Nu: a Dynamic Aspect-Oriented Intermediate Language Model and Virtual Machine for Flexible Runtime Adaptation

Robert Dyer
Dept. of Computer Science, Iowa State University
rdyer@cs.iastate.edu

Hridesh Rajan
Dept. of Computer Science, Iowa State University
hridesh@cs.iastate.edu

Abstract

A variety of dynamic aspect-oriented language constructs are proposed in recent literature with corresponding, compelling use cases. Such constructs demonstrate the need to dynamically adapt the set of join points intercepted at a fine-grained level. The notion of morphing aspects and continuous weaving is motivated by this need. We propose an intermediate language model called *Nu*, that extends object-oriented intermediate language models with two fine-grained deployment primitives: *bind* and *remove*. These primitives offer a higher level of abstraction as a compilation target for dynamic aspect-oriented language constructs, thereby making it easier to support such constructs.

We present the design and implementation of the *Nu* model in the Sun Hotspot VM, an industrial strength virtual machine, which serves to show the feasibility of the intermediate language design. Our implementation uses dedicated caching mechanisms to significantly reduce the amortized costs of join point dispatch. Our evaluation shows that the cost of supporting a dynamic deployment model can be reduced to as little as $\sim$1.5%. We demonstrate the potential utility of the intermediate language design by expressing a variety of aspect-oriented source language constructs of dynamic flavor such as CaeserJ's deploy, history-based pointcuts, and control flow constructs in terms of the *Nu* model.

Categories and Subject Descriptors D.1.5 [*Programming Techniques*]: Object-oriented Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features — Control structures; Procedures, functions, and subroutines; D.3.4 [*Programming Languages*]: Processors — Code generation; Optimization; Run-time environments

General Terms Design, Human Factors, Languages, Performance

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

1. Introduction

Dynamic constructs have received a lot of attention in the past 3-4 years of aspect-oriented programming literature [3, 4, 5, 8, 10, 14, 15, 30, 31, 33, 35, 36, 37]. A number of supporting use cases have also appeared e.g. in runtime monitoring, runtime adaptation to fix bugs or add features to long running applications, runtime update of dynamic policy changes, etc. Such use cases drive the need for more dynamic aspect-oriented (AO) language constructs.

Some of these proposals have investigated support for dynamic constructs by translating them to static constructs [6, 10, 35, 36]. Such translations demonstrate the need for a more flexible deployment model [5, 10]. In particular, the need to dynamically adapt the set of join points intercepted at a finer-grained level then currently available is demonstrated for existing dynamic constructs such as cflow [6, 10]. In some cases, a finer-grained deployment model enables simpler implementations e.g. in the case of temporal assertion checking using aspects, where each proposition is represented as an advice, advice representing the following propositions need not be checked until the preceding, enabling propositions are found true [35].

In this work, we propose an intermediate language (IL) model that supports finer-grained runtime deployment at the level of advice-like constructs. The rationale for supporting such constructs at the intermediate language level is to provide a higher level of abstraction as a compilation target for dynamic aspect-oriented language constructs, compared to object-oriented intermediate language models, thereby making it easier to support such constructs. Such support at the intermediate language level can be used as a building block for a variety of dynamic constructs in high-level aspect-oriented languages.

Our intermediate language model, which we call *Nu*, extends the object-oriented intermediate language model with two new atomic deployment primitives, *bind* and *remove*, and a point-in-time join point model [23]. The effect of these primitives is to manipulate *advising relationships*. For the purpose of this paper, by advising relationship we mean a many-to-one relation between join points and a delegate. If a point in the execution of a program and a delegate are in an advising relationship, the execution of the join point is extended by the delegate. The effect of the *bind* primitive is to dynamically create such an advising relationship. The effect of the *remove* primitive is to destroy a specified advising relationship. Our intermediate language model has the following properties:

- It is simple. Only two new primitives are added to the object-oriented intermediate language model.
- It is flexible enough to be able to accommodate the requirements of a broad set of dynamic and static source language constructs such as AspectJ's statically deployed *aspects* [18], CaesarJ's *deploy* [24], control flow constructs such as AspectJ's *cflow* and history-based pointcuts [8, 36].
- It provides a higher level of abstraction as a compilation target for dynamic aspect-oriented language constructs.

AOSD'08, March 31 – April 4, 2008, Brussels, Belgium.

- It allows compilers to maintain the conceptual separation present in the source code in the object code as well. *Nu* supports what Bockisch *et al.* have called *structure-preserving compilation* [5]. The intermediate code now mirrors the design, which among other things is important for the efficiency of incremental compilers [4, 32] and dynamic adaptation.

