To appear in 6th Usenix Cnference of Object-Oriented Technoologies and Systems (COOTS 2001)

Multi-Dispatch in the *Java Virtual Machine*: Design and Implementation

Christopher Dutchyn[*] Paul Lu Duane Szafron Steve Bromling Wade Holst[†]

July 27, 2000

Abstract

Mainstream object-oriented languages, such as C++ and Java[1], provide only a restricted form of polymorphic methods, namely single-receiver dispatch. In common programming situations, programmers must work-around this limitation. We describe how to extend the Java Virtual Machine to support multiple-dispatch and examine the complications that Java imposes on multiple-dispatch in practice. Our technique avoids changes to the Java programming language itself, maintains source-code and library compatibility, and isolates the performance penalty and semantic changes of multiple-dispatch to the program sections which use it. We have micro-benchmark and application-level performance results for a dynamic *Most Specific Applicable* (MSA) dispatcher, a framework-based *Single Receiver Projections* (SRP) and a tuned SRP dispatcher. Our general-purpose technique provides smaller dispatch latency than programmer-written double-dispatch code with equivalent functionality.

1 Introduction

Object-oriented (OO) languages provide powerful tools for expressing computations. One key abstraction is the concept of a *type hierarchy* which describes the relationships among types. Objects represent instances of these different types. Most existing object-oriented languages require each object variable to have a programmer-assigned *static type*. The compiler uses this information to recognize some coding errors. The *principle of substitutability* mandates that in any location where type T is expected, any sub-type of T is acceptable. But, substitutability allows that object variable to have a different (but related) *dynamic type* at runtime.

```
class Point {
 int x, y;
 void draw(Canvas c)
      { // Point specific code }
 void translate(int t) {x+=t; y+=t;}
 void translate(int tX,int tY)
      {x+=tX; y+=tY;}
}

class ColorPoint extends Point {
 Color c;
 void draw(Canvas C)
      { // ColorPoint code }
}

// same static, different dynamic types
Point Pp = new Point();
Point Pc = new ColorPoint();

// static multi-dispatch
Pp.translate(5); // single int version
Pp.translate(1,2); // two int version

// dynamic uni-dispatch
Pp.draw(aCanvas); // Point::draw()
Pc.draw(aCanvas); // ColorPoint::draw()
```

Figure 1: Dispatch Techniques in Java

Another key facility found in OO languages is method selection based upon the types of the arguments. This method selection process is known as *dispatch*. It can occur at compile-time or at runtime. In the former case, where only the static type information is available, we have *static dispatch* (method overloading). The latter case is known as *dynamic dispatch* (method overriding or virtual functions), and object-oriented languages leverage it to provide polymorphism—the execution of type-specific program code.

We can divide OO languages into two broad categories based upon how many arguments are considered during dispatch. *Single-dispatch* languages select a method based upon the type of one distinguished argument; *multi-dispatch* languages consider more than one, and potentially all of the argu-

[*] {dutchyn,paullu,duane,bromling}cs.ualberta.ca, Dept. of Computing Science, University of Alberta

[†] wade@csd.uwo.ca, Dept. of Computer Science, University of Western Ontario

[1] Java is a trademark of Sun Microsystems Inc.

ments at dispatch time. For example, Smalltalk [13] is a single-dispatch language. CLOS [24] and Cecil [6] are multi-dispatch languages. For uniformity, in the rest of this paper, we use the term *uni-dispatch* to denote single-receiver dispatch and *multi-dispatch* to denote multi-method dispatch. This is especially important because we will also discuss double dispatch which consists of a sequence of two uni-dispatches and we want to avoid confusing double dispatch with multi-dispatch.

C++ [25] and Java [14, 15] are dynamic uni-dispatch languages. However, for both languages, the compiler considers the static types of all arguments when compiling method invocations. For this reason, we can regard these languages as supporting static multi-dispatch. Figure 1 illustrates both dynamic uni-dispatch and static multi-dispatch in Java.

