Structuring Extensions in System Infrastructure Software using Aspects

by

Jennifer Ellen Baldwin
B.Sc., University of Victoria, 2004

A Thesis Submitted in Partial Fulfillment of the Requirements
for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Structuring Extensions in System Infrastructure Software using Aspects

by

Jennifer Ellen Baldwin
B.Sc., University of Victoria, 2004

Supervisory Committee

Dr. Y. Coady, Supervisor (Department of Computer Science)

Dr. J. Weber-Jahnke, Department Member (Department of Computer Science)

Dr. W. Myrvold, Department Member (Department of Computer Science)

Dr. E. Wohlstadter, External Examiner (Department of Computer Science, University of British Columbia)

Supervisory Committee

Dr. Y. Coady, Supervisor (Department of Computer Science)

Dr. J. Weber-Jahnke, Department Member (Department of Computer Science)

Dr. W. Myrvold, Department Member (Department of Computer Science)

Dr. E. Wohlstadter, External Examiner (Department of Computer Science, University of British Columbia)

ABSTRACT

Many significant system extensions are hard to modularize. Consequently, their addition to a software system can jeopardize fundamental software engineering principles such as maintainability, understandability and evolvability. For example, the distributed Java Virtual Machine (dJVM) is a cluster aware implementation of a JVM in which distribution was retroactively added as an extension to an existing system. The prototype implementation of the dJVM relies on a patch file applied to IBM's Jikes Research Virtual Machine (RVM), introducing distribution code into roughly 55% of the original 1166 Java files.

In order to better determine the efficacy of modern modularization techniques such as aspect-oriented programming (AOP) in the context of system extensions, we offer up a case study based on distribution. The thesis of this work is that aspects can enhance extensibility of low-level system infrastructure software and be effectively integrated with existing software practices for introducing widespread change.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction and Related Work

Software engineering is integral to the cost, correctness and reliability of software. In order to assess the quality of software, certain crucial yardsticks are used. These so-called "-ilities" include modularity, understandability and evolvability. These properties are important to uphold since they have a proven impact on our software [48]. Extensibility is another valued attribute of software. Systems that support a wide range of extensions are easier to adapt to multiple contexts in cost effective ways. But what happens when we combine system-wide extensions and software engineering practices? Often times, the modularity of the system is broken by developers extending the system in unanticipated ways and hence the quality of the software is degraded.

This chapter introduces the motivation behind this work. In the first section, we discuss the difficulty in effecting widespread modifications associated with system extensions. An example of such an extension is *distribution*, that is, distributing a workload across multiple machines. In the second section, a concrete example of adding distribution functionality to a system is shown in the context of a distributed Java Virtual Machine (dJVM). The current means of effecting this implementation is via a *patch*, or a difference listing between the modified and original versions of the system. *Aspect-oriented programming (AOP)* is proposed as an alternative for consideration in the context of widespread system extensions. AOP promises to better consolidate and structure distribution code thereby improving adherence to software engineering principles. In the third section, patches and preprocessor directives are introduced as well as the concepts of AOP and the syntax of *AspectJ*, an

aspect-oriented extension to the Java programming language. Finally, in the last section, we discuss why distribution is crosscutting and relate this to previous work in modularizing this particular concern.

1.1 Widespread System Extensions

In the 1970s, Dijkstra argued that designers do not understand the large programs that they write since the problems are typically not adequately decomposed into their parts [35]. With each part in relative isolation, it is possible to understand these programs better but also prove the correctness of the programs more easily. Parnas also recognized that modularity can bring many benefits to understanding of system behaviour as a whole [52]. He also suggested that the decomposition of a system into modules should be more concern based than execution based. Murphy's work showed that a structural view of relevant parts of a large system allows developers to reason in isolation about individual concerns [51]. She also developed techniques to reason about the structural intent of a system.

When extensions are added to an existing system, these widespread modifications are often scattered amongst the modules of that system. They may also be *tangled*, meaning that they are difficult to distinguish from the implementation of other, unrelated code. The current approaches to implementing these widespread modifications to low-level systems code is by the use of patching and preprocessor directives, both of which produce no performance overhead. More recently however, work has investigated the use of aspects in this domain [28, 30, 42, 34].

Essentially, *patching* allows a developer to modify original files in place, thereby producing new files, and taking the difference listing between them. Patching [18] is discussed further in Section 1.3.1. One of the key problems associated with patches is that it is very difficult to reason about the impact a patch has on a system. The use and problems of patches within Linux is investigated further in Section 1.3.2.

Preprocessor directives, which are also discussed in Section 1.3.1, supply commands

within source code that are used to tell the preprocessor what to include or exclude for compilation. They provide more local control than compile-line options and are not included in the binary unless the preprocessor is directed to do so. In the Linux kernel, developers must contend with over 4,000 preprocessor flags defined to control compilation. In this context, it is often hard to tell what is and what is not part of the build configuration, as the code is activated with these directives.

In short, the same phenomenon applies to both patches and preprocessor directives. It is difficult for developers to reason about the high-level changes that these mechanisms make to a system. Additionally, these textual-based modifications are exceedingly fragile and can ultimately shackle modifications to outdated versions of a system. As a result, both patch and preprocessor directives provide no modularization for critical system extensions inherently related to system evolution, and in fact pose obstacles to further evolution.

A classic example of a widespread system extension is distribution. The case study presented here considers the distributed Java Virtual Machine (dJVM), which adds distribution functionality to a Java Virtual Machine via means of a patch. Though the design of distribution has been carefully mapped out, we argue that the design is hard to understand in this form. Other examples of this phenomenon can be obtained from extensible operating systems research such as SPIN [31] and Vino [54]. Extensible systems must provide an interface through which developers can extend the system in a principled manner. SPIN demonstrated that semantic properties of extensions are important, and that a wide, or poorly defined, interface can be exceedingly difficult to manage due to the extensive semantic properties that cannot be adequately expressed for extensions.

1.2 Case Study Background: Virtual Machines and the dJVM

A *platform* is the underlying hardware and software for a computer which defines an environment in which a software system can be developed. A *Virtual Machine (VM)* is an execution platform with software added to it that either makes it appear as a different platform or gives the appearance of multiple platforms. The advantages of this include improved security, portability and flexibility [55]. Virtual machines provide a layer between the application and the operating system and therefore can provide added security. Portability is increased because the virtual machine runs the same no matter what the base hardware and/or operating system. This built-in security and portability make virtual machines a prime candidate for a distributed system since little extra work is needed in order to protect the host machine or to make the system run on different architectures. *Java virtual machines (JVMs)* are simply virtual machines that run Java programs securely and machine-independently. The following subsections introduce an example of a JVM, the Jikes RVM, followed by a description of how this is made distributed in the dJVM. The Jikes RVM and dJVM systems provide the code base necessary to evaluate the application of aspects to widespread system extensions.

1.2.1 Jikes Research Virtual Machine

The *Jikes Research Virtual Machine (RVM)* [23] is a unique open source project that was developed by IBM Watson Laboratories. It is also the first Java virtual machine written itself in Java [56, 24] and is a lightweight substitute for the original JVM by Sun Microsystems. However at the time being, the RVM runs only on Linux based operating systems. A basis for almost 100 publications over the last five years and averaging several hundred CVS commits per month, the RVM is host to most of today's state-of-the-art Java virtual machine technology. Its code base affords researchers the opportunity to experiment with

a variety of design and implementation alternatives within an otherwise stable and consistently well-maintained infrastructure.

1.2.2 Distributed Java Virtual Machine

The *distributed Java Virtual Machine*, or *dJVM* [58][1], developed at the Australian National University in Canberra, is a ground-breaking effort to add distribution to the original Jikes RVM. The impetus behind distributing the JVM is largely performance-based. Distribution stands to make Java an even more attractive alternative for systems that need reliable and efficient response times, such as high performance server applications. Many server applications are multi-threaded, exhibiting loose coupling between those threads. The resulting tradeoff between computation and communication makes the use of a cluster of workstations feasible. The dJVM targets a 96 node, 192 processor machine running Linux, but also runs on general nodes connected via Ethernet.

The dJVM was built by adding annotations and distribution code in place, that is, interleaving it with code from the existing system. These modifications can then be applied by users in the form of a patch file, enabling users to seamlessly introduce distribution to their RVM, given a compatible version. However, the dJVM is not implemented by a set of in-place code changes alone, the implementation also adds many new classes to the system which provide mechanisms for performing buildtime and runtime modifications to code as well as changing the application/VM code, on demand, to allow distribution management code to function.

The design of the dJVM is based on a single system image (SSI), as shown in Figure 1.1, hiding the underlying distribution from Java applications, and shielding programmers from this additional complexity. The dJVM is aware of the cluster however, and must try to maximize opportunities to make applications run efficiently. Given that the dJVM is the first distributed implementation of a JVM written entirely in Java, it offers an interesting study of design alternatives, especially when compared with other SSI designs, such as

[1] The dJVM patches and documentation can be obtained from http://djvm.anu.edu.au/

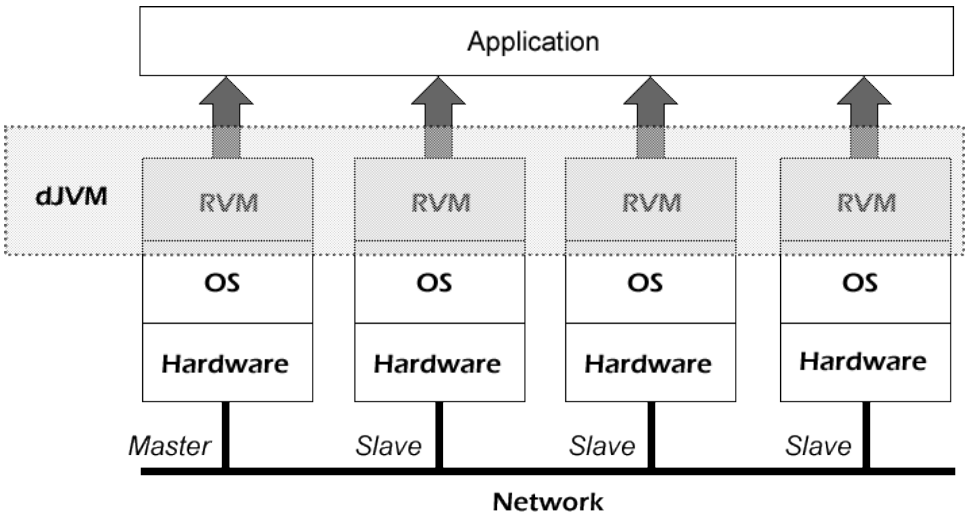

Figure 1.1. *The SSI architecture for the dJVM.*

the cluster aware JVM, cJVM [25] and the Java-Enabled Single-System-Image Computing Architecture (JESSICA) project [50]. Unfortunately, however, in its current form, it is hard to reason about the ways in which the dJVM patch modifies the RVM. Though the design has been carefully developed, it is difficult to map it to the implementation when it is presented in terms of line numbers and modifications to over half of the 1166 Java files in the RVM.

1.3 Effecting Widespread Changes for System Extensions

The following subsections discuss two approaches to effecting large changes within a system. The first is the traditional method of patching with preprocessor directives, which is used widely by the systems community. The second method we introduce is aspect-oriented programming, a relatively new programming paradigm that aims to improve modularity.

1.3.1 Patch and Preprocessor Directives

As mentioned previously, the current strategy for introducing distribution to the Jikes RVM is by using patches. As a development tool, patches allow new functionality to be devel-

```
diff --ignore-whitespace old/README new/README > name.patch

patch  b  < name.patch
```

Figure 1.2. *Commands to create and apply a patch.*

oped in situ, relative to the existing functionality of the system. The first line of Figure 1.2 shows the command used to create a patch file by using the *diff* utility. The `--ignore-whitespace` command tells diff to ignore white space differences such as tabbing. This option was forgotten by dJVM developers so much of the dJVM patch file contains inconsequential modifications. The second line shows how we apply a patch file to a system. The `-b` option saves a copy of the original contents of each modified file, before the differences are applied, in a file of the same name with the suffix `.orig` appended to it. If this file already exists, it is overwritten. If multiple patches are applied to the same file, the `.orig` file is written only for the first patch. Patching is discussed in further detail in the following subsection.

In Figure 1.3, we see that the `VM_DynamicTypeCheck` class implements an empty tag interface, in this case the `DVM_LocalOnlyStatic` interface, on lines 22 through 24. We also see that this interface is introduced by means of a *preprocessor directive*, `RVM_WITH_CLUSTER`. Like most other systems level implementations, Jikes is very complex, not very modular, and uses nonstandard language mechanisms to increase performance. One of these language mechanisms is Jikes' own support for preprocessor directives.

1.3.1.1 dJVM Patch Example

In terms of semantic leverage, the dJVM patch file itself is convoluted and hard to understand. Example code from the dJVM patch is shown in Figure 1.3 on page 9. The first four lines show the files to which the patch applies. Lines prepended with a "–" are being removed from the code and those prepended with a "+" are being added to the code.

We can determine where these lines are being added or removed by the specified relative offset at the beginning of the code modification section. An example of a relative offset is "`@@ -348,10 +352,16 @@`". The first number indicates the starting line in the original, unchanged file for code in this section. The second number indicates the total number of lines in the section before any changes have been made. The third number indicates the start position in the file after previous changes to the file have already been made and the fourth number indicates the total number of lines of code in the section after changes have been made.

Each section of code includes three lines before and after modified lines. These numbers are also included in our offset values. In our example, reading commences at line 348 from the original file and we will be reading 10 lines. In our new file, we will start reading at line 352 and finish by having 16 lines in our section after modifications have been made. Some things are immediately evident from looking at the patch. For example, the `//$Id` value that is being changed (file name, version, date, author, etc.) is often applied to files that have no other changes.

1.3.2 Patch in Systems Code

In terms of case studies on the use of patch files, Fiuczynski, Grimm, Coady and Walker [39] investigated how systems from embedded systems to supercomputers are running Linux. In order for developers to create these tailored systems, they will often start with a mainline kernel and then apply patches or patch sets for their particular application. These patches make system-wide changes and therefore are crosscutting – easily affecting over 100 files. An example given by Fiuczynki [38] discusses his personal experience maintaining the Linux kernel used for PlanetLab [19] which is a distributed overlay platform designed to deploy planetary-scale network services. Each of these machines runs a customized version of Linux that allows it to act as a virtual server, so that other projects running on the same machine do not collide. At one point, this kernel was modified by 28 patches and lagged eight minor releases behind the 2.4 kernel release. Now at version

```
diff -Naur rvmOld/src/vm/classLoader/VM_DynamicTypeCheck.java
rvm/src/vm/classLoader/VM_DynamicTypeCheck.java
--- rvmOld/src/vm/classLoader/VM_DynamicTypeCheck.java  2002-12-13
18:26:45.000000000 +0000
+++ rvm/src/vm/classLoader/VM_DynamicTypeCheck.java     2004-02-11
04:43:59.000000000 +0000
@@ -1,7 +1,7 @@
 /*
 * (C) Copyright IBM Corp. 2001
 */
-//$Id: Introduction.tex,v 1.11 2006/08/21 00:40:55 jbaldwin Exp $
+//$Id: Introduction.tex,v 1.11 2006/08/21 00:40:55 jbaldwin Exp $
 package com.ibm.JikesRVM;

 /**
@@ -69,7 +69,11 @@
 * @author Bowen Alpern
 * @author Dave Grove
 */
-public class VM_DynamicTypeCheck implements VM_TIBLayoutConstants {
+public class VM_DynamicTypeCheck implements VM_TIBLayoutConstants
+                                                                    //-#if RVM_WITH_CLUSTER
+                                                                    , DVM_LocalOnlyStatic
+                                                                    //-#endif
+{

  /**
   * Minimum length of the superclassIds array in TIB.
@@ -348,10 +352,16 @@
    if (LHSType == RHSType) return true;
    if (!LHSType.isResolved()) {
      LHSType.load();
+     //-#if RVM_WITH_CLASS_TRANSFORMER
+     LHSType.transform();
+     //-#endif
      LHSType.resolve();
    }
    if (!RHSType.isResolved()) {
      RHSType.load();
+     //-#if RVM_WITH_CLASS_TRANSFORMER
+     RHSType.transform();
+     //-#endif
      RHSType.resolve();
    }
    int LHSDimension = LHSType.getDimensionality();
```

Figure 1.3. *Jikes patch code for the dJVM.*

2.6, the focus is to keep the patch count to a minimum in order to keep close to the latest mainline kernel release. Nevertheless, several large patches are still needed.

The main problem with the patching approach is that it requires non-trivial effort to maintain a small crosscutting extension between minor kernel upgrades. Even more of a problem, is the challenge of integrating multiple patches, as there is often significant overlap. Conflicts when merging changes can only be remedied with textual comparison. Even though line conflicts may be semantically independent of one another, the code at such locations must still be analyzed by a programmer. Therefore maintaining variants of the Linux kernel are error prone and time consuming. Additionally, developers who wish to mainline their kernel extension must repeatedly go through this process. This can take anywhere from one to three years before full integration for any non-trivial change when adequate time for review and acceptance is involved.

1.3.3 AOP and AspectJ

In the realm of aspect-oriented programming (AOP) [47], *concerns* refer to elements in software systems that represent some identifiable entity. *Scattered* concerns are concerns whose implementations spread over more than one module, where a module refers to a separate unit of software. *Tangled* concerns are concerned whose implementations are intermingled in such a way that they cannot easily be separated. Intuitively, some scattered and tangled concerns have inherent structure that is not otherwise obvious due to their lack of modularity. When these scattered and tangled implementations can be structured using AOP, they are known as *crosscutting concerns*. *Aspect-oriented software development (AOSD)* [1] is a recent paradigm, developed at Xerox PARC in the 1990s, which facilitates the modularization of crosscutting concerns. In short, an *aspect* is a modular unit leveraging a few new linguistic mechanisms to support the inherent structure of a crosscutting concern.

AspectJ is an aspect-oriented extension to the Java programming language created at Xerox PARC. AspectJ was integrated into the Eclipse framework [10] in December 2002,

enabling AspectJ to become one of the most widely-used aspect-oriented languages today. In March 2005, AspectWerkz [6] merged with AspectJ to form a single language. Other AOP languages for Java include JBoss AOP [14], which is integrated with the JBoss application server and can use XML and annotations to express pointcuts, and Spring AOP [21] which was designed for enterprise-level systems. The following section provides an introduction to the language mechanisms of AspectJ including their syntax.

The constructs introduced with AspectJ are *advice*, *join points*, *pointcuts*, *inter-type declarations* and aspects. Advice is the code that will be modularized in an aspect. In essence, the functionality of the crosscutting concern. Join points are well-defined points of execution in a program, for example when specified methods are called or when an exception is thrown. Sets of these join points for which some advice will be applied are referred to as pointcuts. Inter-type declarations are used to add fields, methods and interfaces into class declarations. Aspects can contain advice combined with pointcuts as well as inter-type declarations, and are similar to classes in that they can be instantiated and can contain state and methods.

The first section describes why AOP is useful and presents some examples of this crosscutting behaviour. The next section focuses on how to write aspects at a high-level by introducing syntax for the AspectJ language. The last section briefly describes how aspect code is introduced to the base system via the AspectJ compiler [4] using a process known as *weaving*.

1.3.3.1 Why use AOP?

Object-Oriented Programming (OOP) has been successful in its adoption to software engineering because of the way it structures software to represent real-life problems. OOP systems comprise a collection of modules, referred to as *objects*, that interact with each other. This is in contrast to imperative languages where the collection of modules is made up of *functions* organized into files. OOP improves software engineering properties, or "ilities", which are measurements for the assessment of the quality of software engineer-

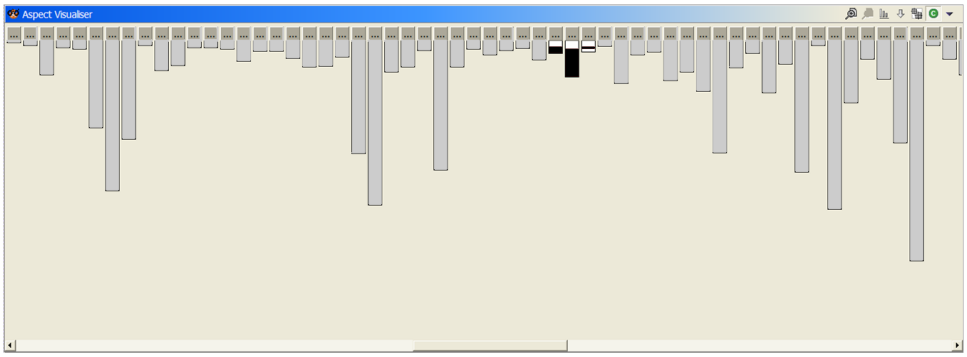

Figure 1.4. *Socket creation in the Tomcat Webserver.*

ing activities and products. Figure 1.4 shows an example of a good object-oriented design where each rectangle in this example represents a class and in this case, also a module. These classes are bigger or smaller in size depending on the lines of code within the class. The black lines show relevant lines of code for socket creation in the Apache Tomcat Webserver and the light grey rectangles show modules that are not modified when adding this new functionality. The modular design for socket creation is apparent since this code is contained within only three classes [33].