An important consideration for such dynamic models is the performance overhead of supporting them. Previous research results have shown that support for such dynamic aspect-oriented models outside the virtual machine (VM) can be prohibitively expensive [3, 31]. Following Bockisch *et al.* [5], we argue that efficient support is possible for such constructs by utilizing extra information available inside the VM. To that end, we discuss strategies that contribute to near negligible overhead for *Nu*'s runtime flexibility.

In summary, this work makes the following contributions:

- a simple, flexible, and dynamic intermediate language model;
- an implementation of the *Nu* model as an extension of the Sun Hotspot Java Virtual Machine (Hotspot JVM) [28], which serves to show the feasibility of supporting the proposed model in a production level virtual machine;
- a caching technique to reduce amortized join point dispatch overhead for dynamic deployment models;
- an implementation in a VM for the point-in-time join point model;
- and, an analysis of a set of techniques to optimize our highly dynamic deployment model.

In the following, we describe our intermediate language design. Section 3 illustrates the potential utility of the intermediate language design by showing strategies to support a variety of dynamic and static aspect-oriented constructs by translating them into our intermediate language model. We then describe our implementation strategy to support the *Nu* intermediate language model in the Hotspot JVM in Section 4. A novel caching scheme is discussed in Section 5. We evaluate the performance of our VM in Section 6. Section 7 discusses related work. 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 dynamic and static constructs in AO source languages. This section introduces the join point model adopted by our approach. We then illustrate new primitives using an example. Throughout this section, possible optimizations are analyzed and marked as **[Opt: 0]**.

2.1 Nu's Join Point Model

One central concept in common 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 [18], 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 example join point is marked in Figure 1.

Instead of AspectJ's join point model, we adopted a finer-grained join point model for *Nu*, proposed by Masuhara *et al.* [23]. Masuhara *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 the *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

```
// 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

of a method call [23]. They showed that this model is sufficiently expressive to represent common advising scenarios.

In the *point-in-time* model, 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. These three join points eliminate the need for three different types of advice: *before*, *after returning*, and *after throwing* advice. The *before execution*, *after returning execution*, and *after throwing execution* become equivalent to *execution*, *return*, and *throw* respectively. Figure 2 illustrates this model. Two join point shadows in the method `Hello.main()` are marked as being shadows for the join points "execution of the method `Hello.main()`" and "return of the method `Hello.main()`". Similarly, 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.

```
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

At this time, *Nu* does not support *around* advice (see Section 8 for more details). Interested readers are referred to Masuhara *et al.*'s work [23] for more detail. We have also explicitly decided to not support static crosscutting mechanisms, such as inter-type declarations in AspectJ [18]. These constructs are largely static and they can be easily supported by high-level language compilers using static weaving techniques [7, 12].

Our adoption of this model was in part driven by the clarity it gives to the semantics of fine-grained dynamic deployment. One issue that arises with the deployment of dynamic aspects is when the aspect being deployed advises a join point that is 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 [17]. For example, assume that an aspect `a` is deployed during the execution of a method `m`. This aspect contains an after advice that intercepts the join point "execution of `m`". Note that in the region-in-time model we are still in the scope of the join point "execution of `m`". The question is whether to invoke `a` when `m` returns. Using

	bind	remove
Stack Transition	..., `Pattern`, `Delegate` → ..., `BindHandle`	..., `BindHandle` → ...
Description	Associates the execution of all join points matched by `Pattern` to invoke `Delegate`	Eliminates the advising relationship represented by `BindHandle`
Exceptions	`NullPointerEx` - thrown if any argument is null `IllegalArgumentEx` - thrown if the stack is malformed	`NullPointerEx` - thrown if the `BindHandle` is null `IllegalArgumentEx` - thrown if the `BindHandle` is stale

Figure 3. Specification of Primitives in *Nu*

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*.

```
public class AuthLogger {
  protected static BindHandle id = null;
  protected static Pattern loginPat;
  protected static Delegate logDel;
  static { // Static initializer
    /* create new Method and Execution objects */
    Method m = new Method("*.login");
    loginPat = new Execution(m);
    logDel = new Delegate(AuthLogger.class,
        AuthLogger.class.getMethod("log",
        new Class[0]));
  }
  public static void enable() {
    if (id == null) id = bind(loginPat, logDel);
  }
  public static void disable() {
    if (id != null) { remove(id); id = null; }
  }
  public static void log() {
    // record the time of login
  }
}
```

Figure 4. Bind and Remove in an Example Program

An example is given in Figure 4. For ease of presentation, the corresponding high-level language code is shown. In this figure and in the rest of the presentation, special forms of **bind(..)** and **remove(..)** will be substituted where the intermediate language primitives would normally appear. 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 an instruction to push `id` on the stack followed by a remove primitive.

