IOWA STATE UNIVERSITY
Digital Repository

Computer Science Technical Reports | Computer Science

2007

Nu: Towards a Flexible and Dynamic Aspect-Oriented Intermediate Language Model

Robert Dyer
Iowa State University, rdyer@iastate.edu

Rakesh Bangalore Shivarudra Setty
Iowa State University, rakesh.setty@yahoo.co.in

Hridesh Rajan
Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/cs_techreports

Part of the Programming Languages and Compilers Commons, and the Software Engineering Commons

Recommended Citation
Dyer, Robert; Setty, Rakesh Bangalore Shivarudra; and Rajan, Hridesh, "Nu: Towards a Flexible and Dynamic Aspect-Oriented Intermediate Language Model" (2007). *Computer Science Technical Reports*. 320.
http://lib.dr.iastate.edu/cs_techreports/320

This Article is brought to you for free and open access by the Computer Science at Iowa State University Digital Repository. It has been accepted for inclusion in Computer Science Technical Reports by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.

Nu: Towards a Flexible and Dynamic Aspect-Oriented Intermediate Language Model

Abstract
The contribution of this work is the design, implementation and evaluation of a new aspect-oriented intermediate language model that we call Nu. The primary motivation behind the design of the Nu model is to maintain the aspect-oriented design modularity in the intermediate code for the responsiveness of incremental compilers and source-level debuggers. Nu extends the object-oriented intermediate language model with two primitives: bind and remove. We demonstrate that these primitives are capable of expressing statically deployed constructs such as AspectJ's aspect, dynamic deployment construct such as CaeserJ's deploy as well as dynamic control flow constructs such as AspectJ's cflow by presenting compilation techniques from high-level languages to Nu for these constructs. Moreover, these compilation techniques also serve to show that aspect-oriented design modularity is indeed preserved in the Nu intermediate code. We also present the design and implementation of a prototype extension of the Sun Hotspot virtual machine that supports the Nu model, which serves to show that it is feasible to implement Nu in a production level virtual machine. A key concern for dynamic language models is the performance overhead of their implementation. Our performance analysis results show that method dispatch time is not degraded in our prototype implementation. Also, advice dispatch time remains fairly close to the manually inlined version.

Keywords
Nu, invocation, incremental, weaving, aspect-oriented intermediate languages, aspect-oriented virtual machines

Disciplines
Programming Languages and Compilers | Software Engineering

This article is available at Iowa State University Digital Repository: http://lib.dr.iastate.edu/cs_techreports/320

Nu: Towards a Flexible and Dynamic Aspect-Oriented Intermediate Language Model

Technical Report # 07-06, Dept. of Computer Science, Iowa State University, June 03, 2007

Robert Dyer Rakesh B. Setty Hridesh Rajan
Dept. of Computer Science, Iowa State University
{rdyer, rsetty, hridesh}@cs.iastate.edu

ABSTRACT

The contribution of this work is the design, implementation and evaluation of a new aspect-oriented intermediate language model that we call *Nu*. The primary motivation behind the design of the *Nu* model is to maintain the aspect-oriented design modularity in the intermediate code for the responsiveness of incremental compilers and source-level debuggers. *Nu* extends the object-oriented intermediate language model with two primitives: *bind* and *remove*. We demonstrate that these primitives are capable of expressing statically deployed constructs such as AspectJ's aspect, dynamic deployment construct such as CaeserJ's deploy as well as dynamic control flow constructs such as AspectJ's cflow by presenting compilation techniques from high-level languages to *Nu* for these constructs. Moreover, these compilation techniques also serve to show that aspect-oriented design modularity is indeed preserved in the *Nu* intermediate code.

We also present the design and implementation of a prototype extension of the Sun Hotspot virtual machine that supports the *Nu* model, which serves to show that it is feasible to implement *Nu* in a production level virtual machine. A key concern for dynamic language models is the performance overhead of their implementation. Our performance analysis results show that method dispatch time is not degraded in our prototype implementation. Also, advice dispatch time remains fairly close to the manually inlined version.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.3.4 [**Programming Languages**]: Processors — Code generation; Incremental compilers; Run-time environments

General Terms

Design, Human Factors, Languages

Keywords

Nu, invocation, incremental, weaving, aspect-oriented intermediate languages, aspect-oriented virtual machines

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

1. INTRODUCTION

Aspect-oriented (AO) techniques provide software engineers with new possibilities for keeping crosscutting conceptual concerns separate at the source code level [10, 18]. The compilers for some AO approaches then transform the aspect-oriented source code to object-oriented intermediate code by inserting calls to and fragments from (the now modularized) crosscutting concerns into other concerns [7, 12]. This is done to produce intermediate code that is compliant with current object-oriented virtual machines.

The design decision to remain compliant with current object-oriented virtual machines was made to encourage early adopters, who may not want to change their virtual machines. Early adoption, in turn, facilitated a large scale empirical evaluation of new language design ideas. Now that aspect-oriented software development techniques have shown promise [8, 33], it makes sense to ask whether relaxing the requirement to remain compliant with current intermediate languages can be beneficial.

In this work, we propose an extension of object-oriented intermediate language (IL) models to better support aspect-oriented languages. Our extension, which we call *Nu*, consists of two new atomic primitives: *bind* and *remove*. The effect of these primitives is to manipulate what we call *advising relationships*. For the purpose of this paper, we define an advising relationship as a many-to-one relation between points in the execution of a program and a delegate. If a point in the execution of a program and a delegate are in an advising relationship, the execution of the point is followed by the execution of the delegate. The effect of the *bind* primitive is to dynamically create an advising relationship. The effect of the *remove* primitive is to destroy the specified advising relationship.

Our intermediate language extension is both simple and flexible enough to be able to accommodate the requirements of a broad set of source language constructs. Our results show that the combination of the two primitives in our intermediate language design allows us to model statically deployed constructs such as AspectJ's *aspect* [17], dynamic deployment constructs such as CaesarJ's *deploy* [26] as well as control flow constructs such as AspectJ's *cflow*.

Another nice property of the *Nu* model is that it allows compilers to maintain the conceptual separation that was present in the source code, in the object code as well. The intermediate code now mirrors the design, which among other things is important for incremental compilation, source-level debugging, and dynamic adaptation of aspect-oriented programs.

We have extended the Sun Hotspot Java Virtual Machine (Hotspot JVM) [29] to support our intermediate language design, which serves to show that it is feasible to support the *Nu* model in a production level virtual machine. We have also modified the AspectJ [17] compiler to generate *Nu* code showing that our extended IL can support portions of this widely used language.

In what follows, we describe our intermediate language design. Section 3 describes strategies to compile various static and dynamic AO language constructs to our intermediate language model. We then describe our implementation strategy to support the *Nu* intermediate language model in the Hotspot JVM in Section 4. Section 6 discusses related work, Section 7 discusses some benefits of our approach, Section 8 discusses future work and Section 9 concludes.

2. NU: A DYNAMIC AO IL MODEL

The key requirements for our IL model is to remain simple, yet flexible enough to be able to support both static and dynamic constructs in AO source languages and at the same time preserve aspect-oriented design modularity in the object code. This section introduces the join point model adopted by our approach. We then illustrate new primitives using an example.

2.1 Nu's Join Point Model

The central concept in AO approaches is the notion of a join point. A join point is defined as a point in the execution of a program. For example, in AspectJ [17], the "execution of the method `Hello.main()`" in Figure 1 is an example of a join point. This join point may possibly occur at a location in the source code, popularly referred to as the *shadow* of the join point. The shadow of the join point shown is marked in Figure 1.

```
// Source Code
public class Hello {
  static void main(String[] arguments) {
    System.out.println("Hello");
  }
}
// Intermediate Code
static void main(java.lang.String[]);
  /* AspectJ join point shadow for "execution
  of the method Hello.main" starts here */
    getstatic       #2; //System.out
    ldc             #3; //String Hello
    invokevirtual   #4; //Method println
  /* AspectJ join point shadow ends here */
    return
```

Figure 1: Illustration of the AspectJ Join Point Model

Instead of AspectJ's join point model, we adopted a finer-grained join point model for *Nu*, proposed by Endoh *et al.* [11]. Endoh *et al.* called the join point model of AspectJ-like languages a *region-in-time* model since a join point in these languages represents duration of an event, such as a call to a method until its termination. They proposed a join point model called *point-in-time* model in which a join point represents an instance of an event, such as the beginning of a method call or the termination of a method call [11]. They showed that this model is sufficiently expressive to represent common advising scenarios.

