repeated overhead of advice lookup and invocation.

Furthermore, this type of advice weaving requires that the interpreter perform advice lookup on every join point execution in the program. Since it is likely that the majority of join point executions have no applicable advice, this could be a major source of overhead. At a minimum, interpreter weaving requires at least one conditional branch for each join point execution. Any pre-processing of pointcuts to improve lookup speed for the interpreter to get to this best case could have large improvements on the execution performance of the program.

On the other hand, modern JVMs have evolved to use a staged execution architecture that includes an interpreter for initial execution. The reason that interpreters are viable in this architecture, despite their relatively poor performance, is that if any method is executed for even a moderate amount of time, it can be recompiled.

Modern optimizing compilers use code stack and program counter sampling to estimate the amount of time spent executing a method rather than counting how many times a method is executed to determine if it should be optimized. This is not only a more accurate measurement for the cost
using the JIT optimizing compiler. Because of this, slower execution times in the interpreter are acceptable because it strikes a good balance between startup and steady state performance [3,46].

3.1.5 Just-in-Time Compiler

JIT compiler based advice lookup and invocation is handled in a similar fashion to how it is handled in the offline compiler. The primary difference is that offline weaving performs lookup on bytecode, and targets bytecode for advice invocation, while weaving in the JIT compiler performs lookup on bytecode and targets native machine code, or an IR language, for advice invocation.

Targeting the native instruction set gives the JIT compiler some advantages when performing advice invocation. One such advantage is that it no longer has to maintain the consistency between bytecode and parallel data structures, like the constant pool, which frees it from a possible source of overhead. The other primary advantage is that since the compiler is targeting native code, it is no longer constrained by the semantics of bytecode. Having full access to the machine code enables the JIT compiler to optimize the code implementing some of the AspectJ language semantics that is not possible to do with the constrained bytecode semantics.

Additionally, like with bytecode based lookup, the majority of the advice lookup code is compiled away so that it need only be performed once at JIT time. The only portions of advice lookup that need to be performed at runtime are those parts which require dynamic information available at only the join point rather than at the join point shadow.

Like interpreter based weaving, JIT weaving follows the spirit of the staged execution architecture of modern Java VMS.

3.1.6 Hybrid Approaches

Hybrid approaches combine any of the above approaches to do either advice lookup or advice invocation in multiple subsystems of the JVM.

and benefit analysis of optimization, but it also allows methods which execute for a long time, but are only called once to be optimized via on-stack replacement.
For example, if an implementation followed the spirit of the typical JVM architecture, it could perform advice lookup and invocation in both the interpreter and the JIT compiler. During startup, advice weaving in such an implementation would be handled by the interpreter, which executes the code more slowly but avoids the startup overhead of optimizing compilation and full weaving. Hot methods in a longer running program would be recompiled by the JIT compiler. The JIT weaver would then generate native code for the advice invocation, and that method would no longer have the performance degradation associated with dynamic advice dispatch in the interpreter.

Additionally, since the interpreter has already executed at least part of the method being compiled, it can leave behind profiling information discovered during execution that the JIT compiler can use to improve both its compilation overhead and its overall optimization of the method. Since lookup has to be handled by both interpreter and compiler, it is likely that the lookup results during interpretation of the method can be used to speed advice lookup during JIT compilation.

Other hybrid approaches are also possible. Offline weaving combined with online approaches could be used to perform some expensive optimizations in the front end compiler, while leaving some aspects to be woven by a load time weaver that preserves some user-configurability.

### 3.1.7 Other VM Subsystems

There are other systems in the VM discussed in Section 2.4 that we have not explicitly covered here, namely the garbage collector and stack and memory models. These systems are not appropriate for implementing advice weaving. They provide functionality for other features of the programming language, and hence need to be considered when integrating advice weaving into the VM, but they are not places that advice weaving can take place.

They do impose constraints on AspectJ implementations, however. For example, implementations in both the interpreter and JIT compiler must be careful to not violate stack invariants imposed by the VM’s GC when implementing advice invocation.
Table 3.1: Related Work Coverage of Design Space

### 3.2 Related Work

In this section we discuss other work in this area. Where possible, we classify each AOP implementation in terms of our categorization of the design space to better highlight areas not previously addressed. Each AOP implementation is summarized in Table 3.1 on page 43 based upon where that implementation performs different phases of advice weaving.

#### 3.2.1 AJC

ajc is the standard AspectJ weaver. Early versions implemented advice weaving as a pre-processor on the Java source language by inserting Java statements which correspond to advice execution directly into the source code.

However, more recent versions of the compiler work as a processing step on the Java class files and JBC directly. Because JBC based weaving is more flexible than source code based weaving, the compiler can operate in a number of different
modes: it can operate during the compilation of a program (static); it can operate
as a post-processing step on already generated class files (static); and it can
operate as a pre-load time step where the weaver is integrated into the loading
infrastructure external to the VM (loadtime). In this way, the weaver performs its
functions as classes are loaded into the VM.

Because of this implementation, it has less restrictions on its ability to do weav-
ing on code for which the original source code is not available. So, it can perform
JBC based weaving on some third party libraries for which only the jar files are
available.

The primary advantage of this type of weaver is that in its most common mode
of operation, all steps of the weaving process are done offline, as discussed in Sec-
tion 3.1.1. Briefly, this implementation compiles out as much statically provable
information as possible. So, the cost of the weaving is primarily born by offline
processes not affecting the runtime performance of the program. But, as men-
tioned, it cannot take advantage of VM-specific optimizations because there is no
integration with the VM at all.

ajc also requires the use of a small runtime library for runtime management and
accounting of some of the more complex AspectJ constructs. For example, much of
the implementation of the cflow pointcut infrastructure is provided through calling
into the AspectJ runtime library.

3.2.2 Aspect Werkz

AspectWerkz\textsuperscript{4}[18] implements an AspectJ-like language on top of traditional Java
using a separate Extensible Markup Language (XML) file or embedded annotations
to specify advice and pointcuts instead of new language syntax. AspectWerkz, like
ajc, can use a static or load time weaving approach, based on bytecode modifi-
cation, together with a runtime library. It also implements dynamic weaving by
hooking into VMS that support Java Hotswap, so that weaving can occur at run-
time. In most other ways, AspectWerkz functions similarly to ajc and other offline
weavers. So, because AspectWerkz is not integrated into the JVM, it is unable to
take advantage of optimizations available to VM JIT compilers.

\textsuperscript{4}Note that circa January 2005, the ajc and AspectWerkz projects merged. Here, we refer to the
most recent version of AspectWerkz before the merge.
Its primary differences with ajc come from its ability to do dynamic weaving and its source language implementation which allows the use of XML and annotations to denote aspects, advice, and pointcuts. However, because these two projects have merged, much of the functionality that used to be unique to AspectWerkz is now available in ajc itself.

3.2.3 ABC

abc [1, 7, 8] is another implementation that uses the offline compiler portion of the architecture space to facilitate advice weaving. abc was meant to be an alternate compiler to ajc which focuses on optimizing the woven code. Since it is also not integrated with the VM, it benefits and suffers from the same broad advantages and disadvantages of ajc with respect to VM integration.

However, abc demonstrated that many AspectJ optimizations can, and perhaps should, be implemented in the front end compiler. It supports complex whole-program analysis which can significantly optimize some AspectJ constructs such as around advice and the cflow pointcut [8]. Additionally, it performs many special and common case optimizations that ajc does not perform. Many of these optimize specific use-cases of AspectJ language features without slowing down the general case implementation. For example, many AspectJ programs are single threaded, so it is a simple and useful optimization to store cflow accounting information in static variables for the main thread only. This optimization greatly speeds up cflow implementation for the main execution thread, without sacrificing the performance of other threads in a general implementation.

The abc compiler is based upon an open, research-oriented compiler framework which further allows researchers to more easily implement experimental language features [7] by making it very simple to add new types of join points and pointcuts to the language.

3.2.4 JAsCo

JAsCo [72, 74] operates in a manner conceptually similar to AspectWerkz. It uses Java compliant interfaces to control aspect deployment and weaving from outside the VM itself. And so, it is based on bytecode weaving, and hence suffers from the
same semantic constraints that other bytecode weavers suffer from.

However, unlike AspectWerkz, JAsCo uses a hook based approach for most of its weaving: it instruments method bytecode to make callbacks into the JAsCo infrastructure which perform pointcut matching to determine applicable advice at runtime. However, it also has the ability to do static bytecode based weaving which inlines the advice lookup and dispatch instead of using hooks. This dichotomy is similar to the interpreter/JIT compiler dichotomy inside the JVM. Advice is first executed through hooks. But, if that advice is executed a sufficient number of times, JAsCo exploits the Hotswap capabilities of the JVM to replace the hook-instrumented code with fully inlined advice dispatch.

Using the Hotswap API also allows the JAsCo implementation to be fully dynamic: it allows the runtime deployment and undeployment of aspects and their advice. So, it implements capabilities that are not present in AspectJ.

3.2.5 PROSE

PROSE [65] was one of the first implementations which integrated AOP functionality into the VM itself. PROSE does not implement AspectJ specifically, but it implements an AOP language which supports much of the functionality of AspectJ’s dynamic JPM. It does not support the use of inter-type declaration (ITD)s.

The earliest versions of PROSE used the JVM debugging interface to add hooks into a running program so that it could interrupt execution at specific events (dynamic join points) and execute any applicable advice.

However, more recent versions integrated the implementation directly into the VM execution layers, using the the Jikes RVM as its base VM. It supports two types of weaving: the first is a hooked based weaving approach in which hooks are added to the machine code generated by the Jikes RVM’s baseline JIT compiler. These hooks call back into the VM where PROSE implements a dynamic dispatch mechanism for advice lookup and invocation.

The second type of weaving that PROSE supports is runtime JBC based weaving on the class model. In this case, the JBC is instrumented to support advice lookup and invocation.

However, PROSE does not add any additional support for ITDs or AOP specific
optimizations. Its implementation was designed to address the needs of middleware applications under heavy runtime adaptation and customization, so it supports these two levels of weaving to effect runtime aspect deployment. So, its focus is on support for dynamic deployment and undeployment of aspects.

3.2.6 JRockit

JRockit [12] is a JVM implemented by BEA. It has proposed support for AOP at the VM level [75]. JRockit supports pointcut and advice like AOP languages with a subscription based model using a provided API. That is, the programmer registers for event notifications on allowable events (representing the available join points) and uses filters to use dynamic information to narrow down the class of events.

The JRockit model is flexible enough to implement most of AspectJ’s dynamic join point model, however some constructs, like cflow, must be implemented on top of their provided API. However, it additionally allows the ability to add and remove advice at runtime. However, the model does not allow the use of ITDS.

3.2.7 Steamloom

Steamloom [16, 44], like PROSE, integrates advice weaving into the VM. It also uses the Jikes RVM as its implementation base.

It implements a fully dynamic AOP language where the programmer has full control over aspect deployment. It does this by implementing a JBC based weaver on the VM’s class model. Although the implementation is not complete, a complete implementation would support a super-set of the AspectJ language because it would contain constructs that allow the programmer to deploy and undeploy aspects at runtime programatically. These dynamic deployment features are not available in AspectJ.

Furthermore, Steamloom adds some JIT support in the form of additional, internal-only, bytecodes. These bytecodes allow the bytecode-based weaver to communicate additional information to the JIT compiler to implement advice instance retrieval operations as well as optimize the cflow pointcut [17]. The necessity of these internal bytecodes to aid in the implementation of optimization demonstrates that the semantics of JBC are too limited to efficiently implement all
of AspectJ’s constructs.

Cflow optimization is achieved by instrumenting the JIT compiler to use an additional word of storage per thread to store cflow guard bits. Each guard bit in the word corresponds to a cflow boundary pointcut in the program. The bit encodes whether the owning thread is within the cflow of the boundary pointcut. This encoding allows thin guards [2] used in the JIT compiler to speculatively optimize cflow tests by assuming that there is no cflow match.

Steamloom requires that the VM perform all weaving operations. So, it is not compatible with a frontend compiler doing some weaving pre-runtime. And, because it implements a language in which the user has control over aspect deployment, it provides an API from which to control the weaver. However, this additional, non-optional API functionality changes the VM’s interface breaking compatibility.

Steamloom’s evaluation has focused on steady state performance of the implementation rather than whole performance and startup performance of the VM. Their class model runtime weaver is a heavy-handed implementation for initial simplicity, and hence it has not been optimized for startup.

3.2.8 Nu

Nu [29, 30] has the goal of improving the development efficiency of AspectJ-like languages by moving weaving further along the tool chain into the VM. Nu adds two new bytecodes implemented by the VM and targeted by frontend compilers to implement the semantics of AspectJ-like languages. This type of implementation requires that the interface between the frontend compiler and the VM changes to accommodate the non-standard additional bytecodes.

Nu implements this functionality in the Sun Hotspot VM through an implementation that supports only the interpreter portion of the VM. The implementation is similar to PROSE in that it is hook-based: each method invocation has additional assembly instructions added to it, before the interpreter dispatches the actual method invocation, to call back into the VM runtime. Logic in the runtime decides whether there are any applicable advice to execute and arranges execution if appropriate. A two-level caching strategy is implemented to improve the speed of this
The current implementation of Nu supports only method invocation join points. The addition of other join points would require the insertion of more hooks in the interpreter to catch their execution and force call backs into the VM runtime.

Nu’s evaluation has focused on micro-measuring the specific cost of advice weaving and dispatch operations as well as the specific costs related to their caching mechanisms. Because Nu does not yet integrate with the JIT compiler in the Hotspot VM it is not possible to perform useful macro measurements of the implementation.

### 3.2.9 Naseer’s Interpreter Dispatch

Naseer [63] created an interpreter dispatch based implementation of AspectJ based on the Jikes RVM. Naseer’s implementation loads the standard output of the ajc compiler configured to produce code for the ajc load time weaver. So, the implementation maintains compatibility with the ajc load time weaver. This approach simulates an interpreter in the Jikes RVM using the baseline compiler.

The implementation constructs an internal representation of pointcuts and advice together with a lookup table data structure to implement efficient pointcut matching. This method creates one table for each target type pattern appearing in any pointcuts in the program called type pattern tables (TPTS). Each type pattern table (TPT) contains pointcuts which could match a join point who’s target type matches the static target type pattern of the TPT. TPTs are additionally broken down into three pointcut lists sorted by the join point kind that they match: call, execution, or other.

As classes are loaded by the VM, each type name is used to match against the type patterns associated with each TPTS. Any matching TPTS are assembled into a list to be associated with the loaded class. The pointcuts reachable by traversing the TPTS in this list are all of the pointcuts that could potentially match a join point whose target is the class.

Because the target type of a join point is available immediately to an interpreter, and the only join points which do not have a target type are adviceexecution join points, dispatching on target type can retrieve associated TPTS efficiently. And
the tabular structure improves pointcut matching efficiency by reducing the number of pointcuts which must be consulted by the interpreter at every join point execution.

Like Nu, Naseer’s implementation contains a caching strategy, albeit a one-level caching strategy, which saves the results of matching per join point shadow for future join point executions.

3.2.10 ALIA

Other work by Bockisch et al. [15] presents an architecture that contains a metamodel which explicitly represents AOP language features as first class entities inside the VM. In this way, the architecture is designed to allow the decoupling of language feature implementation from the frontend compiler by allowing the compiler to target a code model that explicitly supports AOP constructs. This design is similar to the way that ajc implicitly determines a load time interface because of its implementation of a load time weaver, however the ALIA interface is much more general: it supports a super-set of AspectJ’s functionality, including the ability for the programmer to control aspect deployment at runtime. And so its implementation requires that it supports runtime advice weaving.

The meta-model proposes a hierarchy of classes and runtime objects which a frontend language compiler can target and use to implement the AOP-specific functionality required for that language. Because this meta-model interface is designed to support a wide range of AOP language implementations, not just AspectJ, it must be general enough to support features not present in AspectJ, or any other, AOP language.

So, although this language makes many AOP constructs first class entities inside the VM, and hence allows for some amount of optimization, there is a trade-off between the generality of the interface and how much optimization this generality allows. This trade-off has not yet been explored.

This work has a reference implementation that shows how the interface they propose can support the execution of AspectJ as the frontend AOP language. This reference implementation also includes an implementation of the meta-model which they propose in the Jikes RVM.
3.2.11 CaesarJ

CaesarJ [62] is an AOP language like AspectJ and is based on Java. It introduces support for separating aspect implementations into components which are reusable with different bindings.

Additionally, it includes a set of semantics that robustly deals with dynamic aspect deployment. Dynamic deployment is handled by scoping the deployment of an aspect with the dynamic extent of a normal Java block and the deploy keyword. This type of deployment scopes pointcut matching like a cflow pointcut because it constrains the deployment to the dynamic extent of the block in which the aspect is deployed, but also further restricts the applicability of deployed advice to the thread which entered the deploy block.

It supports AspectJ semantics by supporting static, unconditional deployment of aspects by adding the deploy keyword to a static field declaration which instantiates an aspect binding.

Scoping of aspect deployment in this way is important to maintain a coherent set of semantics for deploying an aspect atomically in an executing program which may have matching join points currently executing by some threads.

3.2.12 Virtual Machines

Interlisp-D

Interlisp-D [22] was the reimplementation of Altolisp (itself an implementation of the Interlisp language) on a machine architecture\footnote{The “D-machines”: Dorado and Dolphin.} that had more powerful support for the types of operations that Interlisp required. It mainly included the identification of parts of the Interlisp language which comprised its core functionality: the parts of the language for which the rest of the language could be implemented on top of. This core part of the language specification was known as the Interlisp-D Virtual Machine.

This VM was much more low level than modern Java VMs: it was primarily a thin layer of microcode on top of the D-machine’s native instruction set. The microcode primarily provided efficient implementation of Interlisp’s core set of sys-
tem requirements: memory allocation and garbage collection, low-level input/output, etc.

Modern Java VMs differ primarily in the higher level language that they implement. It is more divorced from the target hardware than an Interlisp-D VM implementation was.

**Smalltalk**

Smalltalk [27, 37] is a pure OO programming language developed at Xerox Palo Alto Research Center. Smalltalk contained early program visualization tools in the form of a development environment containing class library code browsers and editors. This language and environment combination was designed to enable interactive program development, and allow the interleaving of program editing and execution in a live program image.