Figure 4 shows the 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. Since patterns are first-class objects available in the high-level language, they are re-usable. **[Opt: 1]** This allows for possible optimizations by compilers, such as locating commonly used sub-patterns that can be created once and re-used.

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

Like strings in Java, patterns are *constant*; their values cannot be changed after they are created. Since patterns are constant, the virtual machine that implements the *Nu* model does not have to worry about a pattern instance changing after it has been created, which allows for the following optimizations inside the virtual machine:

- **[Opt: 2]** When a pattern is created, a mirror native (C++) object can be created inside the virtual machine that will be much faster to access for pattern matching purposes, compared to accessing Java objects. By making patterns constant, we eliminate the requirement to maintain the consistency between the pattern and its mirror C++ object.
- **[Opt: 3]** For patterns that use regular expressions, at the time of their creation, a non-deterministic finite-state automaton can be created and stored in the mirror native object for significantly faster matching [25]. By making patterns constant, we once again eliminate the requirement to maintain the consistency between the regular expression contained inside the pattern and its mirror non-deterministic finite state automaton contained inside the C++ object.

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 `loginPat` 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`. Both these types are part of *Nu*'s standard library, which is an integral part of *Nu*'s virtual machine implementation. The pattern serves to select the subset of join points in the program. The delegate points to a method that provides the additional code that is to execute at these join points. Two exceptional conditions apply:

- If the top two items of the stack do not contain such well-formed objects, i.e. a pattern and a delegate in that order, an exception of type `IllegalArgumentException` is thrown.
- If either of the top two items of the stack is **`null`**, an exception of type `NullPointerException` is thrown.

In Figure 4, the static initializer of **`class`** `AuthLogger` creates a delegate to the method `AuthLogger.log()` and stores it in the static field `logDel` 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 `loginPat` and the delegate `logDel`, which enables logging of authentication attempts in the system.

After the bind primitive finishes, the pattern and the delegate are popped off of the stack and a unique identifier, described in Section 2.2.4, is pushed on to the stack.

The bind primitive dynamically creates an advising relationship between the join points matched by the pattern and the supplied delegate. On completion of a *bind*, when a join point executes, each delegate supplied with a pattern that matches that join point will intercept its execution. Delegates are invoked *in the same order* in which they were bound. Delegates are invoked at most once per join point (for reasons described in Section 3.2).

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).

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

The language is defined with a per-thread semantics. This means that *bind* and *remove* primitives only affect the advising relationships on the same thread that they 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 (described in detail in the next section). An example is shown in Figure 4, where 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.

If the `BindHandle` instance is stale, i.e. if the advising relationship corresponding to the supplied identifier is already removed, an `IllegalArgumentException` is thrown. If the supplied identifier is **`null`**, an exception of type `NullPointerException` is thrown.

2.2.4 Bind handles

The unique identifier returned by a *bind* primitive is an *immutable* object representing the advising relationship. This unique identifier is an object of *opaque* type `BindHandle`, which is also part of *Nu*'s standard library. A type is *opaque* if there is no way to find out its representation, even by printing. This identifier may only be created by the virtual machine. The following optimizations are feasible for bind handles:

- **[Opt: 4]** Similar to patterns, the internal representation of the bind handle can also be mirrored as a C++ object. The representation of the opaque Java object can contain a pointer to its mirror C++ object and vice-versa.

  During a *remove*, the pointer in the Java object corresponding to the bind handle can be redirected to *null*, marking the bind handle as *stale*. This will allow for an easy check for *stale* bind handles. Note that, if a stale bind handle is supplied to the *remove* primitive, an exception of type `IllegalArgumentException` is thrown.
- **[Opt: 5]** The C++ objects for bind handles can be allocated on a separate, small (non garbage-collected) heap. A specialized and very fast garbage collector can be run more often on this heap, which will traverse the C++ object to Java object link to check if the Java object representing the bind handle has fallen out of scope. In other words, it will compute whether the Java object for the bind handle can be garbage collected. If so, this means that the advising relationship corresponding to that bind handle will never be removed in the thread's lifetime because the semantics of the *remove* primitive requires the original bind handle. Such advising relationships can be safely optimized using advice inlining techniques similar to that used by Steamloom [4, 5].

3. Expressing AO Constructs in Nu

In this section, we describe strategies for compiling static and dynamic AO constructs to the *Nu* IL model. The rationale for this section is to demonstrate that the IL model is flexible enough to support static, dynamic, control flow, and history-based constructs in AO languages. Moreover, it also shows, by giving a translation, that compilation of these constructs generates modular object code, which is an additional benefit of the *Nu* 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. An aspect `World` that implements this simple functionality is shown in Figure 6. The source code equivalent (for ease of presentation) of the *Nu* object code that will be generated for this aspect follows in Figure 7.

