# Runtime Aspect Weaving Through Metaprogramming

Jason Baker Wilson Hsieh

University of Utah, School of Computing
50 South Central Campus Drive, Room 3190
Salt Lake City, Utah 84112–9205

{jbaker,wilson}@cs.utah.edu

## ABSTRACT

We describe an extension to the Java language, Handi-Wrap, that supports weaving aspects into code at runtime. Aspects in Handi-Wrap take the form of method wrappers, which allow aspect code to be inserted around method bodies like advice in AspectJ. Handi-Wrap offers several advantages over static aspect languages such as AspectJ. First, aspects can be woven into binary libraries. Second, a wrapper in Handi-Wrap is a first-class Java value, which allows users to exploit Java mechanisms to define and weave wrappers. For example, wrappers can be passed explicit constructor arguments, and wrapper objects can be composed. Finally, methods in all Java classes, including anonymous classes, can be wrapped. A prototype of Handi-Wrap is implemented in a compile-time metaprogramming system for Java, called Maya; we briefly describe how Maya's features support Handi-Wrap.

## 1. INTRODUCTION

In aspect-oriented programming, weaving advice into a method amounts to redefining the method as the composition of its body with advice. However, current aspect languages do not allow advice itself to be defined compositionally. For example, in AspectJ [26] aspect definitions may only be reused through inheritance. We present an extension to Java, called Handi-Wrap, that allows all of Java's code reuse techniques, including object composition and parameterized constructors, to be used with method advice.

Object composition is a particularly important reuse technique in languages that use a fixed single-inheritance hierarchy, such as Java. Unlike languages that support mixins [15, 24], Java does not allow classes to be built from several class and mixin definitions. Instead, a single class must be specialized with new functionality. Object composition allows classes to be reused by linking instances together at runtime, and is valuable for building aspects as well as classes.

In Handi-Wrap, method wrappers are first-class values. Wrappers can be explicitly constructed with `new`, and like classes, wrappers can be defined within class declarations or method bodies. Weaving is accomplished through `wrap` statements. Handi-Wrap integrates method advice into the component implementation language more tightly than languages such as AspectJ. Although this integration allows aspect code to be cleanly separated from other code, it does not require separation. In fact, Handi-Wrap's tight integration with Java allows more kinds of weaving to be expressed. Wrappers can be woven into classes with lexically scoped names since weaving statements can appear in any scope. Wrappers can also be woven into methods of anonymous classes through a new declaration modifier.

Dynamic aspect weaving provides other benifits. First, wrappers may be passed explicit constructor parameters and take implicit parameters from their lexical environment. Second, a method need not be recompiled when a wrapper is woven into it. Allowing weaving on binaries eases separate development. In addition, it shortens the compile/edit/debug cycle when development aspects such as tracing and contract enforcement are used. Finally, development aspects can be woven into a running program, using an interpreter such as BeanShell [21] or Kawa [6].

Normally, Handi-Wrap requires that wrappers have exactly the same signatures as the methods that they wrap. Since an aspect system that restricts reuse of advice to methods with identical signatures would not be especially useful, Handi-Wrap supports increased polymorphism by defering some type checking to runtime. Handi-Wrap allows wrappers to be defined on arbitrary argument and return types. Additionally, wrappers may be polymorphic over the number of arguments they take.

Handi-Wrap is implemented in Maya as an extension to the Maya programming language [2, 3]. Maya is a compile-time metaprogramming system for Java that allows users to define language extensions. Maya allows us to extend Java's syntax and enforce extended type checking rules, and to integrate aspect oriented features more tightly into the component language than previous systems.

In summary, Handi-Wrap makes several contributions to aspect-oriented language design:

- Wrapper code may be reused through both inheritance and object composition.
- Wrappers can take explicit constructor parameters, and inner wrappers can take implicit parameters from their lexical environment.
- Methods in anonymous classes can be wrapped.


*AOSD* 2002, Enschede, The Netherlands

- Aspect and component code may be compiled separately.

The rest of this paper is organized as follows. Section 2 introduces Handi-Wrap features through a series of examples. Section 3 provides an overview of Handi-Wrap's current implementation. Section 4 evaluates the cost of Handi-Wrap's flexibility. Section 5 describes related work. Finally, Section 6 summaries our conclusions.

## 2. DYNAMIC METHOD WRAPPING

This section introduces Handi-Wrap's syntax and semantics through several examples. These examples show how Handi-Wrap can be used to guide optimizations such as caching, to implement caching, to track down bugs, and to weave wrappers into methods of local classes. These examples also demonstrate how object composition, explicit constructors, and separate compilation increase the expressive power of wrappers.

Handi-Wrap is defined in terms of two core classes: `Procedure` and `Wrapper`. Procedure objects encapsulate method and composable wrapper bodies. The class `Wrapper` is an abstract factory that composes aspect code with wrappee procedures. The abstract method `Wrapper.generate(Method thisMethod, Procedure wrapped)` produces a wrapped definition of `thisMethod` from a `Method` object that contains symbolic information and a callable `Procedure` to be wrapped. The weaver calls `generate()` to compute a woven method definition. In most cases, the programmer does not need to know the details of `Wrapper.generate()`, but at times it is useful to define `generate()` explicitly.

Macros that implement the wrapper interface also implement simple static checks. To safely wrap a method taking an argument of type T, the wrapper must accept arguments of type T or a supertype. To safely pass the argument on to the target method, the wrapper must take an argument of type T or a subtype. Therefore, argument types must be invariant. Return types must be invariant by similar reasoning.