It also included reflection mechanisms in terms of metaclasses in an effort to maintain the property that everything in the language is an object.

The implementation problems encountered by implementing Smalltalk (and Self Section 3.2.12) VMS are some core problems in VM development.

**Self**

Self [46, 47] is a pure OO programming language implementation closely following along the lines of the Smalltalk programming environment except that it is a prototype-based OO language. It was also designed to be an interactive programming environment.

However, this interactive programming environment posed problems that modern VMs have attempted to solve. They require a trade-off between low latency responsiveness to the programmer and efficient execution of long running programs. Low response latency means that long pauses caused by optimizing compilation are not acceptable, but removing the optimizing compiler means long running programs become inefficient.

Self-93 introduced steps towards solving this problem by adopting a compilation system that supported both of these modes: it could dynamically recompile the “hot spots” of an executing program. It used a fast baseline compiler to generate
initial code to reduce interactive latency problems due to slow compilation at the price of producing poor code. The “hot spots” of the program were then recompiled with the optimizing compiler to mitigate the poor code quality produced by the fast compiler.

This implementation was the basis for many modern Java VM implementations, which although not interactive in the same way, use this insight and accompanying analysis in making the trade-off between quick startup and the efficiency of long running programs.

We use the insights gained from this work to guide our implementation of AspectJ.

**Hotspot**

Hotspot is one of the two major modern Java VMs (the other being IBM’s J9 VM). It is the official reference implementation of the JVM specification produced by Sun Microsystems. Until recently, the source code of the Hotspot VM was only available under special academic licensing.

Hotspot is implemented in C/C++ and follows the architecture described earlier (Section 2.4). It implements an interpreter as well as two dynamic optimizing compilers, controlled by a dynamic profiling system, which are used to recompile methods at different optimization levels.

**J9**

J9 is the other major production Java VM produced by IBM. Its source code is not publicly available.

Like Hotspot, J9 uses the combination of an interpreter and an optimizing compiler controlled by an adaptive optimization system. It is also implemented in a combination of C/C++ and assembly code following the architecture described in Section 2.4. J9 supports many different configurations to support a variety of devices from handheld mobile devices to realtime systems.
**Jikes RVM**

The Jikes Research Virtual Machine (RVM) [3, 13, 19, 21] is an open source VM implemented almost entirely in Java. The only portions of the Jikes RVM that are not implemented in Java are the portions that bootstrap the VM which follow the protocol for binary execution of the host platform.

The Jikes RVM is geared toward researchers in VM technology, and so it supports a wide-range of different research, including research in garbage collection, JIT compilation, dynamic profiling, and optimization, as well as support analysis of object lifetimes and execution profiles of Java programs.

Finally, because the Jikes RVM is a Java-in-Java VM, it allows virtually any Java code to be used inside the VM. This fact, coupled with the fact that the VM is written in a high level language like Java allows programmers to implement and test out new research techniques quickly.

Because much of our work is based on the Jikes RVM, it is discussed in much more detail in Section 2.6.

### 3.3 Summary

Table 3.1 shows where in the design space of AOP implementations, as related to the VM architecture, each piece of related work falls.

The table reveals two large gaps of unexplored space. The first gap is the implementation of both advice lookup and invocation in the internal class loader. Although ajc has the ability to implement both of these phases in the external loader, this type of implementation poses performance problems as discussed in Section 3.1.2.

The other gap is advice lookup and invocation in the JIT compiler. Steamloom and PROSE both have JIT compiler support, but neither of these implementations perform lookup or weaving in this portion of the VM architecture.

Furthermore, it is interesting to note which of these implementations are actually partially hybrid implementations. Firstly, ajc’s compile-time and load-time approaches are compatible with each other if the compiler is configured correctly. So, ajc actually implements a hybrid approach where lookup and invocation are

---

5Naseer’s interpreter dispatch implementation.
available in both the frontend compiler and at load time in the external class loader. Secondly, both Steamloom and PROSE implement a partial hybrid implementation by having JIT support for advice invocation. Although neither of these approaches implement a full JIT weaver.

The last point to note is that no implementation uses a design which implements a hybrid approach combining both offline and online, VM-internal advice weavers. That is, no approach takes advantage of both the offline compiler and the VM internal systems in support of advice weaving. All VM-based approaches use the offline compiler as a source compiler only.
Chapter 4

Architecture

This chapter presents our primary design, which we call the AJVM\(^1\) of an AspectJ implementation using a hybrid architecture. Figure 4.1 shows an annotated version of the JVM architecture in Figure 2.4. Changes from the typical JVM architecture are highlighted in italics.

Although modern JVM implementations each make their own design decisions and have their own specific constraints, our architecture is based on modern JVM architectures in general and on the generic design space of AspectJ implementations, and so it is applicable in many JVM implementations. Our architecture offers a significant degree of flexibility to work around the specific design constraints of JVM implementations.

Our architecture centers around the on-disk format for compiled AspectJ programs. Many of the design decisions surrounding this on-disk format are left up to the implementer, although we discuss certain trade-offs here. For example, our architecture allows, but does not require, that an implementation have advice weaving support in the optimizing JIT compiler. Implementations exploiting this portion of the architecture can allow additional optimizations not available to those which do not.

We discuss each piece of the architecture in turn below.

\(^1\)We use AJVM in upper case to refer to the architecture specification, whereas our various implementations are written in lower case.
4.1 Front End Compiler

The front end compiler is responsible for translating the source code language AspectJ into a compatible class file format that can be read by an AJVM implementation. As discussed in Section 2.4 and Section 2.5, JvMs define a sophisticated on-disk format for Java classes. Our architecture uses compatible class file attributes to encode AspectJ specific information in the on-disk representation.

To enable the VM support and optimizations allowed by our architecture, an on-disk format must meet several requirements:

1. aspects must be represented by Java classes, and annotated as aspects;
2. advice bodies must be represented by Java methods, and annotated as advice;
3. aspect instance retrieval methods must be annotated;
4. pointcut definitions must be represented;
5. cflow accounting methods must be annotated; and
6. there must be sufficient data to determine which aspects have been statically woven, thereby making it possible to recover original, unwoven class files.

Any on-disk representation that meets these requirements could be implemented by an instance of our architecture and realize all of the benefits. However, different implementations would then not be compatible with each other, and further, we would not maintain backward compatibility with ajc. So, we use ajc’s implicit on-disk format as the concrete target format for our architecture. ajc’s on-disk format meets all of our identified requirements above with the additional benefit of backward compatibility.

We start by describing the different types of annotations supported by this format followed by the implications that these annotations have and how they are used in practice.

The on-disk format uses both class and method attributes to encode AspectJ semantics. Each of these sets of attributes is laid out below.

- **WeaverVersion** contains the version information of the compiler that the containing class was compiled with. Version information contains both a major and minor version number, where the major version indicates format incompatibility. This attribute tells the program reading the class file which format the rest of the attribute data is in for proper parsing.

- **Aspect** indicates that the containing class is an AspectJ aspect. It also contains the PerClause pointcut for aspects that use the perthis, pertarget, or percflow aspect instantiation models (see Section 2.3). This annotation satisfies requirement 1.

- **SourceContext** contains information about the source code file such as file name.

- **PointcutDeclaration** stores one named pointcut declaration per attribute. An aspect containing more than one named pointcut will encode these pointcuts one per PointcutDeclaration attribute in the class file. This annotation partially satisfies requirement 4 (it does not represent unnamed pointcuts).
- **WeaverState** stores state information about the amount of weaving the frontend compiler has already performed on the given class. For example, it includes notes about which aspects have been woven into the class, as well as information necessary to retrieve the unwoven bytes of the class file. This annotation satisfies requirement 6.

- **Advice** marks the annotated method as containing the body of an advice. The annotation also contains all other information necessarily associated with an advice and pointcut pair: the advice kind (before, after, around, etc), the actual pointcut, and extra information specific to around advice. This annotation satisfies requirement 2 and completes the satisfaction of requirement 4 together with the `PointcutDeclaration` above.

- **AjSynthetic** marks the annotated method as synthetic. A synthetic method does not give rise to any join points at runtime even though the implementation is functionally the same as a standard method call.

- **MethodDeclarationLineNumber** contains source code line number information about the annotated method.

Compilation of AspectJ into its on-disk code format is a combination of these Java attributes and traditional Java class file constructs.

Aspects are compiled into standard Java classes: their fields and standard methods are compiled to members of the class. Advice bodies are compiled to Java methods and, since advice are not named in AspectJ, the compiler generates unique names for them. As discussed above, an aspect is distinguished from a standard Java class by the `Aspect` annotation, and advice are distinguished from standard methods by the `Advice` attribute.

Furthermore, all class files representing aspects have at least two AspectJ-specific synthetic methods, and one synthetic field attached to them. Since aspect instantiation is not controlled by the programmer, the two static methods `aspectOf()` and `hasAspect()` are created by the front end compiler to retrieve (and create if necessary) an instance of an aspect and check if an aspect instance is available respectively. These methods take addition parameters if the aspect uses `perthis`, `pertarget`, or `percfow` instantiation models. These
two methods together with a synthetic static field in the aspect provide a full implement-
mentation of the instance management for the given aspect in plain Java. Note that
this implementation can be overridden by an aspect-aware VM as discussed below.
This collection of methods satisfies requirement 3.

Finally, the front end compiler also targets a small AspectJ runtime library that
is implementable in plain Java. This library contains routines that implement some
AspectJ mechanisms including cflow accounting, such as a counter or stack, and
join point reflection code which can expose properties of a matched join point
within an advice body. Note that here also, although the runtime library is imple-
mentable in plain Java, it can be overridden by a suitably aware VM, and this API
satisfies requirement 5.

4.2 Runtime Weaver

The runtime weaving is responsible for three main tasks, generally ordered:

1. arranging loading of necessary aspects by the VM;
2. reading AspectJ-specific attributes from loaded class files and annotate the
   internal representation appropriately; and
3. completing any weaving that was not done by the front end compiler.

The only part that the weaver is required to do is the third step: completing any
weaving that was not done by the frontend compiler. In most cases, however, it is
necessary to do at least some of the first two steps to accomplish the third.

Arranging for all necessary aspects to be loaded by the VM is a requirement par-
tially dictated by the AspectJ language semantics. Since AspectJ does not specify
any dynamic deployment or undeployment semantics, it is assumed that all appli-
cable aspects apply for the entire execution of the program. To implement this, the
runtime weaver must ensure that aspects are loaded before any of their advice may
apply. And, because the weaver cannot know which advice apply at a particular
join point before that advice is loaded, the weaver must typically load all necessary
advice before executing the main method of the program. The only case where this
Figure 4.2: Example aspect configuration file showing a configuration which requires that the runtime ensure the DisplayUpdating and AccessControl aspects are used in the program.

is not required is when the frontend compiler has statically determined where the aspect applies before execution begins.

Which aspects the weaver loads before the program is executed is specified by an aspect configuration file. This file can be automatically generated by the frontend compiler, or it can be generated or modified by hand to vary the aspect configuration of a program on a per-run basis. Aspects can even be encoded in the file itself. Figure 4.2 shows an example of such a definition file.

Reading the AspectJ-specific attributes stored in the class file is only required by the implementation to complete any weaving not done by the frontend compiler. To do this, it is not strictly required that the internal representation of loaded aspects and classes be annotated. It is only necessary that the weaver maintain enough data to properly weave newly loaded classes. Additional annotations provide the interpreter and JIT with additional semantic information about the program that allows them to more efficiently execute or compile the code.

Finally, the main purpose of the weaver is to complete any weaving that hasn’t already been done. This ability allows non-changing aspects to be woven at static compile time while maintaining the ability for a deployed system to have different runtime aspect configurations: each run of the program could specify a different set of aspect configurations, and the runtime weaver is responsible for the late-binding of the configured advice.

There is no restriction that the weaving happen while a class is loaded by the VM. To properly implement the semantics of AspectJ, it is only necessary that weaving occurs before or during the execution of the program’s join point shadows. In practice, however, this is constrained by the hosting VM because of its internal
representation of the program and the hooks it makes available. For example, it is unreasonable to perform bytecode weaving at an instruction level granularity in many VMS because modifying the instruction stream during execution would be prohibitively expensive.

So, for example, on one end of the spectrum is ajc’s load time weaver (see Section 3.2.1). It is implemented in plain Java using public, VM-external JVM APIs, and it conforms to the specification of the runtime weaver as laid out above. It ensures all necessary aspects are loaded before the program begins execution; it preserves AspectJ annotations to support late weaving (these annotations are not preserved into the VM); and any weaving not completed by the frontend compiler it finishes during class loading.

On the other hand, an implementation could combine a load time annotation parser to extract the AspectJ data with an interpreter and JIT based implementation to complete the weaving process. The interpreter would perform dynamic advice dispatch as join points are executed during startup, and any hot methods that are compiled by the JIT would use code generation based weaving.

Although not a focus of this work, this architecture also supports AspectJ’s static join point model, ITDS. This join point model allows the insertion of members into matched types. Typical implementations of ITDS will instrument on-disk class files similarly to how advice weaving occurs. A runtime implementation could instrument classes in the same way, during loading, or by changing the class schema in the VM’s internal representation.

However, in modern JVMS it is impractical to change the schema of a class that has already been used and has live objects because of memory allocation considerations. So, an efficient implementation of ITDS would instrument classes before their first instantiation in the program.

4.3 JIT Compiler

The JIT compiler is the only part of the architecture that is optional. An implementation can choose to include modifications to the standard JVM JIT which uses data preserved by the load time weaver to better optimize AspectJ code.

An implementation can use the JIT compiler in two ways. It can use it as a full
blown weaver, which statically weaves the entire method during code generation; or, it can leverage the existing optimizing framework present in the JVM by providing additional assertions about the code being optimized based on the AspectJ data preserved by the load time weaver.

Both of these uses allow the JIT compiler to optimize AspectJ code in ways that are not possible otherwise because an alternate implementation is constrained to the semantics of Java bytecode. And, although in most cases, Java bytecode semantics are sufficient for expressing AspectJ programs optimally, in some cases the bytecode is too general to optimize. Some of these cases are discussed in more detail in Section 4.4.2.

It is useful to note, however, that it is possible that weavers implemented entirely in the JIT compiler may be much more complicated than bytecode weavers. Part of the reason is that operating at a lower level of abstraction, but also, in VMS like the Jikes RVM, it is difficult to integrate with a dynamic optimization framework that is tightly coupled to the bytecode representation of the program.

### 4.4 Discussion

#### 4.4.1 Consequences

This architecture was designed to have several important consequences both for the ease of adoption of AspectJ, and aspect-aware VMS, and also for the efficient execution of AspectJ programs.

The two primary features of this architecture are that it allows late-binding of advice, and that a supporting implementation can be integrated into the VM, enabling AspectJ-specific optimizations unavailable to VM-external approaches.

There are two significant advantages to enabling late-binding of advice. The first is that it can prevent entire programs from being recompiled when there are local changes within an aspect. Recompiling large programs can take a significant amount of time which reduces the utility of unit test suites and fast edit-compile-debug cycles. Secondly, late-binding allows deployed systems to change their applicable aspects without being rebuilt (although they would require a restart). This allows users to configure their systems using aspects.
The architecture is both forward and backward compatible with existing JVMS, and the standard AspectJ compiler ajc. A legacy implementation using either compile time, post-compile time or load time weavers can still use a VM implementing this architecture. Because the AspectJ annotations are optional past the load time weaving stage, any JBC woven code implementing the semantics of AspectJ is still fully executable by this architecture, albeit, some JIT optimizations may not apply when they otherwise could.

Secondly, any frontend compiler implementation which conforms to the optional meta-data format can take advantage of any subset of the optimizations that are supported by the VM without sacrificing the ability to run on a standard JVM. Hence, modifying a front end AspectJ implementation to target this architecture, and take advantage of potential VM optimizations, does not preclude that implementation from also running on traditional Java VMS using a plain Java load time weaver (ala the ajc load time weaver).

The architecture supports both global and local offline optimizations, as well as global and local online optimizations independently and compatibly. Because the architecture explicitly permits offline weaving, it does not preclude the possibility of the offline global code analysis required to facilitate the cflow optimization in abc [8] (see also Section 3.2.3). In addition, that once this analysis has been performed, any remaining cflow accounting and execution routines in the code can still be annotated as such and so amenable to further optimization inside the VM. This is discussed further in Section 4.4.2.

Another smaller, but similar benefit, is that production builds of programs which require no late advice binding can perform all advice weaving statically, minimizing online weaving overhead but still enabling further online JIT optimization of the woven code.

4.4.2 Optimizations

Aspect Instance Retrieval

In Section 2.3.3 we discuss how AspectJ semantics dictate that all advice executions occur in the context of an instance of the enclosing aspect. The creation and
selection of which aspect instance is used is not directly controlled by the programmer, but instead specified by the instantiation and association clause of aspect declarations. The primary instantiation model is `issingleton`, however as discussed, more sophisticated models such as `perthis` and `pertarget` are also available.

Requiring an aspect instance for advice invocation is similar to how (non-static) method dispatch requires an object instance for methods to be dispatched against. However, method dispatch requires the object instance to be immediately available at the time of dispatch, and so the program necessarily has the object instance available at the time of the call. For advice dispatch the aspect instance must be retrieved before the advice is executed at a matching join point. This instance retrieval is a source of overhead for advice execution, especially when using a instantiation model other than `issingleton`.

Our architecture allows optimization of both implicit aspect instance retrievals at join points at which an advice applies and explicit instance retrievals where a programmer invokes the `aspectOf` method on an aspect directly. In both cases, meta-data provided by the frontend compiler and preserved into the VM by the runtime weaver identifies which methods correspond to aspect instance retrievals.

Since `issingleton` aspects have one instance, in a bytecode implementation this instance is typically stored in a static field in the aspect itself, which is then retrieved by the AspectJ-required `aspectOf` method when needed. This retrieval can be slightly optimized by an implementation with JIT support by identifying the `aspectOf` call and replacing it with a direct reference to the static field of the aspect which holds the aspect instance. This optimization works in concert with the frontend compiler which has compiled the aspect to a Java class which contains a field to store this instance, and it has generated the static initialization methods in the aspect to create the instance at load time.