Uni-dispatch limits the method selection process to consider only a single argument. This is a substantial limitation, and standard programming idioms exist to overcome this restriction. As a motivation for multi-dispatch, we will describe one programming idiom that signals the need for multi-dispatch, describe how it can be replaced by multi-dispatch, list the advantages of using multi-dispatch to replace the idiomatic code, and provide the timing cost of using multi-dispatch with one of our current multi-dispatch algorithms.

1.1 Double Dispatch

Double dispatch occurs when a method checks an argument type and executes different code as a result of this check. Double dispatch is illustrated in Figure 2(a) (from Sun's AWT classes) where the `processEvent(AWTEvent)` method must process events that are instances of different classes in different ways. Since all of the events were placed in a queue whose static element type is `AWTEvent`, the compiler lost the more specific dynamic type information. When an element is removed from the queue for processing, its dynamic type must be explicitly checked to pick the appropriate action. This is an example of the well-known container problem [5].

Double dispatch suffers from a number of disadvantages. First, double dispatch has the overhead of invoking a second method. In this example, the penalty is reduced because only one argument and no return values are involved. Second, the double dispatch program is longer and more complex; this provides more opportunity for coding errors. Third, the double dispatch program is more difficult to maintain since adding a new event type requires not only the code to handle the new event, but another cascaded `else if` statement.

The need for double dispatch develops naturally in several common situations. First, consider binary operations [4], such as the `compareTo(Object)` method defined in interface `Comparable`. The programmer must ascertain the type of the `Object` argument before continuing to perform a type-specific comparison. Another common place for double dispatch is in drag-and-drop applications, where the result of a user action depends on both the data object dragged, and on the application catching the dropped information. A generic drag-and-drop schema must force the programmer to test data types, and re-dispatch to a more specific method. A third location is in event-driven programming. As we saw in Figure 2, applications are written using base classes such as `Component` and `Event`, but we need to take action based upon the specific type of `Component` and `Event`. Indeed, the need for multiple-dispatch is ubiquitous enough that two of the original design patterns, *Visitor* and *Strategy*, are workarounds to supply multiple-dispatch within single-dispatch languages.

Consider the AWT example, if dynamic multi-dispatch was available in Java. The same program text might resemble Figure 2(b). For clarity, we maintain the case statement to select among `MouseEvent` categories, but a more complete factoring of `MouseEvent` into `MouseButtonEvent` and `MouseMotionEvent` would eliminate the remaining double dispatch. Our *Full Multi-Dispatch* timing is for this more elegant factoring. We no longer need to double-dispatch on any events. Instead, we can use that method name for each of the more precise methods, and rely upon the dynamic multi-dispatcher to select the correct method at runtime based upon the *dispatchable arguments* in addition to the *receiver argument* (the instance of `Component`). Individual component types can still override the methods that accept specific event types (eg. `KeyEvent`, `FocusEvent`), and will do so without invoking the double dispatcher.

The multi-dispatch version is shorter and clearer. However, it requires the Java Virtual Machine (JVM) to directly dispatch an `Event` to the correct `processEvent(AWTEvent)` method. Our modified JVM provides this facility, and correctly executes the multi-dispatch code above. Furthermore, Table 1, a subset of Table 4. shows that multi-dispatch is as fast as double-dispatch.

```
package java.awt;

class Component {

 // double dispatch events to subComponent
 void processEvent(AWTEvent e) {
  if (e instanceof FocusEvent) {
   processFocusEvent((FocusEvent)e);
  } else if (e instanceof MouseEvent) {
   switch (e.getID()) {
   case MouseEvent.MOUSE_PRESSED:
    ...
   case MouseEvent.MOUSE_EXITED:
    processMouseEvent((MouseEvent)e);
    break;
   case MouseEvent.MOUSE_MOVED:
   case MouseEvent.MOUSE_DRAGGED:
    processMouseMotionEvent((MouseEvent)e);
    break;
   }
  } else if (e instanceof KeyEvent) {
   processKeyEvent((KeyEvent)e);
  } else if (e instanceof ComponentEvent) {
   processComponentEvent((ComponentEvent)e);
  } else if (e instanceof InputMethodEvent) {
   processInputMethodEvent((InputMethodEvent)e);
  }
  // other events ignored by Component
 }

 void processFocusEvent(FocusEvent e) {...}

 void processMouseEvent(MouseEvent e) {...}

 void processMouseMotionEvent(MouseEvent e) {...}

 void processKeyEvent(KeyEvent e) {...}

 void processComponentEvent(ComponentEvent e) {...}

 void processInputMethodEvent(InputMethodEvent e) {...}
}
```