Unfortunately, there are still some concerns that cannot be expressed in terms of objects. These concerns exist in many modules within a system and therefore cannot be easily extracted into their own object. Or at least, they cannot be extracted without de-modularizing some other software concern. It is often said in the AOSD community that these concerns do not fit into the *dominant decomposition* of the system [57].

The most common, and easy to understand, example of a crosscutting concern is that of tracing which prints out relevant information at the start and end of every method within the system for which we wish to retrieve tracing information. Figure 1.5 shows tracing in the Tomcat Webserver [33]. The light grey rectangles, as above, are untouched by tracing code and the black lines represent the implementation of the tracing concern. It is easy to see from this example why tracing is referred to as a scattered concern, since part of its functionality is within almost every class, and why it is so difficult to encapsulate this

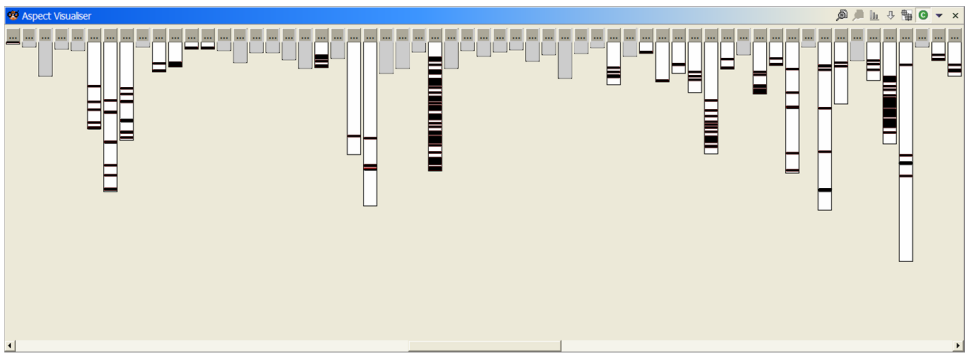

Figure 1.5. *Tracing in the Tomcat Webserver.*

within the object-oriented model.

The tracing example is a simple example shown with the intention of introducing AOP. The problem is that it gives the idea, all too often, that tracing is the one and only use for AOP. In this thesis, we introduce many more aspects with regards to distribution.

It is important to mention that it is not only object-oriented systems that suffer from crosscutting concerns. There is ongoing research to develop aspect languages for non OO languages, such as AspectC [2], which is an aspect-oriented extension of C.

1.3.3.2 An Aspect

We mentioned previously that aspects have the same granularity as Java classes. Therefore, an aspect looks much the same as a Java class except it makes use of the `aspect` keyword instead of `class`. Figure 1.6 shows a tracing aspect that will print out the method signature at the beginning and end of every method within a system. We can also use `public` and `private` to set the visibility of the aspect, and also include variables and methods, much the same as for Java classes. Aspects also include pointcuts, inter-type declarations and advice in a modular unit of a crosscutting implementation. These constructs are discussed further in the following sections.

```
public aspect Tracing {
    before(): execution(* *.*(..)) {
        System.out.println("Entering " + thisJointPoint.getSignature());
    }

    after(): execution(* *.*(..)) {
        System.out.println("Exiting " + thisJointPoint.getSignature());
    }
}
```

Figure 1.6. *An example tracing aspect.*

1.3.3.3 Join Points and Pointcuts

A critical element in the design of any aspect-oriented language is the join point model. Join points are certain well-defined points in the execution of the program. AspectJ provides constructs for many kinds of join points, most importantly method call or execution join points but also points such as initialization of a class and the getting and setting of private variables. These can be viewed in the AspectJ Quick Reference Guide [3].

In AspectJ, pointcuts pick out a set of join points in the program flow. For example, `call(void Point.setX(int))` will pick out every point where the `setX` method, which takes a single `int` parameter, and returns `void`, of the `Point` class is called. We can also use wildcard expressions such as `call(void Point.set*(..))`. This pointcut is associated with all calls to all methods that start with "`set`". Similarly, the "`!`" identifies specific joint points not included in the pointcut. The "`..`" construct signifies that this method can have any number and type(s) of parameters. In the same fashion, we can replace `call` with `execution` in order to pick out the execution of the method itself as the join point. Naming of these pointcuts can provide a powerful way to express intent, as shown in Figure 1.7.

1.3.3.4 Inter-Type Declarations

Inter-type declarations are used to add fields, methods and interfaces into class declarations. In order to do this, we define the variable or method as we would regularly but include

```
pointcut settingX(): execution(void Point.setX(int));

before(): settingX() {
    System.out.println("Entering " + thisJoinPoint.getSignature());
}
```

Figure 1.7. *Before advice with a named pointcut in AspectJ.*

```
aspect PointObserving {
    private Vector Point.observers = new Vector();
    ...
}
```

Figure 1.8. *Inter-type declarations in AspectJ.*

the class name to fully reference it. For example, in Figure 1.8, we see that the vector `observers` will be introduced to the `Point` class.

The introduction of interfaces is an inter-type declaration that uses the `declare parents` keywords. This construct can be used for both `implements` and `extends`. An example of `Point` implementing the `Serializable` interface is shown in Figure 1.9.

1.3.3.5 Advice

In order to actually introduce crosscutting code, we use advice. Advice is a body of code that will run at specified join points. The three types of advice are *before*, *after* and *around*. Figure 1.6 showed an example of before and after advice. In Figure 1.10, we see the same tracing code introduced with around advice. The most important thing to notice about this aspect is the `proceed` keyword which decides to allow computation at the join point to

```
aspect PointSerializing {
    declare parents: Point implements Serializable;
}
```

Figure 1.9. *The declare parents inter-type declaration.*

```
public aspect Tracing {
    around(): execution(* *.*(..)) {
        System.out.println("Entering " + thisJointPoint.getSignature());

        proceed();

        System.out.println("Exiting " + thisJointPoint.getSignature());
    }
}
```

Figure 1.10. *A tracing aspect that uses around advice.*

proceed. By omitting `proceed`, we can prevent the execution of the original method. We can also use around advice to supply different parameters to the method on proceeding.

1.3.3.6 Weaving and the AJC Compiler

Bytecode is the intermediate representation of Java code before it becomes *machine code*, which is directly executed by the machine. The AspectJ compiler, in its current version, takes Java source or bytecode as input and introduces, or *weaves* [46], aspect code via bytecode transformation. This means that if provided with Java source code as input, it must first compile this to bytecode before weaving. This differs greatly from the initial release of AspectJ (version 1.0) which was only able to weave into Java source code. This enabled users to use a preprocessing option which meant they did not necessarily have to compile the modified source as a step in the compilation. Instead, the AspectJ compiler copies the modified source code to a temporary directory where the user is either able to compile this code with the compiler of their choice or easily view the woven source code without the aid of a decompiler (decompiling is discussed in Section 4.3.3). The structure of this woven code is discussed further in Section 4.3.2.

1.4 Distribution: A Composition of Crosscutting Concerns

Distribution is often cited as being an example of a crosscutting concern, or a feature that does not fit into the dominant decomposition of a system. The original work that argues this point was by Lopes and Kiczales, who were also both involved in the creation of aspect-oriented programming (AOP) [47] at Xeroc PARC. They developed a new object-oriented language framework for distributed programming called D [49]. D uses AOP to develop base functionality of distributed applications so that the programmer does not have to explicitly deal with distribution and synchronization.

Returning to the reasons why distribution is scattered and tangled within a system, we refer to work by Fabry which discusses distribution as a set of co-operating aspects [37]. In this work, Fabry argues that even though distribution is a crosscutting concern, it is too broad to be considered just one aspect. Instead, three aspects are presented: concurrency, replication and remote method invocation. This concurrency aspect is one of the first aspects addressed by the AOP community, with explicit support for it in earlier versions of AspectJ. The concurrency issue that would be addressed by this aspect is thread synchronization. *Replication* is a method of sharing data between different computers in a distributed system. For example, if changes are made to data on one server, this server must propagate the data to the other servers in order to keep the data consistent. The remote method invocation aspect ultimately attempts to make it appear that there is no distinction between a local method call and a remote method call. Ideally with the addition of these aspects to a system, it would be transparent that distribution is actually taking place.

The above three aspects are integral to distribution and it is easy to see how concerns such as these would be difficult to separate from the base code and yet separate from each other. For example, if we were to insert remote method invocation code before every remote method call, we would probably do so by executing a local method that contained this code so that the remote invocation code would be contained. However, the call to this method would then be scattered and not systematically enforced and therefore still not modular.

1.5 Chapter Summary

Retroactively introducing significant extensions to a system is inherently difficult. It requires widespread change that is unlikely to fit modularly within the existing dominant decomposition of a system without compromising the modularity of another concern. An example of such an extension is distribution. A current implementation of distribution within a complex system can be studied within the dJVM, which retroactively adds this extension to the Jikes RVM. This extension was developed in place within the original system, and used a difference listing between the original and modified files to create a patch file. This patch file does not lend itself to understandability or evolvability. We believe this example to be highly representative of system infrastructure issues as patches have also been shown to create many problems for Linux kernel developers as well.

The thesis of this work is that aspects can enhance extensibility of low-level system infrastructure software and can be effectively integrated with existing software practices for introducing widespread change. By enhancing extensibility we mean that software engineering principles are better adhered to relative to traditional approaches, and by effectively integrating we mean that current techniques will not be compromised.

The structure of this thesis is as follows. The second chapter discusses the scattered and tangled nature and actual concerns of distribution within the dJVM. The third chapter follows up on this by presenting aspects for these concerns. The fourth chapter provides an analysis of the aspect-oriented version and also describes a vision for related tool support. Finally, future work and conclusions are presented in the fifth chapter.

Chapter 2

Concerns of Distribution in the JVM

This chapter starts by assessing the crosscutting nature of distribution within the dJVM. Though distribution has been previously cited as having crosscutting structure [37, 36, 49], no previous work has considered this particular code base. After this assessment, the design elements of distribution in the dJVM as separate concerns and how they are implemented within this system are discussed. These concerns include those for compilation, class loading, method invocation, thread identity, data access and object allocation, and placement.

2.1 Crosscutting of Distribution in the dJVM

The dJVM introduced 283 new Java classes to the RVM and additionally, the distribution patch modified 84% or 974 of the 1166 Java source files within the Jikes RVM. The number of lines of code that accomplished this was 40,177 lines in length. Of these lines 12,392 lines were being removed from the original source files and 19,398 lines were being added. This is a significant number of changes within the system as almost every file was touched by the distribution code.

Upon further inspection, it was found that the dJVM developers had forgotten to ignore white space differences in addition to CVS tags when taking a difference listing between the files. So the first step in creating our AspectJ implementation was to go through the distribution patch and manually remove inconsequential modifications such as white space. After this step was complete, it was found that only 645 files or 55% of the system had

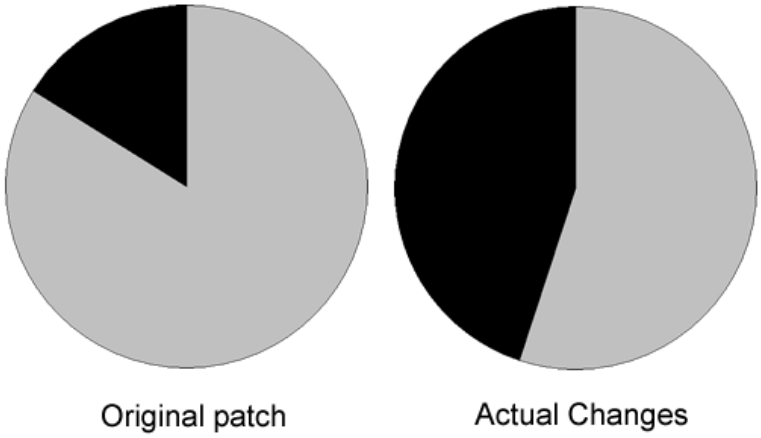

Figure 2.1. *Number of classes in the RVM which have modifications.*

actual modifications, as a result of distribution, made to them and the patch file contained 36,221 lines. The number of lines being removed dropped to 11,072 and the number of lines being added dropped to 18,408. These results are summarized in Figure 2.1 where light grey represents the number of files within the system that have modifications and black represents those that went unmodified. The above results were produced by running perl scripts on the patch file. These perl scripts are shown in Appendix D.

We started by eliminating these inconsequential modifications, such as white space differences and CVS tags, from the patch. Once the patch contained only valid changes to RVM functionality, the next step was to extract related code segments, constituting individual concerns, and attempt to structure them as aspects. The following chapter describes categories of changes and concerns that we identified as potential aspects of distribution.

2.2 The Concerns

We previously identified that the dJVM was based on a single system image (SSI). In order to achieve this, three important categories of modifications to the RVM were made [58].

1. **Infrastructure** – Several infrastructural components needed to be altered in order to

Category	Concern
Infrastructure	Baseline / Optimizing Compiler
VM Modifications	Class Loading Remote Method Invocation Thread Identity Remote Data Access
Object Allocation and Placement	Object Location

Table 2.1. *Categories for concerns of distribution.*

effect distribution with Jikes. These include inter-node communication, the building and booting processes, and the use of system libraries.

2. **VM Modifications** – In order to manipulate remote data in addition to local data, the class loading, method invocation and object access mechanisms needed to be modified.

3. **Object Allocation and Placement** – Crucial to distribution is the allocation and placement of objects on different machines. Mechanisms to provide both local and remote allocation of objects needed to be constructed.

These categories are respectively in the following sections and are outlined with separate associated concerns in Table 2.1. These associated concerns are important to consider in order for us to effectively investigate how they are currently implemented and what they achieve at a high-level. The next chapter revisits these same concerns in an attempt to make code look more like the design.

2.2.1 Infrastructural Modifications

In terms of core infrastructure, a *master-slave* architecture is used in which a master controls all of the global data and classes in the system for all of its slave nodes. Classes and data are *global* if they are shared amongst every member of the cluster. The master is the arbitrator of class loading and the logical owner of global data. This centralized class loading ensures a common identification of Java classes. The obvious advantage of this

is that there is a centralized point of coordination but the disadvantage is that it creates a bottleneck. However, this bottleneck is only a problem at startup time. Once classes are loaded, they may be compiled and instantiated locally. A globally used class though is still initialized at the master node. Here we consider just one of several possible concerns associated with this infrastructure of the dJVM, that is the compiler.

2.2.1.1 Baseline and Optimizing Compiler

There are two different compilers in the Jikes RVM: the baseline compiler and the optimizing compiler. Jikes RVM does not interpret bytecode; rather it compiles each method to machine code and executes the machine code natively. In an adaptive Jikes RVM configuration, the *baseline* compiler performs this initial translation of a method from bytecode to machine code. This translation occurs quickly but unfortunately the resulting machine code typically runs slowly.

In order to improve the performance of the machine code, methods that are either frequently executed or computationally intensive are identified via a sampling mechanism and recompiled by the *optimizing* compiler [26]. The optimizing compiler performs some aggressive optimizations that produce competitive performance compared to production JVMs. The optimizing compiler implementation far exceeds the baseline compiler in size and complexity.

The initial design of the dJVM mainly targets the baseline compiler; further development will be on the optimizing compiler [58]. Changes were in fact made to both compilers by the dJVM. However, modifications to the optimizing compiler were never completed by the dJVM developers since there was a small amount of code analysis to insert some appropriate type checking to enable the optimizing compiler to propagate type information correctly through its internal representation.

2.2.2 VM Modifications

The vast majority of changes within the system were VM modifications. Here, we discuss changes to the class loading mechanism, remote method invocation, thread identity, and remote data access.

2.2.2.1 Class Loading

Loading classes into the JVM alters the type information maintained by each node in the system. Type information for a class consists of `VM_Class`, `VM_Field` and `VM_Method` objects which describe interfaces, fields and methods respectively. Type information is replicated on each node, or machine, within the cluster, or set of machines, that statically or dynamically references a type. Replicate type objects, like all replicated objects, must have the same global identity, but may have different local identities. Furthermore, the internal dictionary identities for type information are required to be consistent.

A node that joins a cluster must have an initial set of types that have the same representation and identity as the corresponding types on all other nodes in the cluster. Class loading effectively happens in three stages:

1. **Build** – Where a set of core types are built into the executable image.
2. **Boot** – An additional set of core types are loaded prior to the node joining a cluster.
3. **Run** – Types that are loaded after the node has successfully joined the cluster.

The first two phases must result in a set of type information that is consistent with the other nodes in the cluster. The third stage is maintained by the distributed class loading system.

In the initial system class loading during the run stage is done through a central class loader. This is done for simplicity, as other areas will play a far more critical role in runtime performance of long running programs.

2.2.2.2 Method Invocation

The two extremes to achieve distribution are: (1) migrate data to the site of the computation, or (2) migrate computation to the site of the data. Which is better is a tradeoff between cost of migrating all/part of the computation and the cost of migrating data. Initially, all instance method execution takes place on the node where the object instance resides. By contrast all static methods are executed locally. When non-static methods associated with remote objects are called, they are invoked remotely and execute on the node in which they are located. The aim of this approach is to improve performance by executing methods where they are located (local/remote), combined with locating objects where they are needed [58]. Type information is also optimized along these lines. With each remote reference, the type of the object referenced is cached. Consequently, remote invocations can be resolved to the particular method to be executed with local type information, and then executed remotely.

When a class is loaded by the class loader, a transformation can take place. In order to support distribution, proxy method declarations for all non-static methods are generated, along with proxy code for all non-abstract methods. This is to enable access to non-local methods. Thus, the dynamic resolution mechanism within the Jikes RVM can use the locally cached type information and the proxy method type to compile and execute the correct proxy code.

Figure 2.2 shows a section of the patch that adds proxy/stub method code. These changes were made to the `VM_Method` class.

2.2.2.3 Thread Identity

A key issue of distribution in the dJVM is thread identity which is used because a global thread has a local thread on each node to support it. Not only must local threads map consistently to the same global identity, but local lock operations must consistently use the same local threads. To do this a thread reuse scheme is introduced where a local thread is bound to a global thread. Binding a local thread happens on demand, i.e. when the first

```
@@ -196,6 +250,34 @@
     return (modifiers & ACC_ABSTRACT) != 0;
   }

+
+  //-#if RVM_WITH_CLUSTER
+  /**
+   * Indicates whether the method is a proxy method
+   */
+  public final boolean isProxy() {
+    // We may set the proxy modifier when this is first declared,
+    // which may be before the class is read and transformed.
+
+    // if (VM.VerifyAssertions) VM.assert(declaringClass.isRead());
+    // if (VM.VerifyAssertions) VM.assert(isLoaded());
+    return (modifiers & ACC_PROXY) != 0;
+  }
+
+  /**
+   * Indicates whether the method is a stub method.
+   */
+  public final boolean isStub() {
+    // We may set the proxy modifier when this is first declared,
+    // which may be before the class is read and transformed.
+
+    // if (VM.VerifyAssertions) VM.assert(declaringClass.isRead());
+    // if (VM.VerifyAssertions) VM.assert(isLoaded());
+    return (modifiers & ACC_STUB) != 0;
+  }
+
+  //-#endif
+
   /**
    * Space required by this method for its local variables, in words.
    * Note: local variables include parameters
```

Figure 2.2. *Remote invocation in the patch.*

request for a global thread to execute locally is made. That local thread remains bound until there are no local resources associated with it. Figure 2.3 shows the changes made to both the `VM_Thread` and the `ThreadSupport` classes. Note that the modifications made to these classes are related but when navigating through a 40,000+ line text file, the correlation between changes would be hard to notice.

2.2.2.4 Remote Data Access

One important design decision that is particularly challenging to comprehensively map to the implementation is that of static (global) versus instance (local) variables. Static variables may be local within their node or within the entire application. Local variables are always held within their host node. The Jikes table of contents holds static fields and has two unused bits in descriptors for these fields. The dJVM usurps these two bits to indicate whether a variable is static or local, thus indicating how the data should be accessed. When data is being kept locally, an empty interface called `DVM_LocalOnlyStatic` must be implemented by any class that contains static data that is always accessed locally [58]. Empty interfaces are otherwise called tag interfaces and are used commonly in Java, for example when an object implements the `Serializable` interface. The purpose of these empty interfaces is to tag a class as a member of a certain set. Figure 1.3 on page 9 shows how the `DVM_LocalOnlyStatic` interface is introduced via preprocessor directives and patching on lines 22 through 24.