Furthermore, because `issingleton` aspects are created once and are in use for the entire run of the program, it is a guarantee at runtime that the aspect instance field is not null. So, this optimization can include hints to the VM to remove redundant null checks.

Bytecode based implementations of the `perthis` and `pertarget` (collectively called `perobject`) models require either the use of hash maps, or modify-
ing the participating objects with additional interfaces and fields.

A hash map solution uses the participating object as a key and the correspond-
ing aspect instance as the value in the map. Before each execution of advice in
that aspect, the appropriate instance is then retrieved using the this or target
object at the join point.

An alternate solution is to store the aspect instance directly in the participating
object by modifying its class schema to keep a private field for aspect instances.
Aspect instances are then retrieved by invoking the static aspectOf method on
the aspect with the relevant object passed as a parameter. Because the object im-
plements an aspect instance retrieval interface, the aspectOf method invokes an
interface method on the object to retrieve the aspect stored in that object. This
second solution is the one used by ajc.

Our architecture allows the optimization of these instance retrievals by avoid-
ing method and interface calls entirely and replacing them by a getfield in-
struction. This optimization isn’t available outside of the VM because of visibility
restrictions in the JBC, and because of the heterogeneity of objects that advice in
one aspect may apply to. Note that, however, this optimization also works in con-
cert with the frontend compiler and the runtime weaver parts of the architecture by
relying on the appropriate aspect instance fields being available.

Although this approach depends on the use of specific fields chosen because
of ajc’s implementation, our architecture and meta-data could also support other
approaches to aspect instance storage such as aspect instance tables [43]. Furth-
more, more complicated instance management could also be used, such as storing
instances of singleton aspects in immortal GC space [13, 34]. The use of these
other optimization approaches only depends on the coordination of different parts
of one implementation of our architecture rather than the structure of our architec-
ture itself.

Advice Invocation

Although many bytecode based implementations implement advice invocation us-
ing a JBC invokevirtual instruction, there is another difference between a
normal method call and an advice dispatch. In the case of advice dispatch, not
only is the aspect instance not immediately available, but it is the presence of a concrete aspect plus advice and pointcut that caused the advice invocation. In other words, because the aspect is the cause of the invocation, we know the exact type of the aspect acting as the target of the invokevirtual. This is in contrast to a normal method call in which the target of the invokevirtual could vary at runtime due to subclass relationships.

Using this information at the site of the advice dispatch, an implementation can provide an additional hint to the JIT compiler’s optimization engine that the target aspect of the advice call is an exact type and that the JIT need not consider the case of having to invoke the method indirectly through a method dispatch table. Furthermore, it allows the optimizer to inline the advice body into the calling method without a guard protecting against incorrect method dispatch or dynamic class loading that is typically required when assuming an exact type and generating specialized code.

However, some JIT compilers have the ability to deduce the same information that our architecture provides for this optimization in some cases. One way a compiler can deduce this information is to perform OSR [32] of the executing method. A method compiled with OSR has more guarantees regarding the exact type and location of objects because it has been compiled specifically for the current execution of the method and will not be used for subsequent executions.

So, although this optimization may enable the VM to optimize advice dispatch more quickly, the same information can be derived in the right context by a sophisticated JIT compiler and adaptive compilation system. In particular, the IBM J9 VM supports a particularly comprehensive set of optimizations capable of deriving these annotations independently in many cases.

CFlow

This architecture also allows the implementation of previously discovered optimizations, such as the VM supported cflow optimization by Bockisch et al. [17]. This optimization takes advantage of the insight that because of the thread-based nature of the cflow pointcut, it is much more efficient for the VM to support it specifically than to implement thread-based cflow accounting in JBC such as in ajc.
Part of our architecture specification includes identifying specific calls for cflow accounting routines which can be implemented in JBC but are identified by metadata annotations from the frontend compiler. These routines implement cflow counter and stack classes managed on a per-thread basis which help to implement the semantics of the pointcut. An implementation of our architecture is able to take advantage of this fact by creating VM-internal classes which implement the runtime framework classes depended upon by the program. These classes implement the same functionality inside the VM, and would be used in place of the JBC-based implementation if available.

The VM-internal implementation of these methods has full access to infrastructure unavailable to external libraries, so they can implement the functionality more efficiently. More specifically, these methods can access fields of internal VM thread objects directly which can store references to the cflow counters and stacks associated with that thread. It is a quick operation to lookup the currently executing thread, and hence the cflow counters and stacks that need to be modified and consulted. This lookup efficiency affects the performance in crossing cflow boundaries, testing cflow conditions, and extracting values matched in a cflow pointcut.

Furthermore, our architecture can support the additional micro optimizations proposed by Bockisch et al. by integrating support for cflow boundary and testing into the JIT.

Finally, the architecture has the additional benefit of supporting known cflow optimizations in the frontend compiler. For example, abc implements much of its cflow logic in a runtime framework (although it is not compatible with ajc's framework, it could be made compatible). However, abc also implements more sophisticated control flow analysis for optimizing its cflow implementation that allows it to remove some of the cflow boundary and cflow tests from the code altogether. The checks and boundary updates that cannot be eliminated must still be executed by the VM, and an implementation of this architecture could optimize the remaining checks as discussed above. By coupling optimizations implemented statically by the frontend compiler and runtime optimizations in the VM, our architecture combines the benefits of both.
4.4.3 Language and VM Applicability

Our architecture is designed to support implementations of the AspectJ language. Because of this, it makes several assumptions about invariants of the language that constrain some portions of it to AspectJ. However, much of the benefits of the architecture are also realizable in other AOP languages.

For example, even though CaesarJ supports dynamic aspect deployment, it does have constructs for the static deployment of aspects. Any aspects that have been deployed statically would meet all of the assumptions that we make for AspectJ aspects, and hence would be supported fully by our architecture.

However, dynamic or programatic deployment of aspects requires more much accounting than a static aspect implementation like AspectJ.

Additionally, our steady state optimizations are not AspectJ-specific. Any AOP pointcuts and advice language which requires that advice are executed in the context of an aspect instance could benefit from our instance retrieval and aspect type hint optimizations.

Our architecture is based on modern JVM architectures and so it is applicable in many JVM implementations to support AspectJ. However, our architecture should also be generalizable to AOP implementations in other languages which use virtual machine-like runtime environments. Our architecture could apply to these other VMs if they contained a similar overall architecture to a JVM. Namely, it requires a semantics similar to AspectJ pointcuts and advice, and a virtual machine containing infrastructure for dynamic code loading.

For example, an implementation of pointcuts and advice in languages like Python or Smalltalk could be implemented at the VM level. The applicability of our optimizations would depend on the exact semantics of the AOP addition to those languages. However, their implementations would also benefit from using our architecture to support bytecode rewriting at runtime, especially in VMs that contain only interpreters such as CPython, the primary Python implementation.

Furthermore, more generally, our architecture demonstrates that non-local language features that may have traditionally be implemented using source code or bytecode rewriting can be implemented efficiently in a runtime system by amortizing rewriting costs.
Chapter 5

Implementation

We present several implementations of our architecture designed to cover both a range of design decisions available to implementers of our architecture specifically, and to cover parts of the design space (Table 3.1) previously unexplored.

Because pieces of our architecture are modular, there is overlap between several of our implementations. Specifically, the frontend compiler is the same for all of our implementations, so this is discussed first in Section 5.2.

The focus of the rest of our implementation work has been in the Jikes RVM. We describe our VM-internal bytecode-based weaving implementations in Section 5.4. These implementations operate at different levels of granularity and use different rewriting implementations, but they all operate on the bytecode of the program either during class loading or during runtime. These implementations all share the same JIT compiler support for steady state optimization purposes.

We then present a second partial architecture implementation in the Jikes RVM that uses a code generation approach using the JIT compilers in Section 5.3. And finally, we present a partial architecture implementation in a second VM, IBM’s J9 VM, in Section 5.5, to demonstrate that our architecture is general enough to be implemented in different JVMS.

However, we first summarize the different pieces of our implementation in an overview to better describe how the pieces fit together.
5.1 Overview

Recall, again, our Java VM architecture diagram, Figure 2.4, from previous sections. Here we use this architecture to briefly name and describe the parts of our four implementations. This diagram is repeated in Figure 5.1 and annotated with the names of components of our implementations.

Our implementation space consists of 5 component implementations. Two of the components are bytecode-based runtime weavers. We call these aclv1 and aclv2. These two components are standalone implementations but they can be combined with JIT optimizations for improved execution performance. They are both implementations in the Jikes RVM and are described in more detail in Section 5.4.2 and Section 5.4.3.

Two of the components are modifications to JIT compilers to support AspectJ-specific optimizations, but these do not implement advice weaving themselves: they both depend on receiving pre-bytecode-woven code by a static compiler or aclv1 or aclv2. These two components are ajit-ibm, implemented in J9, and ajitv2 implemented in the Jikes RVM. These are described in Section 5.5 and Section 5.4.1.
Table 5.1: Implementation Components Summary. The table summarizes which implementation sub-components compromise each of our complete implementations.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>FE ajc</th>
<th>aclv1</th>
<th>aclv2</th>
<th>ajit-ibm</th>
<th>ajitv1</th>
<th>ajitv2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ajvm-ibm</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ajvmv1</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ajvmv2</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ajvmv3</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

respectively.

And finally, we present one final component which is an advice weaver implemented fully in the JIT compilers of the Jikes RVM: ajitv1. This implementation requires no previously woven bytecode and is described in detail in Section 5.3.

Because not all of these components form complete AspectJ implementations independently, Table 5.1 summarizes how we combine them into four broad implementations we study: ajvm-ibm, ajvmv1, ajvmv2 and ajvmv3. We revisit this combination and the resulting consequences in Section 5.6.

Finally, because all of our implementations use the AJVM architecture, we use ajc as the frontend compiler for all of them.

5.2 Front End Compiler (ajc)

The front end compiler is the only part of the architecture that does not allow a significant degree of freedom of implementation: it must produce code that conforms to the on-disk format. To maintain backward compatibility with existing AspectJ programs compiled with ajc, our architecture specifies an on-disk format compatible with the output of ajc. For this reason, all of our implementations use the ajc compiler as a front end compiler.

Typically, the ajc compiler will perform as much advice weaving as possible during compilation. Since advice weaving depends on pointcuts to identify the join point shadows in the source code where advice dispatch may occur, aspects must be known to the compiler when compiling standard Java classes to perform static
weaving.

When classes are individually compiled without knowledge of configured aspects, no static advice weaving occurs, and instead this is left up to the load time or run time weaver to complete. This is the standard way we use the compiler and it is how compilers are used in general.

The compiler can also be configured to leave behind an aspect definition XML file which informs the load time or runtime weaver which aspects are required by the program using the “-outxml” command line option. Alternatively, this file can be created or edited by hand to modify the aspect configuration of a program on a per-run basis as described in Section 4.2.

5.3 JIT Based Weaving in the Jikes RVM (ajvmv1)

Our JIT based weaver implements the proposed architecture using code generation in the JIT compilers in the Jikes RVM. Because Jikes RVM does not use an interpreter directly, it does not require a mix of interpreter and JIT weaving approaches for a correct implementation.

This was our first implementation and hence it did not have as refined support as the later implementations. Although it could be modified to support ajc’s aspect definitions file, for simplicity the implementation used a simpler file format which specified each aspect to load directly, one per line. The implementation then forced the loading of each of these aspects in turn before executing the main() method of the program.

As each aspect and class is loaded, AspectJ-specific meta-data is extracted by using a visitor pattern that visits each class and method attribute. This method has higher overhead than a more integrated approach, but this implementation was designed to measure the steady state performance of code-generation implementations and so a simpler implementation with less efficient start up performance was acceptable.

Since the attributes serialized in each class file are in ajc’s serialized format, we use ajc’s serialization infrastructure to parse these attributes back into Java objects. This is of primary concern for pointcut representation in which pointcuts are parsed into a tree structure.
Our implementation supports both the baseline and optimizing compilers in the Jikes RVM. The baseline compiler generates native machine code directly from JBC by outputting code vectors which simulate what the Java stack machine would do when interpreting each bytecode. This process is described in more detail in Section 2.6. Our implementation hooks into this compilation loop over the bytecode of each method.

At each bytecode which represents a shadow, pointcut matching is performed on the tree representation of all available pointcuts. Matches on a specific bytecode cause the original machine code to be generated, which implements the bytecode, plus additional machine code that surrounds the original that implements the advice dispatch. Surrounding machine code accesses values off the simulated Java stack directly to setup calls to advice bodies if required.

The optimizing compiler portion of the implementation is similar except that the weaver generates additional HIR around the original HIR for the implementing bytecode as the bytecode is translated into HIR. This code generation must obey many HIR invariants during translation. These include obeying abstract stack invariants such as size and type of entries, ensuring that basic blocks are created and used appropriately, especially for cflow pointcuts, as well as obeying conventions for use of virtual registers. Additionally, the JIT HIR weaver uses the AOS infrastructure to determine whether advice bodies should be inlined into the method containing the join point shadow at which they apply. Because our code generation happens before any other optimizations in the compiler, all our produced code goes through all existing code optimization passes.

This implementation includes only the aspect instance retrieval and aspect dispatch optimizations, however it includes them in both the baseline and optimizing compiler. It supports before advice on both execution and call join points.

The primary contribution of this implementation was the realisation that much of the machine code in the baseline compiler and HIR in the optimizing compiler that our weaver was generating was nearly identical to the code that these two compilers generated for bytecode-woven advice dispatch. The differences between the two only occurred in the places were we had implemented optimizations. And, further, we observed that the optimizing compiler already had a framework for dealing with the types of optimizations we implemented: it only required a few ex-
tra hints about the code it was compiling to achieve the same level of optimization in bytecode-woven code.

Additionally, our implementation had the drawback of being outside the VM’s bytecode-based profiling framework. So, although we benefit from much of the optimizing in the JIT compiler, the VM’s profiling framework was not recording profiling information specific to advice dispatch because it had no corresponding bytecode representation.

We used these insights to produce our second and third implementation of the ajvm architecture described above in the next section, Section 5.4.

5.4 Bytecode Based Weaving in the Jikes RVM (ajvmv2, ajvmv3)

5.4.1 JIT Compiler and Runtime Support (ajitv2)

Both our aspect instance retrieval and advice invocation optimizations are implemented as simple code generation changes or hints to the Jikes RVM optimizing JIT compiler. Both are handled during the first stage of the compilation process where JBC is translated into the optimizing compiler’s HIR. The Jikes RVM HIR is expressive enough to encode most of our optimizations, and yet high level enough that the compiler’s optimization plans are still performed.

The aspect instance retrieval optimization is implemented by slightly modifying the HIR generated for calls to an aspect’s `aspectOf()` method, the method which the bytecode weaver or programmer calls to retrieve an aspect instance. Recall that this method takes no parameter if the aspect is using an `issingleton` instantiation model, and an `Object` parameter otherwise.

Figure 5.2 shows the JBC that is generated by our bytecode-based weaver for a simple before advice on a call join point. Here we see that the aspect instance is retrieved at bytecode index 9, and the actual advice body is invoked at bytecode index 12. This advice was statically matched to apply to join points generated from the call join point shadow at index 15.

Figure 5.3 shows the HIR that is generated for the JBC in Figure 5.2 without these optimizations. Lines 15-18 show the call to `aspectOf()` on the class
Figure 5.2: JBC Generated for a Simple Before Call Advice

Aspect1. The result of the call is protected by a null_check guard to capture null pointer exceptions, and finally line 20-23 shows the actual call to the advice body. Note that this is a full virtual method invocation as noted by virtual"<SystemAppCL, LAspectj1;>. The reference to the advice instance guarded by the null_check guard will cause the null check to be realized in the final code.

Figure 5.4 shows the resulting HIR generated when our optimization is included. Line 15-19 shows the full call to the advice body. Note here that the guard on this call is the TRUEGUARD (the default null guard), the call is virtual_exact, so no virtual method table lookup is necessary, and furthermore the aspect object that is the target of the call is immediately available as object "Aspect1259a22e8".

Note that further optimization of this method by the optimizing compiler without our AspectJ optimizations can produce similar results because of OSR in the Jikes RVM, but this only applies to a single execution of the method. The compiler can prove these invariants when the executing method is optimized while it is still on the stack because live object type information is available during OSR. However, our optimization produces these results on the first optimization pass of the method and does not require OSR, and so the optimization spans across multiple method invocations.

Furthermore, the second optimization pass over this method by both the default optimizing compiler and the compiler including our optimizations produces HIR in which both the aspectOf() call, and the advice body call are inlined into the method.

Additionally, we still benefit from all of the optimizations that the optimization
********** START OF IR DUMP Initial HIR

FOR < SystemAppCL, LBaseClass; >.main ([Ljava/lang/String;);V

-13 LABEL0 Frequency: 0.0

-2 EG ir_prologue 10a([Ljava/lang/String;,d) =

0 guard_move t2v(GUARD) = <TRUEGUARD>

0 EG new t1a(LBaseClass;,x,p) = BaseClass

-2 ref_move l3a(LBaseClass;,x,p) = t1a(LBaseClass;,x,p)

-2 ref_move l4a(LBaseClass;,x,p) = l3a(LBaseClass;,x,p)

7 guard_move t6pv(GUARD) = t2v(GUARD)

7 ref_move l5pa(LBaseClass;,x,p) = t1a(LBaseClass;,x,p)

9 int_move l7pi(Z) = 0

10 goto LABEL2

-1 bbend BB0 (ENTRY)

13 LABEL1 Frequency: 0.0

14 EG call t20a(LAspect1;) AF CF OF PF ZF ESP =

Addr 0x00018dd4,

   static"<SystemAppCL,LAspect1;>.aspectOf()

   ();LAspect1;", <unused> ESP

17 EG null_check t21v(GUARD) = t20a(LAspect1;)

17 EG call AF CF OF PF ZF ESP = Addr 0x0000004c,

   virtual"< SystemAppCL, LAspect1; >.

   ajc$before$Aspect1$1$d3c01dbf ()V",

   t21v(GUARD), t20a(LAspect1;) ESP

-2 ref_move l22a(LBaseClass;,x,p) = l5pa(LBaseClass;,x,p)

0 getstatic t23a(Ljava/io/PrintStream;)=Addr 0x00012e7c,

   <mem loc: Ljava/lang/System;.out>

5 EG null_check t24v(GUARD) = t23a(Ljava/io/PrintStream;)

5 EG call AF CF OF PF ZF ESP = Addr 0x000000b8,

   virtual"< BootstrapCL, LPrintStream; >.

   println (Ljava/lang/String;)V",

   t24v(GUARD), t23a(Ljava/io/PrintStream;),

   string "[BaseClass] In test 1" ESP

24 EG call AF CF OF PF ZF ESP = Addr 0x00000050,

   virtual_exact"<SystemAppCL,LBaseClass;>

   .test2 ()V",

   t6pv(GUARD), l5pa(LBaseClass;,x,p) ESP

27 int_add l7pi(I) = l7pi(I), 1

-1 bbend BB1

30 LABEL2 Frequency: 0.0

33 int_ifcmp t19v(GUARD) = l7pi(I), 0x186a0, <, LABEL1,

   Probability: 1.0

-1 bbend BB2

36 LABEL3 Frequency: 0.0

-3 return <unused>

-1 bbend BB3

********** END OF IR DUMP Initial HIR

FOR < SystemAppCL, LBaseClass; >.main ([Ljava/lang/String;);V

Figure 5.3: HIR for Simple Before Call Advice and Join Point Generated without Optimizations.
Figure 5.4: HIR for Simple Before Call Advice and Join Point Generated with Optimizations.
plan calls for because our optimizations are hints and minor code-generation modifications which affect only the generation of the HIR before any actual optimizations are applied. All of the HIR, LIR, and MIR optimizations that the optimizing compiler performs are performed after our code generation.