(a) Double Dispatch in Java

```
package java.awt;

class Component {

 void processEvent(AWTEvent e) {...}

 void processEvent(MouseEvent e) {
  switch (e.getID()) {
  case MouseEvent.MOUSE_PRESSED:
   ...
  case MouseEvent.MOUSE_EXITED:
   processMouseEvent((MouseEvent)e);
   break;
  case MouseEvent.MOUSE_MOVED:
  case MouseEvent.MOUSE_DRAGGED:
   processMouseMotionEvent((MouseEvent)e);
   break;
  }
 }

 void processEvent(FocusEvent e) {...}

 void processMouseEvent(MouseEvent e) {...}

 void processMouseMotionEvent(MouseEvent e) {...}

 void processEvent(KeyEvent e) {...}

 void processEvent(ComponentEvent e) {...}

 void processEvent(InputMethodEvent e) {...}
}
```

(b) Equivalent Code in Multi-Dispatch Java

Figure 2: Double vs. Multi-Dispatch in Java

Our experience with Swing reinforces our belief that double dispatch in AWT is a significant factor. First, Swing does not operate without AWT; instead `AWTEvent` is accepted by a Swing `JComponent`. Therefore, every mouse click and keyboard press is double-dispatched through AWT into Swing. Next, Swing type-checks, and double-dispatches again! Internally, Swing avoids further double-dispatch by coding the `AWTEvent` type into the selector (e.g. `fireInternalEvent()`). Despite the limitations this imposes on the programmer, it is clear that double-dispatch is still the standard technique in Swing as well.

A multi-dispatch Java Virtual Machine offers new potential to other languages as well. For example, Standard ML, Scheme, and Eiffel have implementations which generate JVM compatible binary files. With a multi-dispatch VM underlying these languages, extending them with multi-dispatch semantics becomes straight-forward. In contrast to Java

Dispatch	Time in μs.	(σ)	Normalized
Double	0.91	(0.00)	1.00
Multi-	1.13	(0.03)	1.25
Full Multi-	0.90	(0.01)	1.00

Table 1: AWT Event Dispatch Comparison (Call-Site Dispatch Time in microseconds)

source-language based multi-dispatch, our JVM-based multi-dispatch offers this same multi-dispatch potential to these other languages as well.

The research contributions of this paper are:

1. The design and implementation of an extended Java Virtual Machine that supports arbitrary-arity multi-dispatch with the properties:
   (a) The Java syntax is not modified.
   (b) The Java compiler is not modified.
   (c) The programmer can select which classes should use multi-dispatch.
   (d) The performance and semantics of uni-dispatched methods are not affected.
   (e) The existing class libraries are not affected.
2. The introduction of a dynamic version of Java's static multi-dispatch algorithm.
3. The first performance results of table-based multi-dispatch techniques in a mainstream language.

We begin by reviewing some important details about Java and the implementation of the uni-dispatch Java Virtual Machine. This is followed by a description of the way we re-designed portions of the JVM to support multi-dispatch. Next we present experimental results for our implementation of several different multi-dispatch techniques. This is followed by a discussion of several complex and difficult issues that must be addressed, and a description of some of the details of our implementation. Finally, we close with a short review of related approaches to multi-dispatch and a description of unresolved issues.

2 Background

The Java Programming Language [14, 15] is a static multi-dispatch, dynamic uni-dispatch, dynamic loading object-oriented language. Our primary design goal is to extend the dynamic method selection to optionally and efficiently consider all arguments, without affecting the syntax of the language or any other semantics. Secondary goals are to retain the dynamic and reflective properties of Java.