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 are generated in a static initializer of the aspect to associate the delegate code to the join 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.

```
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$perSingletonInst;
  static { // Static initializer
    ajc$perSingletonInst = new World();
    /* create new Method and Execution objects */
    Method m = new Method("Hello.main");
    Execution e = new Execution(m);
    logDelegate d = new Delegate(World.class,
        World.class.getMethod("ajc$0",
        new Class[0]));
    bind(e, d);
  }
  //Synthetic method generated for the advice
  public void ajc$0() {
    System.out.println("World");
  }
//Constructor World and helper methods hasAspect
//and aspectOf elided for presentation purposes.
}
```

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

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 instances, the delegate is created. The pattern and delegate instances are then used by the bind primitive to initiate join point interception.

An interesting property of the *Nu* version of the intermediate code for the aspect **class** World and the base **class** Hello (not shown) is that they remain separate in their own object code modules. Also, the object code for the base **class** Hello remains free of the aspect related intermediate code. This shows that *Nu* supports what Bockisch *et al.* have called *struture-preserving compilation* [5]. The intermediate code now mirrors the design, which among other things is important for efficiency of incremental compilers [4, 32].

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. In the case of libraries containing aspects, a synthetic method could be generated and a requirement to call this function at initialization time could be imposed to initialize all aspects in the library.

A strategy similar to AspectJ's load-time weaving can also be used, where an XML file is generated by the compiler containing the details of all aspects in the system. All such aspects are then loaded by a custom class loader.

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.

3.1.3 Opportunities for optimizing static deployment

Note that the generated intermediate code here does not store the bind handle returned by the bind primitive. Therefore, the bind handle is eligible for garbage collection immediately after the bind primitive completes. As discussed previously, if a bind handle can be garbage collected that signifies that the advising relationship is never going to be removed. Therefore, it can be optimized away using techniques proposed by Bockisch *et al.* [4, 5], which have shown to have comparable performance as static-weaving approaches.

Recognizing the opportunity for such optimization allows the *Nu* model to remain flexible in general, but offer comparable performance in cases where only limited power is needed.

3.2 Compiling Control Flow Constructs

Our compilation strategy for the *cflow* and *cflowbelow* constructs is similar to the ideas presented by Hanenberg, Hirschfeld and Unland [10]. We will discuss the cflowbelow case as it is slightly more interesting, pointing out differences from cflow as necessary. Note that in addition to these compilation strategies, optimization strategies proposed by Avgustinov *et al.* [29] can also be applied.

An example usage of this pointcut expression is shown in Figure 8. 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 8. An example usage of the *cflowbelow* construct

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

```
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 = 0;
  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);
    if (ids.empty())
      initialDepth =
        Thread.currentThread().countStackFrames();
    ids.push(handle);
  }
  private void cflow$Remove() {
    remove(ids.pop());
  }
  public void ajc$0() {
    if (initialDepth >=
        Thread.currentThread().countStackFrames())
      return;
    count++;
  }
}
```

Figure 9. The generated code for *cflowbelow*

point, binding the same association relationship multiple 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 9 shows the results of the code generation for the example program in Figure 8. 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.

3.3 Compiling Deployment Constructs

Some aspect languages such as CaesarJ [24] provide declarative constructs for dynamic deployment, such as deploy and undeploy. These constructs are naturally supported by our two primitives. Figure 10 shows a strategy for compiling such constructs.

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

Figure 10. 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.

3.4 Compiling Temporal Constructs

Stolz and Bodden proposed a runtime verification framework, where the static aspect deployment model is utilized to verify properties expressed as linear temporal logic formula over pointcuts [35, 36]. These properties are predicates over a program trace, and have also been called history-based pointcuts. Among others Douence *et al* [9], Bockisch, Mezini and Ostermann [6], Walker and Viggers [38], and Allan *et al.* [8] have argued for aspect language constructs of similar flavor.

An example of such a temporal property is

$$G(call(*Word.set(..)) \rightarrow F(call(*Bit.set(..))))$$

which means that every call to the method `Word.set()` is finally followed by a call to the method `Bit.set()`. This property contains two propositions, call to the method `Word.set()` and call to the method `Bit.set()`.

For checking such a property, Stolz and Bodden [35, 36] create aspects that contain state variables representing the fact that a proposition has been satisfied. For each proposition (pointcut), an advice would be created that would manipulate the state variables in the aspect. The advice and state variables together serve to model the state machine. Figure 11 shows the aspect[1] for our example, based on Stolz and Bodden's example [35, Fig 3.].

A version of the temporal aspect in the *Nu* IL model is shown in Figure 12. First, patterns are created to model pointcuts and delegates to the methods are created. The first pattern and delegate is used for the one-time *bind* on line 11 in Figure 12. The bind handle received from this *bind* is not stored to allow for optimizations.