Data replication and caching is key to providing effective performance, although only type data is replicated in this release of the dJVM. Consequently, reading/writing remotely held data results in a request to the node that owns that data.

Data is stored in either object instances or in class variables. The object faulting scheme described in Section 2.2.3 identifies an object as local or remote, however, the same is not true for class variables. Class variables are held within an array, called Java Table Of Contents (JTOC), and a reference to that array is maintained in a register.

Class variables exist in a single global name space. However, the Jikes RVM uses a set

```
@@ -1378,4 +1455,101 @@
   public boolean isAlive() {
     return isAlive;
   }
+
+  //-#if RVM_WITH_CLUSTER
+  /**
+   * This should be hidden from the user application.
+   * Return the thread identity that this thread is associated with
+   * (no thread identity indicates that it not associated with another thread)
+   * @return The currently associated global thread (or null).
+   */
+  public final VM_Thread getClusterThreadIdentity() {
+    return clusterThreadId;
+  }
+
+  /**
+   * This should be hidden from the user application.
+   * Set the current identity of the thread
+   * @param aClusterThreadId The new identity of the thread.
+   */
+  public final void setClusterThreadIdentity(VM_Thread aClusterThreadId) {
+    clusterThreadId = aClusterThreadId;
+  }
+
+  public final void delegate(DVM_Message message)
+    throws VM_PragmaInline
+  {
+    synchronized (messageLock) {
+      ((DVM_MessageInterruptible)currentMessage).delegate(message);
+    }
+  }

...
```

```
 diff -Naur rvmOld/src/vm/libSupport/ThreadSupport.java rvm/src/vm/libSupport/ThreadSupport.java
--- rvmOld/src/vm/libSupport/ThreadSupport.java 2002-10-08 21:25:01.000000000 +0000
+++ rvm/src/vm/libSupport/ThreadSupport.java    2004-02-11 04:43:55.000000000 +0000
@@ -1,7 +1,7 @@
 /*
  * (C) Copyright IBM Corp 2001,2002
  */
-//$Id: Concerns.tex,v 1.9 2006/08/28 06:41:54 jbaldwin Exp $
+//$Id: Concerns.tex,v 1.9 2006/08/28 06:41:54 jbaldwin Exp $


 package com.ibm.JikesRVM.librarySupport;
@@ -34,6 +34,14 @@
    * Get current thread.
    */
   public static Thread getCurrentThread() {
-    return (Thread)VM_Thread.getCurrentThread();
+    VM_Thread thread = VM_Thread.getCurrentThread();
+     //-#if RVM_WITH_CLUSTER
+    VM_Thread ident  = thread.getClusterThreadIdentity();
+
+    if (ident != null)
+      return (Thread)(ident);
+    else
+    //-#endif
+      return (Thread)thread;
   }
 }
```

Figure 2.3. *Thread identity in the patch.*

of class variables to maintain its runtime support structures. Thread queues, type dictionaries, etc. are held in a global name space. In a cluster, the meaning of these runtime support structures ceases to be global and becomes local to each node in the cluster. Consequently, we have two categories of class variables: those that are node specific runtime support types and those that are global to the system (and in particular the application). This is dealt with by annotating those classes that are node specific.

Though it is not immediately obvious, Figure 2.4 shows code that does a remote array copy of a double array whenever the `arraycopy` method of `VM_Array` class is called. The relevant line numbers in the figure are lines 4 through 10. Notice that with white space modifications, it is difficult to pick out the exact modification made to this method. Also, the class to which this change is being made is not clear unless we wade through text to find it.

2.2.3 Object Allocation and Placement

In terms of memory allocation, the dJVM replaces the Jikes' `VM_Allocator` class with one that directs requests to a standard local allocator or to an allocator on another node. Once the objects are allocated, the important concern becomes how to locate them.

The dJVM uses reference faulting to locate objects. This means that the dJVM tries to access the object locally, and if it fails, it attempts to access the object remotely. It then uses the object's unique universal identifier (UID), unique within the cluster, to locate the object. This is referred to as a *decentralized approach*.

A local identifier (LID) is maintained independently of the global identity. This decouples the memory management at each node, and allows the memories of all nodes to be combined for a larger logical memory. This allows local garbage collection to reorganize memory without interfering with any node, while the cost of maintenance should be small compared with communication costs.

Each object reference either references directly to the location of the object within memory, or to an invalid address that can be interpreted as a LID. Sending a reference to

```
@@ -381,23 +453,32 @@

   // NOTE: arraycopy for long[] and double[] are identical
   public static void arraycopy(double[] src, int srcPos, double[] dst, int dstPos, int
       len) {
+    //-#if RVM_WITH_CLUSTER
+    if (VM_Magic.objectAsAddress(src).isRemote() ||
    VM_Magic.objectAsAddress(dst).isRemote()) {
+      DVM_RemoteAccess.remoteArrayCopy(src, srcPos, dst, dstPos, len);
+
+      return;
+    }
+    //-#endif
+
     // Don't do any of the assignments if the offsets and lengths
     // are in error
     if (srcPos >= 0 && dstPos >= 0 && len >= 0 &&
-   (srcPos+len) <= src.length && (dstPos+len) <= dst.length) {
+        (srcPos+len) <= src.length && (dstPos+len) <= dst.length) {
       // handle as two cases, for efficiency and in case subarrays overlap
       if ((! VM.BuildForRealtimeGC) && (src != dst || srcPos > dstPos)) {
-   VM_Memory.aligned32Copy(VM_Magic.objectAsAddress(dst).add(dstPos<<3),
-               VM_Magic.objectAsAddress(src).add(srcPos<<3),
-               len<<3);
+
+        VM_Memory.aligned32Copy(VM_Magic.objectAsAddress(dst).add(dstPos<<3),
+                                VM_Magic.objectAsAddress(src).add(srcPos<<3),
+                                len<<3);
       } else if (srcPos < dstPos) {
-   srcPos += len;
-   dstPos += len;
-   while (len-- != 0)
-     dst[--dstPos] = src[--srcPos];
+        srcPos += len;
+        dstPos += len;
+        while (len-- != 0)
+          dst[--dstPos] = src[--srcPos];
       } else {
-   while (len-- != 0)
-     dst[dstPos++] = src[srcPos++];
+        while (len-- != 0)
+          dst[dstPos++] = src[srcPos++];
       }
     } else {
       failWithIndexOutOfBoundsException();
```

Figure 2.4. *Data replication in the patch.*

another node requires the object reference or LID to be translated to a UID and vice-versa upon reception.

2.3 Concerns of Distribution that Cannot be Structured

In order to effect distribution within a system, there are some low-level changes that must be made that do not necessarily have a crosscutting structure. For example, configuration flags or addition of extra command-line options that change the behaviour of the VM. There are also changes required to enable compilation to take place and take into account the addition of distribution specific classes. These concerns are not discussed in this chapter since they are implementation concerns and not concerns of design.

2.4 Chapter Summary

This chapter has discussed the nature of the implementation of distribution within the dJVM in which the patch touches 55% of the files within the system. We then investigated the design elements in the dJVM that comprise separate concerns of distribution. These concerns are outlined in Table 2.2. The next chapter introduces these same concerns as aspects, to determine if a more structured implementation of distribution can be made in this form.

Category	Concern	Description
Infrastructure	Baseline / Optimizing Compiler	The dJVM targets only the baseline compiler which converts bytecode to machine code but does no optimizations on the compiled code.
VM Modifications	Class Loading	Class loading affects the type information maintained by each node in the system. A central class loader is used to avoid ownership confusion of classes.
	Remote Method Invocation	Performance is improved by executing methods where they are located, combined with locating objects where they are needed.
	Thread Identity	A global thread has a local thread on each node to support it. Thread identity is used to map these local threads to their global thread.
	Remote Data Access	Data replication and caching is key to providing effective performance. Only type data is replicated in the dJVM.
Object Allocation and Placement	Object Location	Reference faulting is used to locate objects. A unique universal identifier (UID) is used when locating remote objects.

Table 2.2. *Concerns of distribution.*

Chapter 3

AspectJ Implementation

We used AspectJ for the refactoring of the dJVM modifications. Given that Jikes is written mainly in Java, and that AspectJ is a leading industrial strength AOP implementation, it is a suitable choice.

This chapter describes aspects we created based on the dJVM design elements discussed in the previous chapter as well as new aspects that we identified during the migration to AOP. We additionally include distribution aspects as prescribed in related work [37]. We were unable to migrate all modifications from the patch, so we include a description of what was left unimplemented in the final section of this chapter.

3.1 The Aspects

All of the code for the aspect-oriented refactoring of the dJVM is available for download[1]. The first aspects overviewed here mirror those introduced in the previous chapter. Namely, those for compilation, class loading, method invocation, thread identity and data access. We discuss aspects that were discovered while investigating the code base. These aspects, although not outlined in the design of the dJVM, lend themselves to be structured together. We follow up with other aspects we believe to be important in terms of consolidation of distribution, such as those for configuring the behaviour of the VM.

[1] Code is available at: http://www.csc.uvic.ca/ jbaldwin/diva

3.1.1 Baseline and Optimizing Compiler Aspects

The baseline and optimizing compiler changes were put into an aspect so that they could be more easily evolved separately from other dJVM modifications and from all other RVM code. Though changes to the compilers include the fact that the `DVM_LocalOnlyStatic` interface was introduced, we did not include this as part of these aspects. These hierarchy changes are already implemented in an aspect of their own.

The `VM_BaselineCompiler` and `VM_Compiler` were affected most by this aspect. The *constant pool* is a table in each class that contains the values for all constants. However, support for constant pools is also added within `VM_Class`. Typically any reference to a constant anywhere in a class file is using the index of that constant in the table.

The optimizing compiler changes are equivalent to those made to the baseline compiler, therefore we expect these aspects to behave similarly as far as an analysis is concerned. Since the optimizing compiler is not used in the configuration of the dJVM, it has been created but has not been tested or released. The baseline compiler aspects have been released and tested.

3.1.2 Class Loading Aspect

The majority of changes made within the class loading aspect are methods added to the `VM_ClassLoader` class which is responsible for manufacturing type descriptions as needed by the running virtual machine. These methods are responsible for writing and validating data. Modifications were also made to the `VM_Type` class, which describes a Java type, and its subclass `VM_Class`, which describes a Java class type. The boot methods of the `VM` and `VM_ClassLoader` classes were also altered to affect distribution specific booting of the system.

```
/**
 * Indicates whether the method is a proxy method
 */
public final boolean VM_Method.isProxy() {
    // We may set the proxy modifier when this is first declared,
    // which may be before the class is read and transformed.
    return (this.modifiers & VM_Method.ACC_PROXY) != 0;
}

/**
 * Indicates whether the method is a stub method.
 */
public final boolean VM_Method.isStub() {
    // We may set the proxy modifier when this is first declared,
    // which may be before the class is read and transformed.
    return (this.modifiers & VM_Method.ACC_STUB) != 0;
}
```

Figure 3.1. *Remote invocation aspect.*

3.1.3 Remote Invocation Aspect

Figure 3.1 shows some example aspect code from the remote invocation aspect. This functionality corresponds to the portion of patch code shown in Figure 2.2 on page 25. The majority of this aspect is inter-type declarations and in particular those that pertain to proxy/stub methods. Another interesting facet of this aspect is that classes can be marked as not being remotely accessible. Also, if a method of a certain class is invoked, instead of it being invoked remotely, its class will be instantiated on the requesting node before the method is invoked. With these design elements consolidated as an aspect, it is easier for us as developers to understand what rules apply to remote method invocation. Understanding these rules will allow us to more easily debug the system, especially if methods are not being invoked on the correct node.

3.1.4 Thread Identity Aspect

Figure 3.2 shows the aspect that implements thread identity functionality. This functionality corresponds to the portion of patch code shown in Figure 2.3 on page 27. The aspect shows the addition of the `clusterThreadId` to the `VM_Thread` class in addition to get-

```
package com.ibm.JikesRVM;
import com.ibm.JikesRVM.librarySupport.*;

public privileged aspect ThreadIdentity {

    private VM_Thread   VM_Thread.clusterThreadId;

    public final VM_Thread VM_Thread.getClusterThreadIdentity() {
        return clusterThreadId;
    }

    public final void VM_Thread.setClusterThreadIdentity(VM_Thread aClusterThreadId) {
        clusterThreadId = aClusterThreadId;
    }

    Thread around(): execution(public static Thread ThreadSupport.getCurrentThread()) {
        VM_Thread thread = VM_Thread.getCurrentThread();
        VM_Thread ident  = thread.getClusterThreadIdentity();

        if (ident != null)
          return (Thread)(ident);
        else
          return (Thread)thread;
    }
}
```

Figure 3.2. *Thread identity aspect.*

ter and setter methods for this private variable. More interestingly, we see `around` advice that checks whether or not the current thread has a *cluster id*. If so, its cluster identity is returned. We also see that this aspect uses the `privileged` keyword. A privileged aspect is able to access the private methods and variables of classes. It does so by putting public getter and setter methods into the class it is accessing. This aspect shows how a low-level concern can be shown modularly and be more easily understood than in terms of line numbers spread throughout a text file, especially since these changes are spread throughout more than one class. Some of the design decisions behind this aspect are discussed in Section 4.3.2.

```
before(double[] src, int srcPos, double[] dst, int dstPos, int len):
    execution(static void VM_Array.arraycopy(double[], int, double[], int, int))
    && args(src, srcPos, dst, dstPos, len) {

    if (VM_Magic.objectAsAddress(src).isRemote()
            || VM_Magic.objectAsAddress(dst).isRemote()) {
        DVM_RemoteAccess.remoteArrayCopy(src, srcPos, dst, dstPos, len);
        return;
    }
}
```

Figure 3.3. *Data replication aspect.*

3.1.5 Data Replication Aspect

Figure 3.3 shows a segment of the data replication aspect. This functionality corresponds to Figure 2.4 on page 29. Namely, this snippet of aspect code shows that every time a copy of an array of `doubles` is done within the `VM_Array` class, we must also ensure to do a remote array copy if the address of the object is remote. This occurs for all primitive type array copies within this class. This aspect also introduces methods to copy each type of literal within the `VM_Statics` class and also methods to calculate the checksum to ensure that data is correctly copied.

3.1.6 LocalOnlyStatic Aspect

In Section 2.2.2.4 on page 26, we discussed how the dJVM developers deal with local and global static data by implementing or extending the `DVM_LocalOnlyStatic` interface. In many cases, this was the only change made to a class and was the most widespread change throughout the patch. In fact, 509 of the 645 files modified had this as their only change which means that this constituted 44% of the file changes. Ignoring this change, only 11% or 136 files had significant distribution changes made to them, as shown in Figure 3.4. This figure uses black for unmodified files and light grey for files with distribution changes. With `DVM_LocalOnlyStatic` changes removed from the patch, it was only 22,712 lines in length, roughly half of its original size. In this form, 6,619 lines are being added and 13,448

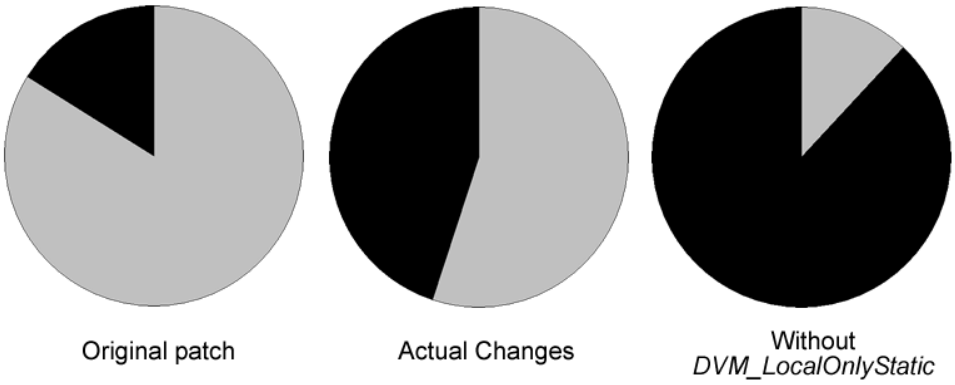

Figure 3.4. *Number of classes modified by the LocalOnlyStatic aspect.*

are being removed. Figure 3.5 shows the AspectJ implementation of the modifications to class hierarchy involving the empty interface `DVM_LocalOnlyStatic`. The corresponding patch is shown in Figure 1.3 on page 9 with the relevant lines being 22 through 24.

3.1.7 Identity Aspect

This aspect is small, however it was difficult to isolate changes in identity from changes made in other aspects. This was due to the lack of refactoring of the code in the dJVM, and results in many changes introduced as `around` advice without a `proceed`. For that reason, this aspect only makes the meaningful change of adding a constructor to `VM_Class` which takes a `DVM_UID`, or unique identifier, as a parameter.

Had we refactored dJVM aspects in a different order, this many not have been the case. It is interesting to note that this coarse grained aspect-aspect interaction would be easier to untangle if the RVM base code was refactored. Figure 3.6 shows an inter-type declaration introduced by the class loading aspect. In this particular piece of advice, the type information is found for the class via the unique identifier.

```
package com.ibm.JikesRVM;
import com.ibm.JikesRVM.memoryManagers.vmInterface.*;
import com.ibm.JikesRVM.librarySupport.*;

public privileged aspect LocalOnlyStatic {

    declare parents: VM_Scheduler || VM_Wait || *Thread || VM_*Queue ||
             VM_*Lock* || VM_Processor* || VM_Proxy ||  VM_Synchronization ||
             (*Map* && !VM_JNIGCMapIterator) || com.ibm.JikesRVM.memoryManagers.JMTk.* ||
             VM_Barriers || VM_Memory || Scan* || SynchronizationBarrier || Util ||
             VM_Finalizer || VM_Handshake || VM_Interface ||
             (*Info && !VM_PendingJSRInfo) || *Table* || VM_DynamicTypeCheck ||
             (*Compile* && !VM_JNICompiledMethod) || (VM_Pragma* && !VM_PragmaException) ||
             VM_DynamicLink ||VM_Entrypoints ||  VM_BaselineException* || VM_Magic* ||
             VM_OutOfLineMachineCode || VM_RecompilationManager || VM_Runtime ||
             VM_StackTrace || VM_Verifier ||
             (VM_JNI* && !VM_JNICompiledMethod && !VM_JNIFunctions) ||
             *Header || *Profile* || *Monitor* || VM_*Class* ||
             VM || VM_Array || VM_Atom || VM_BasicBlock || VM_BootRecord ||
             VM_BuildBB ||  VM_Callbacks || VM_CommandLineArgs || VM_Configuration ||
             VM_EdgeCounts || VM_FileSystem || VM_EventLogger || VM_Field || VM_Lister ||
             VM_Math || VM_Member || VM_Method || VM_ObjectModel || VM_Primitive ||
             VM_Properties || VM_Reflection || VM_Services || VM_Statics ||
             VM_Time || VM_Triplet || VM_Type || VM_UTF8Convert || JikesRVMSocketImpl ||
             FileSupport || ReflectionSupport && !*Constants && !DVM*
             implements DVM_LocalOnlyStatic;

    declare parents: *Constants && !DVM* extends DVM_LocalOnlyStatic;

    /**
     * interface implemented to force access of static variables to be through
     * the local JTOC.
     */
    public static VM_Type VM_Type.LocalOnlyStaticType;

    public final boolean VM_Type.isLocalOnlyStaticType() throws VM_PragmaUninterruptible {
        return this == LocalOnlyStaticType;
    }

    /**
     * Should all accesses to the static fields of this class be to the local
     * JTOC only, i.e. are the static fields unique to this node
     *
     * FIX ME: these methods should return the value of a flag, rather that go
     * through a loop each time.
     */
    public final boolean VM_Class.isLocalOnlyStatic() {

        VM_Class[] interfaces = this.declaredInterfaces;

        if (interfaces == null)
            return false;

        for(int i = 0, n = interfaces.length; i < n; ++i)
            if (interfaces[i].isLocalOnlyStaticType())
                return true;

        return false;
    }

}
```

Figure 3.5. *LocalOnlyStatic aspect.*

```
/**
 * Find a type description, which matches the parameter dictionary identifier
 *
 * @param id type dictionary identifier
 * @return the corresponding type
 */
public static VM_Type VM_ClassLoader.clusterGetType(DVM_UID uId) {
    VM_Type typ = null;

    if(uId.getNode() == DVM_CommunicationManager.getNodeId()) {
        typ = (VM_Type) DVM_UIDManager.find(uId);
    } else {
        DVM_MessageClassGetTypeReq typMsg = new DVM_MessageClassGetTypeReq(uId);
        typMsg.send();
        typ = validRemoteType(typMsg);
    }

    VM._assert(typ != null,"Unable to get type information "+
    "either locally or remotely");
    return typ;
}
```

Figure 3.6. *Local identity code tangled within class loading.*

3.1.8 Concurrency Aspect

The concurrency aspect was identified in the early work of distribution and AOP and consists of changes to thread classes. The most striking change in this aspect is message locking, in that locking variables along with methods to get and set them are introduced to `VM_Thread`. Message locking is part of the communication process and is considered an infrastructure change. This locking mechanism is designed for message receiver threads which process incoming packets and link them together to form a message.