In the *point-in-time* model, corresponding to AspectJ's *call* join point there are three join points: *call*, *reception*, and *failure*. Here, failure is when an exception is thrown by the callee. These three join points eliminate the need for three different types of advice: *before*, *after returning*, and *after throwing* advice. The *before call*, *after returning call*, and *after throwing call* become equivalent to *call*, *reception*, and *failure* respectively. Similarly, corresponding to AspectJ's *execution* join point there are three join points: *execution*, *return*, and *throw*. Here, throw is when the executing method throws an exception. At this time, *Nu* does not support *around* advice (see Section 8 for more details). For more details about the *point-in-time* model, please see Endoh *et al.* [11].

For example, in Figure 2, two join point shadows in the method

```
static void main(java.lang.String[]);
  /* Join point shadow for the join point
  "execution of the method Hello.main" */
    getstatic       #2; //System.out
    ldc             #3; //String Hello
    invokevirtual   #4; //Method println
  /* Join point shadow for the join point
  "return of the method Hello.main" */
    return
```

Figure 2: Illustration of the Point-In-Time Join Point Model

	bind	**remove**
Stack Transition	..., `Pattern`, `Delegate` → ..., `BindHandle`	..., `BindHandle` → ...
Description	Associates the execution of all join points matched by `Pattern` to invoke `Delegate`	Eliminates the association represented by `BindHandle`
Exceptions	`NullPointerEx` - thrown if any argument is null	`IllegalArgumentEx` - thrown if the `BindHandle` passed in has already been removed

Figure 3: Specification of Primitives in *Nu*

`Hello.main()` are marked as being shadows for the join points "execution of the method `Hello.main()`" and "return of the method `Hello.main()`".

Our adoption of this model was in part driven by the clarity it gives to the semantics of dynamic aspect deployment. One issue that arises with the deployment of dynamic aspects is when the aspect being deployed advises a join point already on the stack. With a *region-in-time* model, it is not very clear whether this new aspect should advise the join point already on the stack and the problem is often left to the semantics of the virtual machine [16]. Using a *point-in-time* model, this problem is avoided since a join point is never on the stack.

2.2 New Primitives: BIND and REMOVE

Our IL model adds only two primitives to the object-oriented IL, *bind* and *remove*. The informal specifications including stack transitions and exceptions that might be thrown are shown in Figure 3. As described previously, the effect of these primitives is to manipulate what we call *advising relationships*. At this time we have explicitly decided not to support static crosscutting mechanisms, such as inter-type declarations in AspectJ [17].

An example use of the *Nu* primitives and *Nu*'s standard library is given in Figure 4. The figure shows the intermediate code for **`class`** `AuthLogger`. The objective is to record the time of execution of any method named *login* in the system. Moreover, one should also be able to enable and disable the authentication logger during execution. To implement this logger, we need to specify the intention to select all methods with the name `login`. In the *Nu* model, one would create a pattern to represent this intention.

2.2.1 Patterns in Nu

A pattern is an object of type `Pattern`. It is created by instantiating a set of classes provided by the *Nu* standard library. It is first-class, in that it can be stored, passed as a parameter, and returned from methods. Note, however, they are immutable. Since patterns are first-class objects available in the high-level language, they are re-usable. This allows for possible optimizations by compilers, such as locating commonly used sub-patterns that can be created once and re-used. Since patterns are immutable, the virtual machine that implements the *Nu* model does not have to worry

```
public class AuthLogger {
  protected static BindHandle id;
  protected static Pattern loginPattern;
  protected static Delegate logDelegate;
  static {}; // Static initializer
    /* create new Method and Execution objects */
        ..
    ldc              #4;  //String *.login
    invokespecial    #5;  //Method.<init>
    invokespecial    #6;  //Execution.<init>
    putstatic        #9;  //Pattern loginPattern
    ldc              #7;  //String AuthLogger
    ldc              #8;  //String log
    invokespecial    #10; //Delegate.<init>
    putstatic        #11; //Delegate logDelegate
    return
  public static void enable();
    getstatic        #9;  //Pattern loginPattern
    getstatic        #11; //Delegate logDelegate
    bind                  //Bind
    putstatic        #12; //BindHandle id
    return
  public static void disable();
    getstatic        #12; //BindHandle id
    remove                //Remove
    return
  public static void log();
    // record the time of login
    return
}
```

Figure 4: Bind and Remove in an Example Program

Basic Patterns	Selected JPs
1. Method	Method-related JPs
2. Constructor	Constructor-related JPs
3. Initialization	Static initializer-related JPs
4. Field	Field-related JPs

Patterns 5-10 take a pattern of type 1, 2, or 3 as argument. Patterns 11-12 take a pattern of type 4 as argument.

Filters	Selected JPs
5. Execution	Method executions
6. Return	Method returns
7. Throw	Method throws
8. Call	Method calls
9. Reception	Method receptions
10. Failure	Method failures
11. Get	Field gets
12. Set	Field sets

Figure 5: Patterns Available in *Nu*'s Standard Library

about a pattern instance changing after it has been created.

Figure 5 shows some commonly used patterns available in our implementation. The basic patterns on the left (numbered 1-4) serve to select all join points (JPs) related to methods, constructors, fields, etc. For example, the pattern object returned by `new Method("*.login")` can be used to select *execution*, *return*, *throw*, *call*, *reception*, and *failure* join points for all methods named "login". The filter patterns on the right (numbered 5-12) expect one of the basic patterns as an argument and further narrow down the set of matching join points. For example, if we want to match the "execution of any method named login" we would have to first create the `Method` pattern discussed before. We would then pass this instance as an argument to the constructor of the `Execution` class. The resulting instance is the pattern for "execution of any method named login."

In the example shown in Figure 4, the static initializer of `class AuthLogger` creates this pattern and stores it in the static field `loginPattern` so that it can be used for enabling the logger using the *bind* primitive.

2.2.2 The bind primitive

The *bind* primitive expects two values on the stack: a pattern (discussed previously) and a delegate. The delegate is a first-class, immutable object of type `Delegate`[1]. Both these types are part of *Nu*'s standard library. The pattern serves to select the subset of the join points in the program. The delegate points to a method that provides the additional code that is to execute at these join points. In Figure 4, the static initializer of `class AuthLogger` creates a delegate to the method `AuthLogger.log()` and stores it in the static field `logDelegate` so that it can be used for enabling the logger using the *bind* primitive. The `enable()` method uses the *bind* primitive to create an advising relationship between the join points matched by the pattern `loginPattern` and the delegate `logDelegate`, which enables logging authentication attempts in the system.

After the bind primitive finishes, the top of the stack contains an immutable, unique identifier representing the advising relationship. This unique identifier is an object of type `BindHandle`, which is also part of *Nu*'s standard library. This identifier may only be created by the virtual machine. If the pattern and/or delegate is null, a `NullPointerException` is thrown by the virtual machine. The bind primitive dynamically creates an advising relationship between the join points matched by the pattern and the supplied delegate. The runtime effect of creating this advising relationship is that the method pointed to by the delegate is invoked when execution reaches any join point matching the pattern.

When a join point executes, each delegate supplied with a pattern that matches that join point will be invoked. Delegates are invoked in the same order in which they were bound. Future language extensions may allow ordering constructs; however, at this time we believe they are not necessary since compilers generating *Nu* intermediate code could re-order the *bind* calls (for example when modeling the static deployment model of AspectJ and implementing the *declare precedence* construct). Delegates are invoked at most once per join point.

Upon completion of a call to *bind*, any join point that executes and matches the supplied pattern will invoke the delegate. This behavior is intentional. Consider a tracing aspect, which will output a trace at the entry and exit of a method. If a *bind* call is used to enable the tracing, we want it to take effect immediately (thereby tracing the method exit of the method containing the *bind* call).

The language is defined with a per-thread semantics. This means that calls to *bind* and *remove* only affect the advising relationships on the same thread that the primitives were called from. This semantics is selected to avoid the need to make groups of *bind*/*remove* calls atomic (note, however, that individual calls are atomic). The termination of a thread causes all associations created by that thread to be automatically removed, since reaching a join point in the context of that thread is no longer possible.