The effect of the one-time *bind* is that `afterP1` starts intercepting the join points matched by **`call`**`(* Word.set(..))`,

[1] Please note, that the intention here is neither to discuss AspectJ in detail nor to compare the proposed approach with AspectJ. The intention here is to illustrate the potential utility of the *Nu* intermediate language model.

```
aspect Tcheck {
  pointcut p1(): call(* Word.set(..));
  pointcut p2(): call(* Bit.set(..));
  int p1 = 1; int p2 = 2;
  Formula state = Globally(p1, Finally(p2));
  Set<int> propSet = new Set<int>();
  after(): p1() { propSet.add(p1); }
  after(): p2() { propSet.add(p2); }
  after(): p1() || p2() {
    state = state.transition(propSet);
    if (state.equals(Formula.TT)) {
      // report formula as satisfied
    } else if(state.equals(Formula.FF)) {
      // report formula as falsified
    }
    state.clear(); //reset proposition vector
  }
}
```

Figure 11. Temporal property checking aspect based on [35]

```
1 class Tcheck {
2   protected static BindHandle id;
3   protected static Pattern prop2;
4   protected static Delegate d2;
5   static {
6    /* Create a pattern prop1 for call(* Word.set(..))
7       and a delegate d1 for Tcheck.afterP1.
8       Initialize prop2 to call(* Bit.set(..))
9       and delegate d2 to Tcheck.afterP2 */
10     ...
11     bind(prop1, d1);
12  }
13  int p1 = 1; int p2 = 2;
14  Formula state = Globally(p1, Finally(p2));
15  Set<int> propSet = new Set<int>();
16  public void afterP1() { propSet.add(p1);
17    id = bind(prop2, d2); afterP1P2();
18  }
19  public void afterP2() { propSet.add(p2);
20     remove(id); afterP1P2();
21  }
22  public void afterP1P2() {
23    state = state.transition(propSet);
24    if (state.equals(Formula.TT)) {
25      // report formula as satisfied
26    } else if(state.equals(Formula.FF)) {
27      // report formula as falsified
28    }
29    state.clear(); //reset proposition vector
30  }
31 }
```

Figure 12. Nu's version of the Tcheck aspect

which represents the first proposition in the temporal formula. Once the first proposition is true, i.e. the method `afterP1` executes, besides managing the logic as before, a check for the second proposition is inserted into the system. This is achieved by the *bind* on line 17 in Figure 12. When the second proposition is satisfied, the method `afterP2` executes, which besides managing the logic as before, stops the check for the second proposition as it is no longer necessary.

To use Hanenberg *et al.*'s terminology [10], *Nu*'s version of the aspect `Tcheck` affects only the initial set of join points selected by pointcut **`call`**`(* Word.set(..))`. After the advice on line 16 executes, it morphs to include the join points selected by pointcut `call(* Bit.set(..))`.

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 4.2 we describe the dispatch at join points. Section 4.3 describes the implementation specific details for the *bind* and *remove* primitives. Section 6 details our evaluation of the implementation. Section 6.3 describes our delegate invocation technique.

4.1 Our VM Implementation Strategy

Hotspot uses mixed-mode execution for faster performance [1]. The key idea is that there are often no gains achieved by compiling the entire program to produce native code before running it [1, 28]. The compilation efforts are focussed on performance critical methods [28]. The insight is based on Hölzle and Ungar's work on adaptive optimization of Self [16].

There are three modes of bytecode execution: an interpreter, a fast non-optimizing compiler and a slow optimizing compiler. Hotspot uses runtime profiling to identify a set of performance-critical methods in the Java program. For the parts that are performance critical, the adaptive optimizing compiler produces optimized native code.

Previous studies of Java programs, for example by Krintz *et al.* [20], show that up to 57% of the methods loaded by the VM are never executed. These studies, the results on adaptive optimization [16], and the highly-dynamic nature of our intermediate language model led us to our implementation strategy that normally we should dispatch an advice using a method-dispatch table like strategy instead of using bytecode rewriting and/or native code insertion and removal at join point shadows.

4.2 Advice Dispatch in Nu's VM

Our current VM implementation provides an advice 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 advice dispatch.

The stub generation code in Hotspot uses a macro-assembler to generate generic stubs for the entry and exit 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.

We inserted an advice dispatch mechanism in these stubs. Our advice dispatch mechanism performs three checks, implemented as three `mov`, three `cmpl`, and three `jcc` assembly instructions. These assembly instructions are directly 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. Caching is described in detail in Section 5.

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 by the (correct) branch prediction, minimizing the dispatch overhead.

4.3 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 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.