Since locking operations are directed to the home node of the object or class, this aspect also entails remote locking and unlocking. A check to see if the object is remote is made in all `lock` and `unlock` methods of `VM_ThinLock` and the appropriate method of `DVM_RemoteAccess` is then called. This is shown in Figure 3.7.

```
void around(Object o, int lockOffset) throws VM_PragmaInline : execution(static void
    VM_ThinLock.unlock(Object, int)) && args(o, lockOffset){
    if (VM_Magic.objectAsAddress(o).isRemote()) {
        DVM_RemoteAccess.remoteUnlock(o);
    } else {
        VM_ThinLock.localUnlock(o, lockOffset);
    }
}
```

Figure 3.7. *Remote locking operations.*

3.1.9 PowerPC Aspect

The PowerPC modifications (identified as such because they are in the PowerPC package) were migrated to their own aspect which can be included depending on the setup of the user. This aspect is outlined in Figure 3.8. Looking more closely at the aspect, it adds the functionality necessary to generate inline code for `VM_MagicNames.invokeStubMethod`. There are over 100 other methods for which inline code is generated, and the patch inserts this code within the substantially long 100 way branch. Though this is a very small and specific aspect, we believe separating these changes according to a platform specific configuration, albeit at a very low level, is an important factor for continuing platform development.

3.1.10 Memory Management Aspect

There are two different memory managers that could be used in the version of Jikes that the dJVM modifies. These include the Java Memory Management Toolkit (JMTk), which was designed especially for the RVM, and also the Watson memory managers which came out of IBM's TJ Watson Research Center. We believe it is important to separate changes to memory management from other distribution modifications and also to have separate aspects to perform tasks for each of the separate memory management configurations. Our particular configuration used JMTk so we supply those for download since they have been tested. The aspect for the Watson memory manager would be extremely similar.

```
package com.ibm.JikesRVM;

public aspect PowerPC implements VM_BaselineConstants, VM_AssemblerConstants {

 boolean around(VM_Compiler compiler, VM_Method methodToBeCalled):
    execution(boolean VM_MagicCompiler.generateInlineCode(VM_Compiler, VM_Method))
                  && args(compiler, methodToBeCalled) {

    VM_Atom      methodName       = methodToBeCalled.getName();
    VM_Assembler asm              = compiler.asm;
    int          spSaveAreaOffset = compiler.spSaveAreaOffset;

    if (methodName == VM_MagicNames.invokeStubMethod) {
                        asm.emitL   (T0, 0, SP); // t0 := ip
                        asm.emitMTLR(T0);        // LR := t0
                        asm.emitCAL (SP, 4, SP); // pop ip
                        asm.emitL   (T0, 4, SP); // t0 := object reference
                        asm.emitL   (T1, 0, SP); // t1 := message   reference
                        asm.emitCall(spSaveAreaOffset); // call
                        asm.emitCAL (SP, 8, SP);
                        return true;
    }
                        else return proceed(compiler, methodToBeCalled);
  }
}
```

Figure 3.8. *PowerPC aspect.*

3.1.11 dJVM Configuration Aspect

The dJVM configuration aspect includes several features associated with configuration such as flags, command-line options and library support. The flags that were introduced by this aspect control the behaviour of the virtual machine. These flags include whether or not to build the cluster version, allocation policies and tracing behaviour. We believe that it is important to localize flags within an aspect so that if a user wants to quickly adjust their system, they would know where to do so. This aspect also introduces command line argument constants and also the advice to process those command line arguments.

The configuration aspect also comprises changes to the `com.ibm.JikesRVM.librarySupport` package within Jikes. The changes consisted of the addition of different `sysWrite` methods used for file support and stack trace support within the dJVM. These `sysWrite` methods are important since they accomplish what a `System.out.println()` call would do in a regular, non-distributed application.

`System.out` and `System.err` are directed to the master node and if we wish to print a message on the node upon which the thread is running, we must use `VM.sysWrite()`, which will write to `stderr` on the same node as the thread that called it. Applications run in the dJVM using the regular system calls and end with a thread switching not enabled exception. Other functions of this aspect include the addition of methods to aid in debugging the system.

3.1.12 Dummy Compilation Aspect

The Jikes RVM has a unique build process. Part of this build process includes compiling the `Dummy` class which contains enough references to force the Java compiler to find every class comprising the VM. This aspect simply adds the necessary distribution classes to cause a full distribution compilation. It was important for these changes to be on their own so that as development of distribution continued, the developer would easily be able to add references in one place to control compilation. This aspect is shown in Figure 3.9. Note that this aspect is in the default package and not in the Jikes RVM package as all of the other aspects were. This is because `Dummy` was also in the default package and could not be seen from the aspect if it was not in the same package. Ideally these modifications would reside within the dJVM configuration aspect. However, due to problems with packaging and since the configuration aspect should be kept within the dJVM package structure, this aspect resides on its own.

3.1.13 Runtime Aspect

There are some classes that are used during the runtime of the system. Because certain methods access raw memory or other machine states, perform unsafe casts, or are operating system calls, they cannot be implemented in Java code. These classes include `VM_Runtime, VM_Magic` and `VM_Address.` `VM_Runtime` introduces entry points into the runtime of the virtual machine. Certain methods in the class `VM_Magic` are treated dif-

```
import com.ibm.JikesRVM.*;
import com.ibm.JikesRVM.ClassTransformer.*;
import com.ibm.JikesRVM.BytecodeToolset.*;
import com.ibm.JikesRVM.memoryManagers.vmInterface.VM_Interface;
import com.ibm.JikesRVM.memoryManagers.JMTk.Plan;

/*
 * Dummy class contains enough references to force java compiler to find
 * every class comprising the vm, so everything gets recompiled by just
 * compiling "Dummy.java".
 *
 * Adding the necessary distribution files to cause them to compile.
 */
public aspect DummyCompilation {

    static JBc                            Dummy.qa;
    static LabelAbsolute                  Dummy.qb;
    static LabelRelative                  Dummy.qc;
    static Assembler                      Dummy.qd;
    static AssemblerLabel                 Dummy.qe;
    static Err                            Dummy.qf;
    static Disassembler                   Dummy.qg;
    static DisassemblerDump               Dummy.qh;
    static java.util.Hashtable            Dummy.qz;
    static ClassTransform                 Dummy.ra;
    static NoTransform                    Dummy.rb;
    static Dump                           Dummy.rc;
    static Isomorphic                     Dummy.rd;
    static DVM_UID                        Dummy.u;
    static DVM_ClassTransformAccess       Dummy.za;
    static DVM_ClassTransformDebug        Dummy.zb;
    static DVM_ClassTransformProxy        Dummy.zc;
    static DVM_ClassTransformRemoteInvoke Dummy.zd;
    static DVM_Proxy                      Dummy.zf;
    static DVM_AllocatorPolicyNode        Dummy.zg;
    static DVM_AllocatorPolicyRoundRobin  Dummy.zh;
    static DVM_AllocatorPolicyRandom      Dummy.zp;
    static DVM_ForceProxy                 Dummy.zi;
    static DVM_ForceRemoteAllocation      Dummy.zj;
    static DVM_TracingOn                  Dummy.zl;
    static JikesRVMSocketImpl             Dummy.zn;
    static java.net.InetAddress           Dummy.zo;
    static ClusterApplicationClassLoader  Dummy.zq;
    static DVM_ClassMonitor               Dummy.zr;
    static DVM_DataForGDBWriter           Dummy.zs;
}
```

Figure 3.9. *Dummy compilation aspect.*

```
public static VM_Address VM_Address.makeRemote(int addr) {
    // call site should have been hijacked by magic in compiler
    if (VM.VerifyAssertions && VM.runningVM) VM._assert(VM.NOT_REACHED);

    return new VM_Address((addr << DVM_ReferenceConstants.REMOTE_LG) |
        DVM_ReferenceConstants.REMOTE_REFERENCE_MASK);
}

public boolean VM_Address.isRemote() {
    // call site should have been hijacked by magic in compiler
    if (VM.VerifyAssertions && VM.runningVM) VM._assert(VM.NOT_REACHED);

    return (this.value & DVM_ReferenceConstants.REMOTE_REFERENCE_MASK) != 0;
}
```

Figure 3.10. *Runtime aspect.*

ferently by the compiler. The type `VM_Address` is used to represent a machine-dependent address type. Figure 3.10 shows part of the runtime aspect. Here we see the addition of `makeRemote` and `isRemote` methods for addresses.

3.1.14 DVM_UID

The `DVM_UID` class was added entirely within the patch so we took these changes and placed them within their own class file in the aspects folder. We are unsure how this particular file ended up being introduced in such a fashion. However, we include it in the aspects source folder so that those changes could be removed from the patch. Alternatively, we could have introduced it within the cluster package of the dJVM source, however we wanted to maintain a pure patch to Java approach with which to perform analysis.

3.1.15 What was Not Implemented

There are two patch files present within the implementation of the dJVM. One is for changes to the RVM itself, `jikesrvm-cluster-extns-1.0.2.patch`, and the other is for changes to Java system libraries,
`jikesrvm-cluster-extns-1.0.2-jlibsource.patch`. This latter patch file has also

been stripped of inconsequential modifications but for the sake of simplicity, these changes were left intact. This patch modifies only three system source files and in a very minor way.

The other RVM patch file, `jikesrvm-cluster-extns-1.0.2.patch`, still exists as not all modifications could be ported to aspects. Since the Jikes RVM contains more than just Java files, changes to these files needed to be left within the patch. These files include C files, Java template files and makefiles.

Other reasons for code remaining in the patch are discussed in Section 4.3. To mention these reasons briefly, the result of *advising*, or making modifications to, certain methods within the system caused the system to end with a segmentation fault. It was also found that not all classes could be advised with the `LocalOnlyStatic` aspect due to inheritance relationships.

Finally, some modifications cannot be expressed as aspects. These modifications include change in visibility to a method or variable, the initialization values of variables and whether or not a variable is final. There were also introductions of or references to inner classes and inner interfaces that could not be supported by the AJC weaver.

Since some code had to be left in the patch file, so did a lot of empty space. As discussed in Chapter 1, the patch file is extremely reliant on line numbers and lines of context. If these were to change, applying the patch would fail. Therefore, if a file contained some changes that could be expressed as an aspect and some that could not, the changes that could would have to be replaced by blank lines in order to preserve line numbering. The patch file currently had 3,489 blank lines out of a total of 12,746 lines; a total of 27.4% of the remaining patch file. However, changes to files other than Java files comprise 3,376 lines or 26% of what is left. This means that altogether, 16% of the original patch, with inconsequential modifications removed, is actual Java changes that could not be expressed in terms of aspects.

3.2 Modifications to the Build Process

The largest struggle in the application of AspectJ within the RVM was the build process. Changes were made to three of the build scripts but many issues regarding the build process were found along the way. These problems are discussed in detail in Chapter 4 on future work. The modifications to the build process are discussed in Appendix A.

3.3 Chapter Summary

The aspects motivated by the dJVM design are outlined in Table 3.1. The aspects that were not defined as concerns by designers of the dJVM are shown in Table 3.2. The pertaining category of change and particular concern, if they have one, are also shown. The modules affected by each aspect are shown out of the total 209 Java files used in our configuration. This configuration takes into account that we used only the baseline compiler, Java Memory Management Toolkit and did not use the PowerPC source. We show the modules affected by the optimizing compiler and PowerPC for interest although their percentage out of the total will not be accurate.

During the implementation of distribution, it was also found that some of the dJVM patch could not expressed as aspects. In the end, 16% of the original patch file, with inconsequential modifications removed, was inexpressable as aspects. The next chapter analyzes these aspects in terms of software engineering principles and also how they were able to survive as the RVM itself evolved. We also discuss the negative results we encountered during implementation, and prescribe a possible solution.

Category	Aspect	Description	Classes Affected (out of 209)
Infrastructure	Baseline and Optimizing Compiler	Major changes were made to `VM_BaselineCompiler` and `VM_Compiler`. Support for constant pools is added to `VM_Class`.	9 for baseline compiler, 6 for optimizing compiler
VM Modifications	Class Loading	`VM_ClassLoader` was affected the most, as well as classes that describe Java types. Modifications to the boot process were also made.	14
	Remote Invocation	Introduction of proxy/stub method support as well as remote accessibility restrictions.	20
	Thread Identity	A cluster id is introduced to `VM_Thread` with checks for it made in `ThreadSupport`.	2
	Data Replication	Remote copying support for data is added as well as checksum methods to ensure valid copied data.	9
	Local vs. Global Data	Static data may be static within a single node or globally within the entire application. The tag interface `DVM_LocalOnlyStatic` is added with support to check for it via introduced methods.	109
Object Allocation and Placement	Identity	Local identity is maintained independently of global identity, therefore this id is added as a parameter to `VM_Class`.	2

Table 3.1. *Aspects identified by dJVM design.*

Aspect	Description	Classes Affected (out of 209)
Concurrency	Ensures thread synchronization by introducing message locks to `VM_Thread` with checks for these locks in the locking and unlocking methods of `VM_ThinLock`.	7
PowerPC	Architecture-specific distribution modifications.	1
Memory Management	Although an efficient distributed garbage collection algorithm has not been developed [58], changes to memory management are still needed to run the RVM distributedly. We support changes to JMTk.	7
Configuration	Configuration features that control the behaviour of the VM such as flags, debugging methods, command-line options and console print methods used for file support and stack trace support.	9
Dummy Compilation	References to force Java compiler to find every class for compilation.	1
Runtime	These methods access raw memory or other unsafe operations and cannot be implemented in Java code.	3

Table 3.2. *Aspects identified that are not part of dJVM design.*

Chapter 4

Analysis and Validation

In order to analyze the usefulness of AOP, we compare the original patched version to the aspect-oriented implementation, and consider the relative strengths and weaknesses of each approach. Perhaps the most obvious difference is the number of lines of code with each approach. The patched version at 36,221 lines and the aspect-oriented version has 8,216[1]. The reduced number of lines has intuitive benefits in the realm of maintainability and understandability. The first section of this chapter discusses these benefits with regards to these software engineering principles. The second section discusses the improvements in regard to evolution of the base system. The third section of this chapter discusses negative results encountered during the development of the aspect-oriented version of the system. That is, this section overviews the portions of the patch we could not port to aspects. Possible solutions to these problems are introduced in the last two sections of this chapter.

4.1 Software Engineering Principles

The modularity of a system is known to increase fundamental software engineering principles by isolating related concerns so that they are more easily understood by developers. Along with understandability is the notion that maintainability and evolvability are improved, both of which decrease costs when developing a system desirable particularly in

[1]This is a conservative estimate based on working aspects for one of the memory managers and for the Intel architecture. Changes that could not be implemented with aspects are still written as comments so their line numbers are valid in comparison to those in the patch file.

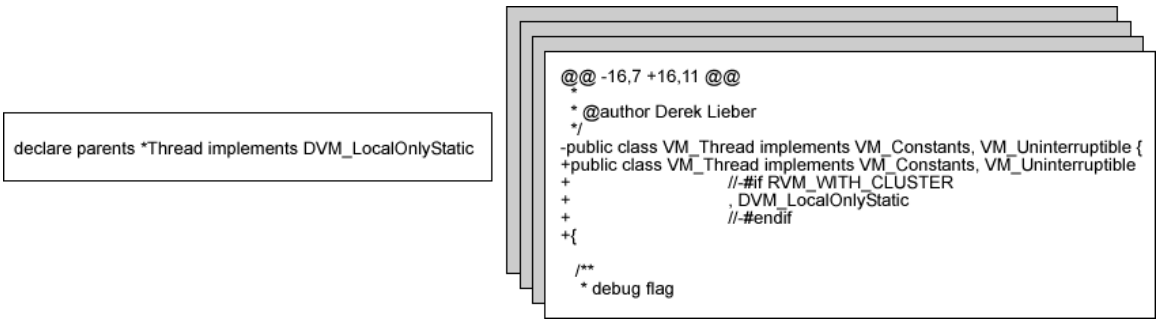

Figure 4.1. *Maintainability improved by consolidated code structure.*

infrastructure software. This section outlines the results of a qualitative analysis in regards to these and other software engineering principles. The principles we look at in addition to maintainability, understandability and evolvability include unpluggability, and flexibility. Evolution is extremely important as AOP is often cited as being able to create a more evolvable system. Therefore we look at evolution in more depth than the previous principles in the following section. The notion of evolvability that we look at in this study is the number of modifications that apply to a new system.

4.1.1 Maintainability

In regards to maintainability, if any modifications needed to be made in the system, we would have to manually go through every place that code exists modify it. With this approach, it is entirely plausible that we would miss one or even more points where this feature existed [34]. With the aspect-oriented approach, it is only necessary to modify the place where the feature is present one time. Figure 4.1 demostrates this point. If we express that every `Thread` class implements `DVM_LocalOnlyStatic` then we only need to modify the code in this one location whereas previously we had to manually modify every `Thread` class, such as `VM_Thread` and `MainThread`. In total there are 15 `Thread` classes within the system that are used amongst all of the configurations, not including the dJVM newly introduced classes.

Another issue is that if we wanted to make a change to distribution for the original

system, the dJVM system would need to be modified and new patches created. This entails a lot of extra work since we need to navigate through many classes and recreate patch files. This is exactly the way in which this functionality is shackled to an older version of the RVM since the changes required to roll the dJVM forward are tedious and error prone.

It is worth mentioning here that attempts to directly modify the patch file should be avoided. The patch file is automaticaly generated and closely dependant on offset numbers. Hence a change to even a small part in the patch code can create disastrous results.

4.1.2 Understandability

Understandability is improved because of the inherent property that AOP structures the points in execution of the program where distribution crosscuts the system. Since AOP localizes changes and the points to where these changes take place, it is easier for programmers to conceptualize what is taking place at a higher level.

As an example, consider the aspect shown in Figure 3.5 on page 38. Though this aspect is relatively straightforward, these changes account for a large portion of the patch, approximately 44%, of the changes in the system. In total, the original patch file consisted of 13,509 lines of code to accomplish what the aspect has done in 64 lines. But besides these dramatic quantifiable changes, we believe there is a more important impact in terms of reflecting the design intent of this aspect.

By inspecting the patch file, the design intent behind the set of classes that should implement the `DVM_LocalOnlyStatic` interface is not obvious. Through personal communication with the dJVM developers, classes which define global structures that are for intranode runtime systems are those that fall into this category. These include classes for the scheduler, thread handler, garbage collector, compilers and mechanisms for maintaining internal type information, Java Table of Contents. The problem is that the rule does not always apply and therefore, some trial and error is needed in order to define the subset of classes accurately. This is due to the distinction between identifying the behavior, versus how the behavior is implemented. There are no nice, clear-cut rules for which classes

this change should incorporate. The approach in the dJVM prototype was to explicitly tag classes with an empty interface for local only static behaviour, thus when the class is examined, it is clear how the statics are treated. On the other hand this approach leads to a large number of trivial modifications which are particularly hard to map between the design and the implementation in their scattered form. If we were able to define a meaningful set in an aspect using specific pointcuts and pattern matching, design intent would become clearer. Experimenting with refactoring of the original RVM was out of the scope of this thesis. However, if the code resembles the design, it has potential to be easier to understand and more tractable to manage. This issue is revisited in Chapter 5 on future work.

4.1.3 Unpluggability

Should we wish to have a version of the system without aspects, we need only rebuild it without using the AspectJ compiler. With patches we would have to roll back to reverse the impact of a patch. Aspects are extremely useful in this respect since they do not actually modify our original source code and the system can be configured with or without them and be rebuilt. The original RVM code does not have to know about the aspect although it may eventually be designed to accommodate it.

Frequently throughout the dJVM patch, the visibility of variables and methods were relaxed, i.e., private variables were made public. This was because in order to distribute the system, functionality between clustered nodes had to become less strict in order to facilitate their communication. This relaxation of access rights suggests that encapsulation must be rethought in the presence of distribution. Since a privileged aspect can access these private variables, this relaxation of visibility may not be necessary and these reassignments in access can then be restricted to within an aspect and can easily be removed.

