Supporting Dynamic Aspect-oriented Features

Robert Dyer and Hridesh Rajan
Iowa State University

Dynamic aspect-oriented (AO) features have important software engineering benefits such as allowing unanticipated software evolution and maintenance. It is thus important to efficiently support these features in language implementations. Current implementations incur unnecessary design-time and runtime overhead due to the lack of support in underlying intermediate language (IL) models. To address this problem, we present a flexible and dynamic IL model that we call *Nu*. The *Nu* model provides a higher level of abstraction compared to traditional object-oriented ILs, making it easier to efficiently support dynamic AO features. We demonstrate these benefits by providing an industrial strength VM implementation for *Nu*, by showing translation strategies from dynamic source-level constructs to *Nu*, and by analyzing the performance of the resulting IL code.

Nu's VM extends the Sun Hotspot VM interpreter and uses a novel caching mechanism to significantly reduce the amortized costs of join point dispatch. Our evaluation using standard benchmarks shows that the overhead of supporting a dynamic deployment model can be reduced to as little as $\sim$1.5%. *Nu* provides an improved compilation target for dynamic deployment features, which makes it easier to support such features with corresponding software engineering benefits in software evolution and maintenance and in runtime verification.

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

General Terms: Design, Human Factors, Languages

Additional Key Words and Phrases: Nu, invocation, weaving, aspect-oriented intermediate-languages, aspect-oriented virtual machines

1. INTRODUCTION

Software evolution and maintenance is a fact of life [Bennett and Rajlich 2000; Lehman 1998]. Enhancements, security, and bug fixes are routinely made to a software system during its usable life. Long running software systems such as web and application servers, automatic teller machines (ATMs), critical control systems often need to balance evolution and availability requirements. As Malabarba *et al.* state, "for a large class of critical applications, such as business transaction systems, telephone switching systems and emergency response systems, the interruption poses an unacceptable loss of availability [Malabarba et al. 2000]". As an example, consider the maintenance needs faced by European banks

The work described in this article is the revised and extended version of a paper presented at AOSD 2008 in Brussels, Belgium. This research was supported in part by NSF grants CNS-07-09217 and CNS-08-08913.
Authors' address: R. Dyer, rdyer@cs.iastate.edu, Computer Science, Iowa State University, Ames, IA 50010. H. Rajan, hridesh@cs.iastate.edu, Computer Science, Iowa State University, Ames, IA 50010.

while updating ATMs from national currencies to Euro [uwe Mätzel and Schnorf 1997; Kniesel 1999]. The 24-hour service typical for ATMs dictate constant availability, whereas the maintenance needs to convert currencies required immediate software update. Often such maintenance needs are critical and unanticipated [Kniesel 1999].

Dynamic aspect-oriented features have shown the potential to support such unanticipated evolution of software systems [Popovici et al. 2002; Popovici et al. 2003]. These features have received a lot of attention in the past 3-4 years of aspect-oriented programming literature [Allan et al. 2005; Avgustinov et al. 2007; Baker and Hsieh 2002; Bockisch et al. 2004; Bockisch et al. 2006; Chen and Roşu 2007; Hanenberg et al. 2004; Hirschfeld 2003; Hirschfeld and Hanenberg 2006; Martin et al. 2005; Stolz and Bodden 2006; Suvée et al. 2003]. A number of other important use cases for these features 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. Better support for dynamic aspect-oriented features thus has important software engineering benefits.

In this article, we describe the design, implementation and rigorous evaluation of our intermediate language (IL) model *Nu* and corresponding virtual machine implementation, which provides dedicated support for dynamic aspect-oriented features. The following section briefly gives background on and motivates the need for these features.

1.1 Dynamic Aspect-oriented Features

Aspect-oriented programming (AOP) [Kiczales et al. 1997] techniques offer software designers improved methods to separate certain types of concerns in a system. For example, consider a thread pooling concern that returns a new thread from a previously allocated pool of threads. If this concern were added to an existing program, every call to allocate a new thread must be replaced with a call to the thread pool. These calls most likely are scattered throughout the program. Using AOP techniques, this change could be implemented in a modular fashion by using declarative constructs to identify thread creation and to replace those allocations with calls to the thread pool. The thread pooling concern is thus localized into a single module, allowing for easier evolution.

The declarative constructs to identify points where threads are created and replace them with calls to a thread pool are static aspect-oriented (AO) constructs. These constructs can be statically composed with the original code to produce the desired result. Certain constructs have a more dynamic nature to them. For example, these constructs might rely on the dynamic control flow of a program and can't easily be statically composed with the original code, without requiring additional logic.

Other use cases drive the need to support unanticipated changes. Consider for example a long-running web application that suddenly shows performance degredation while allocating threads. Using dynamic AO constructs, one could apply the thread pooling concern previously mentioned to temporarily solve the problem while investigating the underlying issue. Once the underlying issue is resolved, the thread pooling concern could be dynamically removed.

Some proposals for dynamic AO features have investigated support for these constructs by translating them to static constructs [Bockisch et al. 2005; Hanenberg et al. 2004; Stolz and Bodden 2006]. For example, Stolz and Bodden propose translating LTL formulas into AspectJ code [Stolz and Bodden 2006]. This translation generates automata to check if formulas are satisfied, updating state for each proposition at certain points of the program using generated advice. At the points where state may change, the AspectJ code adds

additional logic to update the automata and check the satisfiability of the formulas.

In some cases, a finer-grained deployment model enables simpler implementations e.g. in the case of temporal assertion checking using aspects, advice representing the following propositions need not be checked until the enabling proposition(s) are found true [Bodden and Stolz 2006; Stolz and Bodden 2006]. Such translations demonstrate the need for a more flexible deployment model [Bockisch et al. 2004; Bockisch et al. 2006; Hanenberg et al. 2004]. 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 history-based pointcuts [Bodden and Stolz 2006; Stolz and Bodden 2006] and cflow [Bockisch et al. 2005; Bockisch et al. 2006; Hanenberg et al. 2004].

1.2 Contributions

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 simplifying the support for 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 [Masuhara et al. 2006]. 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 an advising relationship. Our IL 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[1] such as AspectJ's statically deployed *aspects* [Kiczales et al. 2001], CaesarJ's *deploy* [Aracic et al. 2006], control flow constructs and history-based pointcuts [Allan et al. 2005; Stolz and Bodden 2006].

—It provides a higher level of abstraction as a compilation target for dynamic aspect-oriented language constructs.

—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* [Bockisch et al. 2004]. The intermediate code now mirrors the design, which among other things is important for the efficiency of incremental compilers [Bockisch et al. 2006; Rajan et al. 2006] 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

[1]Note that not all static language constructs are supported in the current implementation. Please see Section 6.

aspect-oriented models outside the virtual machine (VM) can be prohibitively expensive [Baker and Hsieh 2002; Popovici et al. 2002]. Following Bockisch et al. [Bockisch et al. 2004], 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 to the interpreter (at this time the Just-in-time compiler is not supported) of the Sun Hotspot Java Virtual Machine (Hotspot JVM) [Paleczny et al. 2001], 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 [Masuhara et al. 2006]; and,

—an analysis of techniques to optimize our highly dynamic deployment model.