Polymorphic wrappers may be defined using special syntax. A wrapper that takes a parameter of type `Any` may be woven into a method that takes any reference or primitive type at the corresponding argument position. Similarly, a wrapper that returns `Any` may be woven into methods returning any type, and void methods. Primitive values bound to `Any` variables are boxed, and for the purpose of local type checking, `Any` variables are treated as `Object`. More flexibility could be allowed in a language that supports bounded polymorphism such as GJ [7]. Using GJ, one could achieve safety through polymorphic type variables, and specificity by declaring upper bounds.

Handi-Wrap also defines a new declaration modifier, `rest`. A wrapper's last formal parameter may be a `rest Object[]`; this parameter will be bound to an array of the rest of the actual arguments. The `apply` operator treats rest variables as lists of arguments. Argument lists may also be stored in fields and local variables declared with the `rest` modifier.

### 2.1 Evaluating Cacheable Methods

Caching is an example of the kinds of concerns that Handi-Wrap can express cleanly. The result of any side-effect-free method may be cached, but caching is not always beneficial. Advice can be used to find out where caches are needed, and to implement caching. Separating the memoization of a value from its computation yields concrete benefits. For instance, caches may be woven into both a high-level method and a method it calls. Aspect-oriented programming allows one to easily reevaluate the utility of a high-level cache after a lower-level cache has been added.

One simple caching metric is to count the total number of method invocations, and the number of invocations with distinct arguments. These counts can be obtained by tracing the method, and filtering duplicates from the trace.

Suppose we wish to trace a static method called `ClassPath.lookup(String)` that returns a `ClassPath.Resource`. We can define a wrapper for lookup as follows:

```
ClassPath.Resource wrapper
lookupTracer(String name) {
  System.out.println("lookup("+name+") called");
  return wrapped.apply(name);
}
```

The wrapper declaration above does not mention `ClassPath.lookup` and performs no weaving. The wrapper can be applied to any static method that takes a `String` and returns a `ClassPath.Resource`. The wrapper body is free to examine its arguments, call the wrapped method, and return a value. The wrapper object is bound to a variable called `lookupTracer`.

We use the `wrap` statement to weave `lookupTracer` into `ClassPath.lookup()`,

```
wrap ClassPath.lookup(String) with lookupTracer;
```

The first argument to `wrap` is a method signature, and the second is a wrapper expression. After this statement is executed, calls to `ClassPath.lookup()` are redirected to `lookupTracer`. In this case, we could avoid declaring the wrapper variable `lookupTracer`, and use an anonymous wrapper expression:

```
wrap ClassPath.lookup(String)
with new ClassPath.Resource wrapper(String name) {
     /* body of lookupTracer */
};
```

In addition to the base class `Wrapper`, which is a fundamental part of the Handi-Wrap language, Handi-Wrap provides a library of utility wrappers in `maya.wrap.Wrappers`. One such wrapper implements generic call tracing using `Any` and `rest` arguments:

```
public static final Any wrapper
tracer(rest Object[] args) {
  System.err.println("entering " +
                     fmtMeth(thisMethod, args));
  Any ret = wrapped.apply(args);
  System.err.println("leaving  " +
                     fmtMeth(thisMethod, args) +
                     " => " + ret);
  return ret;
}
```

The implicit parameter `thisMethod` is an object that represents the method being wrapped. When `tracer` is woven into an instance method, `args[0]` contains the receiver, since a method receiver is always counted as a wrapper argument. The static method `Wrappers.fmtMeth()` uses `thisMethod`'s access modifiers to decide how to format the

| | |
|---|---|
| **crashTracer** | Dumps method invocation information when an exception is thrown. |
| **trapper** | A development wrapper that unconditionally throws a runtime exception. |
| **stackDumper** | Prints a stack trace and calls through the wrapped method. |
| **CacheNth** | A wrapper subclass that memoizes the wrapped method's results based using the $n^{th}$ argument as a key. `CacheNth` wrappers are of little value alone, but are useful when composed with other wrappers. |
| **cache1st** | A wrapper that memoizes the wrapped method's results using the sole argument as a key. |
| **cache2nd** | A wrapper that memoizes the wrapped method's results using the second of two arguments as a key. |
| **makePerInstance(Wrapper)** | Takes a wrapper w as argument, and returns a wrapper that applies w on a per-instance basis. |
| **makeConditionalWrapper(Procedure, Wrapper)** | Takes a procedure and a wrapper, and returns a wrapper that calls through `makeConditionalWrapper`'s second argument when the procedure returns true, and calls the method directly otherwise. |
| **staticLocker** | Calls the wrapped method with its declaring class locked. |
| **instanceLocker** | Calls the wrapped method with its receiving instance locked. |

**Table 1: Utility wrappers, classes, and methods defined by `maya.wrap.Wrappers`.**

argument list. Wrappers such as `tracer` cannot explicitly throw checked exceptions, since wrapper declarations cannot contain `throws` clauses. However, `wrapped.apply()` may throw arbitrary exceptions that are immediately propagated to the wrapper's caller.

## 2.2 Composing Caches

`Wrappers` defines several other generic wrappers, wrapper subtypes, and wrapper-generating methods. These objects may be composed to implement concerns such as caching in various ways. Some predefined wrappers are listed in Table 1.

The class `CacheNth` and methods such as `makeConditionalWrapper()` and `makePerInstance()` are of particular interest. `CacheNth` uses standard Java features, inner classes and explicit constructors, to associate caches with methods. The definitions of `CacheNth` and `makePerInstance()` are shown in Figures 1 and 2, respectively.