4.1.4 Flexibility

As previously mentioned, source code files in the dJVM were not only Java files but also C files, template files and makefiles. One of the benefits of the patch was that it was able to modify all of these different files. AspectJ is only capable of modifying Java files, therefore the patch file modifications pertaining to non Java files were left intact and are still used in the build process. There are aspect-oriented languages such as Compose Star [9] that attempt to modify different languages but they are thus far experimental and not widely in use. The other related benefit to using patches is that they are able to modify Javadocs and comments in the original source files. Though it may be arguable as to whether or not these need be changed in the presence of aspects, we account for it here for the sake of completeness.

4.1.5 Summary of Software Engineering Principles

The benefits are summarized in Table 4.1. This table demonstrates the positive and negative results associated with each of patches and aspects in the context of our experience in the dJVM. We found that overall aspects uphold software engineering benefits because in each area of consideration, they outperform patches. The only area in which aspects were not clearly more beneficial is in the realm of unpluggability since patching can be reversed. However, this is not as easily done as with aspects.

4.2 Evolvability

The version of Jikes that the dJVM is built on is version 2.2.0. At that time, major changes were the addition of PowerPC support, the Jikes Memory Management Toolkit (JMTk) and package structure. The current release of the Jikes RVM, as of November 2005, is version 2.4.2. At this point in time, with the introduction of Apple's Intel-based machines, it is not clear that PowerPC support is still needed. This newer version of Jikes has had JMTk

	Patches	Aspects
Maintainability and Understandability	(–) changes are spread out (–) based on line numbers (and very little context)	(+) crosscutting concerns are localized (+) semantic leverage
Unpluggability	(+) easily applied (+/-) possible to remove	(+) RVM code never modified
Flexibility	(+) makefiles, C, Javadoc... (–) cannot adapt to structure changes (–) harder to integrate multiple patches	(–) Java code only (+) based on design intent (+) sustainable (+) can reason about and apply multiple aspects

Table 4.1. *Analysis summary of software engineering principles.*

replaced with the Memory Management Toolkit (MMTk) and also has had its package structure revamped.

This section provides an overview of major evolutionary trends over the period between these two versions and considers the impact this evolution has on the implementation of distribution as a patch versus aspects. This work is preliminary as the new, full implementation of the dJVM for 2.4.2 is not yet available. Therefore, the analysis discussed here is based on code inspection of the two versions of the Jikes RVM, as far as distribution modifications are concerned.

The work presented in [28] revealed the concrete ways in which aspects can improve the internal structure of some of the distribution code, clarify external interaction between the distribution code and the RVM, and reduce code size relative to the patch. These results bode well for easing evolution of distribution in the RVM with aspects. It requires nontrivial effort to both understand and maintain the code in patch form, even across minor RVM upgrades, due to the lack of semantic information and the fragile nature of textual substitution. This section aims to investigate just how evolvable these aspects might be by evaluating whether or not they will still apply to the new system.

4.2.1 Overview of Evolution

A more recent release of the dJVM is not yet available, for a variety of reasons. These include the replacement of the roll-your-own bytecode manipulation tool with the Byte Code Engineering Library (BCEL) [7], and converting over to use the non-blocking `java.nio` libraries. Both have introduced unforeseen complications. The first is that the classpath and RVM classloading mechanism are constantly changing, the second is that the GNU classpath implementation of the non-blocking libraries is not yet stable. Recent efforts in the project have gone into porting the bytecode analysis tools to those that BLOAT (a bytecode level optimizer) [8] provides.

The overall approach of the dJVM was to leverage the classloading mechanism as much as possible to effect both VM and application changes. This will remain the approach for the next version of the dJVM; however, a higher-level specification of those changes, perhaps in the form of aspects, is potentially quite worthwhile. In such a case the mutated classloading mechanism would include and/or be essentially a runtime weaver. However, we must also bear in mind performance is a long term goal.

There will be some changes to the dJVM design itself. One such change will be the inclusion of proxy objects (these will be identical to the original in terms of layout) which can act both as a cache for an object or as a method for redirecting calls. The purpose of this is to enable easy, and hopefully efficient, plug in of caching and replication algorithms. There is potential for aspects in this area in particular, though this implementation will be particularly sensitive and could have a serious impact upon performance and prevent many compiler optimizations.

As for evolution within the Jikes RVM itself, an area that is receiving attention is the removal of the trivial preprocessor directives. In particular, removing those directives that conditionally execute Java code, since a reasonable number of these can be replaced with standard Java condition statements as shown in Figure 4.2. The replacement of these statements should not incur any execution overhead for code generated by the optimizing compiler since it can do constant folding and dead code elimination.

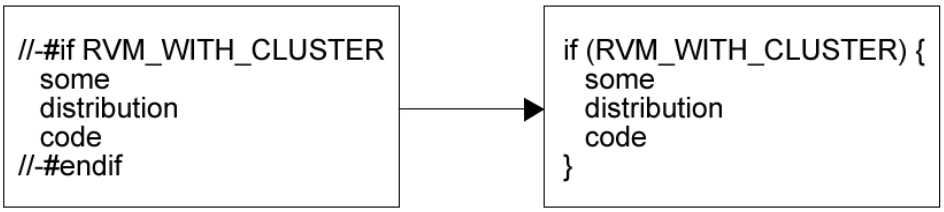

Figure 4.2. *Java condition statements.*

Improving the package structure and the interfaces between those packages, combined with the elimination of preprocessor directives should facilitate research using the Jikes RVM. Furthermore, such improvements would provide a better platform for considering what role aspects can play in the Jikes RVM and separately to the dJVM.

Currently using preprocessor directives for introducing empty tag interfaces is really circumstance dependent. For example if the restructuring of the Jikes RVM allowed a fairly succinct rule for replacing where the `DVM_LocalOnlyStatic` is used (i.e., a rule that someone could keep in their head so that they knew why and where it would be applied) then it would be replaced. However, if the rule becomes complex or obscure then an explicit annotation may be preferred. It is really a matter of cognitive effectiveness, or at least the developer's perception of what that is. An aspect for `DVM_LocalOnlyStatic` is more likely to be effective if the RVM structure is a little cleaner.

The high-level specification of mutations instead of lower level annotations is preferable from the perspective of dJVM development. Aspects would be more appealing if the Jikes RVM package structure and interfaces between those packages were improved, this would in turn help reduce the complexity of the rules identifying where aspects would be applied.

4.2.2 The Toll of Evolution

In the aspect-oriented prototype implementation of distribution, there were five types of aspects that were considered in the evolution analysis. These aspects included hierarchy changes made in the `LocalOnlyStatic` aspect, changes to the optimizing compiler,

functionality for the PowerPC architecture, a large collection of newly added variables and methods to existing classes (inter-type declarations), and a large collection of advice. There will also be a lot of extra functionality required due to new classes introduced to the system during evolution, but that artifact is out of the scope of this study until the new dJVM patch is available.

4.2.2.1 Hierarchy Modifications

Perhaps one of the largest changes to the latest version of the Jikes RVM in terms of distribution is that the Java Memory Management Toolkit (JMTk) has been made independent and is now known as the Memory Management Toolkit or MMTk [32]. Much work has been done on factoring out the VM-specific code, which now resides outside of the Jikes RVM source tree, and it has been reorganized to have an Eclipse-friendly directory/package structure. But it is not just memory management structure that has changed. In fact, some of the code originally located inside of the root package of Jikes has now been migrated into two new packages, `classLoader` and `jni`, and three packages (including their code) have been removed from the old version of the Jikes RVM. These include the `BytecodeToolset`, `ClassTransformer` and `librarySupport` packages.

The inclusion of MMTk means that some of the distribution modifications for the dJVM are lost. However, the aspect which is affected most by this is the `LocalOnlyStatic` aspect in which all of the members of the JMTk package were affected (40% of the changes in this aspect). It may seem logical to attempt to weave into every file within the MMTk package instead. However, this is likely to select too many files and therefore cause problems so the changes to memory management in the `LocalOnlyStatic` aspect will most likely have to be completely revised as a result of this evolutionary change.

Another interesting point about the `LocalOnlyStatic` aspect is that it takes advantage of wildcard expressions which capture different files in each version of the system. For example, `VM_NativeDaemonThread`, which does not exist in version 2.4.2, fell under `*Thread` in version 2.2.0. The AspectJ compiler [4] will not throw a compiler error telling

us that this class does not exist since we did not explicitly define it, therefore making the aspect more robust. This property of AOP makes it evolvable to new versions of a system. Each of these wildcard expressions was evaluated to make sure that they were not specifying any new files that were not supposed to be woven in the new version. In only two out of 15 cases, did the wildcard expression match any new files. In those two cases, it is highly likely that the new files should indeed be matched. In eight out of 15 of those cases, the new version was missing files that were originally caught by the wildcard expressions in version 2.2.0. However, by not explicitly naming these files, our aspect will still compile and run.

The addition of these empty tag interfaces (such as the previously mentioned `DVM_LocalOnlyStatic` as well as `DVM_NoRemoteAccessor`, `DVM_NoProxy`, `DVM_NoRemoteInvoke` and `DVM_NoRemoteAllocation`) was made to 44% of the original 1166 Java files, many more than shown in Figure 4.5. This is due to the fact that many of these files were not in our configuration. However, the files we used and files from other configurations tend to have the same class names but reside in different directories within the source. Therefore, we expect the same number of changes to be unusable, no matter which configuration is used. This also applies to the following aspects, although not to the same extent. For more information on Jikes configuration options, see [15].

4.2.2.2 Optimizing Compiler

The files and methods which were modified in the optimizing compiler aspect all still exist in the new version of the Jikes RVM. Additionally, the types of changes that were made within this aspect are highly evolvable. This is due to the fact that the changes include the addition of getter methods, before advice that is not dependent upon the code that follows it in the original method (shown in Figure 4.3), and lastly, `around` advice which never proceeds to the original method's implementation (shown in Figure 4.4). This type of `around` advice means that if the implementation of the actual method has changed in the new version of the Jikes RVM, we are less likely to need to change the aspect since we

```
before(VM_Address objRef, boolean root): scanAssertion(objRef, root)   {
          // if it is a remotely held object, then ignore as this
          // is outside our scope.
          VM._assert(!objRef.isRemote());
 }
```

Figure 4.3. *Before advice that is not dependent on the method it advises.*

```
Thread around(): execution(Thread ThreadSupport.getCurrentThread()) {

 VM_Thread thread = VM_Thread.getCurrentThread();
 VM_Thread ident  = thread.getClusterThreadIdentity();

 if (ident != null)
     return (Thread)(ident);
 else
     return (Thread)thread;
}
```

Figure 4.4. *Around advice that never proceeds to the original method implementation.*

never use the original method's code.

4.2.2.3 PowerPC

The PowerPC aspect adds the functionality necessary to generate inline code for the `VM_MagicNames.invokeStubMethod` method when the `VM_MagicCompiler.generateInlineCode` method is executed. However, not only does the `invokeStubMethod` no longer exist within `VM_MagicNames`, but the `VM_MagicCompiler` class itself no longer exists and has been absorbed into `VM_Compiler`. As a result, this aspect will need to change completely. Additionally, with the release of Apple's Intel machines, it is likely that PowerPC support will no longer be provided. If this were the case, it shows how useful it can be to provide architecture specific aspects which can easily be evolved or even unplugged in response to hardware evolution.

4.2.2.4 Inter-type Declarations

In regards to inter-type declarations, it is really only important to see whether or not the target classes still exist within the system. As mentioned previously, some of the original RVM files were migrated into a `classLoader` package. In total, nine of the files modified in this aspect are now within this new package. However, if all required packages are imported, the aspect will find the files and weave into them no matter which package they are in. This means the only changes needed in this aspect will be different import statements. However, 11 out of 36 of the files modified in this aspect no longer exist in the new system so the changes made to them will likely require migration to new classes.

4.2.2.5 Advice on Methods

Lastly, changes to methods were evaluated by inspecting the join points and seeing if (a) the classes and methods themselves still existed, and (b) judging by the implementation in the methods, whether or not the advice would still be applicable.

In regards to (a), 10 out of 44 changes are no longer usable because their methods or classes are no longer present. In evaluating (b), since some of the advice was `before` or `after` advice, then as long as the method signature was the same, we would still be able to bind to it. Problems arose, however, when we had `around` advice. Due to difficulty in implementing changes to the middle of a method, and because no refactoring [44] was done to incorporate aspects, the whole method was copied to the around advice with no proceed functionality. In this case, if the method implementation had changed at all, our aspect would be out of date. Because of this, seven out of 44 changes may not work. With refactoring, made especially easy if we had automated refactoring tools, our aspects would have a better chance of survival.

The final results of this indicate that for the most part, the aspect would be applicable to the new system with approximately 35% of the changes being those made to non-existent methods or those in around advice that are likely to require modification.

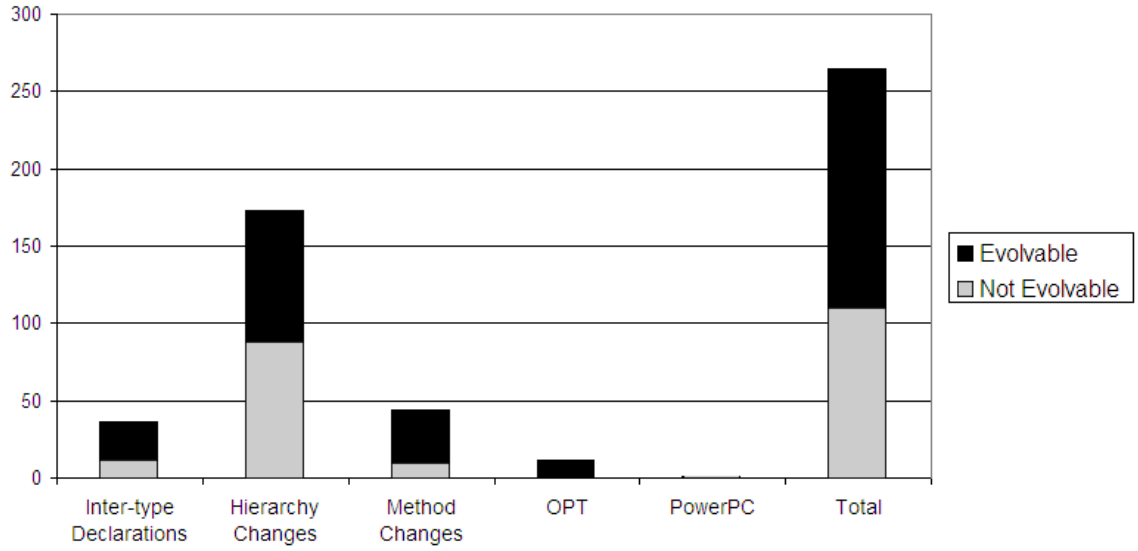

Figure 4.5. *Modifications that are applicable to Jikes 2.4.2*

4.2.2.6 Evolution Summary

The above has shown that overall, aspects are 60% more evolvable than their patch counterpart since patching is rigidly based on line numbers and therefore not at all evolvable. Each aspect performed differently as the RVM itself evolved and different types of advice and inter-type declarations also had different evolution characteristics. The overview of the number of evolvable changes is shown in Figure 4.5 where the light grey space, shown on the bottom of the bars, shows modifications that could not evolve with the system and black space, top portion of the bars, shows those that could.

4.3 Negative Results

Even though aspects improve the structure of distribution, there were various problems found in their actual implementation. Many of these difficulties were due to Jikes' unique build process. Other problems arose due to the way in which the dJVM developers implemented distribution by hijacking Java mechanisms and yet more problems arose due to the

Jikes' boot process in regards to method resolution. These issues are introduced here and possible solutions to them are discussed in the following section.

4.3.1 Class Hierarchies

Figure 3.5 on page 38 shows the `declare parents` construct in AspectJ that is used to modify the class hierarchy to introduce `DVM_LocalOnlyStatic`. The problem with this aspect is that semantically, a superclass implementing the interface should mean that the subclass inherits that property and therefore should not also implement that interface. The dJVM developers had actually overloaded the inheritance construct since `DVM_LocalOnlyStatic` is deliberately not an inheritable property. It is an annotation on a specific class, not a class and its specializations meaning that the annotations are not inherited. This means that no matter what the inheritance structure is, those statics that have local meaning can still be separated from those that have global meaning. This is a deliberate choice since a subtype may want genuinely global statics whereas a supertype may want node local statics. This is similar in concept to that of the other annotations used in the Jikes RVM. Unfortunately, unexpected approaches such as these are used frequently in systems code.

The compiler developers would have made extra effort to put optimizations such as these in place to help the developers. However, without issuing warning messages, developers are left in the dark about what is really happening whilst their code is being woven, therefore adding fuel to the obliviousness fire that AOP already faces. Obliviousness meaning that developers are not necessarily aware of changes made to their base code. The lack of feedback about which classes were actually being woven is discussed further in Section 4.3.3 on page 66.

It is interesting to note that in Sun's Java 5, annotations have been introduced and are also incorporated in the newest version of AspectJ so that annotations can be advised. This could be used instead of the empty interface approach; however, the version of Jikes that the dJVM is based upon is not only written for an older version of the Java specification

but for another version of the compiler altogether, the Jikes Java compiler [12].

4.3.2 Source versus Bytecode Weaving

In an ideal environment, the base system would be compiled and distribution code would then be woven in. As it turns out, since files other than Java files were patched with distribution functionality, these files introduced dependencies. Of particular interest were Java template files which were used to auto-generate other Java files for garbage collection. The auto-generated files could not compile without modifications such as introduced variables and methods that were present in the patch file. Therefore, there was no way to compile the system in order to do bytecode weaving without completely overhauling the build process. The easiest way to introduce distribution code then was to introduce these members with source code weaving before allowing the system to compile. Unfortunately, this meant rolling back to the earliest version of AspectJ which wove into source code instead of byte code.

Advice could actually be woven with bytecode weaving since they introduced no dependency problems in the initial compilation of the system. However, one of the huge drives behind AOP is that of understandability where we strive to make the code look like the design. In the case of the thread identity aspect shown in Figure 3.2, this aspect makes sense in that it has both inter-type declarations and method advice. Splitting these apart for the point of using the latest compilation tools meant sacrificing design. For this reason, we decided to only use source code weaving even though it meant using outdated tools.

In addition to improving design, it was important to have inter-type declarations and advice in the same aspect for another reason. When a private variable is introduced with an inter-type declaration, its name is introduced followed by the aspect's package and class name (as seen in Figure 4.6). This is so that it remains private within the aspect that introduced it which means it differs from the name the bytecode weaver aspects are expecting. For this reason, when a private variable is referenced in a bytecode woven aspect, its visibility would have had to be changed from private to public. The same situation applied

```
private VM_Thread clusterThreadId_com_ibm_JikesRVM_ThreadIdentity;

public final VM_Thread getClusterThreadIdentity() {
  return this.clusterThreadId_com_ibm_JikesRVM_ThreadIdentity;
}

public final void setClusterThreadIdentity(VM_Thread aClusterThreadId) {
  this.clusterThreadId_com_ibm_JikesRVM_ThreadIdentity = aClusterThreadId;
}
```

Figure 4.6. *Private inter-type declarations as introduced by AspectJ.*

to protected variables. An interesting side note was that we could not explicitly introduce protected variables or methods with the source code weaver. Therefore, these variables and methods were introduced with default visibility instead. Figure 4.7 shows how the aspect is decompiled. We see in this figure that the `around` advice becomes a public non-static method with `around` in its method signature. Methods to return the instance of this aspect are added which are used when the `around` advice is called from the advised method.

As mentioned previously, the Jikes RVM was intended to be compiled by the Jikes Java compiler. Therefore, the Jikes RVM contained code that could not be compiled with Sun's Java compiler. This particular code allowed the developers to assign constant variable names to primitive types on the invocation of the Jikes compiler. This is shown in Figure 4.8.

It was thought originally that by using a source code weaver, we would be able to compile our source files normally. However, as it turned out, since the source code weaver still requires all source files to be parsed, errors are thrown that `INSTRUCTION`, `EXTENT` and `ADDRESS` are all types that could not be recognized. For this reason, the Jikes base code had to be modified to change these types to their primitive counterparts. It is unreasonable that we should be able to use a source code weaver in order to use a Java compiler of our choice, yet we are still limited to the textual parsing rules of Sun's specification without the ability to turn them off.