In the following section, we describe our intermediate-language design. Our implementation strategy to support the *Nu* IL model in the Hotspot JVM is discussed in Section 3. A novel caching scheme is discussed in Section 4. We evaluate the performance of our VM in Section 5. Section 6 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. Section 7 discusses related work. Section 8 discusses future work and Section 9 concludes.

2. NU: A DYNAMIC INTERMEDIATE LANGUAGE MODEL

The key requirement 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.

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 [Kiczales et al. 2001], 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 [Hilsdale and Hugunin 2004; Masuhara et al. 2003]. 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. [Masuhara et al. 2006]. Masuhara et al. call 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 propose 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 of a method call [Masuhara et al. 2006]. They show that this model is sufficiently expressive to represent common advising scenarios.

```
// Source Code
public class Hello {
  static void main(String[] arguments) {
    System.out.println("Hello");
  }
}
```

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

Fig. 1. Illustration of the AspectJ Join Point Model (for simplicity some join point shadows are omitted)

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 kinds 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
```

Fig. 2. Illustration of the Point-In-Time Join Point Model [Masuhara et al. 2006] (for simplicity some join point shadows are omitted)

At this time, *Nu*'s implementation does not support *around* advice (see Section 8 for more details). Interested readers are referred to Masuhara et al.'s work [Masuhara et al. 2006] for more detail. We have also explicitly decided to not support static crosscutting mechanisms, such as inter-type declarations in AspectJ [Kiczales et al. 2001]. These constructs are largely static and they can be easily supported by high-level language compilers using static weaving techniques [Böllert 1999; Hilsdale and Hugunin 2004].

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 [Kiczales 2007]. 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. A region-in-time model can solve this problem, but would be unnecessarily complicated, whereas a point-in-time model offers a simple solution.

	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`

Fig. 3. Specification of Primitives in *Nu*

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 is 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 {
    // create Method/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
  }
}
```

Fig. 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. Like strings in Java, patterns are *immutable*; their values cannot be changed after they are 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

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 5-7 take a pattern of type 1, 2 or 3 as argument.
Patterns 8-10 take a pattern of type 1 or 2 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

Fig. 5. Patterns Available in *Nu*'s Standard Library

(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`. 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.

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 to enable the logger via 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 2.2.3, 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 6.2).

Future language extensions may allow ordering constructs; however, at this time we believe they are not necessary since compilers generating *Nu* intermediate code could reorder 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 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 *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.

2.2.4 *The remove primitive.* The remove primitive expects a unique, immutable identifier representing the advising relationship on the stack. It destroys the advising relationship corresponding to the identifier. 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.

3. NU VM : PROOF OF FEASIBILITY FOR THE NU IL MODEL

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 3.4 we describe the dispatch at join points. Section 3.3 describes the implementation specific details for the *bind* and *remove* primitives. A novel caching mechanism is described in Section 4. Section 5 details our evaluation of the implementation.

3.1 Our VM Implementation Strategy

Hotspot uses mixed-mode execution for faster performance [Agesen and Detlefs 2000]. The key idea is that there are often no gains achieved by compiling the entire program to produce native code before running it [Agesen and Detlefs 2000; Paleczny et al. 2001]. The compilation efforts are focussed on performance critical methods [Paleczny et al. 2001]. The insight is based on Hölzle and Ungar's work on adaptive optimization of Self [Hölzle and Ungar 1996].

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. [Krintz et al. 2001], show that up to 57% of the methods loaded by the VM are never executed. These studies, the results on adaptive optimization [Hölzle and Ungar 1996], and the highly dynamic nature of our intermediate-language model led us to our implementation strategy. Instead of using bytecode rewriting which would spend time rewriting methods that may never execute, we should dispatch advice using a method-dispatch table when methods actually execute.

3.2 VM Implementation Overview

Figure 6 shows an overview of the components modified to implement the *Nu* virtual machine. The Hotspot interpreter was modified by adding additional assembly code for advice dispatch (JP Dispatcher). The standard Java Runtime Environment (JRE) has additional Java classes added to it for the *Nu* pattern library. The VM has additional C++ code added to handle *bind* and *remove* calls as well as perform pattern matching at join points. The caching mechanism described in Section 4 adds additional data to the methodOop class as well as a global counter. Additionally it adds code into the classloader to initialize the cache. More details about the caching mechanism are in Section 4.

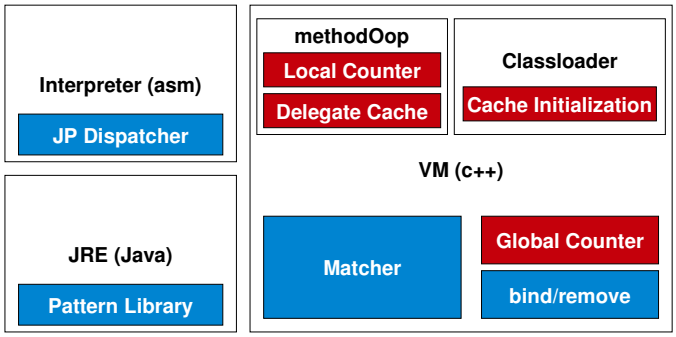


Fig. 6. Overview of *Nu*'s VM Implementation

3.3 Handling Bind/Remove Calls in Nu VM

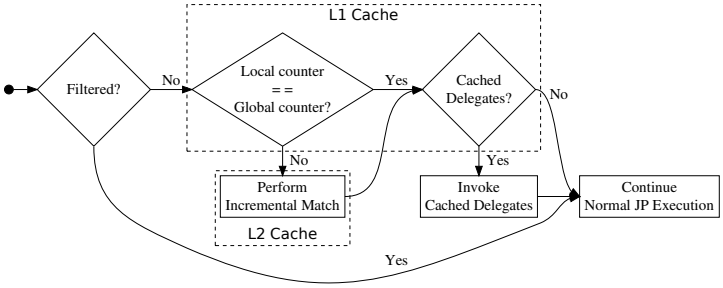
The modified VM handles *bind* calls by storing the pattern and delegate objects into a list. There is one list for each kind of join point and the pattern indicates which join point kind(s) it applies to. It also performs some simple sanity checks (like verifying neither object are **`null`**, if the 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 its local 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.

3.4 Join Point Dispatch in Nu 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.




```
  if (methodOop.counter == -1)              // is this join point filtered?
    goto ContinueJP
  if (methodOop.counter != globalCounter)   // is cache invalid?
    call incrementalMatcher()
  if (cache.head == null)                   // is cache empty?
    goto ContinueJP
  call invokeCachedDelegates()
ContinueJP:
```

Fig. 7. The join point dispatch code

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 an 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, pseudo-code is shown in Figure 7, are directly emitted in the assembly code stubs generated by the VM. The caching mechanism is described in more detail in the following section.

The first check is a filtering check to prevent JRE and *Nu* runtime join point shadows from being advised. Filtered join point shadows use a special value in the cache.

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.

One part not shown in Figure 7 is the exposing of context such as **`this`**, **`target`**, etc to delegates. The signature of the delegate method indicates if such context is needed and *bind* checks for this signature and sets a flag in the *bindHandle*. Before the cached delegates execute, if any delegate needs context exposed the VM generates a **`thisJoinPoint`**

object using information already available on the stack.