Each time a `CacheNth` wrapper is woven into a method, `generate` is called. This method allocates a new cache and returns a procedure that uses the cache. `CacheNth` uses an explicit constructor parameter to select the key argument used in memoization. It also uses an anonymous procedure class to associate code with state. (In more static aspect languages such as AspectJ, such mechanisms are unavailable.) Procedure objects, like methods, are called with arguments and return values. The details of the procedure calling convention are hidden within the `apply` macro.

A `CacheNth` wrapper may be composed with other wrappers. For example, a per-instance cache for the method `C.m` can be created using

```
wrap C.m(Object)
  with makePerInstance(new CacheNth(1));
```

The wrapper returned by `makePerInstance()` will apply the argument to the wrapped method each time a new receiver object is encountered. The wrapper returned by `makePerInstance()` accepts a receiver of any type, which is bound to `thisReceiver` followed by zero or additional arguments stored in `args`. Since `CacheNth` allocates a cache each time it is woven, the result of `m` will be cached on a per-instance basis.

`CacheNth` provides flexibility with a runtime cost. All arguments to `CacheNth` are packed into an `Object[]` on entry, and unpacked when the underlying method is called. To avoid this cost, Handi-Wrap provides specialized wrappers `cache1st` and `cache2nd` to handle unary static and instance methods, respectively.

In addition to the conventional instance advice defined through `makePerInstance()`, Handi-Wrap provides a general mechanism through `conditionalWrapper` declarations. We can trace calls to a method `n` on a specific instance `c` as follows:

```
wrap C.n() with new Any conditionalWrapper C()
  { thisReceiver == c } => Wrappers.tracer;
```

The above statement wraps `C.n()` with a wrapper that traces its execution when `c` is the receiving instance. This wrapper may be applied to instance methods of `C` that take no arguments and return any type. The `conditionalWrapper` syntax is a macro that is expanded as follows:

```
makeConditionalWrapper(
  new boolean procedure(final C thisReceiver) {
    return thisReceiver == c;
  },
  Wrappers.tracer);
```

The ability to advise a specific instance is critical to caching for `ClassPath.lookup()`. In the `ClassPath` definition used by the Maya compiler, `lookup()` is not actually a static method. `ClassPath` is defined in a separate library, and several `ClassPath` objects can be defined on distinct search paths. A `ClassPath` object may be used in a batch compiler. In such a case, caching produces a significant performance improvement, and it is safe to assume that classes do not change behind the compiler's back. However, other clients have different performance and semantic requirements. A `ClassPath` may be used by a simple class loader that never looks up the same resource twice, in which case caching would not be useful. Alternatively, a `ClassPath` object may be used to load plugins or servlets. In such cases, the disk contents may change dynamically and lookup results should never be cached.

A compiler can install a cache on calls to `lookup()` in its own `ClassPath` as follows:

```
final ClassPath path = new ClassPath(pathString);
wrap ClassPath.lookupString(String) with
  new Any conditionalWrapper ClassPath(String _)
  { thisReceiver == path } => Wrappers.cache2nd;
```

The above code shows how concerns can be separated ver-

```
public static class CacheNth extends Wrapper {
  int offset;

  public CacheNth(int offset) { this.offset = offset; }

  public Procedure generate(final Method thisMethod, final Procedure wrapped) {
    final HashMap cache = new HashMap();        // Store results of thisMethod in its own cache

    return new Object procedure(rest Object[] args) {
       Object ret = cache.get(args[offset]);
       if (ret == null && !cache.containsKey(args[offset]))
         {
           ret = wrapped.apply(args);
           cache.put(args[offset], ret);
         }
        return ret;
      };
  }
}
```

**Figure 1: Caching method results based on an arbitrary argument.**

```
public static Wrapper
makePerInstance(Wrapper wrapper) {
  final Map instanceMap = new WeakHashMap();

  // match any receiver type, any return type and
  // any # of args
  return new Any wrapper Any(rest Object[] args) {
    Procedure proc =
      (Procedure) instanceMap.get(thisReceiver);
    if (proc == null)
      {
        proc = wrapper.generate(thisMethod,
                                wrapped);
        instanceMap.put(thisReceiver, proc);
      }
    // thisReceiver must be explicitly passed in
    return proc.apply(thisReceiver, args);
  };
}
```

**Figure 2: Associating a wrapper with each receiver.**

tically between software packages. An application can customize the behavior of library classes such as `ClassPath` by defining aspects on its public interface. Since Handi-Wrap can weave binaries, a library's behavior can be changed without access to source code.

Handi-Wrap's support for caching exploits many dynamic Java features. First, caches are associated with methods through inner class closures. Second, wrapper behavior is controlled through explicit constructor parameters. Finally, wrapper objects are composed to define a variety of caching policies: one cache per method, one cache per instance, and one instance being cached. Although these specific policies can be implemented in a static aspect language, Handi-Wrap's code-reuse mechanisms provide added flexibility. For example, per-instance advice is implemented as a composable wrapper rather than a new language feature.

## 2.3 Controlling Debug Output

Although tracing method invocations is an important debugging tool, sometimes it is still necessary to resort to `println`. Dynamic aspect weaving can be used to selectively enable debug logging to filter out unimportant output.

One of the trickiest bits of code in `mayac`, the compiler for Maya, is the pattern parser. It subjects both templates that generate AST nodes and macro argument lists to a second level of parsing. Sequences of ordinary syntax tree nodes are translated to trees of objects called `RightSymbols`:

```
class PatternParser {
  RightSymbol parse(Object lhs, Environment env)
  { ... }
  ...
}
class TemplateParser extends PatternParser { ... }
class ArgumentParser extends PatternParser { ... }
```

An implementation of the Maya compiler might define several boolean variables to control different types of debug logging. For instance, `Debug.traceParse` would enable logging of every move the LALR(1) parser makes, and `Debug.tracePatternParse` would enable logging of every move made by the pattern parser.