Weaving into source code alleviated dependency issues and there are many other sys-

```
/*   Generated by AspectJ version 1.0.6 */
package com.ibm.JikesRVM;
import com.ibm.JikesRVM.librarySupport.*;

public class ThreadIdentity {

    /**
     * This should be hidden from the user application. Set the current identity
     * of the thread
     *
     * @param aClusterThreadId
     *            The new identity of the thread.
     */
    public final Thread around0$ajc(final org.aspectj.runtime.internal.AroundClosure
        ajc$closure) throws Throwable {
        VM_Thread thread = VM_Thread.getCurrentThread();
        VM_Thread ident = thread.getClusterThreadIdentity();
        if (ident != null) return ((Thread)(ident));else return ((Thread)thread);
    }

    public ThreadIdentity() {
        super();
        ;
    }

    public static ThreadIdentity aspectInstance;
    public static ThreadIdentity aspectOf() {
        return ThreadIdentity.aspectInstance;
    }

    public static boolean hasAspect() {
        return ThreadIdentity.aspectInstance != null;
    }

    static {
        ThreadIdentity.aspectInstance = new ThreadIdentity();
    }

}
```

Figure 4.7. *Source code of around advice within an aspect.*

```
   $HOST_JIKES +F +Z +KINSTRUCTION=byte +KEXTENT=int +KADDRESS=int +E -g +U -classpath
    .:rvmrt.jar:$ASPECTJ1_HOME/lib/aspectjrt.jar Dummy.java
```

Figure 4.8. *Invocation of the Jikes Java compiler.*

tems with a complicated build process where such a source code weaver could be useful. Unfortunately, there are some differing opinions on what it means to use a source code weaver.

In order to understand AOP, it is important to think of aspects as modules on their own and not merely as code injection mechanisms such as a patching approach. By using source code weaving, it might be possible that programmers would see them this way. Of course, how the tool is actually implemented behind the scenes may not be important if we never investigate the temporarily woven directory of source code. But there are also performance concerns – it is faster to manipulate bytecode than to parse and modify text files taking into account the time required to read and write them. However, left as an option, the developer could decide for themselves whether or not an improvement in compilation performance is important.

4.3.3 Implementation Obstacles: Aspects and Segmentation Faults

Debugging is integral to development, especially when programming within a difficult system such as Jikes. During the development of the dJVM using aspects, there were two general types of problems encountered which are associated with hierarchical changes and with low-level inlining requirements. The causes for both problems were completely different but the final results were the same – a segmentation fault.

When implementing empty interfaces for the `LocalOnlyStatic` aspect, the system ended with a segmentation fault and it was initially assumed that a file must have been missed in the list of files provided to the AspectJ weaver. Painstaking effort was made to find the offending files which were `VM_Properties` and `VM_Statics`. In order to discover the problem, the Fast Java Decompiler (JAD)[13] was used to assess how the files were being modified by the AspectJ compiler. The decompiled files told us that these particular classes were not actually being modified at all but the question of why still remained. As implementation continued, we continued to decompile and check that files were actually being woven with the empty interface and the list of files that did not grew (although no

other files caused segmentation faults, these files did cause other problems to occur). By enabling the verbose mode of the ajc compiler, it could be verified which files were actually being woven but why only a subset was being chosen was still not clear. As mentioned previously, the actual reason behind this was that superclasses of those files had already implemented or extended the empty interface. However, with no warning messages being issued, it was difficult to discover why these errors were occurring.

The reason why *inlining*, or introducing code directly into the body of a method without introducing a method call, is so important, over and above performance, is the second reason why segmentation faults occurred. Inlining is discussed in more depth in Section 4.5.2.1 on page 75. At a glance, when certain synchronized methods or methods otherwise related to concurrency and locking in the Jikes RVM were advised, they caused segmentation faults, even when that advice was empty and even when we used the call join point instead of the execution join point. Through personal communication with one of the dJVM developers, we were told that it may be a matter of runtime resolution of the method. Occasionally when a method is compiled it refers to a method that has not yet been compiled and in such a circumstance it will insert code to perform the resolution (and if necessary loading/compilation) of the target method. This also occurs when building the VM since there is some order in which methods are compiled to native code. If we had the power to inline advice at will, these segmentation faults would not have occurred.

4.4 Analysis: Tool Support for System Extensions

Software development tools are steadily becoming easier to use, but these same tools are becoming so "helpful" that they can actually do more harm than good. Tools are supposed to help evolve and adapt systems – ironically however, they are often too rigid to be able to evolve and adapt themselves. In this section, we consider the synergies and interoperability characteristics, as well as the design of what we believe to be necessary for integrated tool support, for patch, preprocessor directives and aspects. This design is established

based on evidence gathered from both OS and VM infrastructure software. Interoperable System Infrastructure Support (ISIS) is proposed as a tool that can better evolve and adapt according to a system's needs [29]. Details of the proposed tool are explained more fully in Section 4.5 .

4.4.1 Intersection: Patches, Preprocessors, and Aspects

Though Table 4.1 highlights the benefits of aspects over patches, we do not conclude that all patches would make good aspects. However, those that contain concerns that would benefit from (un)pluggability, improved internal structure and explicit external interaction could be good candidates. In our experience, in addition to the dJVM, we have come across several patch files for Linux that may fall into this category. These include Linux-Tiny [16] which reduces memory and disk footprint of the mainline Linux kernel and requires changes to 142 files, and VServer [17] which is a virtual private server implementation added to the Linux kernel that requires changes to 211 files in the Linux kernel, respectively.

With respect to preprocessor directives, it is interesting to note that within the dJVM patch, most but not all segments of distribution code were introduced with directives. It turns out that this code was either an oversight or the code involved was never intended for use in the current release of the dJVM. This shows precisely how oversights can occur because of the widespread nature of these changes. So, we are not suggesting that all of the 4,000+ preprocessor flags in Linux would make good candidates for aspects, but we have identified several that could benefit from moving to a more structured implementation for crosscutting concerns.

4.4.2 Existing Limitations: Lack of Support for Extensible Systems

This section discusses limitations of each of the core approaches – patches, directives, and aspects – associated with building extensible system infrastructures. Based on the evidence from the dJVM, we conclude this section by summarizing why we believe there is a need

for interoperable and adaptive tool support for system infrastructure software.

4.4.2.1 Patches and Preprocessor Directives

Without any navigation support or ability to discern where exactly changes were being made in the system, it is hard to understand how distribution works at a high level. To contrast the patch with aspects for meaningfulness, making a change to the third switch statement in the middle of a method does not tell system infrastructure developers much about the changes being made to our system. However, if this change actually meant that before every array copy, we needed to do a remote array copy, then these changes begin to make sense.

4.4.2.2 Inability to use Java and AspectJ Tool Support

Unfortunately the complexity of the Jikes RVM project has meant that it is very hard to even view the source code within Eclipse [10], and currently impossible to actually build Jikes inside Eclipse (Eclipse is an extensible development platform and application frameworks for building software). This is all due to the complexity of Jikes' build process which currently consists of a set of complex bash build scripts, makefiles and a preprocessor. Most notably, the incompatibilities arise from the inclusion of preprocessor statements, especially when they contain complex code such as `if-else` statements, and also because the Jikes' source tree structure differs greatly from its compile-time structure.

The inability to build certain Java-based systems within Eclipse is a large frustration for many developers. The memory management subsystem that is currently employed by Jikes is the Memory Management Toolkit or MMTk [32]. Much effort was involved in having this Java-based memory subsystem be pluggable and in addition, to have it compile and run within Eclipse. The challenges faced in order to have MMTk build within Eclipse are the same that Jikes faces, namely, the use of preprocessor directives and the unique file system structure of the system. The solution, at a high level, would be to eliminate preprocessor directives by creating a "glue" module that defines boolean constants according to pre-

processor directives, and then replace preprocessor directives in the main code with these constants. As for package versus source code structure, the approach used within MMTk should also work for the VM. This would involve dividing the Jikes RVM into relatively autonomous modules which can be edited in Eclipse, using stub packages for interfaces to parts of the system which cannot, and isolating the Eclipse-unfriendly code into (hopefully small) "glue" modules.

4.4.2.3 Interoperability and Adaptivity Between the Approaches

It is not unreasonable to assume that all three approaches considered here will inevitably co-exist in most highly-adaptive system infrastructures. The problem then becomes twofold. First, there is the issue of interoperability for any tool that is designed to support multiple mechanisms for adaptive system behaviour. That is, tool support for adaptive systems must allow multiple mechanisms to be simultaneously considered when a developer has to reason about the system as a whole. Second, this tool must itself be adaptive. This is not only because future mechanisms will be introduced, but also because more extensive tool support for individual mechanisms must be incorporated to work together with other mechanisms throughout the system. In particular, we see great potential for a tool that can be used to adapt primitive mechanisms, such as patches and directives, to more modern mechanisms, such as aspects, where appropriate. As previously mentioned, we believe that such a tool would also allow for the ability to detect possible conflicts between competing mechanisms that concurrently work in support of adaptivity within a system.

4.5 Interoperable System Infrastructure Support (ISIS)

Borrowing from our experience with aspects, these new tools for patches and preprocessor directives will allow us to better visualize how patches and directives may crosscut a system. An example of this would be allowing developers to navigate between patched code and back again, and more interestingly, automating the process of rudimentary aspect

refactoring from a patch file or a set of preprocessor flags.

Our proposal is to build ISIS in a way that is language agnostic, in that either AspectC [2] or AspectJ could be the AOP engine. Our current efforts have been directed towards managing preprocessor directives in JVM code that are not activated for a current build configuration. That is, the tool hides code associated with preprocessor flags not in use. The prototype implementation for this particular feature of ISIS, APProVer (Aspect-Miner and Preprocessor Viewer) [53], is currently being developed and evaluated by a small set of developers working with infrastructure software written in C. APProVer generates specific build views from a single master source. The tool allows developers to more easily refactor large systems without having to sift through a code base saturated with preprocessor controlled sections that do not pertain to a chosen build.

4.5.1 Related Tools

There are two tools that have had a significant impact on the understanding and adoption of AOP. These two tools are the AspectJ Development Tools (AJDT) [5] project and crosscutting visualizers. These tools already exist but at this time, cannot work with complicated systems such as Jikes. They are discussed here because ISIS will incorporate their supporting features and expand their application to more primitive mechanisms commonly found in adaptive systems such as preprocessor directives and patches.

4.5.1.1 AspectJ Development Tools

The AspectJ Development Tools project is an Eclipse plug-in that provides tool support for Aspect-Oriented programming, in particular for AspectJ. It has been argued that aspects are oblivious because the base code never knows that it is being advised. AJDT provides gutter annotations that show where base code is being advised and by what aspects. Conversely, the aspect also contains gutter annotations that tell the developer which join points are being advised. This eliminates the need to decompile code in order to see if the advice

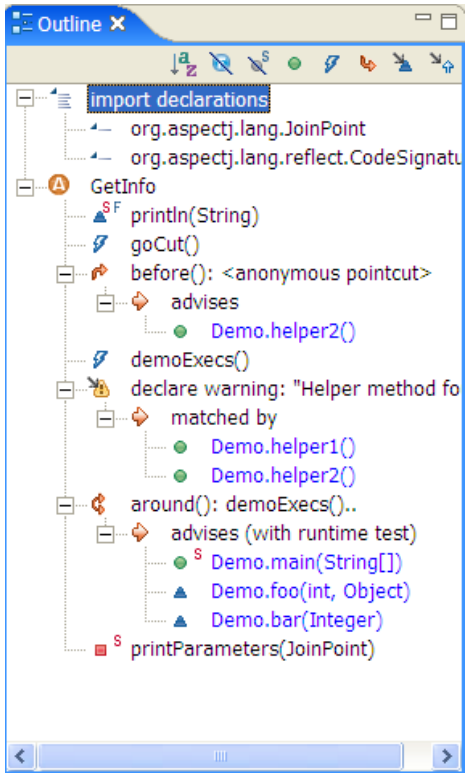

Figure 4.9. *AJDT Outline View.*

indeed has correctly applied and what changes have in fact been made to the base code. AJDT is also capable of showing hierarchical information about an aspect-oriented system as shown in Figure 4.9.

We briefly discussed patch conflicts earlier and it is possible, although less likely, that two aspects would also have conflicts that require resolution. The reason conflicts are less likely with AOP is that aspect-based conflicts would be based on the actual modifications that the aspects make, whereas patch conflicts arise from line numbers as well as the modifications they make. With AJDT allowing us to navigate to join points and showing which other aspects also advise that join point, we are more easily able to detect aspect conflicts. There are other advantages to using AJDT such as the integrated Eclipse debugger. The debug perspective is shown in Figure 4.10.

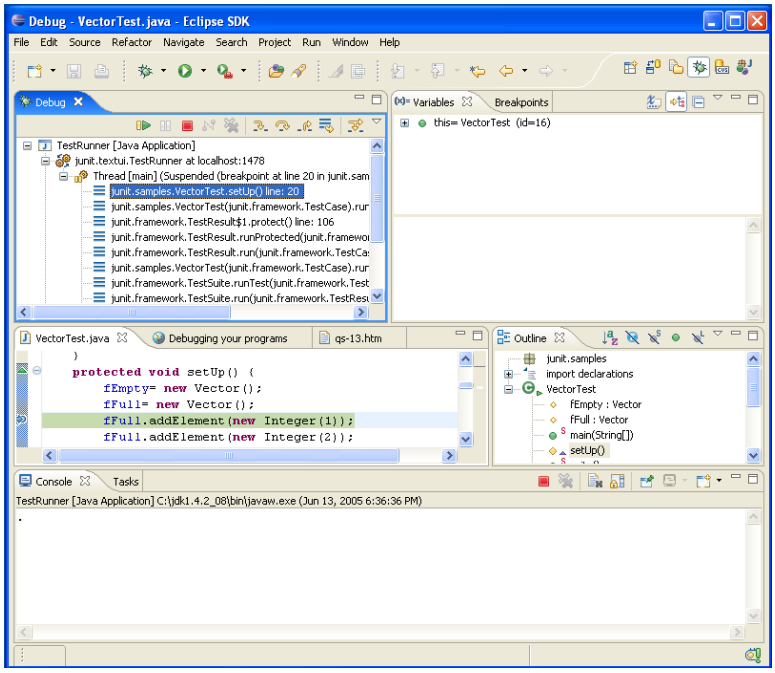

Figure 4.10. *Eclipse's debug perspective.*

AJDT also requires code to compile in order to gather the necessary aspect-specific information that is used for such tasks as the gutter annotations, navigation and visualizations. However, because Jikes is unable to build within Eclipse, the final result is that we are unable to use AJDT and all of the important mechanisms that it provides.

4.5.1.2 Crosscutting Visualization

One of the most interesting tools in the realm of AOP is that of crosscutting visualizers. These tools show rectangles as modules and use color coded lines to show the crosscutting of concerns. There are various tools that exist to show this type of crosscutting behavior. AJDT itself provides its own visualizer, although we were unable to use it due to the issues experienced with AJDT and Eclipse that are outlined in the previous section. We show an image created with the AJDT visualizer in Figure 4.11 that illustrates the crosscutting nature of a debugging aspect.

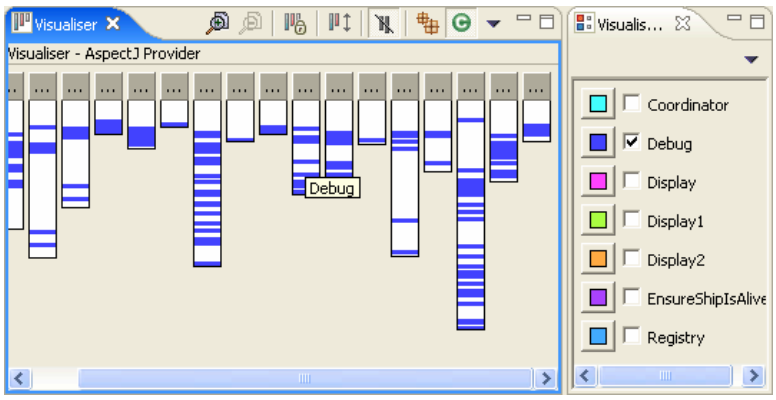

Figure 4.11. *Crosscutting visualizer.*

There is also another popular visualizer called the Aspect Mining Tool (AMT) [45]. AMT requires the user to run the AMT analyzer to extract necessary line-oriented statistics about the system which it then uses along with the aspects and a modified version of the AspectJ compiler to create the crosscutting depiction. Again, since Jikes has such a unique build process, we were unable to use the regular techniques used by AMT to build these program statistics. Instead, we tried to incorporate the data gathering step within the build process of the dJVM to overcome issues between source file structure and compile-time structure. Sadly, this approach did not work since the AspectJ compiler was unable to parse some of the irregular Java code within the Jikes source files. Strangely enough, the latest AspectJ source code weaver was able to cope. This anomaly was most likely due to the aspect mining tool using a different version of the AspectJ compiler or having limitations that restricted it to simple Java constructs.

The Aspect Browser [43] is another mature visualizer which shows the promise of flexibility since it does not rely on code compilation but instead on regular expressions which match aspects textually to the points of execution that they crosscut. This also means that it is capable of showing crosscutting information for more than just Java files so it can be used as a visualizer for other aspect-oriented languages such as AspectC. Currently, the Aspect Browser comes in two forms, one of them being an Eclipse plug-in. However, unlike other

Eclipse plug-ins, without a dependency on compilation, we only need to provide the code as text to the Eclipse project and therefore are still able to use this tool.

4.5.2 Additional Support

It is increasingly common for developers to make use of tools that are so integral to the development process that they are almost part of the language themselves. Example of such tools are inlining and most importantly, debugging. Inlining can be controlled with the aid of a Just-In-Time Compiler (JIT) that aims to improve performance. However, without the control to specify where inlining should take place, we lose control over our own code. Additionally, ensuring correctness of a system, especially system infrastructure without the help of a debugger, can be a harrowing process. We find that debugging support is insufficient in the particular case of Jikes, and this is unfortunately the case for many complicated system infrastructures. The following sections provide more detail regarding the trade-offs associated with problems encountered with inlining and debugging while introducing distribution to the Jikes RVM.

4.5.2.1 Inlining

AspectJ may add overhead to the performance of the dJVM for a number of reasons, the primary reason being that of the increased cost of method calls from the original classes to those within the aspects [27]. Since the impetus behind distributing the RVM was to boost performance, a performance hit may not be acceptable. Therefore, it makes sense that we might want to inline advice to aid performance but as we have seen, there is another, far more important reason for us to inline which was discussed in Section 4.3.3. First, we will discuss why inlining was not made an option in AspectJ.

Inlining can be done much more effectively by a JIT than by a tool has to follow Java's access rules [46]. This is important because code within an aspect must follow Java's standard lexical accessibility rules. However, code within an aspect contains its own methods

and variables that are private within that aspect. Inlining advice then means that those aspect methods/variables will have to be visible to the inlined code and would therefore require increased visibility. Another reason for not inlining may be that if we decide to use a debugger within our woven system, or if an exception is thrown, the stack trace would not include the aspect. The trade-off between performance and debugging support is a decision that we, as the designer of our system, should be allowed to make.

Another AspectJ compiler, the AspectBench compiler for AspectJ, abc [22], supports inlining. The problem here is that a heuristic is used that decides for the programmer whether or not a method should be inlined. The developer can decide to either inline aggressively or not inline at all which means that we still do not have control over when inlining occurs.

4.5.2.2 Debugging Support

The previously mentioned causes of segmentation faults in Section 4.3.3 are two motivating reasons behind why we need a debugger, but we still have not answered the question of why one does not exist within Jikes. The two year old version of the Jikes RVM on which the dJVM is based is version 2.2.0. This happens to be the first version in which the system debugger was removed. The RVM developers decided that users would require a debugger for their own programs that they ran on top of Jikes and not for the RVM itself. Therefore it was concluded that maintaining the debugger outweighed its usefulness and it was removed from the system. Jikes was designed to be a testbed for developers to prototype new virtual machine technologies, so could it really be true that we would no longer need to debug the system itself?

As it stands, debugging within the RVM is exceedingly difficult since there are no good debugging tools. In the worst case it may be necessary to use The GNU Debugger, or *gdb* [11], to examine the assembly code. Since Java bytecode is not machine executable, Jikes has to be loaded through a C boot loader. The boot loader loads enough of the Jikes system that the Java based portion of the system can begin to run, then the full system can

be loaded. Compiling the C code with the `-g` option only includes symbolic information for the C code and not the RVM compiled Java code, and including the `-g` flag for the Java code does not really provide much additional information when the RVM compiles its methods to native code. The particular types of problems, such as segmentation faults, are some of the worst to debug. In an effort to provide a record of our experience with the dJVM, we detail the shortcuts we used to find some point in the image we could execute to and then use gdb to disassemble and step through the machine code or to use some code transforms to insert debugging statements. These details can be found in Appendix C.