4. CACHING TECHNIQUE IN NU 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 (see Figure 7). 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 signature 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 enables incremental matching. Similar to L1 and L2 caches inside a processor, a L1 hit is the least costly operation, followed by a L2 hit (see Figure 8).

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 represent 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 an adaptation of the fast-match technique used by the standard AspectJ compiler (ajc) during compilation [Hilsdale and Hugunin 2004].

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)

Fig. 8. Cache Hits/Misses and Their Respective Costs

Our algorithm for detecting an L1 cache hit/miss is as follows. Each join point shadow (`methodOop`) contains a counter (`methodOop.Counter`) that is initialized to zero, when the class containing the join point shadow is loaded. There is also a global counter (`globalCounter`) for each join point kind (items 5–12, Figure 5) initialized to zero when the VM is initialized. The global counter for a join point kind is incremented on *bind* and *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 valid.

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. Upon exiting, the join point matcher sets the local counter to the current value of the global counter. 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 incremental pattern matcher is called. The incremental matcher is the L2 cache and refers to a simple technique of only matching patterns that have not already been matched against that join point shadow. The join point shadow stores a pointer to 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 (internally, *bindHandle*s are stored in a linked list). The incremental matcher 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 a pointer to the last matched *bindHandle* in the L2 cache.

5. RUNTIME PERFORMANCE OF NU 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 join point dispatch overhead of our VM implementation. We used two standard Java benchmarks for our evaluation: SPEC JVM98 and Java Grande Framework (JGF). 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 JGF method benchmarks with no *bind*/*remove* calls. 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.

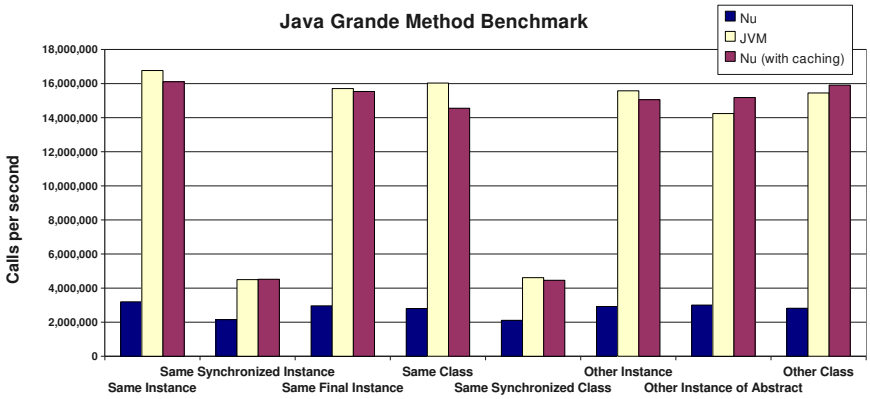


Fig. 9. Comparison of Join Point Dispatch Times using the JGF Benchmark (larger bars are better)

The results for the JGF method benchmarks are shown in Figures 9 and 10. Since the JGF method benchmark repeatedly executes simple methods to obtain the average number of method calls 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

	JVM	Nu (initial)	% of JVM	Nu (current)	% of JVM
Same Instance	$16.77x10^6$	$3.19x10^6$	19.05%	$16.11x10^6$	96.06%
Same Synchronized Instance	$4.50x10^6$	$2.15x10^6$	47.77%	$4.52x10^6$	100.45%
Same Final Instance	$15.71x10^6$	$2.96x10^6$	18.85%	$15.54x10^6$	98.90%
Same Class	$16.03x10^6$	$2.80x10^6$	17.47%	$14.55x10^6$	90.78%
Same Synchronized Class	$4.61x10^6$	$2.11x10^6$	45.71%	$4.46x10^6$	96.63%
Other Instance	$15.57x10^6$	$2.92x10^6$	18.76%	$15.06x10^6$	96.68%
Other Abstract Instance	$14.24x10^6$	$3.00x10^6$	21.08%	$15.18x10^6$	106.61%
Other Class	$15.45x10^6$	$2.82x10^6$	18.24%	$15.91x10^6$	102.98%
Average	$12.86x10^6$	$2.74x10^6$	21.34%	$12.66x10^6$	98.48%

Fig. 10. Comparison of Join Point Dispatch Times Using the JGF Benchmark (larger is better)

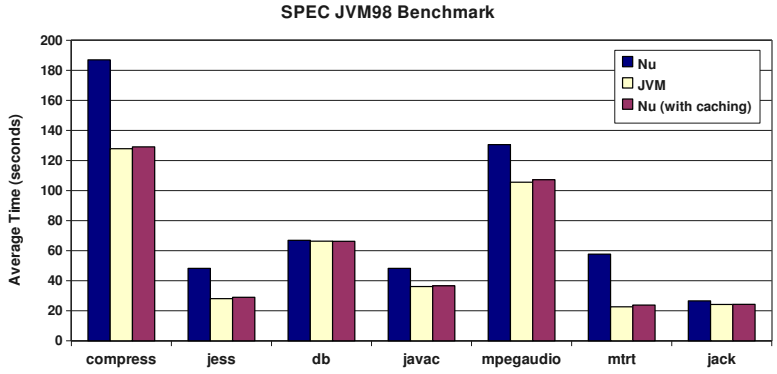


Fig. 11. Comparison of Join Point Dispatch Times using the SPEC JVM98 Benchmark (smaller bars are better)

	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%

Fig. 12. Comparison of Join Point Dispatch Times Using the SPEC JVM98 Benchmark (smaller is better)

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 benchmark are shown in Figures 11 and 12. This benchmark measures the time to execute a set of realistic applications. Similar to the JGF benchmark, our implementation went from a 37% execution time overhead to about 1.5% overhead.

5.1 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 13.

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

Fig. 13. Cache Benchmark Results

Most common AO programs today contain relatively few aspects (and pointcuts) and thus these results show that the performance of our caching mechanism scales well. Note that the results indicate a linear relationship to the number of patterns already bound.

5.2 Bind/Remove Performance

To measure the performance of the *bind* and *remove* primitives, we created another synthetic benchmark. This benchmark contains one class with a method that will be matched by patterns in *bind* calls. 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) *bind* and *remove* calls and determines their averages. This benchmark was run 30 times for each value of `NUM`.

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

		Small	**Medium**	**Large**
Nu	init	61.27	59.82	58.74
Steamloom	init	15.00	16.00	18.00
Nu	deploy	0.60	0.52	0.58
Steamloom	deploy	3.23	28.14	19,126.27
Nu	undeploy	0.026	0.016	0.024
Steamloom	undeploy	1.19	10.55	2781.87

Fig. 14. Deployment Benchmark Results

We measured the deployment and undeployment time of *Nu* and the closest related work, Steamloom [Bockisch et al. 2004]. This measurement was on a synthetic benchmark. Three benchmarks of varying size were used. The aspect (un)deployed for all three benchmarks was a tracing aspect and the number of classes in the system was varied (10, 100 and 1000).