2.2.3 The remove primitive

The *remove* primitive expects the unique, immutable identifier representing the advising relationship on the stack. It destroys the advising relationship corresponding to the identifier. If the advising relationship corresponding to the supplied identifier is already removed, an `IllegalArgumentException` is thrown. For example, in Figure 4 the `disable()` method uses the *remove* primitive to destroy the advising relationship corresponding to the `BindHandle` instance stored in the static field `id`, effectively ceasing logging.

[1] A limitation of our current implementation is that delegate constructors take method names and class names as strings. In the future, we will use Kniesel's approach for type-safe delegates [19].

3. COMPILING AO CONSTRUCTS TO NU

In this section, we will describe strategies for compiling static and dynamic AO constructs to the *Nu* IL model. The rationale for this section is to demonstrate that our model is flexible enough to support static, dynamic and dynamic control flow constructs in AO languages. Moreover, it also shows, by giving a translation, that compilation of these constructs generates modular object code, which is the primary motivation behind the design of the *Nu* intermediate language model.

3.1 Compiling AspectJ Constructs

To illustrate the compilation strategies from AspectJ constructs to the *Nu* IL model, consider a simple extension of the Hello program shown in Figure 1. Now let us assume that we were to write an aspect that would extend the functionality of the method `main()` so that instead of printing "Hello" it prints "Hello" followed by "World" on successive lines. **`aspect`** `World` that implements this simple functionality is shown in Figure 6. The generated object code for this aspect follows in Figure 7.

```
public aspect World {
  pointcut main(): execution(* Hello.main(..));
  after returning(): main() {
    System.out.println("World");
  }
}
```

Figure 6: The World Aspect

```
public class World {
  public static final World ajc$perSingletonInst;
  static {}; // Static initializer
    aload_0
    //Method java/lang/Object.<init>
    invokespecial  #1;
    //Create the static instance of the aspect
    //Method World.<init>
    invokespecial  #12;
    //Store in ajc$perSingletonInst
    putstatic       #9;
    //Create the pointcut as pattern
    ldc            #6;  //String main
    //Method org/nu/lang/pattern/Method.<init>
    invokespecial  #7;
    //Method org/nu/lang/pattern/Execution.<init>
    invokespecial  #8;
    //Create the delegate
    getstatic            #9;  // instance
    ldc            #10; //String ajc$0
    //Method org/nu/lang/Delegate.<init>
    invokespecial  #11;
    bind
    ..
    return
  //Synthetic method generated for the advice
  public void ajc$0();
    getstatic      #13; //Field System.out
    ldc            #9;  //String World
    invokevirtual  #14; //Method println
    return
//Constructor World, and helper methods hasAspect,
//and aspectOf elided for presentation purposes.
```

Figure 7: Compiling an AspectJ Aspect to *Nu* IL

3.1.1 Compiling Aspects, Pointcuts and Advice

Aspects are compiled into intermediate code units in the following way: pointcuts are compiled into pattern object instances, advice code is compiled into delegate methods, and bind primitives

```
public class Hello {
  static void main(java.lang.String[]);
0:  getstatic       #21; //Field System.out
3:  ldc             #22; //String Hello
5:  invokevirtual   #28; //Method println
8:  goto            20
    //Code inserted for aspect invocation
11: astore_1
12: invokestatic    #38; //Method World.aspectOf
15: invokevirtual   #41; //Method World.ajc$0
18: aload_1
19: athrow
20: invokestatic    #38; //Method World.aspectOf
23: invokevirtual   #41; //Method World.ajc$0
26: return
}
public class World {
  public static final World ajc$perSingletonInst;
  static {}; // Static initializer
0:  invokestatic #14; //Method ajc$postClinit
3:  goto         11
6:  astore_0
7:  aload_0
8:  putstatic    #16; //Field ajc$initFailureCause
11: return
//Advice ajc$0, constructor World, and methods
//hasAspect, aspectOf and ajc$postClinit elided.
}
```

Figure 8: An AspectJ Aspect Compiled to Standard Bytecode: the Generated Code for the World Concern is in Gray

are generated in a static initializer of the aspect to associate the delegate code to the joint points matched by the patterns. In the example shown in Figure 7, the generated object code for the method `ajc$0()` contains the advice code.

The generated intermediate code for the static initializer of **`aspect`** `World` contains additional code to first create an instance of the pattern `Method`. This instance is then used to create an instance of the pattern `Execution`. After creating the pattern instance, the delegate is created.

One interesting property of the *Nu* IL model is that the intermediate code for the aspect **`class`** `World` and the base **`class`** `Hello` remain separate in their own object code modules. Also, the object code for the base **`class`** `Hello` remains free from the aspect related intermediate code.

On the other hand, in order to remain compliant to OO intermediate languages, the intermediate code generated by current compilers for AspectJ is not able to maintain this separation. We compiled our *HelloWorld* application using one AspectJ compiler, *ajc*. We disassembled the class files using *javap*, the disassembler for Java. Figure 8 shows the disassembled intermediate code. We used the Java byte code notations to represent the disassembled code.

The generated code for the advice, constructor, and the helper methods `hasAspect()`, **`aspectOf`**`()`, and `ajc$postClinit()` is similar in both the *Nu* version as well as ajc's version, so we elide it for presentation purposes. Unlike *Nu*'s version, the generated code for the static initializer does not contain any additional instructions.

The intermediate code to invoke `ajc$0()` at join points, however, is inserted into **`class`** `Hello` in the method `main()`. As a result, the concern modularized by **`aspect`** `World` ends up being scattered and tangled with the `Hello` concern in the object code. The separation of the `Hello` and `World` concerns at the source level is thus lost during the compilation phase. The *Nu* version requires generating a few extra instructions, but the intermediate code preserves the design modularity of the `World` concern.

3.1.2 Compiling Complex Aspects

The illustrative AO application compiled in the previous section served to provide an example of a basic translation. To preserve the semantics of an aspect in the AspectJ language, compilation of an aspect in a real world AO application needs to account for two additional conditions: deployment as a single unit and whole program deployment of aspects.

First, aspects are deployed as a single unit at the beginning of the program. This requirement is addressed by generating all bind instructions for an aspect inside a transaction in the static initializer or in a synthetic static method `ajc$preClinit()`. A dummy reference to all aspects is inserted in the static initializer of the main application class as the first few instructions. This causes all aspects to initialize before the application execution begins.

Second, aspects in AspectJ advise all threads in the program. In Java, when a thread is created it must be permanently bound to an object with a `run()` method. When the thread starts by calling `Thread.start()`, it will invoke the object's `run()` method. The strategy to deploy aspects for all threads in the program is to generate a set of instructions that execute between the methods `Thread.start()` and the object's `run()` method. These instructions are calls to the static method `ajc$preClinit()` on all aspects in the program. As mentioned previously, the bind instructions are generated in the `ajc$preClinit()` as a transaction. Executing this method deploys the aspects for the new thread.

3.2 Compiling Deployment Constructs

Some aspect languages such as CaesarJ [26] provide declarative constructs for dynamic deployment, such as deploy and undeploy. These constructs are naturally supported by our two primitives. Figure 9 shows a strategy for compiling such constructs. For presentation purposes, instead of the generated intermediate code the equivalent source code is shown. In the source code a notation such as `id = bind(p,d)` represents generating two push instructions for the pattern `p` and the delegate `d` followed by generating the bind primitive, followed by a store instruction to store the result in id. Furthermore, `remove(id)` represents generating a remove primitive after an instruction to push `id` on the stack.

```
class World {
  public static World ajc$perSingletonInst;
  static Pattern p =
      new Execution(new Method("*.main"));
  static Delegate d =
      new Delegate(World.aspectOf(), "ajc$0");
  static BindHandle id;
  static { .. }
  public void deploy() {
    id = bind(p, d);
  }
  public void undeploy() {
    remove(id);
  }
  public void ajc$0() {
    System.out.println("World");
  }
  //Elided generated code for hasAspect()
  //and aspectOf() helper methods
}
```

Equivalent intermediate code: (=> means translates to)

```
id = bind(p, d) => {getstatic p; getstatic d;
                    bind; putstatic id;}
remove(id) => {getstatic id; remove;}
```

Figure 9: Compiling dynamic deployment constructs

The deploy and undeploy constructs are modeled by generating methods that contain the code to bind and remove the pointcuts and delegates in the aspect. The call to deploy and undeploy in the program is replaced by `World.aspectOf().deploy()` and `World.aspectOf().undeploy()` respectively.

The strategies discussed in Section 3.1.2 also apply in this case. This strategy for compiling dynamic deployment constructs also maintains the separation of the aspect modules and base modules.