In order to meet these goals, we chose to modify the Java Virtual Machine [20, 21](JVM) implementation, rather than modifying the programming language itself. Java programs are compiled by `javac`[2] into bytecodes representing primitive operations of a simple stack-based computer. These bytecodes are interpreted by a Java Virtual Machine written for each hardware platform. Our JVM is the "classic" VM (now known as the *Research Virtual Machine*[3]) written in C and distributed by Sun Microsystems Inc. Other JVM implementations exist, and many include *Just In Time* (JIT) compiler technology to enhance the interpretation speed at runtime by replacing the bytecodes with equivalent native machine instructions. At present, we do not support JIT in our techniques.

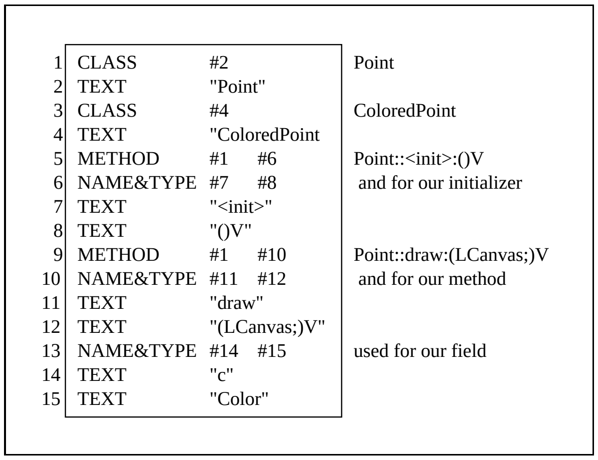
[2]or other compiler

Before we look at how to implement multi-dispatch in the virtual machine, we first need to understand the binary representation that the virtual machine executes, how method invocations are translated into the virtual machine code, and how the JVM actually dispatches the call-sites.

2.1 Java Classfile format

The JVM reads the bytecodes, along with some necessary symbolic information from a binary representation, known as a `.class` file. Each `.class` file contains a symbol table for one class including a description of its superclasses, and a series of method descriptions containing the actual bytecodes to interpret. We need to leverage the symbolic information, called the *constant pool*, to effect multi-dispatch.

Figure 3 shows the layout of the constant pool for the `ColorPoint` class shown in Figure 1.



1	CLASS	#2		Point
2	TEXT	"Point"		
3	CLASS	#4		ColoredPoint
4	TEXT	"ColoredPoint		
5	METHOD	#1	#6	Point::<init>:()V
6	NAME&TYPE	#7	#8	and for our initializer
7	TEXT	"<init>"		
8	TEXT	"()V"		
9	METHOD	#1	#10	Point::draw:(LCanvas;)V
10	NAME&TYPE	#11	#12	and for our method
11	TEXT	"draw"		
12	TEXT	"(LCanvas;)V"		
13	NAME&TYPE	#14	#15	used for our field
14	TEXT	"c"		
15	TEXT	"Color"		

Figure 3: A Simple Constant Pool

Conceptually, the constant pool consists of an array

[3]The Research Virtual machine was initially released as the "classic" reference VM. Sun later renamed it the *Exact* VM. With the advent of the *HotSpot* VM, the classic VM was released as the Research VM.

containing text strings and tagged references to text strings. In Figure 3, class `Point` is represented by a tag entry at location 1 that indicates that it is a CLASS tag and that we should look at constant pool location 2 for the name text. Then, the constant pool contains the text string "`Point`" at location 2. Therefore, a class symbol requires two constant pool entries. Method references are similar, except they require five constant pool entries: the METHOD (line 9), the CLASS entry (line 1), the NAME-AND-TYPE entry (line 10), the TEXT entry containing the selector name (line 11), and the TEXT entry containing the descriptor(line 12).

In our example, constant pool location 9 contains the tag declaring that it contains a METHOD. It references the CLASS tag at location 1, to define the static type of the class containing the method to be invoked. In this case, the class happens to be `Point` itself; but, most often, this is not the case. The METHOD entry also references the NAME-AND-TYPE entry at location 10. This NAME-AND-TYPE entry contains pointers to text entries at locations 11 and 12. The first location, 11, contains the method name, "`draw`". The second location, 12, contains a specially encoded signature "`(LCanvas;)V`" describing the number of arguments to the method, their types, and the return type from the method. In our example, we see one class argument with name "`Canvas`" and that the return type is `void`.