A detailed trace of parsing activity is critical to track down a bug in pattern parsing. For instance, the compiler may fail on a particular template expression that generates a `Statement` node. To track down this bug, we must obtain a trace of the template expression's parse, but sifting through a trace of the entire compilation unit is painful.

A generic wrapper can be used to limit debug output to the dynamic extent of a particular method as follows:

```
Any wrapper traceParsing(rest Object[] args) {
  Debug.traceParse = true;
  Debug.tracePatternParse = true;
  Any ret = wrapped.apply(args);
  Debug.tracePatternParse = false;
  Debug.traceParse = false;
  return ret;
}
```

The `traceParsing` wrapper could be defined more carefully to handle recursion and exceptions, but the above definition suffices for the debugging problem at hand.

Weaving this wrapper into `PatternParser.parse()` reduces the amount of extraneous debug output, but does not eliminate it entirely. For better results, we can use `tra-`

```
class C {
  static private boolean once = false;
  static public Runnable m() {
    class R implements Runnable {
      public void run() { ... }
    }
    if (Debug.traceRuns && !once) {
      // Wrapping R effects all instantiations of
      // R, just as instanceof matches any
      // instantiation of R.  Only trace it once.
      wrap R.run() with Wrappers.tracer;
      once = true;
    }
    return new R();
  }
}
```

**Figure 3: An example of wrapping methods of local classes.**

ceParsing to build a wrapper that enables logging exactly where it is needed:

```
wrap PatternParser.parse(Object,Environment)
with new RightSymbol conditionalWrapper
         PatternParser(Object lhs, Environment env)
         { thisReceiver instanceof TemplateParser
           && lhs instanceof Type
           && (((Type) lhs).getReflectClass()
               == Statement.class) }
         => traceParsing;
```

Wrappers are woven into method definitions, rather than arbitrary <receiver, signature> pairs. Therefore, the above wrapper must be woven into `parse` in its declaring class, `PatternParser`, and it must explicitly check for the receiver class `TemplateParser`. While Handi-Wrap's weaving mechanism is extremely simple, flexible policies could be built on top of it. For instance, wildcard matching could weave a wrapper into a method definition and all overriding definitions.

Like other aspect systems, Handi-Wrap can reduce development time by augmenting `println` and interactive debuggers with tracing, contract enforcement, and so on. Handi-Wrap has an advantage over static aspect weaving systems in that aspects can be woven without recompiling the weaving target.

## 2.4 Wrapping Unmentionable Methods

Handi-Wrap's tight integration with Java permits strictly more weaving than static aspect languages allow. Handi-Wrap extends the language of statements and expressions. In comparison, AspectJ only extends the language of top-level declarations. As a result of Handi-Wrap's expressiveness, wrappers can be woven into methods that cannot be named at the top level. In particular, methods in local and anonymous classes can be wrapped. Figure 3 shows a `wrap` statement that can only appear inside implementation code. Here the local class `R` can only be named within `C.m()`. Notice that `wrap R.run()` applies to all instances of `R`, regardless of whether they are allocated by the current invocation of `m`. In this respect, `wrap` matches the behavior of `instanceof` and casts, because local classes are merely syntactic sugar.

Handi-Wrap also allows methods in anonymous classes to be wrapped:

```
static public Runnable m() {
  return new Runnable() {
    public wrapped(Wrappers.tracer) void run()
    { ... }
  };
}
```

Here, the method modifier `wrapped(Wrappers.tracer)` signals that the method `run()` should be wrapped in a tracer. The expression `Wrappers.tracer` is evaluated and wrapping is performed in the top-level class `C`'s static initializers.

# 3. IMPLEMENTING HANDI-WRAP

Handi-Wrap generates standard Java bytecode. If JVM compatibility were not an issue, an implementation might involve dynamically updating vtables. Given the constraints of the JVM, Handi-Wrap achieves good performance through the use of carefully chosen data structures, and by open-coding a method's original definition in the same JVM method that dispatches wrappers. Handi-Wrap's implementation decisions are based on the assumption that the vast majority of methods will not be wrapped.

Handi-Wrap is implemented as a group of cooperating macros written in Maya. Maya is an extension of Java that allows users to write their own syntax extensions, which are called Mayans. Mayans can reinterpret or extend Maya syntax by expanding it to other Maya syntax: they operate on abstract syntax trees, and their expansion is triggered during parsing as semantic actions. Maya's expressiveness comes from treating grammar productions as generic functions, and Mayans as multi-methods on those generic functions. Mayans may override the translation of builtin productions, and may add new productions to Maya's LALR(1) grammar.

Handi-Wrap's implementation performs a variety of tasks for which Maya is particularly well suited. First, code to enable wrapping is woven into each class declaration in the component program. The prologue woven into each method is generated programmatically, based on the method's signature. Second, macros encapsulate an efficient procedure-calling convention borrowed from the Kawa [6] Scheme to Java bytecode compiler. Third, Handi-Wrap adds a variety of syntactic forms to Java. This is possible since Maya allows its grammar to be extended in arbitrary ways.

While much of Handi-Wrap is implemented through syntax extensions, two important features are not: Handi-Wrap generates code to call a wrapped method's original definition from wrappers dynamically, and code to enable wrapping of precompiled libraries is implemented using a binary rewriter based on the Bytecode Engineering Library [13]. Without the bytecode rewriter, wrappers could only be dynamically woven into methods compiled by Handi-Wrap.