3.3 Compiling Control Flow Constructs

A more surprising result for us was to be able to compile control flow constructs cflow and cflowbelow in AspectJ-like languages. Moreover, each of these strategies produced modular object code. In this sub-section, we describe these compilation strategies. Like the previous section, the equivalent source code is shown for presentation purposes.

The AspectJ language provides a construct called *cflow* to separate crosscutting concerns based on the control flow of the program. The AspectJ programming guide [2] informally defines a *cflow* pointcut as follows: "The cflow pointcut picks out all join points that occur between entry and exit of each join point P picked out by Pointcut, including P itself. Hence, it picks out the join points in the control flow of the join points picked out by Pointcut." The *cflowbelow* construct is similar, except it does not pick out the join points matched by the pointcut itself.

The compilation strategy for the *cflow* and *cflowbelow* constructs are similar. We will discuss the cflowbelow case as it is slightly more interesting, pointing out differences from cflow as necessary.

An example usage of this pointcut expression is shown in Figure 10. In this example, `aspect Counting` uses the *cflowbelow* construct to count the number of calls to the method `Bit.Set()` below the control flow of the method `Word.Set()`. The pointcut expression will select all calls to the method `Bit.Set()` that occur in any join point that occurs between entry and exit of the method `Word.Set()`.

```
aspect Counting {
  int count;
  before(): cflowbelow(execution(* Word.Set()))
            && call(* Bit.Set()) {
    count++;
  }
}
```

Figure 10: An example usage of the *cflowbelow* construct

An example of a current compilation technique for cflow constructs [2] for this example is as follows: first, generate a stack in `aspect Counting`, second, insert instructions to push and pop a unique *identifier* into this stack at the entry and exit of the method `Word.Set()` and third, insert instructions to check whether that *identifier* is present on the stack at every point in the program where a call to the method `Bit.Set()` is possible [25].

Our compilation strategy for the cflow and cflowbelow constructs is as follows: first, generate two new methods, say `cflow$Bind()` and `cflow$Remove()`, making sure that the names are unique in the class (since the class may already contain other methods), second, *bind* these two methods to execute at the entry and exit of the method `Word.Set()`, respectively, and third, generate code in `cflow$Bind()` and `cflow$Remove()` to *bind* and *remove* the code to the actual advice to execute whenever `Bit.Set()` is called. A stack is used to track multiple *bind* calls to `Word.Set()`, allowing the code to *remove* the proper association. Note that since a delegate is invoked at most once per join point, binding the same association relationship multiple

```
class Counting {
  static int count;
  private static Stack /*BindHandle*/ ids;
  private static Call p; //Static pattern instance
  private static Delegate d; //Advice's delegate
  private static int initialDepth;
  static {
    ids = new Stack();
    p = new Call(new Method("Bit.Set"));
    d = new Delegate(
          ajc$perSingletonInst, "ajc$0");
    Method meth = new Method("Word.Set");
    Execution exec = new Execution(meth);
    Delegate delBind = new Delegate(
          ajc$perSingletonInst,"cflow$Bind");
    bind(exec, delBind);
    Delegate delRemove = new Delegate(
          ajc$perSingletonInst,"cflow$Remove");
    Return ret = new Return(meth);
    bind(ret, delRemove);
    Failure fail = new Failure(meth);
    bind(fail, delRemove);
  }
  private void cflow$Bind() {
    BindHandle handle = bind(p, d);
    ids.push(handle);
    initialDepth =
        Thread.currentThread().countStackFrames();
  }
  private void cflow$Remove() {
    remove(ids.pop());
  }
  public void ajc$0() {
    if (initialDepth >=
        Thread.currentThread().countStackFrames())
      return;
    count++;
  }
}
```

Figure 11: The generated code for *cflowbelow*

times will not cause the VM to invoke the delegate multiple times at matching join point shadows.

Some bookkeeping is required to keep track of the execution stack depth in the variable `initialDepth`. Inside the advice body, a check is generated to determine if the stack depth is the same. If the stack depth is the same, then any call being made to `Bit.Set()` is being performed from the initial call to `Word.Set()` — we are not *below* the control flow of `Word.Set()`. In this case, the delegate simply returns without executing the advice body. If the stack depth is larger, then we are below the control flow of `Word.Set()` and may continue executing the advice body. Figure 11 shows the results of the code generation for the example program in Figure 10. As previously mentioned, the equivalent source code is shown for ease of presentation. The only difference between the compilation of cflow and cflowbelow is that the bookkeeping code for stack depth is not generated in the case of cflow.

4. PROTOTYPE VM IMPLEMENTATION

We have extended the Sun Hotspot Java virtual machine (or Hotspot for short) to support the *bind* and *remove* primitives. In our prototype implementation, we mimic these instructions as native methods inside the VM. In the rest of this section, we describe the relevant aspects of Hotspot, our extensions, and a comparison of their runtime performance that serves to support our claim that it is feasible to support *Nu* in an industrial strength VM implementation without significant performance degradation.

In Section 4.3 we describe the dispatch at join points. Section 4.5 describes our delegate invocation technique. Section 4.4 details our evaluation of the implementation. Section 4.6 describes the implementation specific details for the *bind* and *remove* primitives.

4.1 Background: Sun Hotspot JVM

Hotspot uses mixed-mode execution for faster performance [1]. There are three modes of bytecode execution: an interpreter, a fast non-optimizing compiler and a slow optimizing compiler. It uses runtime profiling to identify a set of performance-critical methods in the Java program. The compilation efforts are then focussed on these performance critical methods and the rest of the code is interpreted [29]. The insight is based on Hölzle and Ungar's work on adaptive optimization of Self [15], where a profile-based technique called type feedback is used to direct dynamic optimizations.

For the few parts of the Java program that are executed most often, the adaptive optimizing compiler produces optimized native code. The key idea is that there are often no gains achieved by compiling the entire program to produce the native code before running it. The interpreter uses a macro-assembler to generate generic stubs for the entry of Java methods. These stubs include a check to see if a compiled version of the method exists and if so, directly jumps to the compiled code. If not, the stub will continue executing inside the interpreter. The mixed-mode execution strategies and the preference to use the interpreter loop for execution of most of the program are precisely the reasons for selecting Hotspot for our prototype. The ease of extending the interpreter loop and small number of compilations make it easy to efficiently support dynamic intermediate language designs such as *Nu*.

4.2 Our VM Implementation Strategy

Previous studies of Java programs, for example by Krintz *et al.* [21], show that up to 57% of the methods loaded by the VM are never executed. These studies and the results on adaptive optimization led us to our implementation strategy that in the common case we should interpret a join point instead of statically weaving it like traditional AO compilers that modify the join point shadows in the bytecode. Only frequently executing join point shadows should be dynamically woven.

Our current VM implementation provides a dispatch mechanism at each join point. The focus of the prototype presented in this paper is to optimize this dispatch mechanism. This mechanism handles matching the join point to existing patterns and invoking any corresponding matched delegates. We take advantage of the stub generation code of Hotspot, adding in additional code to perform our dispatch for the join points. We have implemented code for method execution and return join points. At this time, we have only tested the interpreter loop. We plan to continue our investigation into the Hotspot compilation process. Based on an initial review of the Hotspot compiler code, we have determined that the compiler also uses the interpreter stubs for method entry and exits and therefore extending our strategies should be feasible.

4.3 Join Point Dispatch in Nu's VM

At each appropriate point in the interpreter corresponding to the execution of a join point shadow, we added code that performs three checks, implemented as three `mov`, three `cmpl`, and three `jcc` assembly instructions. These assembly instructions are emitted in the assembly code stubs generated by the VM.

The first check is a filtering check to prevent JRE and *Nu* runtime join point shadows from being advised. The second check is a cache validation check that determines if the cached pattern matching results for the join point shadow are valid. If the results

are not valid, an incremental pattern match is performed for the join point shadow and the pattern matching results are cached. The third check determines if there are any cached delegates that need to be invoked at this join point shadow, pending check of any dynamic residues. If the check passes, the delegates are invoked, otherwise the join point shadow execution continues. This code is designed to maximize the use of branch prediction algorithms implemented by most modern processors. If a join point is executed frequently, these checks will be optimized away by the (correct) branch prediction, minimizing the dispatch overhead.