4.5.3 Validation: An Integrated Approach

The following discussion is a vision for the future in regards to what we believe is necessary support in terms of integrating fundamental extension mechanisms within system infrastructure. Previously, we discussed problems with visualization tools. However, if we want to create a generic set of tools to aid in the evolution of patches to aspects, then a patch/preprocessor crosscutting visualizer would help. The tool's visual output would be the same as shown in Figure 4.11 but instead would show how patches and directives crosscut the system. This would be integral in showing whether or not a patch or code controlled by directives might become valuable aspects. Additionally, it could aid the identification of possible conflicts between approaches.

When viewing patched/processed code within an visualizer, it would also be valuable to show which lines of code are involved in a highly accessible way. This would be done via paragraph color coding and would be expanded to allow color coding for different patches, which could ultimately serve as a conflict resolution tool. Additionally, in the case of patch files, it would be advantageous to have a navigation tool that would allow jumping from a patch modification section directly to where it would apply within the base code. This was one of the more tedious, although not difficult, tasks during the implementation of distribution aspects for the dJVM.

Finally, we envision an automatic refactoring tool to create an aspect from a patch or

preprocessor controlled segment. This would entail viewing code that introduces adaptive behaviour from within an IDE that had knowledge about the base system. By right clicking on a segment from a patch or preprocessor directive, the tool would be able to find the point within the original code, evaluate to the best of its knowledge the principled point of execution and extract the changes to an aspect. In the case of a patch, this would remove much of the mundane effort in searching original files for line numbers in order to see where changes were made. This would be especially easy for introducing inter-type declarations via aspects.

A possible point of contention with an automatic refactoring tool is that of intelligence. This tool would only be able to render a chainsaw effect by giving rudimentary separation of concerns by pulling out code into aspects without much refactoring of either the base code or aspect code. It will still remain up to the developer to discern the nuances of design. However, tools such as these would decrease the amount of grunt work required to evolve outdated code to that of modern programming techniques. For example, in our experience with the dJVM, the first refactoring effort contained five aspects. But after careful consideration of the design of distribution, this code was then further refactored into 16 different but coordinated aspects for distribution, shown in Tables 3.1 and 3.2. A further form of refactoring could incorporate refactoring of the base code to further accommodate the aspects. We have begun to assess the point of diminishing returns associated with these different levels of refactoring in other work [41].

4.6 Chapter Summary

This chapter has outlined the benefits we see with an aspect-oriented implementation of distribution within the dJVM. We have discussed such "ilities" as maintainability, understandability, flexibility, unpluggability and most importantly, evolvability. However, it is important to mention that not all of the dJVM implementation was ported to aspects due to insufficient support for the homegrown mechanisms within Jikes that could not be ex-

pressed. Additionally, the lack of debugging support restricted development within this environment, especially since inlining was not supported by the AspectJ compiler.

Finally, we discuss improvements in current tool support for system extensions. We believe that one of the more critical features of evolvable tool support stems from the need to include program analysis technology that detects interference between extensions of a given system. Specifically, when there are multiple means of introducing extensions, some form of cognitive support for reasoning about when code introduced by any one of these approaches conflicts with code from any other approach is critical.

Chapter 5

Future Work and Conclusions

This chapter describes future work for this project and overviews the conclusions of current work. Future work for this particular case study includes extensive testing, both regression and stress testing. We also would like to do performance testing and fully implement distribution on the newest release of Jikes to provide further evolution results. The following sections detail each of these future considerations more carefully. The last section of this chapter summarizes conclusions from this thesis.

5.1 Future Work

First and foremost in future work is testing. The dJVM currently does not provide a test suite and even though the aspect-oriented version of the system has been tested on small examples, it is important to apply regression testing so that we can ensure no other original functionality was corrupted. Additionally, one of the driving motivations behind distributing Jikes was to cause an improvement in performance and there is a possibility that aspects could degrade this. It is important to determine if this is the case, and if so, whether the tradeoff in regards to improved modularity is worthwhile. Finally, we also need to further validate our claims regarding evolution by applying the aspects and distribution as a whole to the newest release of the Jikes RVM instead of by using manual code inspection.

5.1.1 Testing

The Standard Performance Evaluation Corporation (SPEC) provides a suite of tests designed especially for Java virtual machines called the SPEC JVM98 Benchmarks [20]. These tests would require dJVM specific code and alteration to run on the dJVM, but they would provide a standard test suite that would be extremely beneficial for future tests on different versions of the dJVM.

A test suite for the dJVM would be invaluable to ensure that nothing has been broken in the original Jikes system. We would also like to stress test the system in a distributed environment, meaning one similar to the 96 node, 192 processor cluster that the dJVM was originally designed for.

5.1.2 Performance

AspectJ may add overhead to the performance of the dJVM for a number of reasons, including that of increased cost of method calls from the original classes to those within the aspects [27]. The increase in the number of method calls is due to the way that aspects are woven into bytecode. In terms of what we can expect with respect to a performance hit, each aspect itself becomes a compiled Java class with each piece of advice becoming a public non-static method. A call to this method is then placed at the correct point of execution in the original code that we are weaving into. Though we do not anticipate any prohibitive performance penalties, it is premature for us to make this claim. Since the impetus behind distributing the RVM was to boost performance, this may not be acceptable. Therefore, performance testing will also be an important factor to consider in the future.

Ideally, we would use the SPEC benchmarks as mentioned in the previous section to further assess performance. However, due to the fact that dJVM programs have to be written with multiple threads to take advantage of different processors, a test suite such as this would need to be modified to provide valid performance results.

5.1.3 Evolvability

We have already made a preliminary assessment of evolution in Section 4.2 but the work presented in this section is a quantifiable, logical argument based upon manual code inspection. In order to really evaluate the arguments put forth in this section, it is important to apply the existing aspects to the new version of the Jikes RVM. Clearly, these aspects will have to be modified and extended in order to implement a functional distributed virtual machine. In order to do this, when the new patch is released, it will be refactored into aspects in much the same way as it was for the original dJVM system. The aim of reimplementing distribution for another version of the Jikes RVM is to answer the question: would it be less work to modify the aspects rather than reimplement distribution for the new version of the Jikes RVM? If the effort involved in effecting a large change such as distribution into an evolving system can be significantly eased, then changes such as these are less likely to be left behind as the system evolves.

5.2 Conclusions

Extensibility of a system is valued to promote reusability, thereby saving costs in industry as well as providing a stable software platform on which to build. However, when significant extensions to a system are introduced, they can compromise the system by breaking modularity. Lehman and Belady observed this during a study of OS/360 [48]. This behaviour can be caused by scattered and tangled code that does not fit into the structure of the system. In regards to our case study, the Jikes RVM, it is perhaps no surprise that distribution is a concern that is hard to modularize given that the RVM dictates a pre-existing, dominant modular decomposition into which distribution must be retro-fitted. We do not want to dismiss the possibility that this, at least in part, is due to the fact that the Jikes RVM code has been written first – without anticipating an extension such as distribution [40].

This study has shown the precise ways in which both patches and aspects have benefits. Patches are more flexible in that they are able to adapt any type of text file, including

Property	Lines of Code
Original Patch	40,177
Patch Minus Inconsequential Modifications	36,221
Patch Minus LocalOnlyStatic	22,712
Patch for Non-Java Source	3,376
Patch with Functional Aspect Code Removed (including non-Java)	12,746
Blank Lines	3,489
Aspects	8,216

Table 5.1. *Summary of numerical data.*

those created from other languages. They are also able to change comments and Javadocs within the Java files as well as removing entire files. However, patches supply no semantic leverage.

Aspects can structure widespread system changes that involve extensions that can be structured as crosscutting concerns. However, in the current prototype, there were many cases where coarse grained *around* advice was used to make minimal refactoring changes within a midpoint of a particular class method. This was done as a first pass, instead of attempting to further refactor the existing code to expose a more fine grained point of interaction. Not only did the whole method need to be copied to make those changes, producing a relatively large amount of redundant code, but the developer had the tedious job of explicitly coding in the original parameter list from the original method so that the around advice could access them.

Such improvements include better internal structure of distribution code and the consolidation of previously scattered code, in addition to a reduced size. The summary of numbers obtained during analysis are shown in 5.1. We have shown that AOP can improve the modularity of distribution elements that can be structured as crosscutting concerns, and hence benefits developers in the areas of maintainability, understandability and unpluggability.

In regards to evolution, the dJVM, which effects distribution by means of a patch, is based on version 2.2.0 of the Jikes RVM. However, only two years later, we are now on

version 2.4.2 of the Jikes RVM and the developers of the dJVM are trying to catch up. When this task is complete, what version of the Jikes RVM will be the newest? It is a constant struggle for developers to keep large system changes up to date if the system they are based upon is continuously changing. Aspects have been touted as being evolvable due to the fact that they are based on principled points of execution within the system, such as method invocations, whereas the patch is rigidly based on line numbers.

As part of an entire study of using AspectJ to implement distribution in the Jikes RVM, it is important to look at how evolvable those aspects really are in practice. This study has shown that these aspects are more evolvable than the patch but are not perfect. As the system changes, package structure changes, functionality changes or even entire files are removed. It is estimated that roughly 40% of our aspects can no longer be applied for the previously mentioned reasons. Using wildcard expressions such as `*Thread` in the LocalOnlyStatic aspect can help by continuing to capture design intent rather than statically applying changes to named files. In order to really leverage these expressions, we need the Jikes RVM to have a clean structure, basing packages on design or at the least, having an understandable naming convention on which modifications to design can be based. We believe a more aggressive refactoring for aspects would be required before these benefits can be realized.

In this study, we have also gathered evidence from case studies in infrastructure software to establish the ways in which extensible systems require extensible and configurable tools to survive. This reality has spurred us to consider a suite of adaptive tools to help modernize these textual approaches, and better integrate them with more modern mechanisms for adaptive systems. Our proposal for ISIS is designed to allow developers to reason about how patches, preprocessors, and aspects interact not only with the base code, but also each other. We believe that, in the context of system domains in particular, it is critical that ISIS be built in a way that allows adaptation within the tool itself. Having an open and adaptive tool suite would afford system infrastructure developers the same luxuries as their application developer counterparts; they could more easily shift focus from views showing

high level interaction, such as which code came from which patch, and then back to low-level, domain-specific issues such as inlining information, and low level debugging details. The bottom line is that even though infrastructure software may not be considered pretty by developers, it still needs help in the form of modern software practices and tools.

Table 5.2 overviews the contributions and conclusions of this work, and highlights the conclusion of each of the previous chapters. This thesis has shown that aspects are capable of better structuring widespread change within a system and better support software engineering principles, including evolution. From this we conclude that aspects can indeed enhance extensibility to low-level system infrastructure software. However, we also found that these aspects lack tool support in the low-level system infrastructure domain, which put them at a significant loss. From these results we conclude that an adaptable, integrated approach for the tool support dedicated specifically to system extensions is necessary before aspects can be most effectively integrated with existing approaches to extensibility, such as patches and preprocessor directives.

Experimental Phase	Property	Description
Design Mining	Design Concerns	The implementation of distribution within the dJVM touches 55% of the Java files within the system. Three categories of modifications were identified: infrastructural, VM modifications and object allocation/placement.
Aspect Mining	Design Aspects	Aspects that model the design as above were created. One in particular, modifies 44% of Java files in Jikes.
	Implementation Aspects	Aspects that modeled implementation concerns also needed to effect distribution. These aspects added configuration flags, extra command line arguments and debugging support.
Analysis	Maintainability / Understandability	(+) Crosscutting concern is localized. (+) Semantic leverage.
	Unpluggability	(+) Original RVM code is never modified.
	Evolution / Flexibility	(+) Aspects were roughly 40% more evolvable than modifications made with the patch. (+) Can reason about and apply multiple aspects. (+) Based on design intent. (–) Patches can modify C files, template files, makefiles...
Validation	Interoperable System Infrastructure Support (ISIS)	Cognitive tool support for system adaptation is critical for detecting interference between adaptive elements of a given system.

Table 5.2. *Summary of conclusions.*

Bibliography

[1] Aspect-oriented software development. [Online]. Available: www.aosd.net

[2] AspectC: AOP for C. [Online]. Available: http://www.cs.ubc.ca/labs/spl/projects/aspectc.html

[3] AspectJ 5 quick reference. [Online]. Available: www.eclipse.org/aspectj/doc/next/quick5.pdf

[4] AspectJ compiler 1.2. [Online]. Available: http://eclipse.org/aspectj/

[5] AspectJ development tools (AJDT). [Online]. Available: http://www.eclipse.org/ajdt/

[6] AspectWerkz. [Online]. Available: http://aspectwerkz.codehaus.org/

[7] BCEL. [Online]. Available: http://jakarta.apache.org/bcel/

[8] BLOAT: The bytecode-level optimizer and analysis tool. [Online]. Available: http://www.cs.purdue.edu/s3/projects/bloat/

[9] Compose*. [Online]. Available: http://janus.cs.utwente.nl/twiki/bin/view/Composer/WebHome

[10] Eclipse foundation. [Online]. Available: http://www.eclipse.org/

[11] GDB: The GNU project debugger. [Online]. Available: http://www.gnu.org/software/gdb/

[12] IBM Jikes compiler for the Java language. [Online]. Available: http://sourceforge.net/projects/jikes/

[13] Jad - The fast Java decompiler. [Online]. Available: http://www.kpdus.com/jad.html

[14] JBoss aspect oriented programming (AOP). [Online]. Available: http://labs.jboss.com/portal/jbossaop/

[15] Jikes RVM user's guide. [Online]. Available: http://jikesrvm.sourceforge.net/userguide/HTML/userguide.html

[16] Linux-Tiny. [Online]. Available: http://www.selenic.com/tiny-about

[17] Linux-VServer. [Online]. Available: http://linux-vserver.org/

[18] Patch utility. [Online]. Available: http://www.gnu.org/software/patch/

[19] PlanetLab. [Online]. Available: http://www.planet-lab.org

[20] SPEC JVM98 benchmarks. [Online]. Available: http://www.spec.org/jvm98/

[21] Spring framework. [Online]. Available: http://www.springframework.org/

[22] C. Allan, P. Avgustinov, A. S. Christensen, B. Dufour, C. Goard, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, J. Tibble, and C. Verbrugge, "abc: The AspectBench compiler for AspectJ," in *Companion to the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. New York, NY: ACM Press, 2005, pp. 88–89.

[23] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar, "The Jikes research virtual machine project: building an open-source research community," *IBM Systems Journal*, vol. 44, no. 2, pp. 399–417, 2005.

[24] B. Alpern, C. R. Attanasio, J. J. Barton, A. Cocchi, S. F. Hummel, D. Lieber, T. Ngo, M. F. Mergen, J. C. Shepherd, and S. Smith, "Implementing Jalapeno in Java," in *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 1999, pp. 314–324.

[25] Y. Aridor, M. Factor, and A. Teperman, "cJVM: A single system image of a JVM on a cluster," in *International Conference on Parallel Processing*, 1999, pp. 4–11.

[26] M. Arnold, D. Grove, S. Fink, M. Hind, and P. F. Sweeney, "Adaptive optimization in the Jalapeno JVM." in *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Minneapolis, MN, 2000, pp. 47–65.

[27] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble, "Optimising AspectJ," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. New York, NY: ACM Press, 2005, pp. 117–128.

[28] J. Baldwin and Y. Coady, "Are patches cutting it? Structuring distribution within a JVM using aspects," in *Proceedings of the 2005 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*. IBM Press, 2005, pp. 29–39.

[29] J. Baldwin and Y. Coady, "Adaptive systems require adaptive support - When tools attack!" in *Proceedings of the Hawaii International Conference on System Sciences (HICSS)*, 2007, p. 10.

[30] J. Baldwin, J. Zigman, and Y. Coady, "Version 2.*.* and counting! The toll of evo-

lution on aspect-oriented distribution," in *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD) workshop on Linking Aspect Technology and Evolution (LATE)*, 2006, p. 5.

[31] B. N. Bershad, C. Chambers, S. J. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. G. Sirer, "SPIN - an extensible microkernel for application-specific operating system services," in *ACM SIGOPS European Workshop*, 1994, pp. 68–71.

[32] S. M. Blackburn, P. Cheng, and K. S. McKinley, "Oil and water? High performance garbage collection in java with MMTk," in *Proceedings of the 26th International Conference on Software Engineering (ICSE)*. Washington, DC: IEEE Computer Society, 2004, pp. 137–146.

[33] A. Clement and M. Kersten, "AJDT: getting started with aspect-oriented programming in Eclipse," in *EclipseCon 2004 Presentation*, 2004.

[34] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn, "Using AspectC to improve the modularity of path-specific customization in operating system code," in *ESEC/FSE-9: Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY: ACM Press, 2001, pp. 88–98.

[35] E. W. Dijkstra, *A Discipline of Programming*. Englewood Cliffs, NJ: Prentice-Hall, 1976.

[36] J. Fabry, "A framework for replication using aspect-oriented programming," in *Licentiaatsthesis, Vrije Universiteit Brussel, Faculteit Wetenschappen - Departement Informatica.*, 1998.

[37] J. Fabry, "Distribution as a set of cooperating aspects," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP) workshop on Distributed Objects Programming Paradigms*, 2000, p. 5.

[38] M. E. Fiuczynski, "Better tools for kernel evolution, please!" *;LOGIN:*, vol. 30, no. 5, pp. 8–10, 2005.

[39] M. E. Fiuczynski, R. Grimm, Y. Coady, and D. Walker, "patch (1) considered harmful," in *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*. Santa Fe, NM: IEEE Computer Society, 2005, p. 6.

[40] A. Gal, O. Spinczyk, and W. S. Preikschat, "On aspect-orientation in distributed real-time dependable systems," in *Proceedings of the The Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS)*, 2002, p. 11.

[41] C. Gibbs and Y. Coady, "Refactoring for aspects: In search of the tipping point," 2006, submitted.

[42] C. Gibbs, R. Liu, and Y. Coady, "Scalable system infrastructure: Can aspects keep pace?" in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2005, pp. 241–261.

[43] W. G. Griswold, Y. Kato, and J. J. Yuan, "AspectBrowser: Tool support for managing dispersed aspects, Tech. Rep. CS1999-0640, March 2000.

[44] S. Hanenberg, C. Oberschulte, and R. Unland, "Refactoring of aspect-oriented software," in *4th Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World (Net.ObjectDays)*, Erfurt, Germany, 2003, pp. 19–35.

[45] J. Hannemann and G. Kiczales, "Overcoming the prevalent decomposition of legacy code," in *Proceedings of the Workshop on Advanced Separation of Concerns at the International Conference on Software Engineering (ICSE)*, Toronto, ON, 2001, p. 5.

[46] E. Hilsdale and J. Hugunin, "Advice weaving in AspectJ," in *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*. New York, NY: ACM Press, 2004, pp. 26–35.

[47] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. New York: Springer-Verlag, 1997, vol. 1241, pp. 220–242.

[48] M. M. Lehman and L. A. Belady, Eds., *Program evolution: processes of software change*. San Diego, CA: Academic Press Professional Inc., 1985.

[49] C. V. Lopes and G. Kiczales, "D: A language framework for distributed programming," Palo Alto, CA, Tech. Rep. SPL97-010, P9710047, February 1997.

[50] M. J. M. Ma, C.-L. Wang, and F. C. M. Lau, "JESSICA: Java-enabled single-system-image computing architecture," *Journal of Parallel and Distributed Computing*, vol. 60, no. 10, pp. 1194–1222, 2000.

[51] G. C. Murphy, "Lightweight structural summarization as an aid to software evolution," Ph.D. dissertation, University of Washington, 1996.

[52] D. L. Parnas and P. C. Clements, "A rational design process: How and why to fake it." *IEEE Transactions on Software Engineering*, vol. 12, no. 8, p. 874, 1986.

[53] N. Singh, G. Johnson, and Y. Coady, "APProVer: Coping with conditional compilation and configuration," 2006, submitted.

[54] C. Small and M. Seltzer, "VINO: An integrated platform for operating systems and database research," Cambridge, MA, Tech. Rep. TR-30-94, 1994.

[55] J. E. Smith and R. Nair, *Virtual Machines: Architectures, Implementations and Appli-*

cations. San Francisco, CA: Morgan Kaufmann Publishers, 2004, ch. An Overview of Virtual Machine Architectures, p. 21.

[56] A. Taivalsaari, "Implementing a Java virtual machine in the Java programming language," Sun Microsystems, Tech. Rep. SMLI TR-98-64, 1998.

[57] P. L. Tarr, H. Ossher, W. H. Harrison, and S. M. S. Jr., "N degrees of separation: Multi-dimensional separation of concerns," in *International Conference on Software Engineering (ICSE)*, 1999, pp. 107–119.