The results are shown in Figure 14 and also include the initialization time for both VMs. *Nu*'s initialization is about 45ms longer than Steamloom's initialization, however both bind and remove outperform Steamloom's deploy/undeploy in all cases. In particular, note the times for *Nu* are almost constant due to the fact *Nu* does not match join points immediately. Steamloom on the other hand will match and weave the pointcut against all join points. In the Large benchmark, this means that Steamloom must weave into 1000 classes and incurs an overhead of almost 20 seconds.

5.3 Delegate Invocation in Nu 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.

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.

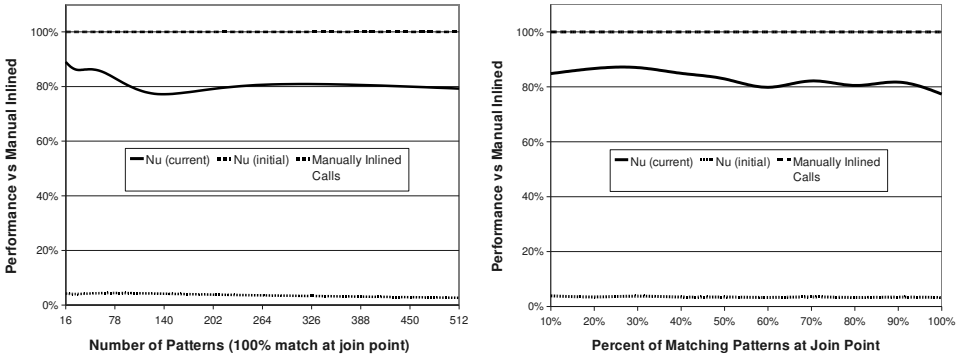


Fig. 15. Invoke Benchmark Results

To measure the performance of our delegate invocation code, we created a benchmark that repeatedly calls a simple test method. 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*. We then measure both copies of the test method (one with manually inlined calls and one with advising relationships to the delegate). 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).

The left of Figure 15 varies the total number of *bind* calls while keeping the percent that match the test method at 100%. The right of Figure 15 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.

5.4 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 JGF 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.

6. THE NU IL MODEL AS A TARGET COMPILATION LANGUAGE

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[2], 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.

6.1 Compiling AspectJ Constructs

In this section, we demonstrate compilation strategies from AspectJ to the *Nu* IL model. 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.

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

Fig. 16. The World Aspect

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. 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 16. The source code equivalent (for ease of presentation) of the *Nu* object code that will be generated for this aspect follows in Figure 17.

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

Fig. 17. Compiling an AspectJ Aspect to *Nu* IL

[2]Note that not all static constructs, such as around advice, are currently supported in *Nu*'s implementation.

6.1.1 *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 17, 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 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* [Bockisch et al. 2004]. The intermediate code now mirrors the design, which among other things is important for the efficiency of incremental compilers [Bockisch et al. 2006; Rajan et al. 2006].

```
public class World {
 static Hashtable<Object,World> ajc$perThisInst = ...;
 static {
  Execution e = new Execution(new Method("Hello.main"));
  /* Delegate to the ajc$0$wrapper method */
  Delegate d = new Delegate(World.class, "ajc$0$wrapper");
  bind(e, d);
 }
 public static void ajc$0$wrapper(JoinPoint thisJp) {
  aspectOf(thisJp.getThis()).ajc$0();
 }
 // remainder same as previous example
 ...
}
```

Fig. 18. Compiling AspectJ's *perthis* Instantiation Model to *Nu* IL

The example in Figure 17 shows a singleton instantiation model for the aspect `World`. Compiling other instantiation models follows a similar structure. For example to compile a *perthis* version of the example aspect, the advice `ajc$0` will now have a wrapper to look up the aspect instance in a table. This compilation technique is similar to the technique proposed by Sakurai et al. for compiling Association Aspects [Sakurai et al. 2004].

6.1.2 *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 `run()`. 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.

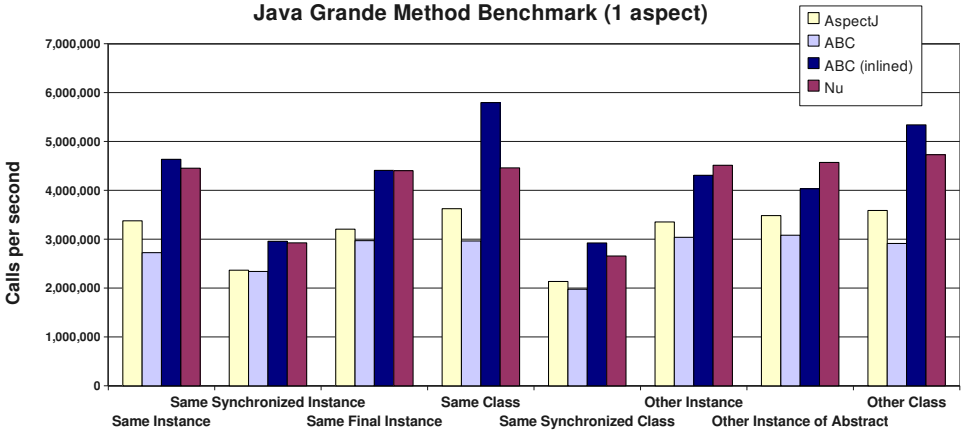


Fig. 19. Performance Comparison of AspectJ Constructs on Java Grande Benchmarks (larger bars are better)

6.1.3 *AspectJ Constructs Performance.* The performance of AspectJ constructs translated to *Nu* IL was measured on the Java Grande method benchmarks. We measured four versions of a simple counting aspect: a version compiled with the standard AspectJ compiler (ajc), two versions compiled with the AspectBench compiler (abc) [Avgustinov et al. 2005] and a version using our compilation strategies to generate *Nu* IL. All versions ran in interpreted only mode, due to the *Nu* VM not currently supporting the Just-in-Time (JIT) compilation process.

ajc usually generates two method calls for an advice invocation: one to fetch an aspect instance and then the actual call to the advice method [Hilsdale and Hugunin 2004]. The idea behind this compilation strategy is that the JIT compiler can inline these calls. Unfortunately since we have to run in interpreted only mode, these inlining optimizations will never be performed by the VM. In order to study the behavior as if the advice had been inlined, we made use of abc, which has an option to enable advice inlining.

Figure 19 shows the results of the benchmarks. The *Nu* IL compiled code is faster than the AspectJ and non-inlined abc compiled code in all cases. The inlined abc version was slower in two cases and faster in the remaining cases. On average, it was only about 5% faster than the *Nu* IL version. This shows that even with the overhead shown in Figure 9, our implementation of the *Nu* IL model performs well for static AO constructs.

So far we have not been very concerned about space overheads of our implementation, primarily because our main objective was to optimize the runtime performance of a highly flexible and dynamic AO system. Despite this, when measuring the maximum memory usage for the base Hotspot VM with no advice (2,195.6 MB), with advice compiled by ajc (2,425.9 MB), abc (2,468.0 MB) and the *Nu* VM (2,682.9 MB) we see only a 10% increase in memory usage compared to the ajc version.