Finally, we implement the cflow optimization in Section 4.4.2 by adding specific runtime support for cflow accounting. Because the compiled program targets a standard API to implement cflow, we have the ability inside the VM to implement parts of the library that are otherwise typically implemented in a standard Java library. Our implementation uses this technique, so cflow accounting calls will be calls directly into the VM rather than to a standard library.

Our implementation includes VM-internal versions of the cflow stack and counter classes that are part of ajc’s AspectJ runtime library. VM-external versions of these classes must use ThreadLocal variables to implement the semantics of cflow properly. However, our implementation has direct, efficient access to the currently executing thread whenever these calls are made, and so these methods are implemented to directly access added fields in the RVMThread class. Efficiently implementing this lookup affects the performance of the crossing of cflow boundaries, testing cflow conditions, and extracting values matched in a cflow pointcut.

Although the primary overhead of cflow is the lookup of the data structures given their thread local nature, additional optimization is available by a tighter integration with the JIT compiler as described in [17] and Section 3.2.7. Our implementation could be extended to implement these additional micro-optimizations, but the macro-optimizations implemented for cflow demonstrate the capability of our architecture to support these kinds of techniques.

5.4.2 VM Load Time Internal (aclv1)

Our first implementation of a runtime weaver inside the Jikes RVM takes advantage of the fact that the Jikes RVM is Java-in-Java. This implementation is based on the Jikes RVM version 2.9.0.1 and supports the full AspectJ language, including ITDS. The weaver has two main parts. The first is the actual advice weaver, which processes classes as they are loaded and performs bytecode weaving based on configured aspects. The second is the meta-data extractor, which also processes
classes and aspects as they are loaded, but preserves necessary AspectJ-specific information for optimization purposes.

This load time weaving implementation is based tightly on ajc’s VM-external load time weaver. We integrated much of the code of ajc (which is written in Java) directly into the VM. So, at VM initialization, a modified version of ajc’s load time weaving framework is initialized which processes the aspect definition file to identify aspects required by the program.

Because this weaver is not tightly integrated into the VM, it does not operate directly on VM-internal data structures. Instead, it intercepts class loading inside the VM while classes are still in their on-disk format. These classes are then modified if necessary by the weaver. Any modified classes are re-encoded into their on-disk format, and replace the original class at the point that the weaver intercepted the class during loading. Any new classes required by weaving, such as the creation of around closure classes, are also created by the weaver to be loaded by the VM when necessary.

The meta-data extractor also operates on classes as they are loaded, but at a point further along the class loading chain. As the VM parses the on-disk format of a Java class to construct an internal representation of that class, it must parse that classes attributes. Since AspectJ meta-data is stored in these attributes, the implementation extends the attribute parser to understand AspectJ attributes. This process adds little overhead to the existing attribute parser because each attribute must be processed regardless.

AspectJ-specific information that is necessary for further optimization of the program is then stored in the VM’s internal representation of the class. This information includes annotating classes that represent aspects and the instantiation model that they use, as well as annotation methods in an aspect that correspond to advice execution or aspect instance retrieval. This information preservation is what allows the JIT compiler to optimize AspectJ code as discussed in Section 5.4.1.

Note that this implementation has several significant drawbacks from a performance perspective. First, because the implementation reuses much of the ajc load time weaving framework code, it is not a carefully integrated implementation in the Jikes RVM. The ajc load time weaver must reconstruct and maintain its own representation of the class hierarchy to perform pointcut matching because it was
implemented as a VM-external weaver. But, this representation is not necessary for a VM-internal approach because the class hierarchy is already represented directly by the VM.

Secondly, the internal weaver is also forced to decode and re-encode Java on-disk formatted class files as they are loaded by the VM because this is the only representation of a class available to a VM-external approach. This adds significant avoidable overhead during program startup.

And finally, because the implementation operates on the on-disk format of Java class files, it must weave entire classes at a time as they are loaded. It does not have the option of a finer level of weaving granularity.

Although this implementation takes a naive, straightforward approach to runtime weaving, the performance overhead does not affect the steady state performance of the program since the weaver operates on each class exactly once and only at load time. The code quality that the weaver generates is not affected by its lack of VM-integration.

So, although this implementation is not ideal, it serves several useful purposes. It demonstrates that an implementation of our proposed architecture is possible. Although this implementation is naive, it fully meets our architectural requirements, and it supports the JIT compiler enough to enable all known AspectJ-specific VM optimizations. And although the startup performance of the implementation is effected, the steady state performance is not.

### 5.4.3 VM Internal Integrated Weaver (aclv2)

Our second VM-integrated bytecode weaver uses the same JIT compiler support as above, but it is a complete re-implementation of the runtime weaver discussed in Section 5.4.2. This implementation is based on the Jikes RVM version 3.1.0.

This internal weaver is also a bytecode based implementation, however it operates directly on the VM’s internal bytecode representation rather than the on-disk format of Java classes. This further allows the implementation to operate at different weaving granularities. For example, it supports weaving entire classes at a time as with the previous implementation, but it also supports weaving a method at a time.
This bytecode weaver supports a subset of the AspectJ language, including call and execution shadows, before, and after returning advice and reference, kinded, within, this, target, args, if, and cflow pointcuts.

Because the organization of our implementation crosscuts many structures in the VM, as well as different concepts and architecturally imposed restrictions, we describe our implementation in three overlapping parts. First, the loading process broadly describes the process the weaver follows to ensure aspects are loaded and advice weaving is properly performed. Secondly, we describe the internal data representations used in the implementation. And finally, we describe in detail both pointcut matching and bytecode munging which implement the actual advice dispatch.

**Loading Process**

Like the previously discussed implementation, this weaver ensures that the necessary aspects are loaded by processing the aspect definition file of the program before execution of the main method begins. Each of the defined aspects is loaded and processed through the standard class loading infrastructure of the VM, and this loading happens after the VM has been initialized, but before the VM begins executing the main() method of the program.

The second part of the implementation deals with reading AspectJ-specific data stored in loaded classes as they are processed by the VM infrastructure. All AspectJ data, both on aspects and classes, is processed as it is dynamically loaded by the VM class loading infrastructure.

Because aspects are explicitly loaded by the VM, rather than dynamically loaded as the program executes, all AspectJ data stored in the aspect files is available before the program begins execution.

With the exception of our secondary pointcut representation (see below), all pointcuts, aspects and advice are parsed into their internal representations immediately after being read from the on-disk attribute formats. Because there are interdependencies between pointcuts via name references, pointcuts are optionally translated into a secondary representation\(^1\) after all aspects have been loaded.

\(^1\)This representation was designed by Naseer et al. [63] for fast interpreter-based pointcut matching.
Data Representation

This process has to deal with two kinds of primary data. First, it must deal with the pointcuts of the program. Pointcuts are of primary concern because they will be used during all subsequent class loading to determine which join point shadows in which methods must be munged to effect advice dispatch. This also includes named pointcuts, which can be referenced from other aspects. Second, it must deal with aspects themselves, including: aspect initialization, the binding between pointcuts and advice bodies, and the meta-data that allows optimization of advice execution.

The translation and representation of on-disk pointcuts into an internal format has an effect both on the performance of the parsing, as well as the performance of matching using those pointcuts.

The internal representation of a pointcut is a tree structure which exactly mimics the structure of the on-disk representation of the pointcut. Typically this tree structure is an exact abstract syntax tree (AST) representation of the pointcut that the programmer wrote in the program. Note that most pointcuts contain many references to program types in either wildcard or non-wildcard type patterns. These references are converted to concrete VM-internal references whenever possible while the representation is constructed. Note that since type references are maintained as references inside the VM, they do not force class loading of referenced types. Each pointcut and advice pair is stored in a global list for later matching.

Optionally, this representation can be used to construct an alternate pointcut storage structure: the tabular structure proposed by Naseer [63]. This structure parses the first representation of a pointcut into type pattern tables and lists of these tables. Type pattern tables organize pointcuts into groups based on the type patterns of the target of the join points that they may apply to. So, for example, if several pointcuts all applied to the non-wildcard List type, they would be grouped into the same entry in the table. Similarly, wildcard type patterns that are used across different pointcuts would also be grouped.

When Naseer’s table structure is used and a class is loaded by the VM, its type name is compared against each non-wildcard and wildcard type pattern in the type pattern tables. Any pattern matches with the classes’ type name means that a
pointcut in that specific type pattern table could match join points which have the
loaded class as a target. Each of the matches is stored in a list in the loaded class.

This representation is designed to reduce the number of pointcuts which must
be consulted to determine matching at a specific join point or join point shadow.
This is discussed further below.

Finally, the weaver must deal with the representation of aspects themselves and
pointcut and aspect associations. This is primarily implemented by extending the
RVMClass class in the Jikes RVM, which represents classes internally, to include
an aspect structure and a list of named pointcut declarations. The aspect structure
identifies that class as an aspect and includes its aspect instantiation model. This
structure is constructed from the Aspect attribute stored with the aspect.

Named pointcuts are constructed according to the first of the pointcut represen-
tations and stored in the list of named pointcut declarations. Each of these pointcuts
is generated from the on-disk PointcutDeclaration attribute stored with as-
pects. These pointcuts can be referenced from other concrete pointcuts in the en-
closing or other aspects.

The memory requirements for these representations are minimal and linear in
the size of the program. Since aspect, advice and pointcuts are just direct repre-
sentations of the on-disk format, they are proportional to the size of the program
representation of these constructs. Naseer’s TPT representation is proportional to
the number of loaded classes and the number of pointcuts because each TPT stores
at most one reference per loaded pointcut, and each loaded class stores a list of
TPTS.

**Pointcut Matching and Advice Munging**

Our implementation uses bytecode-based advice weaving directly on the VM-internal
representation of a method’s bytecode. Because of the Jikes RVM’s design and im-
plementation, it would be prohibitively expensive to implement bytecode-based
advice weaving at a finer granularity than weaving an entire method at a time, so
that is the finest granularity we support. Class level granularity advice weaving is
implemented simply by iterating through each method of a class and performing
the weaving on that method.
```java
public static void processMethod(NormalMethod method) {
    VM_BytecodeList codeList = new VM_BytecodeList(method.getByteCodes());
    List<Shadow> shadows = codeList.getShadowList();

    for (Shadow s : shadows) {
        List<Advice> adviceList = getAdviceList(s);
        for (Advice a : adviceList) {
            Pointcut pc = a.getPointcut();
            VM_ResidueMap rMap = new VM_ResidueMap(s.getNumberParameters());

            if (pc.match(s, rMap).maybeTrue()) {
                munge(codeList, a, s, rMap);
            }
        }
    }
    codeList.finalizeAndInstall();
}
```

**Figure 5.5: Outer Matching and Munging Loops**

Pointcut matching and advice munging are intertwined because munging depends on matches having already been computed. Figure 5.5 shows the general algorithm for matching and munging. Broadly, this algorithm converts a concise byte-array representation of the method’s bytecode into a linked list, checks each pointcut against each join point shadow in the list for a possible match, munges the bytecode list if required, and converts it back to a byte-array representation when complete. The result of matching can be one of three answers: static fail, static match, or maybe match. Only in the first case can we avoid munging the bytecode, because a static match requires unguarded advice execution and a maybe match requires additional dynamic information which can only be checked at runtime.

On line 6 the algorithm retrieves a list of advice. This list contains one pointcut per advice to be checked against the join point shadow. The implementation which does not integrate Naseer’s table structure, `getAdviceList()` will return the global list of advice: every advice loaded by the VM. If Naseer’s table structures have been created, this method instead will consult the type pattern tables in the target class of the shadow `s`. Ideally this list will be much shorter than the global list and contain only advice whose pointcuts match the join point shadow.

Also of primary interest, are lines 2 and 18 which convert between bytecode
public class MainClass {
    int counter = 0;

    void test1() { counter++; }
    void test2() { test1(); }

    public static void main(String[] args) {
        MainClass c = new MainClass();

        for (int i = 0; i < 10000000; i++) {
            c.test1();
            c.test2();
        }
    }
}

Figure 5.6: Example Main Method

public aspect Counter {
    int counter = 0;

    before(): call(void Main*Class.test1()) && withincode(void Main*Class.main(..)) {
        counter++;
    }
}

Figure 5.7: Example Aspect and Advice

representations, line 12 which performs matching, and line 13 which performs munging. We detail each of these processes in turn.

Consider the main method shown in Figure 5.6 and an example aspect shown in Figure 5.7. A front end compiler configured to perform no static weaving would produce the bytecode shown in Figure 5.8 for the main method.

This bytecode is converted into a list representation shown in the diagram Figure 5.9. Note that this representation uses pointers to bytecode objects to represent offsets instead of absolute indices from the start of the method. This representation allows us to easily insert new bytecode in the middle of the method without having to recompute offsets after every change.

Figure 5.10 shows the bytecode of this method after it has been munged. We note the insertion of two additional bytecodes: invokeinternals and invokeinternalv. These are two special bytecodes that use the Jikes RVM
```java
public static void main(java.lang.String[]);

Code:
0: new #1; //class MainClass
3: dup
4: invokespecial #25; //Method "<init>":()V
7: astore_1
8: iconst_0
9: istore_2
10: goto 24
13: aload_1
14: invokevirtual #21; //Method test1():V
17: aload_1
18: invokevirtual #26; //Method test2():V
21: iinc 2, 1
24: iload_2
25: ldc #28; //int 10000000
27: if_icmplt 13
30: return
```

**Figure 5.8:** Example Method Bytecode produced from Figure 5.6

```java
0: new
3: dup
4: invokespecial
7: astore_1
8: iconst_0
9: istore_2
10: goto
13: aload_1
14: invokevirtual
17: aload_1
18: invokevirtual
21: iinc 2, 1
24: iload_2
25: ldc
27: if_icmplt
30: return
```

**Figure 5.9:** Bytecode List Representation of Example Method
method IDs as their parameters instead of constant pool offsets: they refer directly to`Aspect1.aspectOf()` and`Aspect1.abc$before$Aspect1$1$d3c01dbf()` directly instead of referring to string references in the constant pool. Using these special internal bytecodes allows us to insert instructions like this without having to modify the constant pool of the class.

Note also that these bytecodes do not yet have computed offsets and that their insertion has made the offsets of some of the existing bytecode invalid. The bytecode munger needs to recompute these offsets only when finalizing and installing the new method; it is more efficient to perform all necessary munging on the method before recomputing these values.

A pass in`VM_BeckodeList.finalizeAndInstall()` recomputes
public static void main(java.lang.String[]);

Code:
0: new #1; //class BaseClass
3: dup
4: invokespecial #25; //Method "<init>":(V
7: astore_1
8: iconst_0
9: istore_2
10: goto 30
13: aload_1
14: invokeinternals // Method Aspect1.aspectOf();
17: invokevirtual #21;// Method test1:()V
20: invokevirtual #26;// Method test2:()V
23: iinc 2, 1
30: iload_2
31: ldc #28; //int 10000000
33: if_icmplt 13
36: return

Figure 5.11: Example Method Bytecode after Munging

the offsets by traversing the list from the beginning and calculating the index at which each bytecode will be stored in the resulting array, stores the bytecode objects back into an allocated byte array, and replaces the old method bytecode with the new munged bytecode. This process is only performed when bytecode was munged in the method; if there were no matches to any shadows in the method, the munging does not need to perform any of these extra passes.

The final installed bytecode array is shown in Figure 5.11.

Our bytecode weaver targets new bytecodes for each of several existing Java bytecodes that take as a parameter an index into the constant pool. From above, invokeinternals mirrors invokestatic, and invokeinternalmv mirrors invokevirtual. These mirror bytecode implementations are exactly the same as the bytecode they are based upon except for where their reference parameters are retrieved from.

Adding these new bytecodes is straight forward in the Jikes RVM because it uses a bytecode stream class as an interface to the bytecode used by both the baseline and optimizing compilers. This interface hides constant pool access for method references from the JIT compilers, and simple modifications to its implementation also hide the implementation of the invokeinternals and
invokeinternalv bytecode by looking up their method references directly instead of through the constant pool. Since the code that must be generated by invokeinternals is the same as invokestatic, no other changes need to be made to the compilers to support it.

Matching

Line 12 in Figure 5.5 shows the stage of the algorithm at which pointcut matching is required. The matching process is the same regardless of whether TPTS have been enabled in the VM.

The pointcut matching process must return two things: an answer to the question of whether the pointcut matches the join point shadow being checked, and if it does potentially match, a residual mapping for parameters required by the advice. The result of the pointcut matching may be one of three answers: static fail, static match, or maybe match. These indicate that the pointcut either definitely does not apply, definitely applies, or requires dynamic checking to complete matching respectively.