5. 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 reduce the amortized cost of this operation. To that end, we have implemented a two-level caching algorithm for dynamic matching in our advice dispatch mechanism. 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 potentially match that signature. The L1 cache helps avoid calls to the incremental matcher. The L2 cache is inside the matcher and helps avoid duplicate matching. Similar to L1 and L2 caches inside a processor, a L1 hit is the least costly operation, followed by a L2 hit and L2 miss.

Cache Hit/Miss	Overhead of join point dispatch
L1 hit	Cost of equality test (local-bind-counter == global-bind-counter)
L2 hit	Incremental-Match(Join point, List of patterns)
L2 miss	Match(Join point, List of patterns)

Figure 13. Cache hits/misses and their respective costs

Our algorithm for detecting an L1 cache hit/miss is as follows. 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. Global counters are never decremented so that the local caches always know if they are current.

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 [12].

At advice 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.

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. The L2 cache is the incremental matcher and 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, it only performs matching against patterns with newer *bindHandle*s. The incremental match must also check the list of delegates in the L1 cache 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, by setting the local counter in the L1 cache to the global counter's value and storing the last matched *bindHandle* in the L2 cache.

6. Runtime Performance of Nu's VM

To evaluate the runtime performance of our implementation of *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. All measurements were performed on a dual 2.2GHz XEON server with 2GB memory.

The results for the Java Grande method benchmark are shown in Figures 14 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.

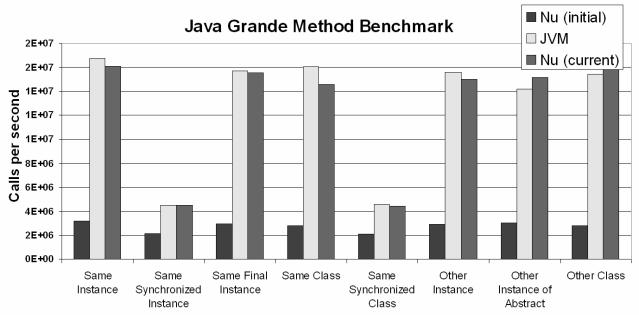

Figure 14. Comparison of Advice Dispatch times with 0 Advice Using the Java Grande Benchmark (larger bars are better)

	JVM	Nu (initial)	% of JVM	Nu (current)	% of JVM
Same Instance	16.765E6	3.194E6	19.05%	16.105E6	96.06%
Same Sync. Instance	4.497E6	2.148E6	47.77%	4.518E6	100.45%
Same Final Instance	15.709E6	2.961E6	18.85%	15.537E6	98.90%
Same Class	16.032E6	2.801E6	17.47%	14.554E6	90.78%
Same Sync. Class	4.613E6	2.108E6	45.71%	4.457E6	96.63%
Other Instance	15.571E6	2.921E6	18.76%	15.055E6	96.68%
Other Abs. Instance	14.240E6	3.002E6	21.08%	15.181E6	106.61%
Other Class	15.449E6	2.817E6	18.24%	15.909E6	102.98%
Average	12.859E6	2.744E6	21.34%	12.664E6	98.48%

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

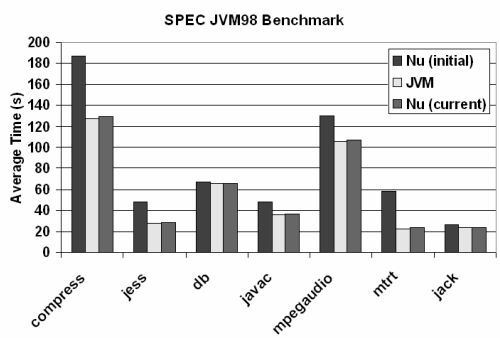

Figure 16. Comparison of Advice Dispatch times with 0 Advice Using the SPEC JVM98 Benchmark (smaller bars are better)

The results for the SPEC JVM98 benchmark are shown in Figures 16 and 17. 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.

6.1 Cache Performance

To measure the penalty for a cache miss, we created a synthetic benchmark. This benchmark determined the baseline performance of calling a method (which has already had its cache initialized). It then creates a number of advising relationships which do not advise the method being measured. We then call the method and measure its performance. This process is then repeated 10,000 times and the results averaged. The results are shown in Figure 18. Note that the measured performance indicates a linear relationship to the number of patterns already bound.