2.2 Static Multi-Dispatch in Javac

The Java compiler converts source code into a binary representation. When it encounters a method invocation, `javac` must emit a constant pool entry that describes the method to be invoked. It must provide an exact description, so that, for instance, the two `translate(...)` methods in `Point` can be distinguished at runtime. Therefore, it must examine the types of the arguments at a call-site, and select between them. This selection process which considers the static types of all arguments can be viewed as a static multi-dispatch.

The Java Language Specification, 2nd Edition [15] (JLS) provides an explicit algorithm for static multi-dispatch called *Most Specific Applicable* (MSA). At a callsite, the compiler begins with a list of all methods implemented and inherited by the (static) receiver type. Through a series of culling operations, the compiler reduces the set of methods down to a single most specific method. The first operation removes methods with the wrong name, methods that accept an incorrect number of arguments and methods that are not accessible from the callsite. This latter group includes private methods when the call-site is in another class, or protected methods from outside of the package.

Next, any methods which are not compatible with the static type of the arguments are also removed. This test relies upon testing *widening conversions*, where one type T_{sub} can be widened to another T_{super} if and only if T_{sub} is the same type as T_{super} or a subtype of T_{super}. For example, a `FocusEvent` can be widened to an `AWTEvent`, because the latter is a super-type of the former[4]. The opposite is not valid: an `AWTEvent` cannot be widened to a `FocusEvent`; indeed a type-cast from `AWTEvent` to `FocusEvent` would need to be a type-checked *narrowing* conversion.

Finally, `javac` attempts to locate the single *most specific* method among the remaining subset of *statically applicable* methods. One method $\mathtt{M}(T_{1,1}, \ldots, T_{1,n})$ is considered more specific than $\mathtt{M}(T_{2,1}, \ldots, T_{2,n})$ if and only if each argument type $T_{1,i}$ can be widened to $T_{2,i}$ for each $(i = 1, \ldots, n)$, and for some j, $T_{2,j}$ cannot be widened to $T_{1,j}$. In effect, this means that any set of arguments acceptable to $\mathtt{M}(T_{2,1}, \ldots, T_{2,n})$ is also acceptable to $\mathtt{M}(T_{1,1}, \ldots, T_{1,n})$, but not vice versa.

Given the subset of applicable methods, `javac` selects one M_t as its tentatively most specific. It then checks each other candidate method M_c by testing whether its arguments can be widened to the corresponding argument in M_t. If this is successful, then M_c is at least as specific as M_t; and, the compiler adopts M_c as the new tentatively most specific method - the method M_t is discarded from the candidate list. If the first test, whether M_c be widened to M_t, is unsuccessful, then the compiler checks the other direction: can M_t be widened to M_c. If so, then the compiler discards M_c from the candidate list.

Unfortunately, both tests can fail. To illustrate this, consider the first two methods in figure 4. The first argument of the first method (`ColorPoint`) can be widened to the type of the first argument of the second method (`Point`). But the opposite is true for the second argument of each method. If we invoke `colorBox` with two `ColorPoint` arguments, both methods apply. If the third method was not present, we would have an *ambiguous method* error. The third method, taking two `ColorPoints`, removes the ambiguity because it is more specific

[4]The JLS separately recognizes identity conversions (a `FocusEvent` can be converted into a `FocusEvent`). `Javac` does not distinguish them, so we do the same for our exposition.

than both of the other methods. It allows both of the others to be discarded, giving a single most specific method.

```
colorBox(ColorPoint p1, Point p2) {...}
colorBox(Point p1, ColorPoint p2) {...}
          // conflict method removes ambiguity
colorBox(ColorPoint p1, ColorPoint p2) {...}
```

Figure 4: Ambiguous and Conflict Methods

Primitive types[5], when used as arguments, are tested at the same time, and in the same way. Primitive widening conversions are defined which effectively impose a standard type hierarchy on the primitive types. The compiler inserts widening casts as needed.