The pointcut matching algorithm takes as parameters a residual map to store dynamic residual information and an object representing the join point shadow to be tested for a match. Pointcuts are matched by traversing the pointcut tree representation. Each node in the pointcut tree specifies properties that the given join point shadow must possess, and these are checked while visiting each node.

For example, && pointcut matching is as simple as:

```java
protected FuzzyBoolean matchInternal(Shadow shadow, VM_ResidueMap rMap) {
    FuzzyBoolean leftMatch = left.match(shadow, rMap);
    if (leftMatch.alwaysFalse()) return leftMatch;
    return leftMatch.and(right.match(shadow, rMap));
}
```

whereas other pointcuts, such as a call pointcut must check several things: whether the join point shadow represents a Java method call; whether the target class name and target method name match the actual object and methods being invoked by the Java method call; and finally, whether the parameters of the Java method call match the required parameters of the call pointcut.

Matching in this case becomes more complicated when wildcard names are
used for type patterns. Since wildcard types cannot be resolved to concrete VM internal classes, every match against a wildcard type requires string comparisons. Non-wildcard type patterns in pointcuts are converted directly to concrete VM class representations, and since each class in the VM is unique, type matching is equivalent to pointer comparison.

The residual map is populated by pointcuts such as cflow, if, and this, target or args since all of these pointcuts require dynamic residual code to complete advice matching at runtime.

For example, the following advice:

```
1 before(TestClass tc): execution(void TestClass.test1()) && this(tc) {
2   ...
3 }
```

will result in a `this` pointcut that both checks for the “this” object being of runtime type `TestClass`, as well as binding it to the first parameter to the advice body. When a successful match occurs for this pointcut, all of the fact that “this” must be of type `TestClass`, the location of the instance of “this” on the stack, and the parameter index where “this” needs to be mapped to in the advice invocation is recorded.

All of this information is necessary so that the bytecode munger can produce proper code.

**Munging**

Line 13 in Figure 5.5 shows the stage at which bytecode munging takes place. Munging is implemented on matched pointcuts one at a time, one shadow at a time.

The first thing the munging process does is compute how much extra local variable space is required to store the parameters required by the advice execution bytecode. Since JBC is a stack language, all of the parameters to the join point shadow are stored in the Java stack. Since some of these parameters will be needed for dynamic checking, or as values passed to the advice body, they need to be either duplicated on the stack or saved into local variables so that they can be used by both the advice invocation and the original join point shadow. The return value of the join point shadow is also allocated space in local variables if it is required by an
after returning advice.

Our bytecode munger makes no attempt at optimizing the storage and retrieval of parameters to the join point shadow. If any of these parameters are required by the advice invocation, then local variable space is allocated for all of the parameters and instructions are emitted to store them all to local variables for ease of implementation. Unused parameters, however, will be discovered later by the optimizing compiler’s constant propagation and define/use optimizations.

Note that this local variable storage is only allocated for non-execution join point shadows. Since the JBC specification dictates that the parameters for execution join point shadows are all stored in local variables when the method begins executing, we do not need to create extra space or emit bytecodes for these.

After bytecode instructions have been emitted to save these parameters into local variables the munger emits bytecode which loads the necessary parameters back onto the stack according to the parameter mappings stored in the residue map created by the matching algorithm. This process not only creates bytecode which loads the parameters back onto the stack in the correct order for the advice body execution, but it also creates the necessary dynamic instanceof checks and branch instructions to guard the advice execution based on the runtime type that the pointcut requires of some of the parameters.

Finally, the code vectors required for cflow testing and if pointcuts are generated.

All of these code vectors are generated into a local array before being adding to the bytecode list of the method itself for clarify of implementation. And, since offsets in the list representation are represented by pointers to target bytecode, creating branch instructions is quick and robust. The addition of new bytecode into the middle of the method does not affect existing branch instructions or the exception table of the method.

Further, as noted above, any generated instructions which require references that would traditionally refer to the constant pool of the class use our internal mirror instructions which reference the members directly.

Although not implemented in our implementation, around advice would require a slightly more complicated implementation in the munger. Although the majority of cases of around advice munging can be implemented without gener-
ating new closure classes and objects, in some cases proper munging requires the
generation of new classes. These classes could be generated eagerly, at the time
of the munging, or lazily when the advice is actually executed. In either case, we
can rely on the class loading infrastructure to recognize the special case of closure
class loading and produce a specialized class accordingly. This implementation
would affect the performance of munging only, and does not add to the overhead
of pointcut matching.

Lastly, munging the bytecode of a method marks the entire method bytecode
list as “dirty.” This means that the individual bytecodes in the list no longer have
correct computed bytecode-indexed offsets for branching, and so when this method
is installed into the VM, these offsets must be recomputed.

In summary, our bytecode weaver takes five passes over the method’s bytecode:

1. The list representation is generated, including join point shadows.
2. All branches and exception table pointers are resolved in the list representa-
tion.
3. Each join point shadow is matched against available pointcuts and munging
occurs if necessary.
4. The offsets of all bytecodes are recomputed if the method has been changed.
5. The list representation is converted back into a byte array of exact size.

And it has two broad configuration options. The first is that it can be configured
to act as a load time weaver, in which it performs bytecode weaving on a class
level granularity as those classes are loaded into the VM. Alternatively, it can be
configured to act as a runtime weaver which weaves bytecode on a method level
granularity: it weaves one method at a time just before that method is executed for
the first time.

The second configuration option is whether the implementation uses Naseer’s
type pattern tables to possibly prune the number of pointcuts requiring matching at
each join point shadow encountered.
5.5 Implementation in J9 (ajvm-ibm)

We have a partial implementation of our architecture in IBM’s J9 VM to demonstrate the suitability of our approach on a second VM from a steady-state optimization perspective. The implementation consists of a rudimentary implementation of the load time weaver which does not perform weaving. Instead, its function is to just extract meta-data from the class files as they are loaded.

When attempting to implement the aspect instance retrieval and aspect dispatch optimizations in J9, it became clear that in our benchmarks the J9 JIT compiler was sophisticated enough to perform the optimizations on its final optimization passes without the additional hints from the provided meta data. It is possible that by providing additional hints, the optimizations could have been arrived at sooner during optimization, but we did not explore this in detail. Steady state analysis and benchmarking takes into account only the quality of the final optimization pass.

However, we use the meta data to implement cflow optimizations for both the interpreter and the JIT compiler. This implementation is similar in concept to the cflow optimization implementation in the Jikes RVM because they both extend the internal thread representation to include references to cflow counters and stacks to improve the performance of cflow boundary accounting and testing.

The implementation in J9 also uses a VM-compatible runtime library to support the cflow pointcut for interpreter execution only. Because J9 is implemented in C/C++, the implementation of the library is part Java and part native C/C++ calls. The Java portion of the library implements the cflow API of the AspectJ runtime library. It works as a loader and wrapper of the J9 compatible native portion.

The native portion of the library implements the machinery that accesses the VM threads directly. It is implemented as a shared object library that is linked into a running VM dynamically when the Java library requires it. Because it is written in C/C++ and provided by the VM, it can access and manipulate the internals of the VM on behalf of the Java code.

When the method is JIT compiled the Java and native library is no longer used to access the VM threads. Instead, the JIT compiler identifies the calls for cflow accounting directly using provided meta data and generates optimized high level IR to access the thread data structure. The JIT compiler is unable to do this on its
own with the libraries created for the interpreter because it cannot optimize through the Java native methods that the library requires.

However, the end result is that the JIT compiler generates similar machine code to that generated by the Jikes RVM implementation.

5.6 Summary

Now that we have described each implementation and sub-component in detail, it is useful to revisit how they fit together overall. Recall from Table 5.1 how the different sub-components are fit together to make complete implementations of our architecture.

ajvmv2 and ajvmv3 are our primary implementations. We use these implementations to evaluate the performance that is achievable in our architecture. ajvmv2 is used to evaluate the steady state performance attainable. Because steady state performance measures the long running performance of a program, it ignores the startup time and hence the fact that ajvmv2 uses a naive bytecode weaver (aclv1) does not have an impact on the results. However, since aclv1 implements the entire AspectJ language, this implementation allows us to benchmark the entire range of AspectJ features with our optimized JIT support (ajitv2).

We use the optimized bytecode weaver (aclv2) in ajvmv3 to evaluate the startup performance attainable by our architecture. Although aclv2 supports only a subset of the AspectJ language, it contains the full pointcut matching and weaving infrastructure required to support the full language, it performs matching on all join point shadows in AspectJ, and so using it to measure startup performance is accurate. Note that ajvmv3 supports a number of configuration options which we vary in our benchmark analysis also (granularity levels and the use of TPTS).

For comparison, the Jikes RVM 3.1.0 has 688 files and 136220 lines of code in the VM proper. This count excludes the GC, because this part of the VM is contained in its own standalone module called the memory management toolkit (MMTK). ajvmv3 contains 787 files and 146421 lines of code. This is an increase of 99 files and 10201 lines of code, however, the weaver module in ajvmv3 contains all 99 of these files and 9959 lines of code. So, ajvmv3 requires only 242 additional lines of code in existing Jikes RVM 3.1.0 files.
Finally, the ajvm-ibm and ajvmv1 implementations demonstrate that our architecture is implementable in alternate VMS, and also that a full JIT-based bytecode weaver (ajitv1) achieves similar IR to a JIT which is supplemented with additional AspectJ-specific optimization hints (ajitv2).
Chapter 6

Design of Benchmark Suite

Comprehensive benchmarking of runtime systems like VMS requires two major components. First, the steady state performance of the system should be analyzed. Steady state performance is largely a measure of the code quality produced by the optimizing JIT compiler in the VM because an analysis of steady state ignores VM startup. Instead it runs benchmarks until a stable state has been reached before taking measurements.

In an AspectJ implementation, steady state measurements measure the quality of the code produced that implements the advice dispatch: it measures the actual cost of advice invocation, instance retrieval, and any residual code for dynamic type checking or cflow accounting. Note that such a measurement ignores all costs associated with pointcut matching and code generation in all approaches except an interpreter approach (which performs all advice weaving dynamically).

The second component of our benchmark suite is an analysis of the startup performance of the implementations. This analysis addresses all initial costs paid by the runtime architecture, including pointcut matching and code generation. Although startup costs are amortized over longer running programs, startup performance is still an important consideration because of short running, script-like programs as well as improved startup of large programs like Eclipse.

Together these approaches capture both runtime parts of an AspectJ implementation: advice lookup and advice invocation. Different designs of the implementation will trade-off the lookup and invocation phases in varying ways having
different overall startup and steady state performance.

Additionally, it is important to consider that AspectJ programs are not run in isolation in a VM. Instead they frequently interact with various runtime services in the VM, and their performance depends heavily on all parts of the VM. There exists a body of research [4, 14, 20, 31, 33, 41, 46], as well as available benchmark suites [14, 20, 70] that explore the challenges of benchmarking VMS and what facets of VM performance are of practical use to users.

Due to these, our benchmark suite is divided up largely into two classifications of benchmarks: macro and micro; and two benchmark harnesses which control how these benchmarks are run to produce a startup or steady state performance measurement.

This chapter consists of into four sections. Section 6.1 discusses the details about how we report and analyze reported measurements, Section 6.2 covers the details of the different benchmark harnesses we use to collect our data, Section 6.3 details the specifics about the actual benchmarks in our suite, and finally Section 6.4 goes over specific limitations of our approach.

6.1 Measurement

6.1.1 Methodology

Accurate benchmarking of even simple runtime systems, such as a non-threaded C program, is an involved task. Modern operating system (OS) interleave the execution of programs with OS services, other systems programs, and also other user programs. This means that even a simple program can exhibit different timing behaviour across runs because of the non-deterministic behaviour in which the program runs.

This task becomes even more difficult when benchmarking programs run in a managed runtime environment, such as a JVM. In these environments the non-determinism comes not only from the OS, but also the execution environment. In a JVM, many subsystems come into play when executing the program, including garbage collection, JIT compilation, and profiling. Furthermore, Gu et al. [41] show that local changes that seem simple and predictable can have global non-
trivial effects on the performance of the system.

Many traditional approaches to benchmarking run a benchmark a given numbers of times and report best, second best, worst, mean, or median performance over the runs collected. These measurement techniques are vulnerable to random effects that can occur in the specific run being chosen as the measurement value. Even if the mean result of all of the runs is reported as the measured value, the result is vulnerable to benchmarks which show a high standard deviation across measurements.

Because of these factors, it is important to measure performance in a way that is repeatable, statistically significant, and thorough. We use a measurement strategy advocated by Georges et al. [33], in which multiple invocations and corresponding measurements are used to compute a mean measurement together with a desired confidence interval.

Computing a mean value together with a confidence interval is a statistically grounded way to measure the timing result of a benchmark. A 95% confidence interval in an experiment should be interpreted to mean if that experiment were run a large number of times that 95% of the computed confidence intervals would contain the true value of the mean.

Furthermore, computing a confidence interval together with a mean value gives us a grounded way of making robust comparisons against different experimental runs and different implementations. If the mean of two different experiments varies, but the confidence intervals between the two overlap, we can conclude that at the 95% confidence level, there is no statistically significant difference between the two measurements: there is a significant probability that the difference between the two mean values is due to random effects rather than an actual difference.

6.1.2 Drawing Comparisons

When drawing comparisons between two different experiments using the above methodology, it is also important to consider exactly what it is that is varying between the experiments. It is critical to only allow variance along one dimension at a time: the dimension that is under scrutiny.

To give an overall picture of the performance of one specific VM, many param-
eters of variance must be considered, but they do not apply in all circumstances. Heap size, garbage collection strategy, and platform all have a significant impact on the performance characteristics of an experiment.

Experimentation with the abc macro benchmark suite confirmed that varying any of the front end compiler, the VM, the garbage collection strategy, or the hardware platform can produce a variety of effects on the behaviour of different benchmarks at low heap sizes. These effects include running times which steadily increase or decrease as heap size increases and sudden large changes in running time induced by small changes in heap size. However, across all the combinations of the above variables that we have tried, there is almost always a heap size (typically around 40MB) above which the running time remains roughly constant across heap size variation.

The behaviour below this heap size is heavily affected by the constraints of a small heap and the need to perform frequent garbage collection, and the performance can be dominated by subtle interactions between the garbage collector and the rest of the program and VM. This effect is especially apparent in the Jikes RVM because it is implemented in Java and uses the same heap both for running a benchmark and for its internal VM structures.

We therefore argue that for our purposes, it is appropriate to ignore performance results at low heap sizes as they are not an accurate representation of the performance characteristics we are attempting to measure. At best, such results might reflect small differences in memory requirements, but they would not be indicative of any inherent differences in execution speed because they would measure subtle interaction with the GC. This approach would not be suitable in cases where the correlation between performance and heap size is of interest, such as when comparing different VMs or garbage collectors, but it is reasonable here because we are measuring the micro and macro performance of AspectJ features and programs.

Running the benchmarks with different GC strategy also has a significant impact on performance, especially on macro benchmarks that are memory intensive. It is possible that there is a subtle interaction between different AspectJ implementations and the garbage collection strategy chosen, but we believe these factors to be largely independent.
Finally, hardware platform variation also has an impact on performance which makes it impossible to compare benchmarks which are not running on the same platform. The inherent problem is the same as when comparing benchmark results across distinct VMS: although one VM implemented on different platforms may show a large amount of code, much of the code is different, including the code generator in the JIT optimizing compiler, parts of the GC which must deal with low level machine code representation, as well as scheduling routines. Even interpreters, if they are highly tuned, will be significantly different on different platforms.

6.2 Harnesses

In this section we describe the different benchmark harnesses we have developed. The benchmark harnesses are the driver programs which control how a specific benchmark is run and how the performance measurement of that benchmark is taken. Section 6.2.1 covers the steady state performance harness, and Section 6.2.2 discusses the startup harness.

6.2.1 Steady State

The purpose of taking a steady state measurement of a benchmark is to determine the performance of that benchmark if it were part of a long running program. This roughly measures the code quality that the JIT produces while optimizing that benchmark, but excludes the cost of the optimization itself.

To measure the steady state performance of a benchmark, we run the given benchmark repeatedly in a loop in the benchmark harness. This exercises the benchmark an increasing number of times in each invocation of the VM. Running the benchmark this way runs the same code of the benchmark over and over again, allowing the VM to fully optimize the code on repeated executions.

The benchmark has reached a steady state when the last $k$ executions of the benchmark produce a coefficient of variation ($\text{COV}$)\textsuperscript{1} that is less than a specific threshold. Choosing a low enough threshold ensures that the JIT compiler and GC in the VM have stabilized during the invocation.

\textsuperscript{1}$\text{COV}$ is defined as the standard deviation of the measurements divided by their mean.
So, the benchmark harness maintains a running window of iterations, and updates the COV after every iteration. Once a stable window of $k$ executions has been reached, the average of the running time of those $k$ executions is reported as the running time of the benchmark.

Because JIT compilation does not happen on every execution of the benchmark, reaching a steady state usually means that the benchmark code has been optimized as much as possible. So, taking a final measurement after reaching the steady state is measuring the performance of the JIT compiled code of the workload of the benchmark, together with any memory subsystem load and I/O it uses. This type of measurement is significantly different than taking a measurement on the first, second or even tenth iteration of a benchmark. Early runs of a benchmark will run a mix of optimized and unoptimized code, as well as include execution costs of JIT compiling of newly identified hot methods.

Furthermore, because the steady state harness uses VM timing functions for measurement, and measurements are taken of the time to execute the benchmark in question only, the final result includes neither the time it takes to invoke the VM itself nor the cost of loading the classes that participate in a benchmark execution.

There are some benchmark and VM combinations that never reach a steady state using this measurement methodology. In these cases, there is too much variance within the $k$ execution window to be able to say that the actual measurement is converging to a specific value. In these benchmarks it is not possible to report a final measurement, because there is never a stable window of executions.

A benchmark may fail to reach a stable state for a number of reasons. It could be that the memory behaviour of the benchmark and the GC collection scheme cause inconsistent behaviour which can occur when the allocation pattern of the benchmark is causing collections every second or third benchmark execution, rather than a similar amount of allocations and collections during every execution.

Any benchmark, macro or micro, can be run using a steady state execution harness. However, the benchmark should not include its own timing and measurement code: all of the timing measurement, looping, and statistical analysis code is contained within the steady state harness. The only information required by the harness is the entry point to the benchmark that it should execute during each loop.