4.3.1 Caching technique in Nu's VM

Matching a join point with a list of bound patterns at runtime is an expensive operation that is a separate research topic on its own; however, caching techniques can be used to minimize the amortized cost of this operation. To that end, we have implemented a two-level caching algorithm for dynamic matching at a join point shadow. Following the terminology of the computer architecture community, hereon we refer to these two caches as the *L1 cache* and *L2 cache*. A join point shadow match result being present or not present in a cache is referred to as a *hit* or *miss* respectively. The L1 cache is maintained at the join point shadow in the form of a list of references to the (delegate, pattern) pairs that have already matched with that join point shadow. In the previous section, the cache validation check that we described pertains to the L1 cache. The L2 cache for each join point kind is maintained inside the pattern matcher in the form of a hash map from the join point shadow signatures to a list of current patterns that match that signature. Similar to L1 and L2 caches inside a processor, a L1 hit is the least costly operation, followed by a L2 hit, then followed by a match.

We implemented a very simple algorithm for detecting a L1 cache hit/miss. Each join point shadow contains a counter that is initialized to zero, when the class containing the join point shadow is loaded. There is also a global counter for each join point kind that is initialized to zero when the VM is initialized. The global counter for a join point kind is incremented on *bind*/*remove* operations, if the bound/removed pattern may match that join point kind.

Patterns internally maintain the information about possible join point shadow kinds that may match during their construction using an iterative scheme. All patterns maintain a fast-match flag. All concrete patterns such as `Execution`, `Call`, etc, statically assign values to this flag that represents matching their specific join point shadow kinds. All dynamic patterns such as `This`, `Target`, etc, match selective join point kinds. When constructed, all `And/Or` composite patterns retrieve the fast match flags from inner patterns supplied as arguments to their constructors and set their own fast match flag to the logical `and/or` of their inner pattern's flag. This scheme is similar to the fast-match technique used by the AspectJ compiler during compile-time [2].

At join point dispatch time, the check for L1 cache hit/miss is simply an equality test between the local counter for the join point shadow and the global counter for that join point kind. At join point match time, the local counter is reinitialized to the current value by the join point matcher. We suspect that better checking techniques might be possible; however, we were able to implement this check using two `mov`, one `cmpl`, and one `jcc` instruction and therefore we did not investigate further in this direction. One nice property of our L1 cache checking algorithm is that it does not require keeping track of all join point shadows of a specific kind that we have seen so far and invalidating their caches during a *bind* operation. This alternative scheme would have reduced the cache check to one `mov`, one `cmpl`, and one `jcc` instruction, but with a significantly expensive *bind* operation.

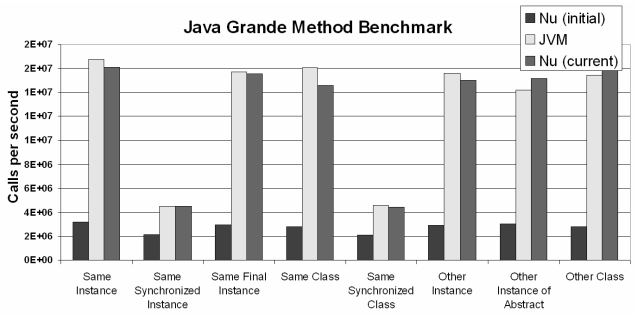

Figure 12: Comparison of Join Point Dispatch times Using the Java Grande Benchmark (larger bars are better)

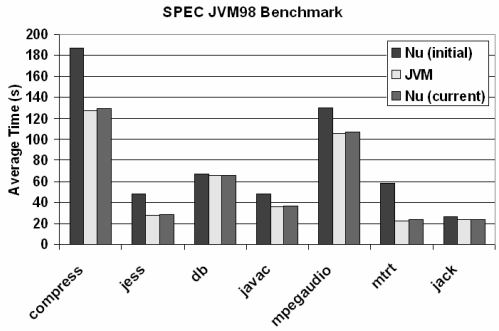

Figure 13: Comparison of Join Point Dispatch times Using the SPEC JVM98 Benchmark (smaller bars are better)

When a join point shadow incurs an L1 cache miss, the pattern matcher is called to perform incremental pattern matching. The overhead of calling the incremental pattern matcher is the cost of an L1 cache miss. Incremental matching refers to a simple technique of only matching patterns that have not already been matched. The join point stores the *bindHandle* of the last pattern it was matched against. When an incremental match is performed, only patterns with newer *bindHandle*s need matched. The incremental match must also check the list of cached delegates to verify none have been *remove*d and if so they are taken out of the join point's L1 cache. At the end of the incremental match, the join point's L1 cache is set to valid.

We compared the performance of our improved join point dispatching technique with a previous version of our runtime with no caching and the unmodified Sun JVM. The results are described in the next section.

4.4 Runtime Performance of Nu's VM

To evaluate the runtime performance of our implementation of

	JVM	Nu (initial)	% of JVM	Nu (current)	% of JVM
check	0.052	0.052	100.90%	0.057	109.86%
compress	127.853	186.968	146.24%	129.068	100.95%
jess	28.086	48.199	171.61%	28.974	103.16%
db	66.346	66.915	100.86%	66.237	99.84%
javac	36.140	48.190	133.34%	36.636	101.37%
mpegaudio	105.596	130.548	123.63%	107.212	101.53%
mtrt	22.651	57.652	254.52%	23.812	105.13%
jack	24.188	26.556	109.79%	24.232	100.18%
Average	51.364	70.635	137.52%	52.028	101.29%

Figure 14: Comparison of Join Point Dispatch times Using the SPEC JVM98 Benchmark (smaller is better)

	JVM	Nu (initial)	% of JVM	Nu (current)	% of JVM
Same Instance	16,765,966.65	3,194,657.43	19.05%	16,105,814.33	96.06%
Same Synchronized Instance	4,497,624.18	2,148,470.59	47.77%	4,518,029.94	100.45%
Same Final Instance	15,709,483.76	2,961,620.07	18.85%	15,537,262.22	98.90%
Same Class	16,032,110.68	2,801,511.89	17.47%	14,554,337.11	90.78%
Same Synchronized Class	4,613,550.68	2,108,878.45	45.71%	4,457,858.00	96.63%
Other Instance	15,571,583.01	2,921,367.39	18.76%	15,055,368.00	96.68%
Other Instance of Abstract	14,240,123.96	3,002,304.42	21.08%	15,181,286.63	106.61%
Other Class	15,449,503.87	2,817,779.49	18.24%	15,909,643.50	102.98%
Average	12,859,993.35	2,744,573.71	21.34%	12,664,949.97	98.48%

Figure 15: Comparison of Join Point Dispatch times Using the Java Grande Benchmark (larger is better)

Nu, we evaluated the performance of the system in the case where no *bind* calls have occurred to determine the dispatch overhead of our VM implementation. We used two standard Java benchmarks for our evaluation: SPEC JVM98 and Java Grande. Since we are advocating modifying a production level VM, it is important that the modifications do not significantly affect the performance of existing applications. To measure the overhead in these cases, we ran the SPEC JVM98 and Java Grande method benchmarks on our modified VM. There were no *bind*/*remove* calls in these benchmarks. We measured the performance of the unmodified JVM, our initial implementation of *Nu*, and our current implementation of *Nu* as described in this paper.

The results for the Java Grande method benchmark are shown in Figures 12 and 15. Since the Java Grande method benchmark executes simple methods repeatedly to obtain the average number of method calls possible per second, this is where our caching implementation really shows up. Our initial version had to perform matching on each method call (even though there were no binds). With caching in place, this match is performed once. Our implementation went from 21.3% to 98.5% of the method calls achieved by the unmodified JVM.

The results for the SPEC JVM98 benchmark are shown in Figures 13 and 14. This benchmark measures the time to execute a set of realistic applications. These results were similar to the Java Grande benchmark. Our implementation went from a 37% execution time overhead to a little over 1% overhead.

4.5 Delegate Invocation in Nu's VM

Due to the lack of delegates in Java, our initial implementation made use of the reflection API and Java Native Interface (JNI) methods. Users passed in strings representing the name of a class and the name of the delegate method and the runtime created a reflection `Method` object representing the specified delegate. This object was then passed into *bind* calls. JNI methods available inside the VM were then used to invoke the delegate where necessary.

Our current strategy still makes use of the reflection API `Method` class for passing in a delegate to *bind* calls. The *bind* implementation makes use of data structures already available inside the VM to keep track of information regarding the delegate, such as class, instance, method, etc. When the VM initially loads, template code for invoking delegates is generated inside the method stubs. This code makes use of the stored information about the delegate, avoiding the need to use expensive JNI methods.