	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 17. Comparison of Join Point Dispatch times Using the SPEC JVM98 Benchmark (smaller is better)

Number of Patterns	0	64	128	192	256
Time (μs)	0.001	1.959	4.030	6.514	8.721

Figure 18. Cache Benchmark Results

6.2 Bind/Remove Performance

To measure the performance of the *bind* and *remove* primitives, we created a synthetic benchmark. This benchmark contains one class with a method that will be targeted by patterns in *bind* calls. The main portion of the benchmark is shown in Figure 19. The benchmark starts with an initial number of pattern/delegate pairs bound. This number was varied from 0 to 2048 and set in `NUM`. It then measures (separately) a *bind* and *remove* and determines the average. This benchmark was run 30 times for each value of `NUM`.

```
for (int i = 0; i < NUM; i++)
  Dispatcher.Bind(pat, del);

for (int j = 0; j < 10000; j++) {
  start = System.nanoTime();
  BindHandle id = Dispatcher.Bind(pat, del);
  count += System.nanoTime() - start;

  start2 = System.nanoTime();
  Dispatcher.Remove(id);
  count2 += System.nanoTime() - start2;
}
```

Figure 19. The bind/remove Benchmark

The results showed that the performance of both primitives was independent of the number of existing advising relationships. The average time taken by the *bind* and *remove* primitives were 11μs and 3.4μs with a variance of approximately 3μs and $1E^{-4}\mu$s, respectively.

6.3 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.

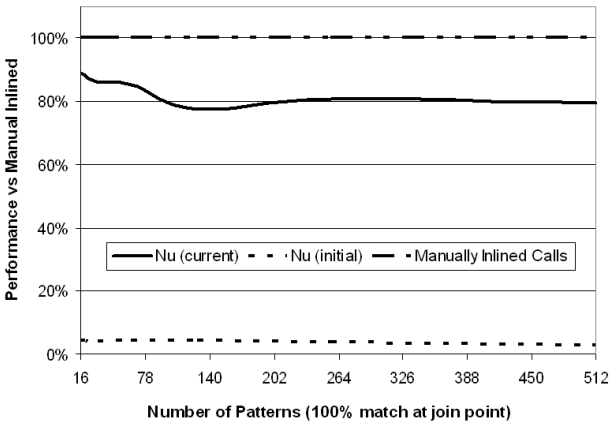

Figure 20. Invoke Benchmark - Varying Number of Patterns

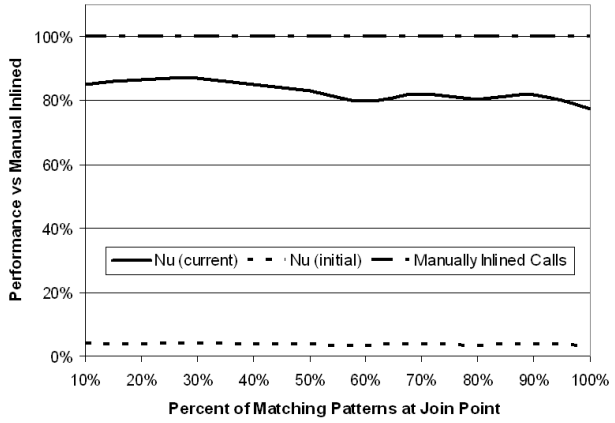

Figure 21. Invoke Benchmark - Varying % Matching Patterns

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. 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 20 varies the total number of *bind* calls while keeping the percent that match the test method at 100%. Figure 21 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.

6.4 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 machines 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.

7. 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.

7.1 Run- and Load-Time Weaving

There are several approaches for run-time weaving such as PROSE [31], Handi-Wrap [3], Eos [34, 33], 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.* [26], 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 [22]. 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 [18] has load-time weaving support. Weave.NET [21] uses a similar approach for the .NET framework. The JMangler framework can also be used for load-time weaving [19]. 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 a larger 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.

7.2 Virtual Machine Support of Aspects

Steamloom [5] 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 [2]. 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 advice.

Steamloom has support for deploying (and undeploying) aspects as a unit. *Nu*'s model allows for a finer-grained level of deployment. Aspects in *Nu* can be deployed in whole, or in part due to the lower-level abstractions provided by the intermediate language primitives. This functionality would need to be simulated in Steamloom using conditional pointcuts or multiple aspects.

Haupt and Schippers propose a delegation-based machine model [11] for AOP support that uses proxy objects and delegation chains to add/remove additional functionality as needed. This model could be considered an implementation of Ossher's proposed machine model based on fragmented objects [27]. Both the delegation-based model and *Nu*'s model aim to be targets for high-level AOP languages, however, the implementation of *Nu* focuses on efficiency and production-level VM support. The delegation-based model is slightly more flexible due to its support of introductions, which is future work for the *Nu* model.

8. Future Work

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

8.1 Language Extensions

Our current implementation does not support *around* constructs in AspectJ-like languages. Masuhara *et al.* have proposed adding two constructs, *proceed* and *skip*, to handle around advice [23]. 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 [27] 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 dispatch. During VM initialization time, this table is initialized with code buffers that contain optimized code for various different types of entry and exit events. In our current implementation, we insert additional instructions into these code buffers. During the execution of a program, an entry and exit is translated to 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 advice dispatch scenarios. For example, if we have not seen any bind instructions yet for a join point kind, there is no need for advice dispatch condition checks. As soon as the VM sees a bind call for a specific join point kind, it checks to see if the entry 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 any list, the entry for that join point kind is replaced with the original entry that does not contain advice dispatch checks. These two modifications should further speed up the join point dispatch by eliminating the need for redundant checks.