2.3 Dynamic Uni-Dispatch in the JVM

Now we turn our attention to dispatching polymorphic call-sites at runtime. Methods are stored in the `.class` file as sequences of virtual machine instructions. Within a stream of bytecodes, method invocations are represented by `invoke` bytecodes that occupies 3 bytes[6]. The first byte contains the opcode (`0xb6` for `invoke-virtual`). The remaining two bytes form an index into the constant pool. The constant pool must contain a METHOD entry at the given index. This entry contains the static type of the receiver argument (as the CLASS linked entry), and the method name and signature (through the NAME&TYPE entry). Figure 5 shows the pseudo-bytecode[7] for invoking the method `Component.-processEvent(AWTEvent)` twice.

From the opcode, `invokevirtual`, the JVM knows that the next two bytes contain the constant pool index of a METHOD descriptor. From that descriptor, the JVM can locate the method name and signature. The JVM parses the signature to discover that the method to be invoked requires a receiver argument and one other argument. Therefore, the JVM peeks into the operand stack and locates the receiver argument. At this point, the JVM has the information it needs to begin searching for the method to invoke. It has the name, the signature, and the receiver of the message.

The JVM Specification (section 5.4.3.3) provides a recursive algorithm for *resolving* a method reference and locating the correct method. Beginning with

[5]Java provides non-object types `byte`, `char`, `short`, `int`, `long`, `float`, and `double`. These are called primitive types.

[6]`Invokeinterface` occupies 5 bytes.

[7]Rather than show constant pool indices, we show their values directly.

```
Component aComponent = new SubComponent(...);
AWTEvent anEvent = new  FocusEvent(...);
FocusEvent aFocusEvent = new  FocusEvent(...);

aComponent.processEvent(anEvent);
aComponent.processEvent(aFocusEvent);
```

(a) Polymorphic call sites in source.

```
...
apush      aComponent
apush      anEvent
invokevirtual    Component::processEvent:(LAWTEvent;)V

apush      aComponent
apush      aFocusEvent
invokevirtual    Component::processEvent:(LAWTEvent;)V
...
```

(b) Polymorphic call sites in bytecodes.

Figure 5: Polymorphic Call-Sites - two views

the methods defined for the precise receiver argument type, scan for an exact match for the name and signature. If one is not found, search the superclass[8] of the receiver argument, continuing up the superclass chain until `Object`, the root of the type hierarchy, is searched. If an exact match is not found, throw an `AbstractMethodError`. This look-up process applies to each of the `invoke` bytecodes.

This look-up process is a time-intensive operation. To reduce the overhead of method look-up, the resolved method is cached into the constant pool alongside the original method reference. The next time this method reference is applied by another `invoke` bytecode, the cached method is used directly.

Once a method is resolved, a method-specific *invoker* is executed to begin the interpretation of the new method. This invoker performs method-specific operations, such as acquiring a lock in the case of `synchronized` methods, constructing a JVM activation record in the case of bytecode methods, or preparing a machine-level activation record for `native` methods.

The Research JVM recognizes a special case in invoking methods: any private methods, final methods, or constructors can be handled in a *nonvirtual* mode. Clearly, each of these situations do not require dynamic dispatch. But, multi-dispatch will need to handle these cases specially.

[8]Java provides only single inheritance.

3 Design

We now have sufficient information to describe the general design for extending the JVM to support multi-dispatch. In short, we mark classes which are to use multi-dispatch, and replace their method invokers with one that selects a more specific method based on the actual arguments. Hence, existing uni-dispatch method invocations are unchanged in any way.

Marking the `.class` files without changing the language syntax is straightforward. We created an empty interface `MultiDispatchable` and any class which will provide multi-dispatch methods must implement that interface. The `.class` file retains that interface name, and the virtual machine can easily check for this at class loading time. Our implementation does not change the syntax of the Java programming language or the binary `.class` file format in any way.

