Layered Method Dispatch With INVOKEDYNAMIC

An Implementation Study

Malte Appeltauer Michael Haupt Robert Hirschfeld
Software Architecture Group
Hasso-Plattner-Institut, University of Potsdam, Germany
{firstname.lastname}@hpi.uni-potsdam.de

ABSTRACT

This paper describes an implementation study in which we use the upcoming INVOKEDYNAMIC bytecode instruction—to be supported by the standard Java virtual machine starting with the release of Java 7—to implement layered method dispatch. We compare the resulting implementation approach with that of the existing JCop compiler, and present preliminary results of comparative benchmarks. In spite of the as of now not optimized INVOKEDYAMIC implementation in the Java virtual machine, performance characteristics of the INVOKEDYNAMIC-based JCop implementation are promising.

1. INTRODUCTION

In object-oriented programming, functionality is encapsulated in methods that can be refined through inheritance mechanisms such as overriding. Inheritance supports behavior specification in fixed hierarchies. However, dynamic variations cannot be explicitly represented this way.

Context-oriented Programming (COP) adds an orthogonal dimension to inheritance by addressing the explicit representation of *behavioral variations* and their *dynamic composition*. In COP, behavioral variations are functionality that should be executed in addition or instead of the base functionality. Behavioral variations are encapsulated into layers, that can be de-/activated at runtime.

We denote a method for which at least one layer provides an adaptation, *layered method*. Its behavioral variations are implemented by *partial method definitions*, which are encapsulated by the layer. The base method represents the method definition that is executed when no active layer provides a corresponding partial method.

Implementation strategies of these concepts depend on reflective access and extendability of the host programming language. In host languages with a powerful meta programming API, layer-aware method lookup and layer composition can be implemented by libraries. Host languages with more restricted meta programing support either require a complex and unintuitive implementation, or a full blown programming model extension. In previous work, we did the latter for the Java programming language and developed two extensions with dedicated COP language constructs [4, 1].

JCop [1], our current context-oriented Java language extension is implemented as a JastAdd [5] compiler extension, offering a convenient syntax extension and generating plain Java bytecode. JCop's layer-aware method dispatch performs a thread-local lookup of the appropriate partial method to be called in a composition object. This lookup is implemented by means of plain Java such as thread local objects, hash tables, and inheritance.

The upcoming Java 7 release includes a powerful extension to the language's meta programming facilities: the INVOKEDYNAMIC (henceforth abbreviated "ID") instruction[1] [10]. With ID and the accompanying Java compiler extensions, it is possible to generate method calls that are resolved at run-time by user-provided code. In other words, ID introduces the ability to specify arbitrary method lookup semantics—including changing late-bound methods.

In previous work [2], we conducted several micro-benchmarks to measure possible performance decreases of our implementation of layer-aware method lookup. Results showed a certain execution overhead of layer-aware lookup compared to a naïve Java implementation of the same behavior. Optimization possibilities of the internal Java representation generated by our compiler are restricted. Especially, thread-local access to layer composition is expensive even when caching mechanisms are used. Therefore, we investigated the applicability of Java's new dedicated support for dynamic invocation to our JCop compiler implementation.

This paper presents a working implementation of layered method dispatch using ID as an alternative to the current JCop architecture. The implementation is a hand-made proof-of-concept style feasibility study; the JCop compiler is currently not capable of generating code adopting this scheme. Section 2 introduces Java's ID feature. In Section 3, we describe the JCop implementation of layer-aware method dispatch and present our ID-based implementation. Section 4 discusses our performance observations, points out some remarks to the current development state of ID, and mentions COP languages with related implementations. Section 5 summarizes the paper.