To measure the performance of our delegate invocation code, we created a benchmark that calls a simple test method repeatedly. A delegate method that increments a static counter is then used to create an advising relationship with our test method. A copy of the test method is created with manually inlined calls to the delegate method. The number of manually inlined calls is equal to the number of advising relationships created using *bind*. Both copies of the test method (one with manually inlined calls and one with advising relationships to the delegate) are then executed and timed.

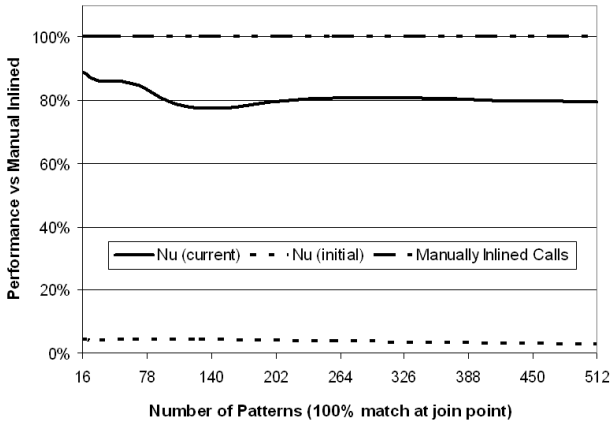

Figure 16: Invoke Benchmark - Varying Number of Patterns

A comparison to AspectJ's advice invocation code was not made, since most typical AspectJ compilers generate two methods at the call site (one to get an instance of the aspect and one to call the advice method).

Figure 16 varies the total number of *bind* calls while keeping the percent that match the test method at 100%. Figure 17 varies the percentage of *bind* calls that match the test method while keeping the total number of *bind* calls at 256. As can be seen from the figures, our delegate invocation technique went from around 4% as efficient as the manually inlined version to around 82%. We believe that as we refine our technique, our invocation mechanism should approach relatively the same efficiency as manually inlining calls to delegate methods.

4.6 Handling Bind/Remove Calls in Nu's VM

The modified VM handles *bind* calls by storing the pattern and delegate objects into a list. There is one list for each type of join point and the pattern indicates which join point(s) it applies to. It also performs some simple sanity checks (like verifying neither object are null, if the delegate is non-static then an instance object was passed in, etc). The VM then stores the pair into all applicable lists, generates and returns a unique `BindHandle` to the caller. The `BindHandle` is an instance of the immutable Java class `BindHandle`, which may only be instantiated by the VM.

For *remove* calls, the modified VM simply removes the pattern/delegate pair matching the passed in `BindHandle` from all lists. Any join point that previously cached the delegate will lazily, on its next execution, recognize the cache is invalid and remove the delegate from the cache.

The class file processor was modified to initialize data structures used at each join point. These data structures consist of several flags for use in caching, a local cached delegate list, and storage

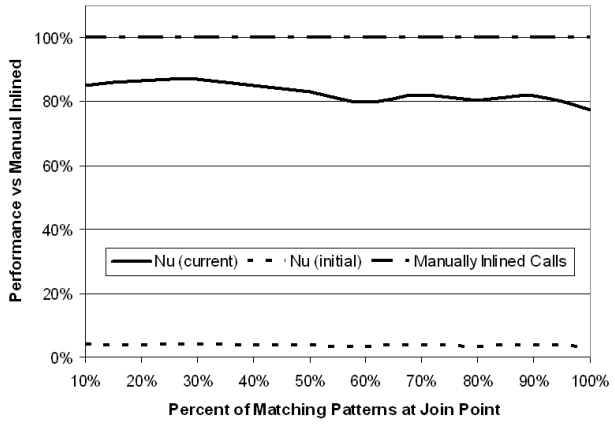

Figure 17: Invoke Benchmark - Varying % Matching Patterns

for the join point's static reflective information (which is created lazily upon first use). The class file processor already accesses the bytecode of potential join point shadows, so no additional iterations were needed for initializing these data structures.

4.7 Summary

Our current prototype implementation serves as a proof of concept of our claim that support for the *Nu* IL model in production level virtual machine is feasible. Starting from our very inefficient implementation, we have improved our join point dispatch by reducing the overhead from 37% to 1.27% for the SPEC JVM98 benchmark and increased our performance on the Java Grande benchmark from 21.34% of the unmodified Hotspot to 98.48% of the unmodified Hotspot. Delegate invocation improved from around 4% as efficient as the manually inlined version to around 82% as efficient.

5. COMPILER IMPLEMENTATION

The strategies for compiling AspectJ constructs were implemented by modifying the AspectJ compiler *ajc* to generate *Nu* intermediate code as described in Section 3. We will refer to this compiler as *ajc-nu* from here onwards. The version of *ajc* modified was 1.5.0. Please note that at the time of this paper, version 1.5.3 of the AspectJ compiler was available but was not used due to various problems when performing incremental compiles.

Their are two key differences between our compiler, ajc-nu, and the original AspectJ compiler, ajc. First, our compiler does not have to weave advice constructs. Note that at this time, we let the original parts of the ajc compiler related to weaving inter-type and declare constructs intact in ajc-nu. Some additional work is added to a pass that analyzes and transforms AspectJ's aspect into classes in the object code. This code now also performs the translation of the AspectJ constructs into *Nu* primitives as previously described. Finally, we have tested our compiler on large-scale projects such as Eclipse IDE and the Azureus P2P system by adding aspects to these system [9].

6. RELATED WORK

Three closely related and complimentary research ideas are runtime weaving, load-time weaving and virtual machine support for AOP. We discuss these ideas in detail below.

6.1 Run- and Load-Time Weaving

There are several approaches for run-time weaving such as PROSE [31], Handi-Wrap [5], Eos [32], etc. A typical approach to runtime weaving is to attach hooks at all join points in the program at compile-time. The aspects can then use these hooks to attach and detach at run-time. An alternative approach is to attach hooks only at potentially interesting join points. In the former case, aspects can use all possible join points, excluding those that are created dynamically so the system will be more flexible. The disadvantage is the high overhead of unnecessary hooks. In the latter case, only those aspects that utilize existing hooks can be deployed at run-time, but the overhead will be minimal for a runtime approach.

Eos uses the second model, i.e. only instrument the join points that may potentially be needed. Handi-Wrap uses the first model, making all join points available through wrappers. PROSE indirectly uses the first model, exposing all join points through the debugger interface. PROSE allows aspects to be loaded dynamically without restarting the system. An additional advantage of indirectly exposing join points through a debugger interface is that new join points (created by reflection) are registered automatically. As observed by Popovici *et al.* [31] and Ortin *et al.* [27], however, performance in both cases is a problem.

A load-time weaving approach delays weaving of crosscutting concerns until the class loader loads the class file and defines it to the virtual machine [24]. Load-time weaving approaches typically provide weaving information in the form of XML directives or annotations. The aspect weaver then revises the assemblies or classes according to weaving directives at load-time. A custom class loader is often needed for this approach.

There are load-time weaving approaches for both Java and the .NET framework. For example, AspectJ [17] recently added load-time weaving support. Weave.NET [22] uses a similar approach for the .NET framework. The JMangler framework can also be used for load-time weaving [20]. It provides mechanisms to plug-in class-loaders into the JVM.

A benefit of the load- and run-time weaving approaches is that they delay weaving of AO programs. A contribution of our approach might also be perceived as delaying weaving, however, we view the interface and corresponding contracts between the language designs and execution model designs as the main contribution of our work. The decoupling between language compilers and the virtual machine achieved by the interface provided by our IL model enables independent research in these areas. Simpler aspect language designs and compiler implementations might be realized without spending significant time on the optimization of the underlying AO execution models. Novel optimization mechanisms for the underlying execution models can be developed independent of the language design as long as it conforms to the interface. The load-time weaving approaches do not provide these benefits.

The bind and remove primitives are similar to install and uninstall messages in AspectS [13]. The difference is that install and uninstall messages are sent to aspects in AspectS, whereas bind and remove can be thought of as messages sent to the virtual machine.