Czarnecki and Eisenecker [12] describe two basic ways to implement aspects. First, source code transformation can be used to statically weave aspects into the base program, as in AspectJ. Second, dynamic reflection mechanisms can be used to weave code at runtime. Handi-Wrap takes a middle approach: compile-time reflection (and in some cases bytecode rewriting) is used to insert minimal hooks that allow dynamic wrapping.

## 3.1 Wrapping

Handi-Wrap builds wrapped method definitions from wrappee procedures through the method `Wrapper.generate()`. The two challenges in implementing wrappers are replacing an original method definition with a wrapper procedure and generating a wrappee procedure corresponding to an original method definition. The first challenge is met by weaving hooks into component code, and the second challenge is met using dynamic code generation.

To support wrapping, Handi-Wrap adds a static field to each class and a prologue to each method body. The field `proc$` holds an array of wrapper procedures. This array is like a vtable except that it includes static methods and excludes inherited methods. All `proc$` entries are initially null. Handi-Wrap inserts a prologue around each method $m_i$; this prologue checks whether the method has been wrapped by comparing `proc$[i]` with null. If a wrapper is defined, $m_i$ calls through `proc$[i]`, boxing arguments and unboxing the return value as needed. Otherwise, the original method definition is executed.

Handi-Wrap uses the Kawa Scheme system's calling convention, which is more efficient than standard Java interfaces such as `Method.invoke()`. Although Handi-Wrap supports procedures that take a variable number of arguments, the allocation of argument arrays can be avoided in many cases. Also, Handi-Wrap avoids dynamically allocating boxed representations of small integers through the flyweight pattern.

To wrap a method $m_i$, we set `proc$[i]` to the procedure returned by `generate()`. If other wrappers have already been woven into $m_i$, the second argument to `generate()` is the previous value of `proc$[i]`. Otherwise, a *bottom procedure* object must be constructed for the method's original definition.

Handi-Wrap makes a method's original definition available by copying it into a new method that can be called from a bottom procedure. Handi-Wrap generates a bottom procedure definition dynamically the first time a method is wrapped. The generated code avoids the overhead of invoking methods through the Java reflection API. One could instead statically generate a `Procedure` subclass for each method in the component code, but this would make distributed binaries inordinately large.

## 3.2 Type Checking

Handi-Wrap implements static type checking as a thin veneer over a dynamically typed design. In practice, the `wrap` statement provides most of the benefits of static type checking. Without `wrap`, weaving would involve a series of tedious and error-prone calls to the Java reflection API in which methods and types are encoded as strings.

The `wrap` statement statically ensures that there is a method to wrap. The `wrap` statement also checks that wrappers are woven into methods with compatible signatures. Often, these checks are performed statically, based on the concrete type of the wrapper, but in some cases these checks must be deferred to runtime. For instance, static information is not available when a method returns an anonymous wrapper instance. Wrappers can also be defined by subclassing `Wrapper`, in which case Handi-Wrap does not see their signatures. In such cases, the JVM enforces dynamic type safety.

Wrapper declarations may not contain throw clauses. This restriction prevents wrappers from throwing checked exceptions explicitly, but does not limit the set of methods that may be wrapped. Since bottom procedures are generated dynamically at the bytecode level, they are subject to Java Virtual Machine type-checking rules, rather than the slightly more restrictive Java language rules. Specifically, bottom procedures can and do hide the declared exceptions thrown by procedures they call, since the JVM does not constrain the exceptions that a method may throw.

Static checking could be improved by introducing wrapper and procedure signatures to the language. For instance, the type of wrappers applicable to `Object.hashCode()` and `System.identityHashCode()` could be written as `wrapper<Object -> int>`. However, a polymorphic type system would be needed to capture the behavior of methods such as `makePerInstance()` and `makeConditionalWrapper()`.

# 4. PERFORMANCE

In this section, we compare Handi-Wrap to a static aspect language, AspectJ, and evaluate the overhead of Handi-Wrap's compile-time transformations. Tests were run on a dual 350Mhz Pentium II machine with 256MB of memory. Tests were run using JDK-1.3.1 using native threads and the Hotspot client compiler under Debian Linux with a version 2.2.12 kernel. In both the Handi-Wrap and AspectJ tests, the ClassPath library was compiled with JDK-1.3.1's `javac`. For Handi-Wrap tests, the compiled library was retrofitted with the required hooks, and the test framework was compiled with `mayac`. For AspectJ tests, the test framework was compiled with `javac`.

We measured the wall-time taken to perform a series of 7797 calls to `ClassPath.lookup()` obtained by tracing a `mayac` run. Of these calls, 1058 had distinct arguments and 104 returned non-null. Each of the 104 successful calls was made with a distinct argument, since successful calls add types to a namespace.

Table 2 shows the number of seconds spent executing the `ClassPath` benchmark in several configurations. The first line shows base measurements with no wrapping. Subsequent lines give the result of adding various caches around `ClassPath.lookup()`: one cache shared between all instances; a cache for each instance defined using AspectJ's `pertarget` association, a hand-coded wrapper, and Handi-Wrap's `makePerInstance()`; and a cache on one particular instance defined using AspectJ's `if` join point designator, a hand-coded wrapper, and Handi-Wrap's `makeConditionalWrapper()`. An optimized wrapper that avoids storing null values into hash tables was also tested. The columns depict several implementation strategies: a static implementation in AspectJ, a hand-coded wrapper, wrappers derived from `cache2nd`, and wrappers derived from `CacheNth`.

In this example, Handi-Wrap imposes an acceptable overhead: Weaving `cache2nd` around a single `ClassPath` instance is roughly 15% slower than weaving AspectJ advice.