2. INVOKEDYNAMIC

The INVOKEDYNAMIC instruction, defined in Java specification request 292, supports the insertion of late-

[1]`http://jcp.org/en/jsr/detail?id=292`

```
import java.dyn.*;

public class IDTest {
  public static void main(String... args) {
    new IDTest().go();
  }
  public void go() {
    System.out.println("Sending message ...");
    InvokeDynamic.<void>message(this, 23);
    System.out.println("Message sent.");
  }
  public void messageImpl(int k) {
    System.out.println(k);
  }
  public static CallSite bootstrap(
    Class<?> cls, String msg, MethodType t
  ) {
    CallSite cs = new CallSite(cls, msg, t);
    cs.setTarget(
      MethodHandles.lookup().findVirtual(
        IDTest.class, "messageImpl",
        MethodType.methodType(
          void.class, int.class)));
    return cs;
  }
  static {
    Linkage.registerBootstrapMethod("bootstrap");
  }
}
```

Figure 1: A simple INVOKEDYNAMIC example.

bound method calls in Java code that the VM resolves at run-time, using user-provided resolution logic[2]. In Java source code, a method call of the form

```
InvokeDynamic.<T>message(arg1, arg2, arg3)
```

denotes sending a `message`, passing the given arguments, and interpreting the result as having the type `T`. No assumptions about the arguments are made; it is *not* the case that `arg1` is necessarily interpreted as the message receiver.

Each such dynamic call corresponds to precisely one `CallSite` object carrying information about the invocation. Most importantly, a `CallSite` instance stores the particular method to which the site is bound. Since that method is not known prior to run-time, the programmer is required to provide a so-called *bootstrap method* per class containing dynamic invocations. The VM will invoke the bootstrap method as it encounters dynamic calls for the first time during execution.

Bootstrap methods are responsible for creating and returning the respective `CallSite` instances, and they can also bind those `CallSite`s to target methods. While the application is running, it is possible to re-bind call sites by assigning new target methods.

The listing in Figure 1 shows an example of how ID can be used in practice. The example is very simple but illustrates how a concrete method is bound to a dynamic call site.

The `go()` method sends `message` dynamically, with two parameters. When the VM executes this method for the first time, it will encounter an unbound ID call site and invoke the bootstrap method that is registered in the static initializer at the bottom of the listing.

[2]For details on the API that we describe here, we refer to `http://java.sun.com/javase/7/docs/api/java/dyn/package-summary.html`.

The `bootstrap()` method accepts three parameters: the class containing the unbound ID call site, the message sent at this location, and the type of method expected there. In this case, a method returning `void` is expected that accepts an instance of `IDTest` and an `int`.

Each bootstrap method is required to return a `CallSite` instance representing the newly bound ID call site. In the example, the call site is created from the passed parameters and the method `messageImpl()` registered as its target. The effect of this is that any subsequent execution of the ID call site in `go()` will lead to executing `messageImpl()`. The binding is established.

The method `messageImpl()` simply prints the passed `int` to standard output. The first parameter from the ID call, `this`, is implicitly used as the receiver of the virtual method call. Note that this does not have to be the case: had the ID call site been bound to a static method instead, it would have had to accept two parameters as given in the ID instruction in `go()`.

Even though the binding is established once the bootstrap method returns the initialized `CallSite`, this does not mean that it is fixed. It is always possible to send `setTarget()` to the `CallSite` to bind it to a new target method.

3. IMPLEMENTING LAYER-AWARE METHOD LOOKUP

In this section, we sketch JCop's method dispatch, describe an ID-based implementation, and discuss the conceptual differences in both approaches. We show both implementations along the classic COP example of a class `Person` that provides a layered method `toString` for which a partial method is defined in the layer `Address` [8]. The JCop implementation of this example is shown in Figure 2, where the address layer is implemented as a member of `Person`[3].

3.1 JCop Mapping

The JCop compiler takes JCop source files containing class and layer declarations and generates Java byte code. Some language constructs, such as top-level layer declarations and layered fields, are first transformed into an intermediate aspect representation that is then woven into the byte code classes, see Figure 3. In the following, we only refer to core COP features that have been developed in our previous work on ContextJ [3]. This features include layer and partial method definitions as class members and control-flow specific layer composition blocks. Since our experimental ID-based implementation only supports this core features, we omit implementation descriptions of extended features such as declarative composition representation, layered fields and top-level layers.