Using the harness in this way allows us to micro measure very specific opera-
tions and feature implementations in micro benchmarks, as well as measure overall macro performance of much larger and longer running benchmarks. The harness also allows us to reuse pre-existing benchmarks.

### 6.2.2 Startup

Most available JVM benchmarking is done by taking steady state measurements \[14, 20, 70\]. But, a steady state measurement does not give the whole picture of the performance of an implementation executing a given benchmark. Steady state measurements ignore important parts of VM performance such as the cost of class loading, JIT compilation, and VM initialization. However, it is difficult to capture all of these facets in one program execution. So, we have a second benchmark harness designed to perform VM startup measurements.

Startup measurement, unlike steady state measurement, includes the execution cost of any subsystems of the VM that are required during the startup of the program. Because of this, the VM itself cannot be used to measure the time it takes to run the benchmark, so there are two parts to the benchmark harness. The first part is a separate program which measures the time it takes for a full VM execution of the benchmark, including VM startup and shutdown. The second part is a variable-iteration harness that runs in the VM to execute the given benchmark in a loop a specified number of times. The variable-iteration harness does not include any statistical analysis code to make decisions about how many iterations to run. The final measurement of an invocation of a startup benchmark, then, is the measured time of one complete execution of the VM and the benchmark iterated the specified number of times.

We also use the startup benchmark harness to report on the unfolding performance of the VM by taking measurements for an increasing number of iterations of the benchmark for each invocation of the VM, starting at 1. Evaluating implementations this way allows us to compare the slopes of the performance curves of each implementation against each other to evaluate both the constant speed difference between the implementations due to startup costs, as well as the execution performance of the implementations compared via the slope of the curves.

This harness, like the steady state harness, can also be coupled with any bench-
Table 6.1: Startup Workload Summary of Eclipse 3.5.0, Tomcat 6.0.20 and the DaCapo 9.10-beta0 benchmark suite. The Table shows the number of methods loaded, executed, and compiled along with the ratio of executed to loaded methods, and compiled to executed methods for each benchmark. These numbers were obtained using JVMTI on the IBM J9 VM.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th># loaded methods</th>
<th># executed methods (exec/loaded)</th>
<th># JIT compiled methods (compiled/exec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eclipse</td>
<td>81663</td>
<td>35445 (43.4%)</td>
<td>3386 (9.5%)</td>
</tr>
<tr>
<td>Tomcat</td>
<td>13797</td>
<td>6380 (46.2%)</td>
<td>584 (9.2%)</td>
</tr>
<tr>
<td>antlr</td>
<td>6122</td>
<td>3438 (56.2%)</td>
<td>851 (24.8%)</td>
</tr>
<tr>
<td>avora</td>
<td>7279</td>
<td>3956 (54.3%)</td>
<td>659 (16.7%)</td>
</tr>
<tr>
<td>bloat</td>
<td>7727</td>
<td>4105 (53.1%)</td>
<td>888 (21.6%)</td>
</tr>
<tr>
<td>chant</td>
<td>2156</td>
<td>1111 (51.5%)</td>
<td>29 (2.6%)</td>
</tr>
<tr>
<td>derby</td>
<td>18892</td>
<td>8188 (43.3%)</td>
<td>1805 (22.0%)</td>
</tr>
<tr>
<td>eclipse</td>
<td>9591</td>
<td>5076 (52.9%)</td>
<td>788 (15.5%)</td>
</tr>
<tr>
<td>hsqldb</td>
<td>8140</td>
<td>3555 (43.7%)</td>
<td>585 (16.5%)</td>
</tr>
<tr>
<td>jython</td>
<td>11369</td>
<td>5382 (47.3%)</td>
<td>1337 (24.8%)</td>
</tr>
<tr>
<td>luindex</td>
<td>5835</td>
<td>2999 (51.4%)</td>
<td>867 (28.9%)</td>
</tr>
<tr>
<td>lusearch</td>
<td>5746</td>
<td>2934 (51.1%)</td>
<td>498 (17.0%)</td>
</tr>
<tr>
<td>pmd</td>
<td>9388</td>
<td>4573 (48.7%)</td>
<td>1059 (23.2%)</td>
</tr>
<tr>
<td>xalan</td>
<td>2596</td>
<td>1370 (52.8%)</td>
<td>35 (2.6%)</td>
</tr>
<tr>
<td>Total</td>
<td>190301</td>
<td>88512 (46.5%)</td>
<td>13371 (15.1%)</td>
</tr>
</tbody>
</table>

mark. However, the results from a measurement do not always correspond to the notion of VM startup. In industry, it is common to employ widely used programs, such as Eclipse (a popular IDE written in Java) or Tomcat (a Java application server implementing Java Servlet and Java Server Pages), run to a pre-determined state of program initialization as a startup benchmarks [61]. Using Eclipse as a startup benchmark would execute the Eclipse IDE until the work bench has completed initialization. For Tomcat, the benchmark would run until the Tomcat server was fully initialized and ready to serve Java applets without actually serving any clients.

To better understand the nature of these startup workloads, Table 6.1 presents a table which summarizes the startup runs of Eclipse, Tomcat and the DaCapo
benchmark suite using an iteration count of 1. The table reports the number of methods loaded, the number of different methods executed and the ratio of executed to loaded, and the number of methods that got optimized by the JIT compiler and the ratio of optimized to executed. The table shows that less than 50%, on average for Tomcat and Eclipse, of the methods loaded by the VM are actually executed and less than 10% of the methods executed are compiled by the JIT compiler.

The DaCapo benchmarks show similar results for the executed to loaded ratio, but they have a much higher average of compiled to executed methods. This suggests that the DaCapo benchmarks are good, but not ideal, representatives of real world programs with respect to startup workload. The higher percentage of compiled methods is not surprising because each of the DaCapo benchmarks runs a significant non-startup workload for each iteration of the benchmark, which gives the JIT compiler many more hot methods to compile.

6.3 Benchmarks

This section describes the benchmark programs that are executed by the supported benchmark harnesses. Section 6.3.1 introduces our suite of micro benchmarks, and Section 6.3.2 discusses the suite of macro benchmarks we use.

6.3.1 Micro Benchmarks

Our suite of micro benchmarks is designed to allow the assessment of different language features in as much isolation as possible.

Micro benchmarks by definition are synthetic. Although they could be small snippets of existing programs which use a desired feature, the benchmark will always isolate that small part of the program code to run separately from the rest of the program. These small program bits are nearly always run in a tight loop, because it is typical that one execution of a feature is too fast to take a meaningful measurement. The cost of the isolated feature execution can be measured to be the total cost of many iterations of the tight loop divided by the number of iterations.

Our micro benchmark suite is a mixture of auto-generated and hand-tuned benchmarks. The majority of the benchmarks have the same layout and class structure and vary only in the specific code that appears in their inner loops.
The class structure of each of the benchmarks is shown in Figure 6.1. We have a simple class hierarchy of three classes, and one or more aspects depending on the benchmark.

The code in both TestClass and TestClass2 is unchanged among benchmarks and is shown in Figure 6.2. Note that TestClass is the entry point to the benchmark, and it always calls the test method, an inherited method from BaseClass, with a parameter specifying the number of times the inner loop should be executed.

The layout of BaseClass is shown in Figure 6.3. This class varies between benchmarks in two places denoted as $methodDecl$ and $shadow$ in the listing.

Finally, the layout of TestAspect is shown in Figure 6.4. This aspect varies between benchmarks in four places: $aspectScope$, $adviceHeader$, $pointcut$, and $counterRef$. These places control the aspect instantiation,
```java
public class BaseClass {
    private int dummyCounter = 0;
    public int counter = 0;
    public static boolean testSane;

    public void test(int numRepeats) {
        for (int i = 0; i < numRepeats; i++) {
            System.out.println("shadow ");
            System.out.println("shadow [20 times]");
        }
    }

    Figure 6.3: Micro benchmark BaseClass listing. $methodDecl$ and $shadow$ denote substitutions which vary among different micro benchmarks.
```

```java
public aspect TestAspect $aspectScope {
    private int counter = 0;

    static {
        System.out.println("\n");
    }

    $adviceHeader: $pointcut {
        $counterRef += 2000000014;
        if ($counterRef == 1) BaseClass.testSane = false;
    }

    Figure 6.4: Micro benchmark TestAspect listing. $aspectScope$, $adviceHeader$, $pointcut$, and $counterRef$ vary among benchmarks.
```
public class BaseClass {
    private int dummyCounter = 0;
    public int counter = 0;
    public static boolean testSane;

    public void method1() {
        dummyCounter += 2000000014;
        if (dummyCounter == 1) testSane = false;
    }

    public void test(int numRepeats) {
        for (int i = 0; i < numRepeats; i++) {
            method1();
            ... [repeat 20 times]
            method1();
        }
    }

    public aspect TestAspect perthis(call(* BaseClass.method1(..))) {
        private int counter = 0;

        before(): call(* BaseClass.method1(..)) {
            counter += 2000000014;
            if (counter == 1) BaseClass.testSane = false;
        }
    }
}

Figure 6.5: Call Micro Benchmark w/ perthis Instantiation

Controlling these six parameters between benchmarks allows us to generate a variety of benchmarks simulating many combinations of AspectJ features. We present two concrete examples of the generated benchmarks before covering the exact cross product of benchmarks we are able to generate.

The first concrete example is shown in Figure 6.5. This is a simple benchmark loop which calls the generated method method1 twenty times in a loop. The method body does a simple counter increment and test (the necessity of this method body is discussed in more detail below), and the pointcut in TestAspect statically matches each of the twenty method calls with a before advice. Finally, the aspect is declared to use perthis object instantiation which declares that there should be a distinct aspect instance for each participating this object. In this case, this means there should be one aspect instance for each callee object of method1.
This benchmark can capture the cost of static call pointcut matching, as well as the cost of a simple before advice together with the cost of retrieving a perthis aspect instance. A similar benchmark without the perthis clause would capture the cost of a single aspect instance retrieval instead.

Note that both the participating method and advice have similar bodies: they both increment and test a counter. Because micro benchmarks are designed to measure the cost of specific features in isolation, the amount of work done in bodies like these must be minimized. However, they cannot be removed. The necessity of these specific bodies is discussed below.

The second example is shown in Figure 6.6. This example shows the use of a cflow pointcut which captures the calling object at the point of entry into the control flow. This capture enforces that the point of entry must be a BaseClass object.

Like the first example, both participating method and advice bodies update and check counters. However, the body of the advice updates and checks a counter on the captured object rather than itself. This example shows that by varying the six parameters, we can produce sophisticated yet targeted micro benchmarks.

Benchmark generation is separated into four categories because of distinct requirements for different parts of the AspectJ language. The categories are:

- **trivial** benchmarks are designed to capture simple statically determinable pointcut and advice on simple shadows. The main point of variance here is the type of aspect instantiation.

- **state** benchmarks are designed to exercise the dynamic pointcut matching and join point value capture.

- **cflow** benchmarks specifically exercise different aspects of the cflow pointcut, which has a complex set of features, and is non trivial to implement.

- **around** benchmarks measure around advice specifically. This is also separated because of its complex options and implementation.

Each of these categories of benchmarks varies along a number of dimensions. By varying each of these parameters we can generate a cross product of different
public class BaseClass {
    private int dummyCounter = 0;
    public int counter = 0;
    public static boolean testSane;

    public void method1() {
        dummyCounter += 2000000014;
        if (dummyCounter == 1) testSane = false;
    }

    public void method0(int d) {
        if (d <= 1) method1();
        else method0(d-1);
        dummyCounter += 2000000014;
        if (dummyCounter == 1) testSane = false;
    }

    public void test(int numRepeats) {
        for (int i = 0; i < numRepeats; i++) {
            method0(1);
            [repeat 20 times]
            method0(1);
        }
    }

    public aspect TestAspect {
        private int counter = 0;

        before(BaseClass c): 
            call(* BaseClass.method1(..)) &&
            cflow(execution(* BaseClass.method0(..))
            && this(TestClass)
            && this(c)) {
                c.counter += 2000000014;
                if (c.counter == 1) BaseClass.testSane = false;
            }
    }
}

Figure 6.6: Cflow Micro Benchmark w/ State Capture
<table>
<thead>
<tr>
<th>Category</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trivial</td>
<td>advice</td>
<td>This parameter controls the type of advice that is used in the aspect. In this category, the available options are before or afterReturning.</td>
</tr>
<tr>
<td></td>
<td>pointcut</td>
<td>Whether to use a call or execution pointcut.</td>
</tr>
<tr>
<td></td>
<td>afterRetShadow</td>
<td>Whether the returning shadow should be a staticMatch, dynamicMatch or dynamicNoMatch.</td>
</tr>
<tr>
<td></td>
<td>afterRetValue</td>
<td>Whether the return value is captured and used in the advice body.</td>
</tr>
<tr>
<td></td>
<td>perThis</td>
<td>Whether to use perThis instantiation or not.</td>
</tr>
<tr>
<td>State</td>
<td>pointcut</td>
<td>Specify this, target or args pointcut.</td>
</tr>
<tr>
<td></td>
<td>shadow</td>
<td>Whether the shadow should statically or dynamically match.</td>
</tr>
<tr>
<td></td>
<td>value</td>
<td>Controls whether the value is captured and used in the advice body.</td>
</tr>
<tr>
<td>Cflow</td>
<td>in-shadow</td>
<td>Whether the cflow test should statically or dynamically match.</td>
</tr>
<tr>
<td></td>
<td>out-shadow</td>
<td>Whether the cflow boundary should statically or dynamically match.</td>
</tr>
<tr>
<td></td>
<td>in-context</td>
<td>Whether to capture the this object from the cflow boundary.</td>
</tr>
<tr>
<td>Around</td>
<td>modParam</td>
<td>Whether or not to modify the around parameter.</td>
</tr>
<tr>
<td></td>
<td>modRetVal</td>
<td>Whether or not to modify the return value.</td>
</tr>
<tr>
<td></td>
<td>twoProceeds</td>
<td>Whether to introduce a second call to proceed().</td>
</tr>
<tr>
<td></td>
<td>adviceCnt</td>
<td>The number of copies of the same advice to simulate multiple around advice on the same join point.</td>
</tr>
</tbody>
</table>

Table 6.2: Micro Benchmark Generation Control Parameters
features. The dimensions for each of these categories of benchmarks is described in Table 6.2.

A perl script, together with template files for the basic layout, and a specification file detailing exactly which parameters and values should be used, controls the generation of all of these benchmarks. Because there are so many parameters, this process generates a significant (50+) number of benchmarks if every option is exercised.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>staticLoad</td>
<td>This benchmark has no workload in the benchmark itself. It is designed to be used to measured how long it takes the VM to startup and shutdown with no appreciable amount of other work. Results produced using this benchmark measure the the cost of starting up and initializing the VM itself. This benchmark also omits any aspect definition file, so that runtime weaving approaches have the ability to short circuit much of their framework.</td>
</tr>
<tr>
<td>staticLoadWithInfr</td>
<td>This benchmark also has no workload in the benchmark itself, but it does include an aspect definition file, although the file does not specify any aspects to be loaded. This forces runtime weaving implementations to load and execute at least part of their infrastructure for dealing with aspects.</td>
</tr>
</tbody>
</table>
1000MethodsLoaded This benchmark instantiates 100 essentially identical classes containing 10 methods, each of which contains 100 method calls. Only one method of one of the classes is executed for each iteration of the benchmark. This benchmark is designed to measure the cost of loading classes without executing those classes by forcing the loading of a lot of bytecode that is never executed by the VM.

500MethodsExec This benchmark is the same as 1000MethodsLoaded, except that instead of executing just one method from one class per benchmark iteration, 50% of the loaded methods are executed during each iteration. This benchmark is designed to execute around half of the loaded code to mimic the class loading characteristics of common startup macro benchmarks from Table 6.1.

interp* These benchmarks were created by Naseer [63] to specifically evaluate his interpreter-based advice weaving implementation. They measure the cost of unadvised method execution as well as statically matched and residually guarded advice.

| Table 6.3: Hand Coded Micro Benchmarks |

However, auto-generation of benchmarks does not generate a benchmark for every desired measurement. For these additional cases, we have a number of hand generated benchmarks to meet specific measurement requirements. Most of the hand generated benchmarks vary the details of the method and advice body rather than the pointcut and advice parameters to create new micro benchmarks. These benchmarks are described in Table 6.3. |
// Workload variables
private int dummyCounter = 0;
public static boolean testSane;

// Workload
dummyCounter += 2000000014;
if (dummyCounter == 1) testSane = false;

Figure 6.7: Standard Unoptimizable Benchmark Workload

The standard body of each method and advice used above constitutes a workload for every iteration of the benchmark. Since this workload is uninteresting with respect to what we are trying to measure (specific AspectJ features), we want the workload to be as minimal as possible.

However, because modern optimizing JIT compilers contain sophisticated code analysis techniques, it is not possible to leave these workloads empty. If the method and advice bodies were left empty, a JIT compiler could detect this and remove the method and advice body calls entirely from execution. Removing these would result in the measurement reflecting only the cost of an empty loop, which could also very well be optimized away.

Even if the workload were simple constant counter increments, after inlining, a compiler could coalesce multiple increments into one large addition\(^2\), effectively reducing the number of body executions by a factor determined by the number of additions coalesced. Both of these possibilities are within reach of modern optimizing compilers, and both of them present problems for measuring the actual overhead of implementing a specific feature.

So, as shown above, as a standard workload we use the code snippet shown in Figure 6.7. Each execution of the workload increments a counter by a large even number. A large constant is chosen to make it difficult for the value of the counter to be bounded, and because the constant is even, we know that the counter will never hold a value of 1. The constant is also chosen to not be divisible by a large power of 2 to avoid the dummyCounter from cycling through a small number of values as it overflows.

\(^{2}\)This optimization was determined to be performed on the Sun Hotspot VM running in server mode.
The increment always appears together with the if statement, ensuring that no increments will be next to each other after inlining. This means that unless the compiler can optimize away the if statement, it will be unable to coalesce nearby counter increments into a single addition. Since the dummyCounter is stored in a field rather than as a local variable, it is difficult for a compiler to prove that the field will never be odd and remove the if.