AspectJ's static nature provides JVMs with more chances for optimization. For instance, AspectJ weaves calls to private methods into code where Handi-Wrap weaves virtual function calls. However, Sun's dynamic compiler does not benefit from these inlining hints: a shared `cache2nd` wrapper is no less efficient than unconditional advice in AspectJ.

Handi-Wrap's support for a variable number of arguments

| Policy | AspectJ | Handi-Wrap | | |
|---|---|---|---|---|
| | | Custom | Cache2nd | CacheNth(1) |
| No advice | 4.243 | 4.309 | 4.309 | 4.309 |
| Shared cache | 0.765 | 0.760 | 0.760 | 0.815 |
| Per-target cache | 0.812 | 0.772 | 0.874 | 0.898 |
| Specific target cache | 0.761 | 0.760 | 0.871 | 0.889 |
| Optimize nulls | | 0.747 | | |

**Table 2: Seconds spent executing the cache benchmark without directory existence cache.**

| Policy | AspectJ | Handi-Wrap | | |
|---|---|---|---|---|
| | | Custom | Cache2nd | CacheNth(1) |
| No advice | 0.944 | 0.965 | 0.965 | 0.965 |
| Shared cache | 0.252 | 0.253 | 0.253 | 0.316 |
| Per-target cache | 0.307 | 0.258 | 0.369 | 0.401 |
| Specific target cache | 0.254 | 0.251 | 0.367 | 0.382 |
| Optimize nulls | | 0.236 | | |

**Table 3: Seconds spent executing the cache benchmark with directory existence cache.**

imposes a considerable cost. A shared `CacheNth` wrapper runs roughly 60ms slower than a `cache2nd` wrapper, or about 7.5 $\mu$s per call. Much of this time is spent packing `CacheNth`'s arguments into an array, cloning the array for calls to the wrapped method, and unpacking arguments from the cloned array.

AspectJ caches take a 50ms performance hit moving from a shared cache to per-target caches. This makes sense, since per-target caching involves operations on a second hash table. Similarly `cache2nd` takes a 115ms hit. This can be attributed to both additional hashing and operations on argument arrays.

When an AspectJ cache is woven into a specific target, the second hash table disappears, along with the overhead it imposes. However, `cache2nd` does not get appreciably faster. This difference is due to the implementation of `makeConditionalWrapper()`. The guard around AspectJ's advice copies `this` into a local variable and compares it against a field, while the guard around `cache2nd` applies a boolean procedure to its argument array that performs the same comparison. Before we compare the `lookup()` receiver against the cacheable instance, it is copied into an array, the array is cloned, and the receiver is copied out of the clone. This sort of code presents a challenge to any optimizing compiler.

Handi-Wrap's overhead can also be measured in the presence of other optimizations. For example, `ClassPath` provides its own caching mechanism. A `ClassPath` object can be defined so that each directory on the path memoizes nonexistent subdirectories. This cache captures redundancy between distinct lookup arguments. For example, we could avoid a system call to look up `./java/lang/String.class` if we learned that `./java/lang` does not exist when looking up `Object`.

The result of the caching benchmark with directory caching enabled is shown in Table 3. Handi-Wrap's absolute overhead remains the same: `cache2nd` on a specific instance is roughly 110ms slower than the equivalent advice. However, the relative cost has increased: Handi-Wrap is now 44% slower than AspectJ, and this overhead represents 31% of the time spent searching the class path. This overhead can be eliminated through hand-inlining, as shown in the custom column. Further improvement is possible if one avoids storing null values in `HashMaps`.

| Benchmark | Original | Retrofitted | % cost |
|---|---|---|---|
| _200_check | 0.395 | 0.399 | 1.013% |
| _227_mtrt | 14.837 | 17.691 | 19.24% |
| _202_jess | 15.426 | 17.601 | 14.10% |
| _201_compress | 50.655 | 61.131 | 20.68% |
| _209_db | 49.213 | 48.471 | -1.508% |
| _222_mpegaudio | 32.294 | 34.152 | 5.753% |
| _228_jack | 18.605 | 19.893 | 6.922% |
| _213_javac | 36.875 | 41.396 | 12.26% |

**Table 4: Seconds spent in SpecJVM benchmarks, with and without Handi-Wrap prologues.**

Most of Handi-Wrap's overhead appears to be introduced by argument arrays. Some fairly simple optimizations can reduce this cost. For instance, the wrapper procedures defined by `makeConditionalWrapper()` and `makePerInstance()` only use rest arguments to call other procedures. We could avoid argument arrays entirely by specializing these procedures on arities up to 4.

In summary, Handi-Wrap's generic wrappers introduce overhead into argument-list processing. This overhead may be acceptable for unoptimized programs, but hand-inlining can be used when the overhead grows too great.

We also measured the cost introduced by Handi-Wrap's method prologues. We ran the SpecJVM98 benchmarks until a minimum execution time was reached. We then retrofitted the benchmark classes with Handi-Wrap method prologues and repeated the test. The results are shown in Table 4. On average, Handi-Wrap imposes a 9.8% overhead, but the overhead varies widely with the frequency of method calls. The multi-threaded raytracer performs poorly since it uses private fields exclusively and includes frequent calls to field getters. Jess also performs poorly, because builtin functions of the expert shell language are implemented as Java methods, and these methods are typically quite small. Compress suffers because of a few small methods called within its inner loop.

User intervention may be needed to achieve acceptable

performance for certain code. An important first step is not applying Handi-Wrap to code such as an LZW compressor that implements a single well-defined concern. Performance can also be improved by making particular methods unwrappable. For instance, the cost of using Handi-Wrap with mtrt drops to 0.5% when prologues are not added to getter methods. Currently, however, Handi-Wrap does not provide a mechanism for such fine-grained control over wrappability.