Steamloom [Bockisch et al. 2004] which is based on the Jikes Research VM (RVM), whereas *Nu* is based on Sun's Hotspot JVM. The difference in baseline VMs complicates a direct comparison to this related work. The Jikes RVM does not use an interpreter, instead opting to baseline compile all code and re-compile with an optimizing compiler the hot segments of code. In order to give the reader some sense of the relative overhead of these approaches, we measured the overhead compared to the baseline VMs when introducing an aspect into these systems.

The results of the JGF benchmark on Steamloom and the baseline Jikes RVM are shown in Figure 20. The Jikes RVM benchmark was run with no advice active to give a baseline performance, while the Steamloom benchmark had the same advice active as in Figure 9. With the advice activated, Steamloom ran at 54% of the baseline Jikes RVM. For comparison, compared to the baseline Hotspot VM, AspectJ ran at 24%, ABC with inlining ran at 33%, and Nu ran at 32%.

	Same Inst	**Same Sync**	**Same Final**	**Same Class**	**Sync Class**	**Other Inst**	**Other Abst**	**Other Class**
Jikes RVM	43.50×10^6	3.63×10^6	40.88×10^6	46.19×10^6	3.64×10^6	40.24×10^6	40.71×10^6	43.12×10^6
Steamloom	18.55×10^6	3.31×10^6	17.43×10^6	17.96×10^6	3.27×10^6	17.32×10^6	17.24×10^6	17.76×10^6

Fig. 20. Jikes RVM Compared to Steamloom on Java Grande Benchmarks (With 1 Advice Active)

In addition to Steamloom, we measured the performance of PROSE [Popovici et al. 2002; Popovici et al. 2003] running on the same version of the Sun Hotspot VM as the previous benchmarks. Older versions of PROSE used the debugger interface to expose join points and dynamically register advice [Popovici et al. 2002]. The results for this version of PROSE averaged around 5500 calls/sec and thus would not even show in Figure 19. Newer versions of PROSE modify the VM and use a stub and advice weaver for improved performance [Popovici et al. 2003]. Unforntunately, even with PROSE 1.4.0 there was a bug that prevented the benchmark from completing. The portion of the benchmark that ran, however, showed improvements of around 57% compared to the earlier version of PROSE. Both versions are considerably slower than any of the other approaches measured.

6.2 Compiling Control Flow Constructs

Our compilation strategy for the *cflow* and *cflowbelow* constructs is similar to the ideas presented by Hanenberg, Hirschfeld and Unland [Hanenberg et al. 2004]. 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. can also be applied such as sharing cflow states and caching thread-local state objects [Avgustinov et al. 2005].

Consider an example usage, where an **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 between entry and exit of the method `Word.Set()`.

```
class Counting {
 static int count;
 static Call pat;
 static Delegate advice;
 static ThreadLocal<Stack<BindHandle>>
     stack = ...;
 static ThreadLocal<Integer> depth = ...;
 static {
  pat = new Call(new Method("Bit.Set"));
  /* Delegate to the ajc$0 method */
  advice = ... ;
  /* Delegate to the Enter method */
  Delegate delEnter = ... ;
  /* Delegate to the Exit method */
  Delegate delExit = ... ;

  Method meth = new Method("Word.Set");
  Execution exec = new Execution(meth);
  Return ret = new Return(meth);
  Failure fail = new Failure(meth);
  bind(exec, delEnter);
  bind(ret, delExit);
  bind(fail, delExit);
 }
 void Enter() {
  Stack<BindHandle> cache = stack.get();
  if (cache.Empty()) depth.set(Thread.
      currentThread().countStackFrames());
  cache.push(bind(pat, advice));
 }
 void Exit() {
  remove(stack.get().pop());
 }
 void ajc$0() {
  if (depth.get() >= Thread.currentThread().
      countStackFrames()) return;
  count++;
 }
}
```

Fig. 21. The Generated Code for *cflowbelow*

Our compilation strategy for the cflow and cflowbelow constructs is as follows: first, generate two new methods, say `Enter()` and `Exit()`, 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 `Enter()` and `Exit()` to *bind* and *remove* the code to the actual advice to execute whenever `Bit.Set()` is called. In the terminology of Avgustinov et al. [Avgustinov et al. 2005] the shadows for `Enter()` and `Exit()` are *update shadows* and the residue in the advice is a *query shadow*. The stack `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 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 `depth`. 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 21 shows the results of the code generation for the example program described at the start of this section. 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 (highlighted in grey in Figure 21) is not generated in the case of cflow.

```
class Counting {
 static int count;
 static Call pat;
 static Delegate advice;
 static ThreadLocal<BindHandle> id;
 static ThreadLocal<Integer> depth;
 static ThreadLocal<Integer> counter;
 static {/* Same as previous impl.*/}
 void Enter() {
  Integer cache = counter.get();
  if (cache == 0) {
   id.set(bind(pat, advice));
   depth.set(Thread.currentThread().
    countStackFrames());
  }
  counter.set(cache + 1);
 }
 void Exit() {
  Integer cache = counter.get();
  counter.set(cache - 1);
  if (cache == 0)
   remove(id.get());
 }
 void ajc$0(){/* Same as previous impl.*/}
}
```

Fig. 22. Optimized Code for *cflowbelow*

A slightly more optimized version is shown in Figure 22. In this version, a counter is added to track if we are entering and exiting the intial call to `Word.Set()` instead of binding every time `Word.Set()` is called. While the performance of bind and remove calls was shown in Section 3 to be small, incrementing and decrementing a counter is significantly faster, as shown in the following section.

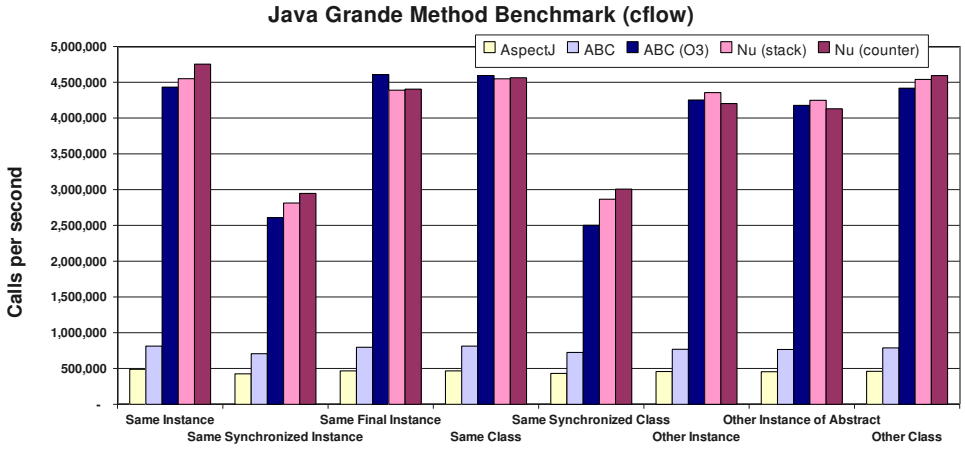


Fig. 23. Performance Comparison of cflow Constructs using JGF Benchmarks (larger bars are better)

6.2.1 *Control Flow Construct Performance.* The performance of control flow constructs was measured on the JGF method benchmarks. Five versions were measured: a version compiled with AspectJ 1.5.3, two versions compiled with abc 1.3.0 at the lowest (O0) and highest (O3 plus inlining) optimization levels and both code generations shown in Figure 21 and Figure 22. Once again, all benchmarks were run in interpreted only mode.