During compilation, a JCop program's AST (abstract syntax tree) is transformed into a plain Java AST that contains additional auxiliary classes and methods. For illustration, Figure 2 shows a source code representation of the generated byte code for our `Person` example. For brevity, the following listings do not present the `Composition` methods `getCurrent()` and `setCurrent(Composition)` that provide access to the thread local composition object, `first()` and `next()` that allow for its traversal, and `addLayer(Layer)` that adds a layer to a composition. For the implementa-

[3]JCop supports layer definition within classes (*layer-in-class*) and as toplevel module (*class-in-layer*) [1].

```
public class Person {
  private String name, address;

  public Person(String name, String addr) {
    this.name = name;
    this.address = addr;
  }

  public String getAddress() {
    return this.address;
  }

  public String toString() {
    return name;
  }

  layer Address {
    public String toString() {
      return proceed() + ", " + getAddress();
  }
}

public class App {
  void m() {
    Person p = new Person("Bob");
    with(Address) {
      System.out.println(p.toString());
    }
  }
}
```

Figure 2: Person example in JCop.

tion of layer lookup, we use inheritance: Layers are transformed into classes. Each application-specific layer is a subtype of `ConcreteLayer`, which in turn inherits from `Layer` (Lines 21–40). They provide *delegation methods* for their partial methods; the actual behavior of partial methods are implemented by *layer methods* which are defined in their corresponding class and called by their delegation methods (Lines 10–14).

At run-time, a composition object holds references to activated layers. It is accessed upon layered method execution to decide to which of the method's variations execution should be delegated (Lines 5–7). The lookup algorithm is implemented as follows:

- If no layer is active, a composition only consists of one `Layer` element denoting the base layer. For each layered method `m` declared in class `C`, `Layer` provides a delegation method that simply calls `m`'s base definition (which is declared in its corresponding class) (Lines 22–25).

- If the first layer in the composition provides a partial method for `m`, it contains a delegation method that calls the partial method representation in `C` (Lines 36–39).

- If the current layer does not provide a partial method, the composition list must be traversed. Since the current layer does not override `m`'s delegation method, the definition of the super class `ConcreteLayer` is executed, which simply delegates the call to the next layer of the composition (Lines 29–32).

Layer activation is implemented by two static methods that replace the `with`/`without` blocks and allow to add and remove items from the list (Lines 45–52).

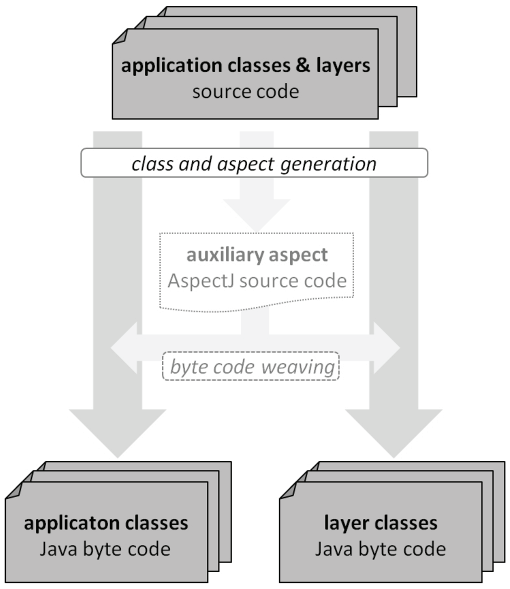

Figure 3: JCop compilation process.

3.2 INVOKEDYNAMIC Implementation

In the JCop implementation, every execution of a layered method performs the aforementioned layer-aware method dispatch. Thus, behavioral variations are adapted as late as possible. This design decision allows for the definition of flexible, "late bound" lookup mechanisms such as event-based layer activation [1].