## 5. RELATED WORK

Handi-Wrap owes much to AspectJ [26]. Method wrappers achieve the same effect as AspectJ advice. However, AspectJ advice can be woven into a rich set of *join points* rather than merely method bodies. AspectJ also provides an abstraction called pointcut not present in Handi-Wrap. AspectJ's join point designators and pointcuts provide a wide variety of functionality, much of which is also available in Handi-Wrap. First, designators establish bindings to advice arguments, a role filled by Handi-Wrap's parameter lists. Second, designators determine when advice applies. In Handi-Wrap, this decision is made by the wrap statement, rather than the wrapper definition. A wrap statement applies a wrapper to a particular method, and can achieve finer-grained control through the use of conditional wrappers. Finally, join point designators may include wildcards.

Handi-Wrap does not allow join-points to be specified through wildcards. It is unclear how wildcards should work in Handi-Wrap. Should wildcard matching be performed statically or dynamically? Should wildcards place restrictions on user-defined classloaders? Should wildcards load classes eagerly, as the current `wrap` statement does, or lazily?

Handi-Wrap's approach to advice reuse is more flexible than AspectJ's abstract pointcuts in several ways. First, all wrappers are reusable, rather than only those coded in a particular style. Hanenberg and Unland [17] describe a systematic but cumbersome technique for achieving reuse in AspectJ. Second, higher-order wrappers such as `makePerInstance()` and `makeConditionalWrapper()` can be defined in the Handi-Wrap language, rather than provided as primitives. User-defined wrappers can implement features of join point designators. For instance, `cflow`'s functionality could be implemented easily and efficiently, and `within` is certainly implementable. These operations provide the building blocks for others such as `this` and `withincode`. Finally, higher-order wrappers fill the roles of both pointcut designators and the aspect instantiation clause `pertarget`.

CLOS [24]'s `:around` method qualifier allows code to be wrapped around all invocations of a generic function. ApectJ's `around` advice and Handi-Wrap's wrappers serve a similar role. Wrappers differ from around methods in two key ways. First, a wrapper applies to an individual method definition rather than a generic function. Second, the order in which wrappers execute is determined by the order in which they are applied to a method, rather than the types of actual arguments.

Other aspect systems such as SOFA/DCUP [18] and Aspectual Components [19] address the need for separate compilation in a component context. Although these systems support dynamic aspect weaving, they do not treat aspects at first-class Java entities; hence, they do not provide the additional reuse mechanisms found in Handi-Wrap. Both systems do support wildcards, and pointcuts are the primary thrust of Aspectual Components. Aspectual Components gives each pointcut a type based on "isA" and "hasA" relationships.

Other aspect systems have been implemented through metaprogramming. AOP/ST [5] is implemented in VisualWorks Smalltalk. AOP/ST dynamically generates branches in the inheritance hierarchy and alters the classes of objects. In AOP/ST, advice is woven into all direct instances of a particular class, whereas in Handi-Wrap advice is woven into concrete method definitions directly.

DCOOL [11] implements AspectJ 0.1's synchronization language in VisualWorks Smalltalk, while Lunau [20] defines a meta-object protocol for Objective-C that supports method wrapping. Interestingly, both these systems weave advice on a per-instance basis, rather than the per-class basis adopted by more recent versions of AspectJ and by Handi-Wrap. Both DCOOL and Lunau's system utilize dynamic reflection, where Handi-Wrap uses compile-time reflection to insert exactly those hooks needed for dynamic weaving. Handi-Wrap is similar to Lunau's system in that both emphasize the importance of composition in aspect code.

There has been a recent surge of interest in bridging the gap between class- and prototype-based inheritance models [9, 22]. Like the dynamic aspect systems described above, these mechanisms allow methods to be overridden on a particular instance. Handi-Wrap is similar to the compound reference model [22] in that both allow a Java method's behavior to be changed at the statement level, rather than through external aspect declarations.

David et al. [14] implement dynamic advice binding in AspectJ. Handi-Wrap follows a similar architecture, in which hooks are added to a method at compile time, and wrappers are defined on these hooks at runtime. However, Handi-Wrap differs in that hooks are applied to all methods, rather than a specific set of methods defined by a pointcut. Handi-Wrap also uses a more efficient implementation, since there are no Java reflection calls on the critical path, and method arguments are not always passed in arrays.

JAC [23] allows Java methods to be dynamically wrapped and unwrapped through a classloader that performs bytecode rewriting. JAC includes a number of features that Handi-Wrap lacks, such as the ability to remove wrappers and the ability to choose the order in which wrappers execute on a particular method call. However, like David's system, JAC's implementation relies heavily on Java's reflection API and appears less efficient than Handi-Wrap. Unlike JAC, Handi-Wrap extends the Java language with new expression, statement, and declaration forms. These extensions make Handi-Wrap easier to use in two ways. First, Handi-Wrap performs limited static type checking, whereas JAC cannot statically verify that class and method names are valid. Second, Handi-Wrap's first class notion of procedures and wrappers, along with the `apply` operator, provide a basis for writing composable wrappers.

Goedicke et al. [16] propose a system in which wrapping is pervasive. A component exports a set of TCL bindings that are natural to its implementation, and the client wraps these commands in XOTCL classes that provide an interface natural to the client. XOTCL's metaobject protocol allows wrapper classes to easily express restrictions and conventions in a component's exported interface. XOTCL also supports transparent method wrapping through a mechanism called per-object mixins. This mechanism allows an imported component's behavior to be further customized.