The results are shown in Figure 23 and demonstrate that both of our compilation strategies fair well when compared to AspectJ and abc. In particular note that our approach

performed similar to the version compiled with abc's highest level of optimizations and fully inlined advice. With these optimizations, abc was able to statically determine exactly where the advice executes and inline it. Our approaches performed similar, but did not require any static analysis.

6.3 Compiling Deployment Constructs

Some aspect languages such as CaesarJ [Aracic et al. 2006] provide declarative constructs for dynamic deployment, e.g. deploy and undeploy, which are naturally supported by our primitives. Figure 24 shows a strategy for compiling such constructs.

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

Fig. 24. 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 6.1.2 also apply in this case. This strategy for compiling dynamic deployment constructs also maintains the separation of the aspect modules and base modules.

6.4 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 [Stolz and Bodden 2006; 2006]. These properties are predicates over program traces, and have also been called history-based pointcuts. Among others Douence et al. [Douence et al. 2004], Bockisch, Mezini and Ostermann [Bockisch et al. 2005], Walker and Viggers [Walker and Viggers 2004], and Allan et al. [Allan et al. 2005] 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 [Stolz and Bodden 2006; 2006] 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 manipulates the state variables in the aspect. The advice and state variables together serve to model the state machine. Figure 25 shows the aspect for our example, based on Stolz and Bodden's example [Stolz and Bodden 2006, Fig 3.].

```
aspect Tcheck {
  pointcut p1(): call(* Word.set(..));
  pointcut p2(): call(* Bit.set(..));
  int p1 = 1; int p2 = 2;
  Formula state = Globally(Implies(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
  }
}
```

Fig. 25. Temporal Property Checking Aspect Based on [Stolz and Bodden 2006]

```
1 class Tcheck {
2  static BindHandle id;
3  static Pattern prop2;
4  static Delegate d2;
5  int p1 = 1; int p2 = 2;
6  Formula state = Globally(
7    Implies(p1, Finally(p2)));
8  Set<int> propSet = new Set<int>();
9  static {
10  /* Create a pattern prop1 for
11  call(* Word.set(..)) and a delegate d1
12  for Tcheck.afterP1.
13  Initialize prop2 to call(* Bit.set(..))
14  and delegate d2 to Tcheck.afterP2 */
15  ...
16  bind(prop1, d1);
17 }
18 void afterP1() { propSet.add(p1);
19   id = bind(prop2, d2); afterP1P2();
20 }
21 void afterP2() { propSet.add(p2);
22   remove(id); afterP1P2();
23 }
24 void afterP1P2() {
25  state = state.transition(propSet);
26  if (state.equals(Formula.TT)) {
27   // report formula as satisfied
28  } else if(state.equals(Formula.FF)) {
29   // report formula as falsified
30  }
31  state.clear(); // reset prop vector
32 }
33 }
```

Fig. 26. Nu's Version of the Tcheck Aspect

A version of the temporal aspect in *Nu* IL model is shown in Figure 26. 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 12 in Figure 26. 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(..))`, 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 19 in Figure 26. 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 [Hanenberg et al. 2004], *Nu*'s version of the **`aspect`** `Tcheck` affects only the initial set of join points selected by the the pointcut **`call`**`(* Word.set(..))`. After the advice on line 18 executes, it morphs to include the join points selected by the pointcut `call(* Bit.set(..))`. As Bodden and Stolz pointed out, dynamically (un)deploying portions of the temporal matching infrastructure in this manner can lead to improved runtime performance [Bodden and Stolz 2006].

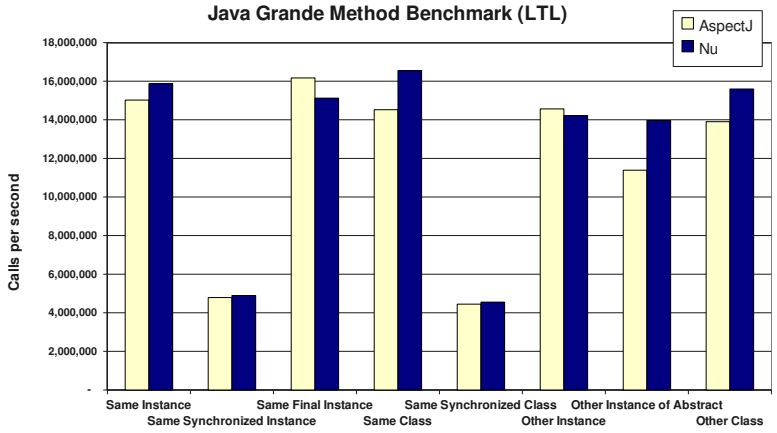


Fig. 27. Performance Comparison of AspectJ and *Nu* Versions of Temporal Constructs (larger bars are better)

6.4.1 *Temporal Construct Performance.* The performance of temporal constructs was measured on the Java Grande method benchmarks. The same framework and formula were used for both the AspectJ and *Nu* versions. Once again, all benchmarks were run in interpreted only mode. The results are shown in Figure 27. Once again, the *Nu* IL version performed similarly to the AspectJ version.

7. RELATED WORK

Three closely related and complimentary research ideas are run-time 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 [Popovici et al. 2002], Handi-Wrap [Baker and Hsieh 2002], Eos [Rajan and Sullivan 2003; 2005], 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 advice 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. [Popovici et al. 2002] and Ortin et al. [Ortin and Cueva 2004], 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 VM [Liang and Bracha 1998]. 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 [Kiczales et al. 2001] has load-time weaving support. Weave.NET [Lafferty and Cahill 2003] uses a similar approach for the .NET framework. The JMangler framework can also be used for load-time weaving [Kniesel et al. 2001]. 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 optimizations for the underlying execution models can be developed independent of the language design as long as it conforms to the interface. Load-time weaving approaches do not provide these benefits.

The bind and remove primitives are similar to install and uninstall messages in AspectS [Hirschfeld 2002]. The difference is that Smalltalk gives reflective access to the method tables, allowing aspects to (un)install advice dynamically, while the Java VM does not have such reflective capabilities and thus need a mechanism such as bind and remove.

7.2 Virtual-Machine Support of Aspects

Steamloom [Bockisch et al. 2004] and *PROSE2* [Popovici et al. 2003] 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 [B. Alpern et al. 2005]. 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 (un)deploying 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.

Haupt and Schippers propose a delegation-based machine model [Haupt and Schippers 2007] 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 [Ossher 2007]. 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.

Golbeck et al. propose lightweight support in virtual machines for AspectJ [Golbeck et al. 2008]. A modified version of the Jikes research virtual machine reads annotations generated by the standard AspectJ compiler (ajc) to provide additional support for the woven aspects in the form of generating more optimized machine code. The virtual machine itself does not perform any advice weaving and thus the language model is quite different from that of *Nu*. Their approach shows potential performance benefits for programs written with AspectJ while our approach tries to be general enough to support multiple high-level languages. Note that both approaches allow execution of AsepctJ code compiled with a standard AspectJ compiler.