As a final note, the object containing the counter that is manipulated also varies between benchmarks. Figure 6.5 shows a benchmark in which the counters are fields in the participating class or aspect. However, in benchmarks which capture state such as in the second example, Figure 6.6 or benchmarks in the “state” set, the counter used in the advice body is a field of the object captured. This usage ensures that the compiler cannot optimize away state capture after having determined the captured object is never used.

### 6.3.2 Macro Benchmarks

Our suite of macro benchmarks is designed to contain a number of benchmarks which represent or are real world programs. These benchmarks should vary in their language feature usage and workload, and they should be realistic in terms of their idiomatic feature use. In other words, they should reflect the programs that programmers actually write.

Because of these requirements, the macro benchmarks in our suite have all been created by other implementers, and are modified only to run in our benchmark harness to take measurements.

This amalgamation of benchmarks consists of the suite of abc AspectJ benchmarks [40], which itself is assembled from a variety of sources, and the DaCapo benchmark suite [14], which we can run in a variety of ways by including aspects that are matching, non-matching, and empty.

These different builds of the DaCapo suite allow us to measure different parts of the infrastructure supporting AspectJ. For example, running the plain benchmark suite allows us to compare different AspectJ implementations when no AspectJ features are used; this is an important measurement to determine the costs of the AspectJ infrastructure even if running plain Java programs. Including match-
ing and non-matching pointcuts and advice allows us to measure the cost of the supporting infrastructure when AspectJ features are used.

6.4 Limitations

Our benchmark suite suffers from several standard limitations. First, there is a lack of access to AspectJ applications that are used in practice. Such AspectJ programs would fill a hole in the macro benchmark suite by providing benchmarks with varied, real-world workloads and idiomatic use of the language in normal programs. Modifying previously written Java programs to include aspects the way we have has the danger of creating non-idiomatic AspectJ programs which do not represent the class of programs we are attempting to measure. This lack is especially important when attempting to measure startup performance of an implementation.

Second, as discussed in more detail above, we have used many synthetic benchmarks. And, although this is standard practice, it is important to keep in mind that synthetic benchmarks do not necessarily represent standard program workloads. For micro benchmarks it makes sense to isolate specific language features in a synthetic benchmark in this way, and the results are interpreted in terms of this limitation. However, synthetic macro benchmarks can be more limited. We partially address this concern by creating synthetic benchmarks or using alternate benchmarks which mimic several important characteristics of the ideal.

Finally, we are also constrained by the implementations and VMS we want to measure and compare. For example, most of our comparisons are using the Jikes RVM as a base VM, but it is not a feature complete implementation: it doesn’t support some graphical user interface (GUI) libraries and some threading libraries. Because of this, the Jikes RVM can run neither the Eclipse GUI nor the Tomcat application server, so we are unable to use these ideal startup benchmarks and must use an alternative set which mimics them (the DaCapo suite).

6.5 Summary

Benchmarking modern runtime environments is a difficult task because of the complexity of the environments and the number of trade-offs each environment may
make. To address this complexity, a benchmark suite must present a whole picture characterizing the performance of a system that it measures. This picture must include not only steady state performance analysis, but also startup performance analysis.

Furthermore, it is important to use a statistically robust technique to take measurements from the benchmarks to ensure that random timing effects present in the environment do not distort the final results.
Chapter 7

Analysis

The previous chapter discussed a suite of benchmarks which covers a wide range of strategies for measuring specific features of an AspectJ implementation. In this section we talk about using these measurement strategies to provide a whole-picture analysis of the performance of an implementation.

Providing a whole picture performance analysis that can be used to compare different implementation strategies is difficult for the same reasons that measuring specific features of an implementation is: AspectJ programs run in a sophisticated managed runtime environment. This environment interacts with the underlying OS and becomes a source of non-determinism between executions of the same program.

Furthermore, the runtime is designed to balance a number of trade-offs in time and space efficiency. Most modern VMS contain both an interpreter for initial execution and optimizing compiler which is conditionally used to improve the performance of specific parts of a program. These two systems attempt balance startup and long running, steady state performance.

So, an analysis has to consider these different facets of a runtime system and how that runtime system is used. The generated code quality of the JIT compiler can be measured by a steady state analysis using both macro and micro benchmarks because a steady state analysis will ensure that the code being evaluated has progressed through the JIT compiler and it will eliminate extraneous startup and GC costs from the measurement. However, such an analysis does not take into
consideration the impact that the JIT compiler has on the overall performance of a program.

A change in the JIT compiler which drastically improves the quality of the generated code could have a very high overhead during its compilation. Since JIT compilation is part of the high cost of program startup, this change could have a negative impact on the startup of a program even if it improves the quality of the code in the long run. Both of these sides must be reported to analyze whether or not the change to the JIT compiler makes an appropriate trade-off.

In this chapter we present a whole picture analysis, using the benchmark suite discussed in the previous chapter, of implementations of our AspectJ architecture and other approaches where appropriate. Our analysis demonstrates the design trade-offs between different implementation strategies of AspectJ, both allowed and disallowed by our proposed architecture.

Furthermore, our analysis suggests a definitive ideal implementation strategy for our architecture which reduces the problem of AspectJ optimization to that of Java optimization: AspectJ program startup time is reduced to near standard Java program startup, memory consumption is minimal and proportional to AspectJ feature usage, and the macro optimization of major areas of the implementation removes the majority of the additional overhead of AspectJ specific language features.

7.1 Start up

7.1.1 Results

We group our startup results into four categories of benchmarks to measure different facets of program startup over five different implementations. We measure both our class- and method-level in-VM weavers (ajvmv3) against the ajc compile time weaver (ajc), the ajc VM-external load-time weaver (ajc ltw), and Naseer’s interpreter dispatch implementation (Section 3.2.9).

The first group of benchmarks are the macro benchmarks intended to measure overall application startup performance. As discussed in Section 6.4, we are limited in our choice of benchmarks because of the practical constraints of which are sup-
ported on the Jikes RVM. Since neither Tomcat nor Eclipse with the GUI enabled will run on the Jikes RVM 3.1.0, we use all of the DaCapo benchmarks (dacapo-9.10-beta0) that do run on the Jikes RVM 3.1.0 as our macro benchmarks.\footnote{The DaCapo suite does contain an Eclipse benchmark, but because it does not use the GUI, this benchmark is supported by the Jikes RVM 3.1.0.} These DaCapo macro benchmarks are run in two ways. The first runs the suite in the default way: the benchmarks remain untouched. The second does another run of the suite but the run contains one configured aspect. The aspect contains a pointcut and advice that do not apply anywhere in the benchmarks. This has the result of not affecting the steady state performance of the benchmarks, because no extra functionality is provided, but it forces the implementations to perform some pointcut matching.

The next three groups of benchmarks are micro benchmarks designed to measure different startup characteristics of the implementations.

The *basic startup cost benchmarks* group consists of four of the hand-created micro benchmarks (see Section 6.3.1): staticLoad, staticLoadWithInfr, 1000MethodsLoaded, and 500MethodsExec.

The *advice execution benchmarks* measure the execution of several combinations of pointcuts and advice involving both statically matched and residually tested advice. These benchmarks are a subset of the auto-generated micro benchmarks (Section 6.3.1).

Finally, the *special advice execution benchmarks* are intended to support comparisons to Naseer’s interpreter dispatch implementation (Section 3.2.9). These are the interp* hand generated micro benchmarks. Because Naseer’s implementation does not support the JIT compiler, the compiler is disabled when taking measurements of all of the implementations for those special benchmarks so that comparisons against this implementation are fair. Measurements of implementations with the JIT disabled are labeled as such.

All implementations use the Jikes RVM 3.1.0 except for Naseer’s implementation, which is based on the Jikes RVM 2.9.2. We use aic 1.5.2, configured to compile the programs in the load time weaving format, as the frontend compiler for all runtime implementations. Further, all of the Jikes RVM builds use the “production” build configuration and the default value of 1 virtual CPU. The results
were obtained using a 3 Ghz dual core Intel Pentium 4 machine with 1GB of memory running Ubuntu 9.10 with kernel 2.6.31.

Figure 7.1 and Figure 7.2 show the results of the macro benchmarks for four implementations: ajc, ajc ltw, and both of our in-VM weaving implementations, labeled as “ajvm - class level” and “ajvm - method level.” These implementations are further run both with and without TPTS enabled; runs without TPTS are annotated with (no tpts). Error bars show the 95% confidence interval for that set of results. The y-axis shows the wall clock running time of the benchmark, and the x-axis groups results by benchmark.

Figure 7.1 shows that in most cases, simply having a runtime advice weaving implementation available does not slow down the startup performance of pure Java because the results show no statistically significance differences.

Figure 7.2 shows the run containing an aspect with one non-applicable advice. The results here highlight the cost paid by each weaving implementation based on the amount of code each must scan to prove that the advice does not apply. This run is more representative of what happens in a program containing advice, and the results show a 1.07x (eclipse) to 1.5x (luindex) cost paid by ajc ltw over the
method level approach. The results show a similar cost when using class level over method level weaving. Recall, however, that the DaCapo benchmarks have a higher percentage of their executed methods optimized than Eclipse or Tomcat during startup, suggesting that these benchmarks have a higher workload which can amortize the differences in startup speed over longer executions.

Figure 7.3 shows an unfolding performance graph of the 1000MethodsLoaded benchmark from the basic startup benchmark group. This benchmark includes all six runtime weaving approaches as well as static ajc, and it is run with the JIT compiler disabled because we include Naseer’s interpreter-based implementation. The x-axis is the number of iterations of the benchmark loop that are executed, and the y-axis is the wall clock running time from VM startup to benchmark completion. Since we do not measure every single iteration individually, actual measured values are marked on the graph with either \( x, +, *, o \) or \( \triangleright \), depending on the implementation. Values between measured points are linearly interpolated.

From this graph we can see not only the startup performance between the implementations when the iteration count is 1, but we can also see the performance of each of the implementations over time compared to one another. The results
Figure 7.3: Unfolding Performance of 1000MethodsLoaded w/ JIT compiler disabled. The legend is annotated with the actual slope value, and the slope relative to ajc in parenthesis.

of these benchmarks are unsurprising in that ajc ltw takes $\sim 7.5s$ (26x) longer to startup than the method level weaver, and class level takes $\sim 0.8s$ (4x) longer to startup than method level when both granularities use TPTS. With TPTS disabled, pointcut matching takes longer, so the difference between the two granularities are magnified. The interpreter dispatch and method level weaver are the quickest to startup. Furthermore, the slopes of the curves, representing performance over time, is relatively the same for all bytecode based weavers because all of them have the opportunity to generate much the same woven bytecode; the interpreter dispatch implementation has a much steeper slope because each join point execution is much more expensive due to repeated pointcut matching and dynamic dispatch.

The majority of the micro benchmark runs follow the same pattern as in Figure 7.3; the final slope of the curves for each implementation measured is the same (with noted exceptions), even though the overall shape and slope of the curves differ between benchmarks. For that reason, instead of graphing each, we summarize
We present a series of bar graphs for each of the benchmark groups like the bar graph shown for the DaCapo results in Figure 7.1. These bar graphs are shown in Figure 7.4 through Figure 7.6. We can use these types of graphs as a summary since, as shown above, the variability for most of the benchmarks is captured in the first iteration. Note for the benchmarks in Figure 7.5 and Figure 7.6 that since nearly all loaded code is executed, we do not see a difference in the performance of class and method level in-VM weaving.

All benchmarks but two showed statistically indistinguishable slopes because of the similarity of their unfolding performance. The exceptions are the cflow benchmarks, and more subtly, some of the state capture benchmarks in which our two in-VM approaches out-perform ajc ltw. These improvements can be accounted for by the differences in the bytecode generated between implementations in the case of state capture, and the steady state optimizations available to in-VM implementations in the case of cflow discussed in more detail below.
Figure 7.5: Advice Execution Benchmarks

Figure 7.6: Special Advice Execution Benchmarks
7.1.2 Analysis

The plain static startup benchmarks shown in Figure 7.4 and the macro benchmarks in Figure 7.1 show that simply having a late binding advice weaving implementation available does not slow down pure Java programs. This is because when no aspects are present, much of the weaving infrastructure can be short-circuited.

The macro benchmarks that contain a single non-matching advice shown in Figure 7.2 show that when there is advice in a program that may apply to any bytecode, the speed of class loading is impacted when the bytecode needs to be scanned. Specifically, ajc ltw is impacted more than the others.

The 500MethodsExec and 1000MethodsLoaded results in Figure 7.4 show the relative performance costs when we micromeasure class loading. These results are interesting because the benchmarks differ only in the number of methods they execute per iteration of the benchmark. Recall that 500MethodsExec executes 50% of the loaded methods, where 1000MethodsLoaded executes 0.1% of the loaded methods. For all implementations except the method level weavers, the performance between the two benchmarks is similar. This is because the class level weavers must process every loaded method whether it is executed or not. So even when the benchmark executes only 0.1% of the loaded methods, these weavers must weave all of the code, and we can see large gains by the method level weavers.

The 500MethodsExec benchmark, which forces weaving of 50% of the loaded methods, highlights the benefits of using additional optimizations like TPTS. Both the ajvm class level and method level weavers are improved significantly when they are configured to use TPTS.

The results in Figure 7.6 show that there is no statistically significant difference in startup between the interpreter weaver and in-VM bytecode weavers. However, Figure 7.3 shows that interpreter dispatch has a significantly steeper slope, and hence higher cost, when repeatedly executing each benchmark. However, in a full hybrid implementation, depending on the execution profile of the program, this overhead is likely to be minimal as a method which is frequently executed will be recompiled by the JIT compiler, relaxing the slope significantly after that point. The slope would only remain steep if the program continued to execute new methods the longer it ran, resulting in no methods being recompiled. However, this
is unlikely to happen because a finite number of methods would require repeated execution in a long running program.

These results taken together suggest that VM-external implementations, although necessary in some cases, have performance limitations when performing runtime weaving for late binding of advice. These limitations are inherent to the fact that a VM-external weaver must duplicate type structures it is unable to access in the VM, and to the fact that it is limited to the class-level granularity of weaving. However, there is an additional cost paid by ajc due to its use of a bytecode manipulation toolkit that is not inherent to VM-external implementations.

Comparing our implementation of in-VM class level weaving to the ajc load time weaver quantifies the cost that ajc ltw pays duplicating in-VM structures and using a convenient but expensive bytecode manipulation toolkit. Further comparing our class level to our method level weavers quantifies the cost that must be paid by a VM-external weaver due to its restriction to class level granularity because our class-level implementation still shows performance degradation over our method level weaver and over interpreter dispatch.

Although the interpreter dispatch implementation does not support the JIT compiler, preventing us from making a more robust comparison, our micro benchmarks show that first run benchmarks startup as quickly using interpreter dispatch as method level weaving. However, interpreter dispatch pays a significant price in runtime code execution performance.

7.2 Steady State

7.2.1 Results

We report a subset of the micro and macro benchmarks described in the previous chapter using the steady state measurement harness described in Section 6.2.1. This set of results spans 5 different implementations, although not every benchmark is run on every implementation.

The implementations are ajc, abc, ajvmv2, Steamloom (all run on or are based on the Jikes RVM), and ajc on J9 and ajc on J9 plus eflow optimizations. We use ajvmv2 in these results because the ACLv1, although slow in startup, implements
the entire AspectJ language allowing us to evaluate more thoroughly our steady state optimizations. See Section 5.6 for more details.

The results from ajc and abc provide a benchmark of steady state performance for implementations that cannot be integrated with the VM: they are constrained to bytecode based weaving with no extra support. ajvm is our implementation of our proposed architecture including the JIT level and runtime library support described in Section 5.4.1. The results from this implementation show what is possible with a VM-integrated approach and steady state optimizations. Steamloom (see Section 3.2.7) does not implement AspectJ so it is not compatible with our benchmarks. However, many of its constructs are similar to AspectJ, and so we ported a number of the benchmarks to Steamloom’s pointcut and advice language to draw comparisons against an alternate VM-integrated approach.

Finally, we report on ajc running on an unmodified J9 and compare those results against ajc running our implementation of the cflow optimizations on J9 (see: Section 3.2.12).

Each of these implementation measurements are constrained to the benchmarks that they support.

It is important to account for the fact that the results of a measurement is in general only an approximation to the true value of the quantity being measured (running time in this case). This is especially important when benchmarking managed languages such as Java because they have extra sources of non-determinism which reduce the reliability of timing measurements. Fortunately, there are well known statistical methods for dealing with measurement error. In particular, we use confidence intervals to determine whether the difference in our measurements of two implementations is likely to represent a difference in their true values, that is, whether the difference is statistically significant. Note that in some cases, there are statistically significant differences that turn out to have little practical significance because they are so small.

Recall from the previous chapter that because the benchmark running times are averages over multiple invocations, and because each invocation is independent, we can compute confidence intervals for the means. We use a confidence level of 95%. If the confidence intervals for two implementations overlap, we conclude, at the 95% confidence level, that there is no statistically significant difference between
them because there is a significant probability that the observed difference is due to random effects.

Our implementation, ajvm, is based upon the Jikes RVM 2.9.0.1 and we use ajc v1.5.3 and abc v1.2.1 both running on the Jikes RVM 2.9.0.1 as points of comparison. We also compare against Steamloom v0.6 which is based on the Jikes RVM v2.3.1. All VM implementations were compiled using the “production” build configuration and are run with the default of one virtual CPU. Furthermore, given the discussion in Section 6.1.2, we ran the benchmarks using a heap size of 256MB.

The results were obtained on a 3Ghz dual core Intel Pentium 4 machine with 1GB of memory running Ubuntu Linux 7.04 with kernel 2.6.20.

The results of these measurements are reported in Figure 7.7 through Figure 7.12. We group the results of the micro-benchmarks into the four categories in Section 6.3.1: around, cflow, state and trivial, and for convenience we label the benchmarks concisely by the features that benchmark exercises.

The simplest around benchmark contains an around advice which increments a counter and then calls proceed; we refer to this one as ‘Simple.’ The next three benchmarks in this category are variants of Simple which either contain two calls to proceed, modify the advice parameter before calling proceed, or do work after calling proceed. These are named ‘TwoProceeds,’ ‘ModParam,’ and ‘WorkAfterProceed,’ respectively. Finally, the ’ClosureFix’ around benchmark tests a fix to ajc’s implementation of around advice that could be considered a performance bug. It consists of a method m1 which increments a counter, a loop which calls m1 repeatedly, and an around advice which matches calls to m1 and also calls m1 again in its body. An if pointcut and boolean variable are used to ensure that the advice will dynamically fail to match the call to m1 in the advice body.