While a number of powerful macro systems, compile-time metaprogramming extensions, and preprocessor toolkits [1, 4, 8, 25] are available for Java, we believe that Maya is best suited for implementing a language extension such as Handi-Wrap. First, Handi-Wrap extends Java's concrete syntax in ways that can't be expressed by simple macro systems such as JSE [1] and compile-time MOPs such as OpenJava [25] and ELIDE [8]. Second, Handi-Wrap uses static type information that is not available in purely syntactic systems such as JSE and JTS [4]. Finally, Maya allows overloading of syntax based on arbitrary static types and does not require the base-level program to explicitly refer to metaprograms. In contrast, OpenJava only allows syntax overloading based on classes that explicitly instantiate a metaclass.

## 6. CONCLUSIONS

Handi-Wrap is an extension to Java that supports dynamic aspect weaving. Handi-Wrap's dynamic nature allows wrappers to be defined compositionally. In addition, wrappers are reusable because they can be defined locally, and can be defined with explicit constructor parameters. Finally, Handi-Wrap includes an expressive library of generic wrappers. An implementation of Handi-Wrap is available at `http://www.cs.utah.edu/~jbaker/maya`. Handi-Wrap achieves efficiency on standard JVMs through the use of carefully chosen data structures and limited use of the Java reflection API.

This paper focused on method wrapping in Handi-Wrap. Some of Handi-War's library wrappers fill the role of language constructs such as AspectJ join point designators. Other features of aspect-oriented languages, such as introductions and wildcards, are not supported by Handi-Wrap. We note that Maya, the language on top of which Handi-Wrap is built, can support method introductions through an implementation [2, 3] of the MultiJava language extension [10], which allows externally defined methods to be introduced into classes.

## 7. ACKNOWLEDGEMENTS

We thank Eric Eide, Alastair Reid, John Regehr, Sean McDirmid, and the anonymous reviewers for their comments and suggestions. This research was supported by the Defense Advanced Research Projects Agency and the Air Force Research Laboratory under agreement number F33615–00–C–1696 and a National Science Foundation CAREER award, CCR–9876117. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation hereon.

## 8. REFERENCES

[1] J. Bachrach and K. Playford. The Java syntactic extender (JSE). In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, Oct. 2001.

[2] J. Baker. Macros that play: Migrating from Java to Maya. Master's thesis, University of Utah, December 2001.

[3] J. Baker and W. Hsieh. Maya: Multiple-dispatch syntax extension in Java. To appear in *Proceedings of the Conference on Programming Language Design and Implementation*, Jun.

[4] D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: Tools for implementing domain-specific languages. In *5th International Conference on Software Reuse*, 1998.

[5] K. Böllert. On weaving aspects. In *ECOOP Workshop on Aspect-Oriented Programming*, 1999.

[6] P. Bothner. Kawa—compiling dynamic languages to the Java VM. In *Proceedings of the USENIX Technical Conference, FREENIX Track*, New Orleans, LA, Jun. 1998. USENIX Association.

[7] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, Vancouver, October 1998. ACM.

[8] A. Bryant, A. Catton, K. D. Volder, and G. C. Murphy. Explicit programming. In *Proceedings of the First International Conference on Aspect-Oriented Software Development*, Apr. 2002.

[9] M. Büchi and W. Weck. Generic wrappers. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 201–225, 2000.

[10] C. Clifton, G. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications '00*, pages 130–146, Minneapolis, MN, Oct. 2000.

[11] K. Czarnecki. Dynamic cool. `http://www.prakinf.tu-ilmenau.de/~czarn/aop/sources.tar.gz`.

[12] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Techniques, and Applications*, chapter 8. Addison-Wesley, 1999.

[13] M. Dahm. The byte code engineering library. `http://bcel.sourceforge.net`.

[14] P.-C. David, T. Ledoux, and N. N. M. Bouraqadi-Saâdani. Two-step weaving with reflection using AspectJ. In *OOPSLA Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, Oct. 2001.

[15] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 171–183, 1998.

[16] M. Goedicke, G. Neumann, and U. Zdun. Design and implemenation constructs for the development of flexible, component-oriented software architectures. In *Proceedings of the Second International Symposium on Generative and Component-Based Software Engineering*, pages 114–128, 2000.

[17] S. Hanenberg and R. Unland. Using and reusing aspects in AspectJ. In *OOPSLA Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, Oct. 2001.

[18] N. D. Hoa. Dynamic aspects in SOFA/DCUP. Technical Report 99/07, Charles University, Prague, Jun. 1999.

[19] K. Lieberherr, D. Lorenz, and M. Mezini. Programming with aspectual components. Technical Report NU-CCS-99-01, Northeastern University, Apr. 1999.

[20] C. P. Lunau. A reflective architecture for process control applications. In *Proceedings of the European Conference on Object-Oriented Programming*, 1997.

[21] P. Niemeyer. Beanshell — lightweight scripting for Java. `http://www.beanshell.org/`.

[22] K. Ostermann and M. Mezini. Object-oriented composition untangled. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, Oct. 2001.

[23] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A flexible solution for aspect-oriented programming in Java. In *Reflection 2001*, pages 1–24, 2001. LNCS 2192.

[24] G. Steele Jr. *Common Lisp, the Language*. Digital Press, second edition, 1990.

[25] M. Tatsubori, S. Chiba, M. Killijian, and K. Itano. *Reflection and Software Engineering*, volume 1826 of *Lecture Notes in Computer Science*, chapter OpenJava: a class-based macro system for Java. Springer Verlag, 2000.

[26] Xerox. The AspectJ programming guide. `http://www.aspectj.org/doc/dist/progguide/`.