6.2 Virtual Machine Support of Aspects

Steamloom [6] and *PROSE2* [30] both aim to achieve an aspect-aware Java VM, to enhance the runtime performance of AOP. Steamloom extends the Jikes Research VM, an open source Java VM [4]. Traditional approaches for supporting dynamic crosscutting involve weaving aspects into the program at compilation. Steamloom moves weaving into the VM, which allows preserving the original structure of the code after compilation and shows performance improvements of 2.4 to 4 times when compared to

AspectJ. It accomplishes this by modifying the Type Information Block to point methods to a stub that modifies the existing byte code to weave in the advice. On the other hand, PROSE2 proposes an enhanced implementation for the original PROSE approach, by incorporating an execution monitor for join points into the virtual machine. This execution monitor is responsible for notifying the AOP engine which in turn executes the corresponding advices.

Our approach and Steamloom are in some sense complimentary. Similar to Steamloom, our approach also advocates support for crosscutting in the execution models. Steamloom investigates techniques to improve the performance of these crosscutting mechanisms provided by the execution model, whereas, our approach focuses on separating the compiler implementations and execution model implementations by defining an interface between the two. Our focus is on providing the basic mechanisms at the interface that can be used as primitives by compiler implementations. Our approach thus potentially allows multiple language models to use the same VM and/or multiple VMs. Each of these VMs may have their own method of weaving.

Steamloom and PROSE2, however, restrict the type hierarchy of aspects. An aspect must inherit from a special class. In languages like Java, this restriction uses up the only available inheritance link. Our approach does not impose any restrictions on programming language constructs, leaving those design decisions to programming language designers and compiler implementers.

7. DISCUSSION

The compilation strategies described in Section 3 demonstrated that a variety of constructs in high-level languages can be supported by our intermediate language model. A key property of all these strategies is that they retain the separation of aspects and base in the object code. Maintaining this separation in object code helps with incremental compilation of aspect-oriented programs and source-level debugging.

Incremental compilation is defined as the property of a compiler such that a small change in syntax or semantic structure requires only a small amount of reprocessing to reflect the change [3]. In a recent report, Lesiecki [23] observed that on an average, incremental compilation of 700 classes and around 70 aspects using the AspectJ compiler usually takes at least 2-3 seconds longer than the near instant compilation time using a pure Java compiler.

The speed of incremental compilation affects the responsiveness of integrated development environments (IDEs), such as Eclipse. As Hölzle and Ungar point out, in the context of object-oriented programming and more specifically the Self language, an IDE must respond as quickly as possible after programming changes in order to increase programmer productivity [15]. Hölzle also argues that "compilation must be quick and non-intrusive, and the system must support full source-level debugging at all times" [14]. These requirements are valid for aspect-oriented IDEs as well. Maintaining the separation of concerns in object code helps achieve these goals.

```
after(): call(* *.read(..)) &&
  && cflow(execution(* java.io.InputStream.*)) {
  /* Do Something */
}
```

Figure 18: Original pointcut expression

```
after(): call(* *.read(..)) &&
  cflow(execution(* java.io.FileInputStream.*)) {
  /* Do Something */
}
```

Figure 19: Modified pointcut expression

To illustrate the issue, let us consider the compilation of the *cflow* construct. In current implementations of aspect-oriented compilers, *cflow* is implemented by generating additional code to perform *runtime checks* and determine if the program is currently in the control flow of a join point (such as the execution of a method). For example, the pointcut expression in Figure 18 will result in the generation of a set of instructions to dynamically check at all program points where a method `read(..)` is called to determine if the program is currently in the control flow of any method in `java.io.InputStream`.

Now, a change in the pointcut expression, say to the pointcut in Figure 19, will have an impact on all program points where the dynamic check was generated. Instead of checking for methods belonging to type `java.io.InputStream`, the generated code will now check for methods belonging to type `java.io.FileInputStream`. A simple change in the source code will thus lead to non-trivial processing.

On the other hand, if the aspect were to be translated to the *Nu* intermediate language model using the strategy described in Section 3.3 (see Figure 11), the change in the pointcut expression would only require localized changes inside the aspect, proportional to the change in the source code, thereby leading to a reduced incremental compilation time and faster responsiveness.

Our recent work [9] has examined this problem in detail by measuring the incremental compilation time of AspectJ [2] programs after making minor modifications to the source code. Figure 20 shows the incremental compilation times of the Azureus peer-to-peer application, a medium-scale system with around 3500 classes, 2000 source files, and 200 KLOC. In most cases, a small change in the aspect resulted in an increase in incremental compilation time of 10 times when compared to the incremental compilation of a Java class in the system. This problem becomes even more apparent with larger systems. Eclipse, a Java integrated development environment, is a large-scale system. Figure 21 shows the incremental compilation time of a subset of Eclipse[2] (over 12000 classes, 7000 source files, and almost 800 KLOC). In most cases, the incremental compilation times in this system are over 30 seconds for very simple changes. The increased incremental compilation time can potentially affect the build-test-debug cycle common in many software development processes.

The benefits of AO modularity are attractive at a large scale; in fact, the majority of the benefits only become apparent during large-scale uses such as IBM WebSphere [8]. Large-scale usage, however, does come with unique performance requirements such as *design-build-test cycle time*, *full build time*, etc. The best AO compilers available today have pushed the performance limits and delivered significant improvements compared to early versions, however, there is only so far they can go without addressing the underlying fundamental problem. Once these issues become apparent to the large-scale adopters, tough decisions on the trade-off of Separation of Concerns vs. performance will have to be made. This work thus creates a timely opportunity to rethink the commitment to object-oriented intermediate languages that the original AO compilers entailed.

8. FUTURE WORK

Our future investigations will focus on two key areas: language extensions and virtual machine optimizations.

[2]A subset was selected due to memory constraints with Java and the large amount of memory required by the AspectJ incremental compiler.

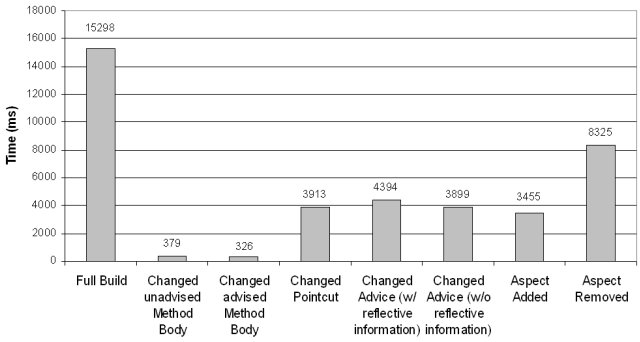

Figure 20: Incremental compile times of Azureus [9]

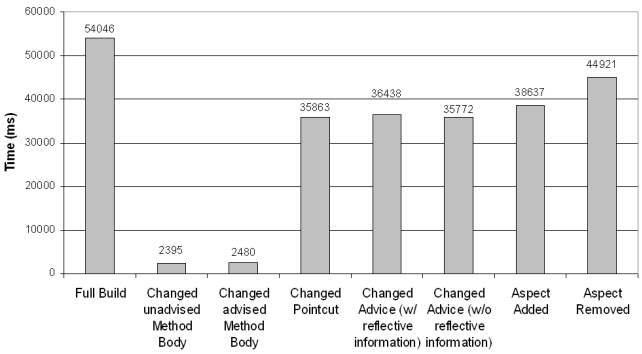

Figure 21: Incremental compile times of Eclipse [9]

8.1 Language Extensions

There are several possible routes for extensions to the *Nu* IL model. One extension would create another IL primitive, say *bindStatic*, which would behave similar to *bind* but with the additional semantics that the bind identifier returned after a *bindStatic* call can never be passed into *remove*. These semantics are useful for static deployment cases and would allow virtual machine implementations to perform optimizations such as code re-writing. Another possible extension which is similar, but would allow the returned identifier to be passed into *remove*, would state that the virtual machine should indeed perform code re-writing for that specific call. This could be used in situations where a compiler is able to statically identify that it is unlikely that *remove* will happen (perhaps through an examination of the control flow graph), allowing the virtual machine to re-write the byte code for faster execution.

Our current implementation does not support *around* constructs in AspectJ-like languages. Endoh *et al.* have proposed adding two constructs, *proceed* and *skip*, to handle around advice [11]. We plan to add and implement similar constructs in our IL model to explore support for around advice in our pointcut model.

Currently, our intermediate language design does not support inter-type declarations. These constructs allow aspects to declare new methods or fields in another type, declare a type extends a new class, or declare a type implements new interfaces. Inter-type declarations can be compiled to the *Nu* intermediate language by directly adding the declarations to the class that it crosscuts. In cases where the declaration affects more than one class, this will require compiling several classes. Clearly, this strategy is not modular since a change in an aspect may affect not only the aspect's object code, but also the object code of each class into which the inter-type declaration is being introduced.