The cflow benchmarks contain a before advice with a cflow pointcut whose inner (boundary) pointcut may match statically (abbreviated as ‘SMatch’), match dynamically (‘DMatch’), or fail dynamically (‘NoMatch’). The dependent pointcut always matches statically. These benchmarks also vary based on whether the cflow pointcut exposes context, indicated by appending ’Context’ to the benchmark name.

The state benchmarks have a before advice with a call pointcut and a state pointcut (this, target, or args) which may match statically, match dynam-
Figure 7.7: Around Microbenchmarks

ically, or fail dynamically. They vary based on the pointcut used and whether or not it exposes a value. These are referred to as 'SMatchThis,' 'SMatchThisValue,' 'SMatchTarget,' and so on.

The trivial benchmarks contain either before or after returning advice and a call pointcut or execution pointcut. For after returning advice, the pointcut may match statically, match dynamically, or fail dynamically, depending on the type of the value returned. These are abbreviated with names such as 'DMatchAfterCall.' In addition, 'PerThis' is appended to the name if the aspect instance model is perthis rather than persingleton.

The macro-benchmarks are labeled with their names and are a subset of the AspectJ benchmarks from the abc group (see Section 6.3.2). Two of these benchmarks, figure and quicksort-gregor, were ported to Steamloom.

Figure 7.7 presents the results of the around benchmarks. On these benchmarks, ajvm is never slower than ajc and is slightly faster when the around advice does work after calling proceed. Except for the case where there are two calls to proceed and ClosureFix, abc is slightly faster than ajvm because of its around advice optimizations (see Section 3.2.3). However, if abc were used as the front end...
As seen in Figure 7.8, ajvm is dramatically faster than ajc in its implementation of cflow. This improvement is a result of the cflow optimizations supported by our implementation. While ajvm outperforms abc in some cases, it should be noted that abc’s expensive inter-procedural cflow optimizations were not enabled. If they had been, it could have been faster, however, as previously mentioned, those optimizations could easily be incorporated into our architecture by extending abc to provide appropriate meta-data. abc outperforms ajc because it optimizes cflow operations for the main application thread, thereby avoiding the overhead of using ThreadLocal for that specific thread. Furthermore, we see that Steamloom’s more JIT-integrated cflow optimizations further improve upon the performance of ajvm as expected.

Figure 7.9 presents the results of the trivial benchmarks. We make the follow-
Running Time (s)

Figure 7.9: Trivial Microbenchmarks

Observations:

- **ajvm** performs significantly better than **ajc** and **abc** when the aspect instantiation model is *perthis* whenever there is a match. This result is because of the *perthis* optimizations described in Section 4.4.2, which make **VM-supported aspect instance retrieval** more efficient.

- **ajvm** is faster than **ajc** on **DMatchAfterExecution** and **SMatchAfterExecution**. This could be attributed to the advice dispatch optimizations (Section 4.4.2), however **ajvm** also performs better on the **NoMatchAfterExecution** benchmark which has no advice dispatch in it.

- **ajvm** is never out-performed by **ajc**. This is important because these two implementations generate similar bytecode during weaving. So, differences between **ajvm** and **ajc** tend to represent differences caused by optimizations rather than bytecode differences.

- **Steamloom** is significantly slower than **ajc** and **ajvm** on both **DMatchAfter**-
Figure 7.10: State Capture Microbenchmarks

Call, NoMatchAfterCall, and SMatchAfterCall, but is significantly faster than all the implementations on DMatchAfterExecution, NoMatchAfterExecution, and SMatchAfterExecution. We were unable to determine the cause of these performance characteristics. However, it appears as though Steamloom is inlining more aggressively in some cases because it generates some intermediate code that is twice as long as ajvm’s in DMatchAfterExecution. We believe that the differences in the the Jikes RVM version that Steamloom and ajvm are based on could account for these varying effects.

Figure 7.10 shows the results of the state benchmarks. As expected, ajvm and ajc have identical performance on these benchmarks. This is not surprising because the state benchmarks use a before advice with a call pointcut. This combination of pointcut and advice was shown to be equally well optimized by a traditional JIT compiler in the trivial benchmark suite. However, abc sees significant speed up over the other implementations when a dynamic match is involved. This result is because abc is much smarter about how it deals with extracting values from the join point shadow context than ajc is. Because ajvm generates similar bytecode to ajc, abc is also quicker than ajvm. However, there appear to be no barriers to
implementing a bytecode weaver that generates as efficient code as abc in this case.

Figure 7.11 shows the results of ajc running on the unmodified and cflow-optimized version of J9. The running time is normalized to the unmodified IBM VM. The first three results show the optimization dramatically improves the performance of the cflow benchmarks in the cases were no cflow state is captured. The final three results show standard J9 performance because our modified implementation does not support context capture, so it defaults to the traditional code path. However, the results are enough to show that this optimization is implementable in our architecture in another VM with significant results.

Finally, the results of the macro benchmarks are presented in Figure 7.12. Two of the macro benchmarks did not stabilize when running on abc; we label these benchmarks with an asterisk and omit abc’s results.

In the majority of cases, there is little or no difference between ajc and ajvm, and furthermore, ajvm is never outperformed by ajc. This confirms that the use of our architecture does not result in a steady state performance penalty. In particular, note that on the plain Java benchmarks, bean-java and figure-java, ajvm and ajc perform identically.
Of interest are the significant performance gains for ajvm on the ants, figure, quicksort-gregor, and quicksort-oege benchmarks. These occur because those benchmarks rely on cflow or cflowbelow pointcuts and so they benefit from ajvm’s integration of a much-improved cflow implementation. ajvm also slightly outperforms ajc on the cona-sim benchmark, which also uses cflow, however, the difference is smaller because it is amortized over the relatively large running time (20s) of the benchmark; abc achieves similar performance for reasons discussed above. Furthermore, Steamloom performs as expected in the two benchmarks that were ported to it: figure and quicksort-gregor.

7.2.2 Analysis

Several conclusions can be drawn from the results of our steady state benchmarks. The primary conclusion is that there are cases where VM support is required to optimize AspectJ effectively. Namely, we show that cflow and advice dispatch and aspect retrieval optimizations have a real effect on the micro and macro performance of AspectJ programs. On the other hand, however, many of our benchmark
results demonstrate equivalent performance between VM-internal and external implementations. The equivalence could be because we have not sufficiently optimized AspectJ execution, but it is more likely that modern JIT compilers already sufficiently optimize Java bytecode so that a plain bytecode rewriting technique achieves similar performance. This idea is reinforced by observing the second and third optimization passes of IBM J9 VM discussed in Section 3.2.12 which produce IR similar to our Jikes RVM optimization without additional AspectJ-specific hints (it is not possible to compare the performance directly of these implementations because they are entirely different VMs).

In other words, most of the semantics of AspectJ constructs can be expressed sufficiently well in Java bytecode that additional information is not required for optimization. So, optimizing AspectJ bytecode execution for these features can be handled by plain Java optimization.

Furthermore, our results show that our implementation, and hence our proposed architecture, do not hurt the steady state performance of the overall VM. The benchmarks show that the inclusion of a runtime weaver and VM-level optimizations perform no worse than ajc the standard VM-external implementation. In some cases, abc performs better than ajvm, but this suggests either that the bytecode produced by the runtime weaver in ajvm can be improved so that it is on par with abc when the improvement is a local code generation difference. The more expensive optimization which uses inter-procedural analysis, which is prohibitively expensive to perform at runtime, could be integrated into our frontend compiler to statically weave the aspects where it might be beneficial.

And lastly, our results from IBM’s J9 VM show both that our architecture is viable in VMs other than the Jikes RVM, and that an implementation of our architecture still benefits from known AspectJ-specific optimizations in other VMs.

7.3 Summary

Taken together, our start up and steady state analyses provide a whole picture performance evaluation of an implementation of our proposed architecture. Using the implementations we described in Chapter 5, and the benchmark suite described in Chapter 6, this analysis shows that our architecture and implementations meet our
performance claims.

The start up evaluation shows that runtime weaving for late binding of advice can be done in a way that it does not prohibitively impact the start up performance of the program. Our steady state analysis shows that our implementation never hurts the performance of the VM, and it performs significantly better than traditional approaches in cases where our AspectJ-specific optimizations apply.
Chapter 8

Conclusions

We conclude with a summary of the technical chapters of the presented research including how our claims and contributions are met by the presented work. We then examine the threats to validity and limitations of our work, followed by a discussion of possible future work.

8.1 Summary

We presented an architecture for AspectJ implementations that is integrated into the VM architecture. We also presented a comprehensive implementation and several smaller architecturally partial implementations of this architecture to demonstrate that it enables optimizations that reduce the problem of optimizing AspectJ to one of optimizing plain Java, it is both forward and backward compatible, and it is only a lightweight addition to modern VMs.

We presented the design of an AspectJ benchmark suite which we used to evaluate our architecture and implementation and other relevant AspectJ implementations. We presented a comparative analysis of our results using the benchmark suite and used it to validate the performance claims of our architecture.

We informed the construction of our architecture and implementations by presenting a map of the design space for AspectJ implementations. The design space breaks down into several areas that align with different subsystems in a modern VM architecture: front end compiler, external class loader, internal class loader, class
model, interpreter and JIT compiler. Additionally, our break down of the design space and phases of an AspectJ implementation identifies the potential advantages and disadvantages of implementing a particular phase in each part of the design space.

Our discussion of related work places other AOP implementations into this design space so as to better understand how other work relates to our own and each other. This placement also reveals parts of the design space that were previously unaddressed: an implementation in the VM-internal class loader, and an implementation in the JIT compiler, as well as new possibilities allowed by an VM-internal implementation based on the level of granularity that weaving takes place.

8.2 Claims and Contributions

We revisit our claims and contributions from Section 1.2 below. In each case we elaborate on how that claim or contribution is addressed by our work.

We start by revisiting our claims:

- Our architecture enables optimizations that reduce the problem of optimizing AspectJ to one of optimizing plain Java because:
  
  The optimizations largely exploit existing JVM infrastructure.
  
  All of our supported optimizations are lightweight additions to the existing optimization framework in the VMS. These are supported either through simple hints to the optimizing compiler, specialized high-level intermediate code generation, or VM-integrated runtime support. In all of these cases, the optimizations implemented fit neatly into the existing VM infrastructure.

- Startup time is reduced to standard Java in most cases, and near standard Java in degenerate cases.

  Our analysis in Section 7.1 shows that implementations of our architecture start up as fast as a plain Java VM running a Java program. Furthermore, our analysis also demonstrates that an integrated implementation of the architecture improves performance over a VM-external
implementation when doing full runtime weaving, and comes close to the startup of pre-woven code.

- **Additional memory consumption is linear: no expensive data structures are required.**

In Chapter 5 we argue that our implementation, ajvm, uses only an additional amount of memory proportional to the size and number of aspects, advice and pointcuts in the program. Even with the inclusion of Naseer’s TPTS for efficient pointcut matching, our memory consumption remains linear in the number of pointcuts in the program.

- **Major areas of our implementation are macro optimized: class loading, advice matching and munging, and steady state code quality.**

Our implementation demonstrates that the major areas of our architecture have been macro optimized. Method level granularity of weaving shows the largest improvement in VM startup time by reducing the number of shadows that must be matched during class loading; our implementation of Naseer’s TPTS further reduces the cost of pointcut matching by reducing the number of pointcuts that must be checked at each shadow we must check; and bytecode munging is improved over ajc ltw by avoiding the use of expensive bytecode manipulation toolkits in favour of a specific more straight forward implementation, integrated pointcut type matching with VM-internal type representation, and adding VM-internal bytecodes to avoid costly constant pool manipulation.

- **Our architecture is both forward and backward compatible.**

We demonstrate this claim in Chapter 4. Our architecture is backward compatible because it supports the execution of existing (compiled with ajc) AspectJ programs. It does not require any modifications to existing programs.

It is forward compatible because the architecture is implementable on a traditional VM using a customized external class loader and runtime library. This support is demonstrated by the fact that the ajc load time weaver implements our architecture and can run on traditional VMs. This also means that
a program can be compiled to target our architecture to reap the benefits provided by an integrated implementation, but can still be run on a traditional VM together with the ajc load time weaver.

• **Our architecture is a lightweight addition to modern VMs.**

This is also demonstrated by our architecture design in Chapter 4, but also by the existence of our integrated implementations. The largest and most potentially intrusive part of our architecture is the addition of the runtime weaver. However, several implementations of our architecture, including the ajc load time weaver and ajvm, show that the runtime weaver is both modular and very loosely coupled to the rest of the VM. The ajvm requires very few modifications to existing Jikes RVM code: only 242 lines of code were added to existing files in the Jikes RVM to implement ajvmv3, and the majority of the additional code (9959 lines of code) is contained in the weaving module.

Furthermore, our supported JIT compiler optimizations are demonstrated to be non-intrusive hints to the optimizing compiler in the best case, and small additions for specialized code generation in the worst case. These optimizations also required little VM code change.

Given that our claims have been validated, our contributions are:

• an architecture of a proposed VM-integrated AspectJ implementation;

• an implementation of the proposed architecture;

• a design of an AspectJ benchmark suite including startup and steady state measurement methodology, and an implementation of the benchmark suite with a combination of existing and novel benchmarks (joint work\(^1\)); and

• an analysis of the data produced by the benchmark suite with conclusions drawn regarding the AspectJ design space and defense of claims.

• a mapping of the AspectJ implementation design space and characterization our work and related work in terms of this mapping;

\(^{1}\)See Preface.
• several partial implementations in the Jikes RVM and J9;

Taking these claims and contributions together, we have shown a VM-integrated AspectJ implementation that can support all known AspectJ optimizations, thereby reducing the problem of optimizing AspectJ execution to that of optimizing standard Java execution. Furthermore, the architecture is such that these optimizations are straightforward to implement; it fits naturally into existing staged JVM architectures; and that it maintains forward and backward compatibility with existing AspectJ compilers and JVMs.

8.3 Threats to Validity and Limitations

• Single Implementation Although we have implemented some parts of the architecture in other VMs, the entire architecture is implemented only in the Jikes RVM. Moreover, this VM does not have an interpreter, and is not a “production-level” VM. It is possible that implementing the architecture in other VMs could cause unforeseen relative performance differences that would affect our conclusions. However, since our architecture is not tightly coupled to the Java VM architecture, and most modern JVMs use a similar overall architecture, we do not foresee any integration problems with other VMs.

• Unforeseen Optimizations It is possible that there exist optimizations for AspectJ in the Java VM that we have not anticipated and that cannot be implemented in our proposed architecture. If that is the case, it would mean that our proposed light weight architecture would require modification to be considered an optimized AspectJ implementation, because it is unable to implement all reasonable AspectJ optimizations. However, we believe that this is unlikely because our architecture allows a VM-internal implementation access to all of the AspectJ semantics via the meta data it provides. If the VM were unable to implement an optimization using this, it is likely that the constraint is imposed by the VM itself rather than our architecture.

• Inappropriate Benchmarks As discussed in several sections, benchmarking adaptive software systems like the Java VM is a difficult procedure, and it
is very difficult to narrow down the exact effect of a given change. Given this fact, it is possible that our developed benchmark suite doesn’t fully represent the extent of our changes, and the quantitative difference between our work and Ethe’s. However, we have selected a comprehensive suite that contains both benchmarks used to evaluate previous work, as well as new macro, micro, and auto-generated benchmarks to supplement the former, as well as startup and steady state benchmark harnesses.

- **Incomplete AspectJ** Our runtime bytecode weaver does not support all of the AspectJ language semantics. It is possible that a complete implementation of these semantics would affect the startup performance of our implementation. However, the missing functionality is largely repetitive semantics applied to different bytecode in the program. Additionally, the missing functionality does not present different or more difficult pointcut matching and munging hurdles, but rather has a different affect on the steady state performance of the program.

### 8.4 Future Work

There are several areas available for future exploration from our work. One primary area is the exploration of additional mechanisms for micro-optimization of both implementations of our architecture, as well as support of AspectJ.

For example, our work does not thoroughly explore micro-optimization of pointcut matching and bytecode munging. Pointcut matching optimizations tend to be of two forms: the optimization either reduces the number of pointcuts required for matching or it optimizes the pointcut matching itself. One method of optimizing pointcut matching would be to compile pointcuts into an executable code vector with constant parts of the pointcuts removed. This code vector could then be optimized by the JIT compiler the same way the rest of the Java program is optimized. Hashing of code vectors could reduce the memory requirements of compiled pointcut code.

The interpreter-based pointcut matching on its AST could be improved by a heuristic reordering of the pointcut tree. A reordering could elevate more discriminating pointcut properties higher in the AST representation to cause those properties
to be matched before others. Checking more discriminating properties first could cause the matching process to fail faster than if those properties were checked later.

Finally, bytecode munging could be improved by further optimizing our generation procedure. We used a relatively naive data structure and algorithm for doing bytecode munging on methods. Although our experiments show that this approach is relatively efficient, it should be possible to optimize this implementation to reduce the number of passes over the bytecode required. It may be possible to reduce the number of passes to two. The first pass would identify branch instructions and match against all shadows. After this phase the VM would have computed all matching advice and which shadows they apply to. It could use this information to pre-compute the size of the woven method, as well as compute new offsets for branch instructions. Finally, it could then use this information to generate a new bytecode array of exactly the right size, which would take one pass to populate.

The effect that these types of micro optimization would have on overall program performance would also need to be evaluated.
Bibliography


→ pages 50

→ pages 35, 47

→ pages 47, 67, 79

→ pages 44

→ pages 26, 54

→ pages 98, 103


[37] A. Goldberg and D. Robson. Smalltalk-80: The Language and its Implementation. Addison Wesley, Xerox Palo Alto Research Center, 1983. → pages 52


[40] A. Group. Aspectj benchmarks. [http://abc.comlab.ox.ac.uk/benchmarks](http://abc.comlab.ox.ac.uk/benchmarks). \[\text{pages} 115\]