However, this strategy might cause performance penalties if methods are frequently executed with the same composition chain. Furthermore, this late-bound lookup is not required to support basic COP features. Alternatively, the lookup table can be manipulated on composition change, for which we can employ ID. We developed an ID-based version of layer lookup for Java for which we adapted our JCop implementation. Figures 5 and 6 present the source code of our running example using ID. We modified JCop lookup logic as follows:

- Layered methods delegate to an ID object, instead of performing a composition lookup (Fig. 5, Line 5). On first execution, a bootstrap method creates a `CallSite` object that is stored in a thread-local hash map in `Composition`.

- Layers provide a `MethodHandle` list for their partial methods instead of delegation methods (Fig. 5, Line 28). This list can be collected and generated at compile time using static analysis.

- Composition statements (`with`/`without`) trigger recomposition in terms of updating call site target objects (Fig. 5, Lines 35, 37 and Fig. 6, Lines 5–6). First, all partial method call sites defined by layers included in the new composition are collected. The required call sites are gathered using the hash table of `Composition`, where the `MethodHandle` objects of the layers are used as keys (Fig. 6, Lines 13–14). Finally, a new `MethodHandle` pointing to the partial method definition (methods with suffix '_<*layer name*>') is set as

```java
public class Person {
  ...
  // original method, triggers delegation
  public String toString() {
    Composition c = Composition.getCurrent();
    Layer first = c.first();
    return first.Person$$toString(this, c);
  }
  // partial method for layer Address
  public String toString(Address l,
                         Composition c) {
    return c.next(l).Person$$toString(this, c) +
           ", " + getAddress();
  }
  // base method containing the original code
  public String toString$$base() {
    return getName();
  }
}

public class Layer {
  public String Person$$toString(Person tar,
                                 Composition c) {
      return tar.toString$$base();
  }
}

public class ConceteLayer extends Layer {
  public String Person$$toString(Person tar,
                                 Composition c) {
    return c.next(this).Person$$toString(tar, c);
  }
}

public class Address extends ConcreteLayer {
  public String Person$$toString(Person tar,
                                 Composition c) {
    return tar.toString(this, c);
  }
}

public class App {
  void m() {
    Person p = new Person("Bob");
    try {
      Composition c = Composition.getCurrent();
      Composition old = c.addLayer(Address);
      System.out.println(p.toString());
    }
    finally {
      Composition.setCurrent(old);
    }
  }
}
```

Figure 4: Internal representation of layer-aware method dispatch in JCop.

target for the call sites.

- The `proceed` pseudo method is also implemented by ID calls and points to the next partial method of the composition (Fig. 5, Lines 9–11). Their target objects are redirected by calling the methods `with` and `without`.

The complete ID-based COP implementation consists of some more classes that are not shown here for brevity. The application-specific implementation presented in Figure 5 contains boilerplate code, such as defining layer classes and method handles for their partial methods, dynamic invocation statements and explicit layer (de)activation statements that developers should not have to care about. However, such code could be eventually be generated by a JCop compiler.

```java
 1 public class Person {
 2  ...
 3  // original method, triggers delegation
 4  public String toString() {
 5    InvokeDynamic.<void>Person$$toString(this);
 6  }
 7  // patial method for layer Address
 8  public String toString$$Address() {
 9   InvokeDynamic.
10    <void>#"proceed:Person$$toString:Address"(this)
11    + ", " + getAddress();
12  }
13  // base method containing the original code
14  public String toString$$base(){
15   return getName();
16  }
17 }
18
19 public class Layer extends Layer {
20  ...
21 }
22
23 public class ConcreteLayer extends Layer {
24  ...
25 }
26
27 public class Address extends ConcreteLayer {
28  public MethodHandle[] getPartialMethods() {...}
29  public String getName() {...};
30 }
31
32 public class App {
33   void m() {
34     Person p = new Person("Bob");
35     with(Address);
36     System.out.println(p.toString());
37     without(Address);
38   }
39 }
```

Figure 5: Representation of layer-aware method dispatch using INVOKEDYNAMIC.