We also plan to investigate using existing frameworks inside Hotspot to detect frequently dispatched advice. This advice could then be inlined using either byte code reweaving or natively using Hotspot's JIT compilers. Hotspot's de-optimization framework could possibly be used to remove previously inlined advice.

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 use cases for dynamic aspect deployment have emerged, including ours [3, 5, 30, 31].

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. It is motivated by the need to provide a richer compilation target for dynamic constructs compared to object-oriented intermediate languages. We illustrated that a variety of AO language constructs such as static, dynamic, control flow, and trace-based can be expressed in terms of the *Nu* model. An additional benefit that we observed was that representation of these constructs in *Nu* preserved the 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 negligible performance degradation of method dispatch time compared to the unmodified JVM. The speed of invoking the delegate also remains fairly close to the manually in-lined method call because of our caching mechanisms.

Our work can also be viewed as a proposal to establish an interface and corresponding contracts between the language designs and execution model designs. This interface will decouple language compilers and the virtual machine potentially enabling 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 they conforms to the interface. Moreover, the effect of such optimization techniques is likely to benefit all such language implementations; hence, the perceived benefits of improvements is likely to be greater.

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, Rakesh B. Setty, Giora Slutzki, Eric Van Wyk, Moshe Y. Vardi and the anonymous reviewers for their insightful comments.

References

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

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

[3] J. Baker and W. Hsieh. Runtime aspect weaving through metaprogramming. In *AOSD '02*, pages 86–95.

[4] C. Bockisch, M. Arnold, T. Dinkelaker, and M. Mezini. Adapting virtual machine techniques for seamless aspect support. In *OOPSLA '06*, pages 109–124.

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

[6] C. Bockisch, M. Mezini, and K. Ostermann. Quantifying over dynamic properties of program execution. In *Dynamic Aspects Workshop (DAW05)*.

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

[8] C. Allan *et al*. Adding trace matching with free variables to AspectJ. In *OOPSLA '05*.

[9] R. Douence, P. Fradet, and M. Sudholt. *Trace-based aspects*, pages 141–150.

[10] S. Hanenberg, R. Hirschfeld, and R. Unland. Morphing aspects: incompletely woven aspects and continuous weaving. In *AOSD '04*, pages 46–55.

[11] M. Haupt and H. Schippers. A machine model for aspect-oriented programming. In *ECOOP*, pages 501–524.

[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.

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

[14] R. Hirschfeld. AspectS - aspect-oriented programming with squeak. In *NODe '02: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 216–232.

[15] R. Hirschfeld and S. Hanenberg. Open aspects. *Computer Languages, Systems & Structures*, 32(2-3):87–108, 2006.

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

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

[18] 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.

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

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

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

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

[23] H. Masuhara, Y. Endoh, and A. Yonezawa. A fine-grained join point model for more reusable aspects. In N. Kobayashi, editor, *APLAS*, volume 4279 of *Lecture Notes in Computer Science*, pages 131–147.

[24] M. Mezini and K. Ostermann. Conquering aspects with Caesar. In *AOSD '03*, pages 90–99.

[25] G. Myers. A Four Russians algorithm for regular expression pattern matching. *Journal of the ACM (JACM)*, 39(2):432–448, 1992.

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

[27] 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.

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

[29] Pavel Avgustinov *et al.* Optimising AspectJ. In *PLDI '05*, pages 117–128.

[30] A. Popovici, G. Alonso, and T. Gross. Just-in-time aspects: efficient dynamic weaving for java. In *AOSD '03*.

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

[32] H. Rajan, R. Dyer, Y. Hanna, and H. Narayanappa. Preserving separation of concerns through compilation. In *SPLAT 06*.

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

[34] H. Rajan and K. J. Sullivan. Classpects: unifying aspect- and object-oriented language design. In *ICSE '05*, pages 59–68.

[35] V. Stolz and E. Bodden. Temporal assertions using AspectJ. In *Fifth Workshop on Runtime Verification (RV '05)*.

[36] V. Stolz and E. Bodden. Tracechecks: Defining semantic interfaces with temporal logic. *Software Composition*, pages 147–162, 2006.

[37] D. Suvée, W. Vanderperren, and V. Jonckers. Jasco: an aspect-oriented approach tailored for component based software development. In *AOSD '03*, pages 21–29.

[38] R. J. Walker and K. Viggers. Implementing protocols via declarative event patterns. In *FSE-12*, pages 159–169.