A more general problem is support for multi-dimensional separation of concerns and HyperJ constructs in the virtual machine. Fortunately, researchers are beginning to identify possible directions. For example, recently Ossher [28] identified a runtime model based on fragmented objects as a basis, which appears to be a promising direction for future extensions of the *Nu* model.

8.2 Optimizations

We have planned several optimizations to further optimize the dispatch time of our prototype virtual machine. Additional optimizations for improved pattern matching and delegate invocation are also planned. In the rest of this section, we will briefly describe these optimizations.

8.2.1 Further Improved Join Point Dispatch

The Hotspot VM keeps a list of tables for efficient interpretation. During VM initialization time, this table is also initialized with code buffers that contain optimized interpretation code for various different types of entry and exit events in the interpreter. In our current optimization, we insert additional instructions into these code buffers. During the execution of a program, the interpreter simply jumps to different entries in these tables as appropriate.

We plan to implement strategies to swap entries in this table such that an entry always points to the most optimal code buffer. At VM initialization time, we will generate multiple generic code buffers, each optimized for specific join point dispatch scenarios. For example, if we have not seen any bind instructions yet for a join point kind, there is no need for dispatch condition checks. As soon as the VM sees a bind call for a specific join point kind, it checks to see if the interpreter table is already initialized to support dispatch of that join point kind. If not, it replaces the entry with the right code buffer. Note that these modified entries will not be generated for every join point instance, just for each join point kind.

On a remove, the VM will check to see if there are any more binds remaining in the list of a join point kind. If there are no more binds in a list of join point kinds, the interpreter replaces the entry for that join point kind with the original entry that does not contain dispatch checks. These two modifications should speed up the join point dispatch by eliminating the need for redundant checks.

8.2.2 More Efficient Join Point Matching

The language implementation techniques for aspect-oriented quantification mechanisms, i.e. matching join points against a (possibly large) set of pointcut predicates, have not received much attention. This is primarily because most aspect-oriented approaches today employ compile-time deployment of aspects, where the cost of quantification is a small percentage of total compilation time. Recently, however, many usecases for dynamic aspect deployment have emerged, including ours [31, 5, 6, 30].

An implementation challenge for languages providing dynamic deployment constructs is to efficiently determine the set of join points that are matched by the aspect being deployed (or removed). This is primarily because in this case the cost of matching may become a significant portion of the cost of the deployment operation.

In the future, we will look into efficient join point matching mechanisms. In particular, a decision tree-based approach for matching join points against a set of pointcuts may potentially reduce the cost of matching. Unlike previous approaches implemented in AO compilers that treat each pointcut individually, one can maintain all pointcuts in the system in a single decision tree, which allows us to utilize more implication relationships resulting in a faster matching process.

9. CONCLUSION

In this paper, we introduced *Nu*, an AO IL model that adds two primitives to object-oriented IL models. These primitives are geared towards a class of AO languages called pointcut-advice languages. *Nu* exhibits two key properties: first it is able to express a number of deployment models and constructs proposed in AO literature and second that representation of these constructs in *Nu* preserves AO design modularity in the object code.

We also described a prototype virtual machine implementation that supports the *Nu* IL model. Our performance analysis showed that there is minimal performance degradation of method dispatch time compared to the unmodified JVM. The speed of invoking the delegate also remains fairly close to the manually inlined method call because of our caching mechanisms.

Acknowledgements This work is supported in part by the National Science Foundation under grant CNS-0627354. We would like to thank Prem Devanbu, Youssef Hanna, Mats Heimdahl, Gregor Kiczales, Gary Leavens, Juri Memmert, Harish Narayanappa, Giora Slutzki, Eric Van Wyk, and Moshe Y. Vardi for insightful comments.

10. REFERENCES

[1] O. Agesen and D. Detlefs. Mixed-mode bytecode execution. Technical Report TR-2000-87, IBM Research, June 2000.

[2] AspectJ programming guide. http://www.eclipse.org/aspectj/.

[3] L. V. Atkinson, J. J. McGregor, and S. D. North. Context sensitive editing as an approach to incremental compilation. *The Computer Journal*, 24(3):222–229, 1981.

[4] B. Alpern et al. The jikes research virtual machine project: Buliding an open-source research community. *IBM Systems Journal*, 44(2), 2005.

[5] J. Baker and W. Hsieh. Runtime aspect weaving through metaprogramming. In *AOSD '02*, pages 86–95, USA, 2002. ACM Press.

[6] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual machine support for dynamic join points. In *AOSD '04*, pages 83–92, 2004.

[7] K. Böllert. On weaving aspects. In *Proceedings of the Workshop on Object-Oriented Technology*, pages 301–302, London, UK, 1999. Springer-Verlag.

[8] A. Colyer and A. Clement. Large-scale aosd for middleware. In *AOSD '04*, pages 56–65, USA, 2004. ACM Press.

[9] R. Dyer, R. Setty, Y. Suvorov, and H. Rajan. A comparison of weaving techniques from the aspect-oriented incremental compilation perspective. Technical Report 07-XX, Iowa State University, Department of Computer Science, June 2007.

[10] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming: Introduction. *Commun. ACM*, 44(10):29–32, 2001.

[11] Y. Endoh, H. Masuhara, and A. Yonezawa. Continuation join point. In *FOAL 06, A workshop affiliated with AOSD 2006*, March 2006.

[12] E. Hilsdale and J. Hugunin. Advice weaving in aspectj. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 26–35, New York, NY, USA, 2004. ACM Press.

[13] R. Hirschfeld. Aspect-oriented programming with AspectS. In M. Akşit and M. Mezini, editors, *Net.Object Days 2002*, October 2002.

[14] U. Hölzle. *Adaptive optimization for self: reconciling high performance with exploratory programming*. PhD thesis, Stanford, CA, USA, 1995.

[15] U. Hölzle and D. Ungar. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Trans. Program. Lang. Syst.*, 18(4):355–400, 1996.

[16] G. Kiczales. Personal communication with Hridesh Rajan at AOSD'07, 2007.

[17] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *ECOOP 2001*, pages 327–353. Springer-Verlag, Hungary, June 2001.

[18] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Finland, June 1997. Springer-Verlag.

[19] G. Kniesel. Type-safe delegation for run-time component adaptation. In *ECOOP '99: Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 351–366, London, UK, 1999. Springer-Verlag.

[20] G. Kniesel, P. Costanza, and M. Austermann. Jmangler-a framework for load-time transformation of java class files. In *SCAM 2001*, pages 100–110. IEEE Computer Society, 2001.

[21] C. J. Krintz, D. Grove, V. Sarkar, and B. Calder. Reducing the overhead of dynamic compilation. volume 31.

[22] D. Lafferty and V. Cahill. Language-independent aspect-oriented programming. In *OOPSLA '03*, pages 1–12, USA, 2003. ACM Press.

[23] N. Lesiecki. Applying AspectJ to J2EE application development. In *AOSD '05*, 2005.

[24] S. Liang and G. Bracha. Dynamic class loading in the java virtual machine. In *OOPSLA '98*, pages 36–44, USA, 1998. ACM Press.

[25] H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In *Compiler Construction (CC2003)*, 2003.

[26] M. Mezini and K. Ostermann. Conquering aspects with caesar. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 90–99, New York, NY, USA, 2003. ACM Press.

[27] F. Ortin and J. M. Cueva. Dynamic adaptation of application aspects. *Journal of Systems and Software*, 71(3):229–243, 2004.

[28] H. Ossher. A direction for research on virtual machine support for concern composition. In *VMIL '07: Proceedings of the 1st workshop on Virtual machines and intermediate languages for emerging modularization mechanisms*, page 5, New York, NY, USA, 2007. ACM Press.

[29] M. Paleczny, C. Vick, and C. Click. The java hotspot server compiler. In *Java Virtual Machine Research and Technology Symposium*, 2001.

[30] A. Popovici, G. Alonso, and T. Gross. Just-in-time aspects: efficient dynamic weaving for java, 2003.

[31] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *AOSD '02*, pages 141–147, USA, 2002. ACM Press.

[32] H. Rajan and K. J. Sullivan. Eos: instance-level aspects for integrated system design. In *ESEC/FSE-11*, pages 297–306, Sept 2003.

[33] D. Sabbah. Aspects: from promise to reality. In *AOSD '04*, pages 1–2, USA, 2004. ACM Press.