8. FUTURE WORK

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

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 call does not return a bind identifier and thus can never be removed. These semantics are useful for static deployment cases and would allow virtual machine implementations to perform optimizations such as code re-writing.

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 [Masuhara et al. 2006]. 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 [Ossher 2007] 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 decrease the dispatch time of our prototype VM. Additional optimizations for improved pattern matching and delegate invocation are also planned. In this section, we will briefly describe these and other 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. 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 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 [Baker and Hsieh 2002; Bockisch et al. 2004; Popovici et al. 2002; Popovici et al. 2003].

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.

Sewe et al. used ordered binary decision diagrams (BDD) for evaluation of dynamic residues [Sewe et al. 2008]. Dynamic residues appear due to partial evaluation of pointcuts performed by static compilers [Masuhara et al. 2003]. Sewe et al. convert those dynamic residues into an ordered BDD, allowing them to evaluate all residues for a specific join point while only evaluating each atomic residue once. Similar techniques might be applicable to our implementation.

In the future, we will look into efficient join point matching mechanisms. One direction is a decision tree-based approach for matching join points against a set of pointcuts[Dyer and Rajan 2008]. 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 utilizing implication relationships and results in a faster matching process.

8.2.3 *Additional Identified Optimizations.* 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 cached for re-use.

Additionally, since patterns are immutable, a virtual machine that implements the *Nu* model needs not worry about a pattern instance changing after creation, which allows for the following optimizations inside the virtual machine.

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 immutable, we eliminate the requirement to maintain the consistency between the pattern and its mirror C++ object.

For patterns that use regular expressions, at the time of their creation a deterministic finite-state automaton can be created and stored in the mirror native object for faster matching [Myers 1992]. By making patterns immutable, we once again eliminate the requirement to maintain the consistency between the regular expression contained inside the pattern and its mirror deterministic finite state automaton contained inside the C++ object. A similar strategy is feasible for bind handles, where 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.

Additionally, the C++ objects for bind handles can be allocated on a separate, small heap ignored by the standard garbage collector. Instead, a specialized and very fast garbage collector can be run more often on this second 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 life-time 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 those used by Steamloom [Bockisch et al. 2006; Bockisch et al. 2004], which have shown to have comparable performance to static-weaving approaches.

This optimization is likely to be helpful for static deployment of aspects. If the generated intermediate code for statically deployed aspects does not store the bind handle returned by the bind primitive, the bind handle is eligible for garbage collection immediately. Recognizing the opportunity for such optimization allows the *Nu* model to remain flexible in general, but offer comparable performance in cases where limited power is needed.

9. CONCLUSION

Dynamic aspect-oriented language features support unanticipated software evolution [Popovici et al. 2002; Popovici et al. 2003] and have been the focus of recent research [Allan et al. 2005; Avgustinov et al. 2007; Baker and Hsieh 2002; Bockisch et al. 2004; Bockisch et al. 2006; Chen and Roşu 2007; Hanenberg et al. 2004; Hirschfeld 2003; Hirschfeld and Hanenberg 2006; Martin et al. 2005; Stolz and Bodden 2006; Suvée et al. 2003]. Important use cases exist for these features in the form of runtime monitoring, runtime adaptation to fix bugs or add new features, runtime update of policy changes, etc. Better support for such dynamic features shows important software engineering benefits.

Existing proposals for these dynamic features have investigated support using static

translations [Bockisch et al. 2005; Hanenberg et al. 2004; Stolz and Bodden 2006]. These translations however incur additional design-time and runtime overhead due to the lack of support for these constructs at the intermediate language level. To solve this problem we propose the *Nu* intermediate language model, which adds two new constructs to existing object-oriented IL models: bind and remove.

Using these two simple constructs, high-level AO language constructs of a dynamic nature can easily be supported. Additionally, the VM implementation for *Nu* shown in this paper incurs a relatively small overhead ($\sim$1.5%) when compared to an unmodified Java VM due to our novel caching mechanism. The overhead associated with deploying and undeploying aspects was also shown to be significantly lower than existing approaches, thus showing the feasibility of supporting our IL model. Our IL model and VM implementation thus gives better support for dynamic AO features and in turn better support for use cases of these features such as runtime monitoring, runtime adaptation, etc.

Acknowledgements

This work is supported in part by the NSF under grants CNS-07-09217 and CNS-08-08913. We thank Prem Devanbu, Youssef Hanna, Mats Heimdahl, Gregor Kiczales, Gary Leavens, Juri Memmert, Harish Narayanappa, Rakesh B. Setty, Giora Slutzki, Eric Van Wyk, and Moshe Y. Vardi, anonymous AOSD 2008 reviewers and TOSEM referees for helpful comments that served to improve the quality of this draft.

REFERENCES

AGESEN, O. AND DETLEFS, D. 2000. Mixed-mode bytecode execution. TR-2000-87, Sun Microsystems, Inc.

ALLAN, C., AVGUSTINOV, P., CHRISTENSEN, A. S., HENDREN, L., KUZINS, S., LHOTÁK, O., DE MOOR, O., SERENI, D., SITTAMPALAM, G., AND TIBBLE, J. 2005. Adding trace matching with free variables to AspectJ. In *OOPSLA '05*. ACM, 345–364.

ARACIC, I., GASIUNAS, V., MEZINI, M., AND OSTERMANN, K. 2006. An overview of CaesarJ. *Transactions on Aspect-Oriented Software Development I*, 135–173.

AVGUSTINOV, P., CHRISTENSEN, A. S., HENDREN, L., KUZINS, S., LHOTAK, J., LHOTAK, O., DE MOOR, O., SERENI, D., SITTAMPALAM, G., AND TIBBLE, J. 2005. abc: an extensible AspectJ compiler. In *AOSD '05*. ACM, 87–98.

AVGUSTINOV, P., CHRISTENSEN, A. S., HENDREN, L. J., KUZINS, S., LHOTÁK, J., LHOTÁK, O., DE MOOR, O., SERENI, D., SITTAMPALAM, G., AND TIBBLE, J. 2005. Optimising AspectJ. In *PLDI '05*. ACM, 117–128.

AVGUSTINOV, P., TIBBLE, J., AND DE MOOR, O. 2007. Making trace monitors feasible. In *OOPSLA '07*. ACM, 589–608.

B. ALPERN ET AL. 2005. The Jikes research virtual machine project: Building an open-source research community. *IBM Systems Journal 44,* 2, 399–417.

BAKER, J. AND HSIEH, W. 2002. Runtime aspect weaving through metaprogramming. In *AOSD '02*. ACM, 86–95.

BENNETT, K. H. AND RAJLICH, V. T. 2000. Software maintenance and evolution: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*. 73–87.

BOCKISCH, C., ARNOLD, M., DINKELAKER, T., AND MEZINI, M. 2006. Adapting virtual machine techniques for seamless aspect support. In *OOPSLA '06*. ACM, 109–124.

BOCKISCH, C., HAUPT, M., MEZINI, M., AND OSTERMANN, K. 2004. Virtual machine support for dynamic join points. In *AOSD '04*. ACM, 83–92.