4. DISCUSSION

This section briefly presents results of a simple performance assessment of the ID implementation and comments on performance improvements to be expected in the future. Moreover, it discusses related work.

4.1 Performance Observations

Our main motivation to develop an ID-based version of COP are performance issues in previous implementations, as discussed along micro-benchmarks in previous work [2]. Direct VM support of dynamic invocation seems promising. We have applied a subset of the same micro-measurements to our COP/ID implementation and briefly wrap up the results in the following.

In this benchmark, we measure layered method invocation throughput per millisecond. Layered methods are called in a loop running for ten seconds. During that time, the layer composition is fixed, i. e., call site targets are not changed, which would cause expensive JIT compilation. Thus, this benchmark only measures the pure execution time of an ID call. For more details on this benchmark, see [2].

We ran the benchmark on an Intel Core 2 Duo machine with 3 GB RAM running Ubuntu Linux system (kernel 2.6.31-20 with SMP). The JVM in use was Java 7 build 17.0-b09. The measured throughputs of the JCop/ID and plain Java-based JCop implementations are roughly on par.

```
public class Composition {
 ...
 public static void with(Layer layer) {
  Composition.getCurrent().addLayer(layer);
  updateTarget(layer);
  updateProceedChain(next(layer));
 }

 private static Hashtable<String, CallSite> cSites
  = new Hashtable<String, CallSite>();

 public static void updateTarget(Layer l) {
  for(MethodHandle handle : l.getHandles()) {
   CallSite cs = cSites.get(handle.toString());
   if(cs != null) {
    MethodType handleType = handle.type();
    MethodType handleRtype.returnType();
    MethodType rType =
        MethodType.methodType(handleRtype);
    String mName = cs.name() + "_" + name;
    MethodHandle target = MethodHandles.lookup().
       findVirtual(cs.callerClass(), mName, rType);
    cs.setTarget(target);
   }
  }
 }

 private static updateProceedChain(Layer layer){
   // implementation similar to updateTarget();
 }

 //bootstrap method
 private static CallSite createCallSite(Class c,
                  String name, MethodType type) {
  CallSite cs = new CallSite(c, name, type);
  MethodType param = type.dropParameterTypes(0,1);
  MethodHandle h = MethodHandles.lookup().
      findVirtual(c, name, param);
  callSites.put(h.toString(), cs);
  Layer l = Composition.getCurrent().first();
  updateTarget(l.getName(), h);
  return cs;
 }

 public static void register(Class c) {
  MethodType bt = MethodType.
      methodType(CallSite.class, Class.class,
          String.class, MethodType.class);
  MethodHandle bm = MethodHandles.lookup().
      findStatic(Composition.class,
          "createCallSite", bt));
  Linkage.registerBootstrapMethod(c, bm);
 }
}
```

Figure 6: `Composition` provides access to layer composition and updates call sites of layered methods.

This first simple measurement was followed by a more elaborate benchmark for dynamic recomposition. The activation of a layer requires an update of a set or list of active layers. To measure the costs of this update, we compared the execution time of five methods (`method_1`–`method_5`), where `method_i` contains the incrementation of a class variable `counter_i`. For each method, five layers (`Layer1`–`Layer5`) provide a partial method with the same body. Thus, the execution of a method results in the same behavior, independently from the layer composition. We ran `method_1`–`method_5` without active layers, which is the reference value for this benchmark, and with the successive activation of one to five layers.

Results from the second measurement exhibit a clear advantage of the ID-based over the plain JCop implementation. In the case where no layer was active, the ID-based approach had a throughput some 160 times larger than JCop, while being 1.75 times slower than a pure Java method. For activated layers, we observed a performance advantage of roughly 48 to 38 times for 1–5 layers.