[58] J. N. Zigman and R. Sankaranarayana, "Designing a distributed JVM on a cluster," in *Proceedings of the 17th European Simulation Multiconference*, Nottingham, United Kingdom, 2003, p. 8.

Appendix A

Modifications to the Build Process of the dJVM

We chose our aspects to have the file name extension of `.java` whereas normally aspects have the extension of `.aj`, although both extensions are supported in AspectJ. The reason for this is that as part of the build process in Jikes, source files are copied from the source folder structure to the compile-time structure. When the aspects folder was introduced to the system, its Java files were automatically copied along with the others. This meant that we did not have to add support for the aspects in this part of the build process.

Since we used source code weaving in building the aspect-oriented version of the dJVM, we needed to weave the files before compilation took place. Therefore changes were made to invoke the AspectJ compiler as part of the `jbuild.tool` script. This script generates class files for VM tools that are not part of the build. This was the first time source files were actually compiled in the system so it was important to ensure they were woven previous to the tool compilation. In order for ajc to find the correct system files, we also needed to include `rvmrt.jar` on the boot classpath. The `aspectargs` file that is shown in the invocation of ajc is a list of the source files to compile. This includes the aspects themselves but also all of the files that were referenced by the aspects **and** the files that were referenced by those files and so on. For this reason, the list is significantly long and lists 593 files. These changes are shown in Figure A.1.

Next, changes to the system compilation script `jbuild.compile` were required. The

```
# JEB ajc alteration

   print -n "(compiling sources with ajc)"

   ajc_invocation="$ASPECTJ1_HOME/bin/ajc -classpath
       $ASPECTJ1_HOME/lib/aspectjrt.jar:.:rvmrt.jar
    -bootclasspath rvmrt.jar -preprocess -argfile $RVM_BUILD/aspctargs"

   if ${ajc_invocation}; then
     print -n "(sources compiled with ajc) "
   fi

   cp ./ajworkingdir/com/ibm/JikesRVM/*.java ./com/ibm/JikesRVM/
   cp ./ajworkingdir/com/ibm/JikesRVM/memoryManagers/vmInterface/*.java
       ./com/ibm/JikesRVM/memoryManagers/vmInterface
   cp ./ajworkingdir/com/ibm/JikesRVM/memoryManagers/JMTk/*.java
       ./com/ibm/JikesRVM/memoryManagers/JMTk
   cp ./ajworkingdir/com/ibm/JikesRVM/librarySupport/*.java
       ./com/ibm/JikesRVM/librarySupport

# end of JEB ajc alteration
```

Figure A.1. *Modifications to the Jikes tool script.*

biggest change here was to ensure that the woven `Dummy` class was in the working directory so that the full compilation of distribution code occurred. The second change was to include the AspectJ runtime jar, `aspectjrt.jar`, file on the classpath. If this file was not on the classpath, a `NoAspectBoundException` was thrown during either the image linking process or during the execution of the dJVM. This is shown in Figure A.2.

Finally, modifications to the `jbuild.linkimage` build script were made. This file included a list of the RVM primordial files. These primordial files will be the ones that are included in the boot image. The reason that not all aspects are listed here is because only the ones with runtime dependencies had to be included. That means only those aspects with advice that introduced method calls to themselves were included.

In order to simplify the process of installation, a script that copies the above build scripts to the compilation directory was created. This script is shown in Figure A.4. Instructions on building and installing the aspect-oriented version of the dJVM are shown in Appendix B.

Another issue that was solved by using source code weaving was that of reconfigu-

```
# JEB ajc alteration

   cp $RVM_BUILD/RVM.classes/ajworkingdir/Dummy.java $RVM_BUILD/RVM.classes/Dummy.java

# end of JEB ajc alteration

   cd $JAL_BUILD/RVM.classes
   rm -f Dummy.class


# JEB ajc alteration

   print -n "(compiling classes with jikes) "
   $HOST_JIKES +F +Z +KINSTRUCTION=byte +KEXTENT=int +KADDRESS=int +E -g +U
    -classpath .:rvmrt.jar:$ASPECTJ1_HOME/lib/aspectjrt.jar Dummy.java
   print -n "(classes compiled) "

# end of JEB ajc alteration
```

Figure A.2. *Modifications to the Jikes compile script.*

```
   print '[Lcom/ibm/JikesRVM/BaselineCompiler;'            >> $JAL_BUILD/RVM.primordials
   print '[Lcom/ibm/JikesRVM/Utility;'                 >> $JAL_BUILD/RVM.primordials
   print '[Lcom/ibm/JikesRVM/Identity;'                >> $JAL_BUILD/RVM.primordials
   print '[Lcom/ibm/JikesRVM/DataReplication;'             >> $JAL_BUILD/RVM.primordials
   print '[Lcom/ibm/JikesRVM/Concurrency;'                 >> $JAL_BUILD/RVM.primordials
   print '[Lcom/ibm/JikesRVM/ClassLoading;'                >> $JAL_BUILD/RVM.primordials
   print '[Lcom/ibm/JikesRVM/MemoryManagement;'            >> $JAL_BUILD/RVM.primordials
```

Figure A.3. *Modifications to the Jikes link image script.*

```
#!/bin/ksh

mkdir -p $RVM_ROOT/rvm/src/vm/aspects/
cp --reply=yes ./aspects/*.java $RVM_ROOT/rvm/src/vm/aspects/
cp --reply=yes jbuild.tool $RVM_BUILD
cp --reply=yes jbuild.linkImage $RVM_BUILD
cp --reply=yes jbuild.compile $RVM_BUILD
cp --reply=yes aspctargs $RVM_BUILD
```

Figure A.4. *The file copying script.*

ration in the Jikes' build process. Every time a reconfiguration was done with different parameters, the source files within the system changed. When weaving with the bytecode weaver, only those files that were compiled were woven into. If a reconfiguration changed the compiled files used in the build, the aspect would fail in compilation due to the file not being part of the system. Only one configuration is supported for the dJVM. But it is important to mention that if aspects were supported in more than one configuration, source code weaving would ease the amount of effort involved in modifying the build process.

A.0.1 Building with Bytecode Weaving

Originally, weaving took place via bytecode weaving before problems were found with this approach. We still discuss the modifications to the build process in order to achieve bytecode weaving since this is how the most recent version of AspectJ effects changes and it is how most systems will accomplish them.

The ajc bytecode weaver allows the developer to weave into a jar file and output a jar file. As the final step in the compilation, the Jikes build process creates two jar files, one for source files (*jksvmsrc.jar*) and the other for class files (*jksvm.jar*). Logically it makes sense to weave into this last jar file. However, the build process creates an image of the RVM as a following step, outside of the compilation process and this image is created from class files in the expanded folder and not within the *jksvm.jar* file.

For this reason, it was necessary to create our own jar file from the expanded code, weave into it with the ajc compiler, and then re-expand the woven class files. As input, the ajc needed to know about all of the source files it would be weaving into. For this, we created a JAR file (*infiles.jar*) with all of the source files from the previous build as its contents.

The aspects are then woven into this jar file, creating another jar file with the aspect code contained within; this file was temporarily called *outfiles.jar*. The code, including the newly compiled code, from this jar file was then extracted to overwrite the previously compiled version of the system. This file then overwrote the original executable dJVM, the

jar file entitled *jksvm.jar*. Since the corresponding code that is dependent on the aspects also needed to be recompiled, these files were listed in an `aspctargs` file, whose pathname is supplied at the end of the ajc command. The changes to effect bytecode weaving in the `jbuild.compile` script are shown in Figure A.5.

Weaving with this approach meant that not changes need to be made to the `jbuild.tool` script. It also meant that the `aspectargs` file only need to contain the aspects for compilation. Changes to the primordial list and to `RVM.sources` would still be the same.

```
# JEB ajc alteration

   cp $RVM_BUILD/RVM.classes/ajworkingdir/Dummy.java $RVM_BUILD/RVM.classes/Dummy.java

# end of JEB ajc alteration

   cd $JAL_BUILD/RVM.classes
   rm -f Dummy.class

   print -n "(compiling classes with jikes) "
   $HOST_JIKES +F +Z +KINSTRUCTION=byte +KEXTENT=int +KADDRESS=int +E -g +U
    -classpath .:rvmrt.jar:$ASPECTJ1_HOME/lib/aspectjrt.jar Dummy.java
   print -n "(classes compiled) "

# JEB ajc alteration

   print -n "(compiling classes with ajc)"
   rm -f $RVM_BUILD/RVM.classes/com/ibm/JikesRVM/TestHierarchy.class
   rm -f infiles.jar
   rm -f outfiles.jar

   jar -cM0f infiles.jar ./com/ibm/JikesRVM/*.class
       ./com/ibm/JikesRVM/BytecodeToolset/*.class
       ./com/ibm/JikesRVM/ClassTransformer/*.class
       ./com/ibm/JikesRVM/librarySupport/*.class
       ./com/ibm/JikesRVM/memoryManagers/JMTk/*.class
       ./com/ibm/JikesRVM/memoryManagers/vmInterface/*.class

   ajc_invocation="$ASPECTJ15_HOME/bin/ajc -showWeaveInfo -verbose -classpath
       $ASPECTJ15_HOME/lib/aspectjrt.jar:.:rvmrt.jar -bootclasspath rvmrt.jar -inpath
       infiles.jar -outjar outfiles.jar -argfile $RVM_BUILD/aspctargs2"

  if ${ajc_invocation}; then
     print -n "(classes compiled with ajc) "
     jar -xf outfiles.jar
  fi

# end of JEB ajc alteration
```

Figure A.5. *Jikes compile script involving bytecode weaving.*

Appendix B

Installing the AspectJ Distributed Java Virtual Machine (AJVM)

B.1 Host and Utility Requirements

Currently, the AspectJ Distributed Java Virtual Machine requires the host computer to have an i386-compliant instruction set running Linux 2.4. The restriction of an i386 host arises from the Jikes Research Virtual Machine, and in particular because of its host-dependent compilers. The system has been run with Linux Kernel 2.4.9 and 2.4.20.

An independent (host) version of Java must be installed in a directory `$java`.
SUN JDK v1.3.1_13, available at:
http://java.sun.com/products/archive/j2se/1.3.1_13/index.html and installed in `/usr/java/jdk1.3.1_13`. Although the IBM SDK v1.3.1 is available we have experienced problems with it.

You will also need the following four other tools which are necessary to build the system:

AspectJ - you will need AspectJ version 1.0.6 to compile this version of the DJVM (which does source code weaving). Available at http://www.eclipse.org/aspectj/downloads.php

Kshell - public domain ksh version 5.2 or later, normally installed in /bin/ksh, available with Redhat (and other) Linux distributions. The majority of scripts used in the Jikes RVM use ksh.

Jikes Java compiler - jikes version 1.18 or later, available at http://sourceforge.net/project/showfiles.php?group_id=128803, normally installed in `/usr/bin/jikes`, is a Java source compiler written in C and independent of any Java runtime libraries, used to compile the JikesRVM source.
Note: Version 1.18 is the only version which we can guarantee will work. We have experienced problems with versions 1.13 through 1.17.

Unzip - public domain `unzip` by Info-Zip, version 5.50 or later (although an earlier version would probably work).

B.2 Building the AspectJ Distributed Java Virtual Machine

Please Note: You will need to increase the memory required (the default is 64MB) by the AspectJ 1.0 compiler. To do so, edit the ajc script to say -Xmx128M as seen below.

```
"$JAVA_HOME/bin/java" -classpath
"$ASPECTJ_HOME/lib/aspectjtools.jar:
$JAVA_HOME/lib/tools.jar:$CLASSPATH" -Xmx128M
org.aspectj.tools.ajc.Main "$@"
```

The following environment variables (these are examples, use your own user account)

should be set (you will need to edit i686-pc-linux-gnu for your configuration):

```
ASPECTJ1_HOME           /home/jbaldwin/aspectj1.0}

(wherever you plan to download and extract this project to)
RVM_DOWNLOAD            /home/jbaldwin/Desktop/aspectdjvmdownload
RVM_ROOT                /home/jbaldwin/aspectrvmRoot
RVM_BUILD               /home/jbaldwin/aspectrvmBuild
RVM_TARGET_CONFIG       $RVM_ROOT/rvm/config/i686-pc-linux-gnu
RVM_HOST_CONFIG         $RVM_ROOT/rvm/config/i686-pc-linux-gnu
RVM_HOST_JAVA_HOME      $java
PATH                    $RVM_ROOT/rvm/bin:$RVM_HOST_JAVA_HOME/bin:$PATH
CLASSPATH               .:$RVM_BUILD/RVM.classes:$RVM_BUILD/RVM.classes/rvmrt.jar:
                          $RVM_BUILD/RVM.classes/jksvm.jar
RSH                     either ssh or rsh
```

These environment variables are also necessary for configuring and building the Jikes distributed Java Virtual Machine, as well as compiling and executing applications for the AJVM.

Obtain and unpack the file `AJVM.tar.gz`. This tar file includes `jikesrvm-2.2.0.tar.gz` which has been modified from the original so that it would compile with Sun's Java compiler. You need this version and not the version downloaded from the official site.

For your convenience, the extra files that this build depends on have been made available for download from the project page. These need to be in the same directory as the unpacked project tar file.

`classpath-0.05.tar.gz`

`jlibraries.tar.gz`

`jlibsource.tar.gz`

B.3 Building the AspectJ DJVM

The distributed JVM source tree is built using the script `DJVMinstall`. This script creates the standard non-distributed Jikes RVM source tree in the `$RVM_ROOT` directory.

Next, run the script `DJVMbuildClassPathJar`. This should be run, from the same directory, immediately after DJVMinstall has been run. This script expects the file `classpath-0.05.tar.gz` to be in the current working directory.

Warning: During the course of the build of the classpath libraries, a few unsightly warning messages, may be observed. However, the library build should, nevertheless, succeed.

Once the above steps have been completed, the dJVM may be built largely in the same way as the standard non-distributed JVM is built.

Change directory to `$RVM_ROOT/rvm/bin`
Run the `jconfigure` script, giving as a parameter, the configuration file:

```
ClusterBaseBaseMarkSweep}}}
```

Change directory to `$RVM_DOWNLOAD`
Run the `copyFilesForBuild` script
Add the aspects folder to the `RVM.sources` script, for example:

```
/home/jbaldwin/aspectrvmRoot/rvm/src/vm/aspects
```

Change directory to `$RVM_BUILD`
Execute the script `jbuild`

B.4 Running the DJVM

To run the dJVM in a non distributed environment, use the command `$RVM_ROOT/rvm/bin/rvm`.

To run in a distributed environment, continue reading:

Disable the firewall (on Linux and on the router)
To check your IP addresses:

```
%su
%tcsh
%ifconfig
```

Next, create a `DJVMHosts` file with the IP address of each machine on a separate line. The first line should be the IP for the local host. For example: `192.168.1.2`
`192.168.1.1`

Run "`rvmCluster className`" from the same directory as is the `DJVMHosts` file or

"`rvmCluster`
`-DJVMHostsFile=/home/jbaldwin/aspectrvmRoot/rvm/bin/DJVMHosts`
`className`"

You must have an account and home directory with the same username and home directory path as the master node. You must also have `$RVM_ROOT` and `$RVM_BUILD` with the same folder names and installations on the other machine AND under the same home user account.

Appendix C

Debugging the dJVM

When a segmentation fault occurs very early in the boot process then it is possible that the fault occurs during the execution of code resident in the RVM image. If this is so debugging is significantly easier. When a segmentation fault occurs (that is not a null pointer check) then the state of the registers is dumped out, including the program counter for the instruction that caused the segmentation fault. If the fault occurs in a method compiled into the image then we can use the program counter given by the segmentation fault to find the method in the RVM.map file in the build directory (the program counter may not match exactly but will be after the start of the method). Once this location is found, we can use gdb to load the RVM bootstrap C program and set a break point in the C code after the C code to load the image (but before the call into the image code). We then run the RVM and when the break point is reached then set a break point at the start of the method being called. Next, step through as necessary/disassemble the method code and set a break point somewhere appropriate before the instruction that will cause the segmentation fault. Finally, run to that break point. Then we can establish what is going on and what values are incorrect and relate that back to the Java code - not necessarily particularly straight forward.

If the segmentation fault does not occur in the initial image but in a method compiled after execution starts then things are more difficult. A suggestion is to make a new method in something like `VM`, that method will be used as a marker since it will be included in the RVM image (note its location from the `RVM.map` file) and have that method do a `VM.sysWrite()` with some message. Since the call is probably made post class loading

being enabled (local class loading) then from the start onward, we can insert calls to the new method we put into the VM. Then we try to find the latest point in the code that we can insert the call and have it run. Once we have done that, we include a single call to the method we made in the Java code. Then we run the VM using gdb as above, only this time we set a break point on the newly introduced method, run to that point, and set a break point on the return instruction from that method (the method should be very short) and run to that break point. Stepping through the code will return to the calling method and disassembling that method will tell us what is going on.

An alternative is to again put a method in the VM that is built into the image and make a transform to insert a call to that method in the method(s) that we are interested in, within the running VM. This can be done to report information such as passing references to be dumped out, or to make a place holder for using gdb.

Appendix D

Perl Scripts

This section contains the perl scripts used to produce the patch statistics discussed in this work. Figure D.1 on the next page shows how to find the total number of files modified by a patch file. Figure D.2 on page 107 shows how to find the number of files whose only change was implementing the `DVM_LocalOnlyStatic` interface. Finally, Figure D.3 on page 108 shows how to find the total number of white space sections left in the final version of the patch file.

```
#!/usr/local/bin/perl

$files = 0;
$sections = 0;
$linesR = 0;
$linesA = 0;

open(handle,"jikesrvm-cluster-extns-1.0.2.patch");

@lines = <handle>;
$ending_value = scalar(@lines);

for($counter=0 ; $counter < $ending_value ; $counter++) {
    $sentence = $lines[$counter];

    if ($sentence =~ /^diff /)
    {
        $files = $files + 1;
    }
    if ($sentence =~ /^@@ /)
    {
        $sections = $sections + 1;
    }
    if ($sentence =~ /^-/ && $sentence !~ /^--- /)
    {
        $linesR = $linesR + 1;
    }
    if ($sentence =~ /^\+/ && $sentence !~ /^\+\+\+ /)
    {
        $linesA = $linesA + 1;
    }
}

close(handle);

print("Files Modified: $files \n");
print("Sections Modified: $sections \n");
print("Total Lines: $ending_value \n");
print("Lines Removed: $linesR \n");
print("Lines Added: $linesA \n");
```

Figure D.1. *Overall patch statistics.*

```
#!/usr/local/bin/perl

$file = "";
$count = 0;

open(handle,"jikesrvm-cluster-extns-1.0.2.patch");

@lines = <handle>;
$ending_value = scalar(@lines);

for($counter=0 ; $counter < $ending_value ; $counter++)
{
    $sentence = $lines[$counter];

    if ($sentence =~ /^diff /)
    {
        @array = split(/ /, $sentence);
        $file = $array[3];
        $count = 0;
    }

    if($sentence =~ /.+DVM_LocalOnlyStatic/ && $sentence !~
        /.+extends.+DVM_LocalOnlyStatic/
        && $sentence !~ /.+import .+DVM_LocalOnlyStatic/ && $count == 0)
    {
            print("$file");
            $count = 1;
    }
}

close(handle);
```

Figure D.2. *Classes implementing DVM_LocalOnlyStatic.*

```
#!/usr/local/bin/perl

$count = 0;
$newsection = 0;
$linesinsection = 0;

open(handle,"jikesrvm-cluster-extns-1.0.2.patch");
#open(handle,"test");

@lines = <handle>;
$ending_value = scalar(@lines);

for($counter=0 ; $counter < $ending_value ; $counter++)
{
    $sentence = $lines[$counter];

    if ($sentence =~ /^@@/)
    {
        if($linesinsection != 0)
        {
            $count = $count + $linesinsection + 7;
        }

        $linesinsection = 0;
        $newsection = 1;
    }

    if ($newsection == 1 && $sentence =~ /^-/ && $sentence !~ /^--- /)
    {
        $newsection = 0;
    }

    if ($newsection == 1 && $sentence =~ /^\+/ && $sentence !~ /^\+ *\n/)
    {
        $newsection = 0;
    }

    if ($newsection == 1 && $sentence =~ /^\+ *\n/)
    {
        $linesinsection = $linesinsection + 1;
    }

    if ($newsection == 0 && $sentence =~ /^\+ *\n/)
    {
        $count = $count + 1;
    }
}

close(handle);

print("Lines: $count \n");
```

Figure D.3. *White space left in patch.*