BOCKISCH, C., KANTHAK, S., HAUPT, M., ARNOLD, M., AND MEZINI, M. 2006. Efficient control flow quantification. In *OOPSLA '06*. ACM, 125–138.

BOCKISCH, C., MEZINI, M., AND OSTERMANN, K. 2005. Quantifying over dynamic properties of program execution. In *Dynamic Aspects Workshop (DAW05)*. 71–75.

BODDEN, E. AND STOLZ, V. 2006. Efficient temporal pointcuts through dynamic advice deployment. In *Workshop on Open Aspect Languages*.

BÖLLERT, K. 1999. On weaving aspects. In *Workshop on Object-Oriented Technology*. Springer, 301–302.

CHEN, F. AND ROŞU, G. 2007. MOP: An efficient and generic runtime verification framework. In *OOPSLA '07*. ACM, 569–588.

DOUENCE, R., FRADET, P., AND SUDHOLT, M. 2004. *Trace-based aspects.* Addison-Wesley, 141–150.

DYER, R. AND RAJAN, H. 2008. A decision tree-based approach to dynamic pointcut evaluation. In *VMIL '08*. ACM.

GOLBECK, R. M., DAVIS, S., NASEER, I., OSTROVSKY, I., AND KICZALES, G. 2008. Lightweight virtual machine support for AspectJ. In *AOSD '08*. ACM, 180–190.

HANENBERG, S., HIRSCHFELD, R., AND UNLAND, R. 2004. Morphing aspects: incompletely woven aspects and continuous weaving. In *AOSD '04*. ACM, 46–55.

HAUPT, M. AND SCHIPPERS, H. 2007. A machine model for aspect-oriented programming. In *ECOOP '07*. Springer, 501–524.

HILSDALE, E. AND HUGUNIN, J. 2004. Advice weaving in AspectJ. In *AOSD '04*. ACM, 26–35.

HIRSCHFELD, R. 2002. Aspect-oriented programming with AspectS. In *Net.Object Days '02*.

HIRSCHFELD, R. 2003. AspectS - aspect-oriented programming with squeak. In *NODe '02*. Springer, 216–232.

HIRSCHFELD, R. AND HANENBERG, S. 2006. Open aspects. *Computer Languages, Systems & Structures 32,* 2-3, 87–108.

HÖLZLE, U. AND UNGAR, D. 1996. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Trans. Program. Lang. Syst. 18,* 4, 355–400.

KICZALES, G. 2007. Personal communication with Hridesh Rajan at AOSD'07.

KICZALES, G., HILSDALE, E., HUGUNIN, J., KERSTEN, M., PALM, J., AND GRISWOLD, W. G. 2001. An overview of AspectJ. In *ECOOP '01*. Springer, 327–353.

KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C. V., LOINGTIER, J.-M., AND IRWIN, J. 1997. Aspect-oriented programming. In *ECOOP '97*. Springer, 220–242.

KNIESEL, G. 1999. Type-safe delegation for run-time component adaptation. In *European Conference on Object-Oriented Programming (ECOOP)*.

KNIESEL, G., COSTANZA, P., AND AUSTERMANN, M. 2001. Jmangler-a framework for load-time transformation of Java class files. In *SCAM '01*. IEEE CS, 100–110.

KRINTZ, C. J., GROVE, D., SARKAR, V., AND CALDER, B. 2001. Reducing the overhead of dynamic compilation. *Software: Practice and Experience 31,* 8, 717–738.

LAFFERTY, D. AND CAHILL, V. 2003. Language-independent aspect-oriented programming. In *OOPSLA '03*. ACM, 1–12.

LEHMAN, M. 1998. Software's future: managing evolution. *Software, IEEE 15,* 1 (Jan/Feb), 40–44.

LIANG, S. AND BRACHA, G. 1998. Dynamic class loading in the Java virtual machine. In *OOPSLA '98*. ACM, 36–44.

MALABARBA, S., PANDEY, R., GRAGG, J., BARR, E., AND BARNES, J. F. 2000. Runtime support for type-safe dynamic java classes. In *ECOOP '00: Proceedings of the 14th European Conference on Object-Oriented Programming*. Springer-Verlag, London, UK, 337–361.

MARTIN, M., LIVSHITS, B., AND LAM, M. 2005. Finding application errors and security flaws using pql: a program query language. In *OOPSLA '05*. ACM, 365–383.

MASUHARA, H., ENDOH, Y., AND YONEZAWA, A. 2006. A fine-grained join point model for more reusable aspects. In *APLAS '06*. Lecture Notes in Computer Science, vol. 4279. Springer, 131–147.

MASUHARA, H., KICZALES, G., AND DUTCHYN, C. 2003. A compilation and optimization model for aspect-oriented programs. In *CC '03*. Springer, 46–60.

MYERS, G. 1992. A four russians algorithm for regular expression pattern matching. *Journal of the ACM 39,* 2, 432–448.

ORTIN, F. AND CUEVA, J. M. 2004. Dynamic adaptation of application aspects. *Journal of Systems and Software 71,* 3, 229–243.

OSSHER, H. 2007. A direction for research on virtual machine support for concern composition. In *VMIL '07*. ACM, 5.

PALECZNY, M., VICK, C., AND CLICK, C. 2001. The Java HotSpot server compiler. In *Java Virtual Machine Research and Technology Symposium*.

POPOVICI, A., ALONSO, G., AND GROSS, T. 2003. Just-in-time aspects: efficient dynamic weaving for Java. In *AOSD '03*. ACM, 100–109.

POPOVICI, A., GROSS, T., AND ALONSO, G. 2002. Dynamic weaving for aspect-oriented programming. In *AOSD '02*. ACM, 141–147.

RAJAN, H., DYER, R., HANNA, Y., AND NARAYANAPPA, H. 2006. Preserving separation of concerns through compilation. In *SPLAT '06*.

RAJAN, H. AND SULLIVAN, K. J. 2003. Eos: instance-level aspects for integrated system design. In *ESEC/FSE-11*. 297–306.

RAJAN, H. AND SULLIVAN, K. J. 2005. Classpects: unifying aspect- and object-oriented language design. In *ICSE '05*. ACM, 59–68.

SAKURAI, K., MASUHARA, H., UBAYASHI, N., MATSUURA, S., AND KOMIYA, S. 2004. Association aspects. In *AOSD '04*. ACM Press, USA, 16–25.

SEWE, A., BOCKISCH, C., AND AND, M. M. 2008. Redundancy-free residual dispatch. In *FOAL '08*. ACM.

STOLZ, V. AND BODDEN, E. 2006. Temporal assertions using AspectJ. *Electronic Notes in Theoretical Computer Science 144,* 4, 109–124.

SUVÉE, D., VANDERPERREN, W., AND JONCKERS, V. 2003. Jasco: an aspect-oriented approach tailored for component based software development. In *AOSD '03*. ACM, 21–29.

UWE MÄTZEL, K. AND SCHNORF, P. 1997. Dynamic component adaptation. 97-6-1, Union Bank of Swizerland.

WALKER, R. J. AND VIGGERS, K. 2004. Implementing protocols via declarative event patterns. In *FSE-12*. ACM, 159–169.