We consider these very promising results, as the ID implementation is still under development at the time of this writing and optimizations are pending [9]. Given that a not-yet-optimized ID implementation performs at least as well as an approach utilizing large amounts of infrastructure (hash tables etc.), we expect that performance will increase even more once the ID implementation is complete.

4.2 Related Work

Several other COP languages also implement a layer-based recomposition that taking place at *composition time*, where method references are re-linked at any occurrence of *with* and *without* [2]. In these cases, no additional lookup is required on method execution, since methods already point to the desired variation. This approach is well suited for domains in which partial methods are frequently executed but layer composition (triggering method re-linking) is relatively stable.

COP extensions related to our ID approach make use of their host languages' support for dynamic delegation, such as *ContextS* [7] and *cj/delMDSOC* [11], much like ID does for Java. Despite the conceptual similarities, the concrete implementations differ from our approach due to different language abstractions. ContextS is based on the dynamic aspect language *AspectS* [6], which manages dynamic recomposition. CJ's implementation dynamically modifies delegation chains between object fragments to organize method bindings.

5. SUMMARY

We have described a possible application of the upcoming ID instruction to the implementation of context-oriented programming languages. While the evaluation results are at first might not enthuse, it needs to be noted that the mechanism in question is likely to be optimized prior to the final release of Java 7.

Our ongoing and future work in this field is concerned with providing compiler support for the ID-based JCop implementation, and with improving the way this new instruction is used. In the long run, dedicated support for late-bound message dispatch will be a cornerstone of language hosting on the JVM.

Acknowledgements

We are grateful to John Rose for his insightful comments on the INVOKEDYNAMIC implementation.

6. REFERENCES

[1] Malte Appeltauer, Robert Hirscheld, Hidehiko Masuhara, Michael Haupt, and Kazunori Kawauchi. Event-based Software Composition. In *Proceedings of International Conference on Software Composition 2010*, Lecture Notes in Computer Science. Springer-Verlag, June 2010. to be published.

[2] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, Jens Lincke, and Michael Perscheid. A Comparison of

Context-oriented Programming Languages. In *COP '09: International Workshop on Context-Oriented Programming*, pages 1–6, New York, NY, USA, 2009. ACM Press.

[3] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, and Hidehiko Masuhara. ContextJ - Context-oriented Programming for Java. 2009. submitted.

[4] Malte Appeltauer, Robert Hirschfeld, and Hidehiko Masuhara. Improving the Development of Context-dependent Java Applications with ContextJ. In *COP '09: International Workshop on Context-Oriented Programming*, pages 1–5, New York, NY, USA, 2009. ACM Press.

[5] Görel Hedin and Eva Magnusson. JastAdd: An Aspect-oriented Compiler Construction System. *Science of Computer Programming*, 47(1):37–58, 2003.

[6] Robert Hirschfeld. AspectS - Aspect-Oriented Programming with Squeak. In *NODe '02: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 216–232, London, UK, 2003. Springer-Verlag.

[7] Robert Hirschfeld, Pascal Costanza, and Michael Haupt. An Introduction to Context-Oriented Programming with ContextS. In *Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007, Braga, Portugal, July 2-7. 2007, Revised Papers*, volume 5235 of *Lecture Notes in Computer Science*, pages 396–407, Berlin, Heidelberg, Germany, 2008. Springer-Verlag.

[8] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented Programming. *Journal of Object Technology*, 7(3):125–151, March-April 2008.

[9] John R. Rose. Personal communication, April 17, 2010.

[10] John R. Rose. Bytecodes meet combinators: invokedynamic on the jvm. In *VMIL '09: Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages*, pages 1–11, New York, NY, USA, 2009. ACM.

[11] Hans Schippers, Michael Haupt, Robert Hirschfeld, and Dirk Janssens. An Implementation Substrate for Languages Composing Modularized Crosscutting Concerns. In *Proc. SAC PSC*. ACM Press, 2009.