This allows us to retain compatibility with existing programs, compilers, and libraries. Certainly, any class that does implement our marker interface has different semantics for dispatch. But, that does not change the semantics of existing single-dispatch programs or libraries. The programmer retains complete responsibility for designating multi-dispatchable classes. This allows the developer to consciously target the multi-dispatch technique to known programming situations, such as double dispatch.

At multimethod invocation, our multimethod invoker executes instead of the original JVM invoker. Our invoker locates a more-precise method based on the types of the invocation arguments, and executes it in place of the original method.

The *non-virtual* mode invocations need to be handled specially. Constructors are never multi-dispatched. We found that constructor chaining within a class could cause infinite loops. Private and final multimethods are still multi-dispatched.

We implemented two different dispatch techniques. The first one, *MSA* implements a dynamic version of Java Most Specific Applicable algorithm used by the `javac` compiler. The other technique, *SRP* [17], is a high performance table-based technique developed in our research group at the University of Alberta. We examine a framework-based system and a tuned implementation. We defer a detailed look at the implementations, in order to present the results of our experiments. For implementation details, please refer to section 6.

4 Experimental Results

So far, we have performed four different micro-benchmarks, and tested against a multi-dispatch implementation of Swing/AWT.

The first benchmark experiment demonstrates backward compatibility and minimal overhead for uni-dispatch applications, by running the `javac` compiler on the multi-dispatch virtual machine, and using it to recompile itself. The other three tests exercise multi-dispatch, and demonstrate multi-dispatch correctness, comparison to double dispatch, and multi-dispatch performance as arity increases. None of the benchmarks use more than one thread.

For our application-level tests, we modified Swing to use multiple-dispatch. We also converted AWT, because Swing depends heavily on AWT to dispatch the events into top-level Swing components.

All experiments were executed on a dedicated intel-architecture PC equipped with two 550MHz Celeron processors, a 100MHz front-side bus, and 256 MB of memory. The operating system is Linux 2.2.16 with `glibc` version 2.1. The Sun Linux JDK 1.2.2 code was compiled using GNU C version 2.95.2, with optimization flags as supplied by Sun's makefiles.[9]. The table-based multi-dispatch code [23] was compiled using GNU G++ version 2.92.2, with options `-ansi -fno-implicit-templates -fkeep-inline-functions -O2`. The Sun JDK only supports the `green` threading model, constructed on the Linux implementation of pthreads.

We report average and standard deviations for 10 runs of each benchmark.

We tested three different virtual machines. The first JVM, *jdk* is the standard JDK 1.2.2 Linux runtime, running without the JIT compiler. This VM serves as a baseline for comparing the remaining four multi-dispatch systems. The three multi-dispatch virtual machines *jdk-MSA*, and *jdk-SRP*, and *jdk-NSRP*, differ only in the dispatch technique. None of the four multi-dispatch virtual machines use a JIT compiler. For the first two experiments (Tables 2 and 3) we report user+system time in seconds, along with normalized values against the *jdk* runtime. For the third and fourth experiments (Table 4 and Figure 7), we describe individual dispatch times in microseconds, ignoring other costs. In the final benchmark, Swing, we report execution times for a synthetic application that creates a number of components and inserts 200,000 events into the event queue.

[9]Typical flags are `-O2`

4.1 Javac - Compatibility Test

The first experiment requires the runtime to load and execute the `javac` compiler to translate the entire `sun.tools` hierarchy of Java source files into `.class` files. This hierarchy includes 234 source files encompassing 49,798 lines of code (excluding comments). Each compilation was verified by comparing the error messages[10] and by checksumming the generated binaries. Each virtual machine passed the test; the timing results are shown in Table 2. These times came from the Unix `time` user command, and provide an average and standard deviation from 10 runs.

JVM	Time in sec.	(σ)	Norm.
jdk	65.41 + 0.25	(0.39)	1.00
jdk-MSA	67.38 + 0.31	(0.14)	1.03
jdk-SRP	68.22 + 0.45	(0.25)	1.05
jdk-NSRP	67.13 + 0.51	(0.35)	1.03

Table 2: Compatibility Testing and Performance (User+System Time to Recompile `sun.tools` in seconds)

The negligible differences between the uni-dispatch and multi-dispatch execution times clearly demonstrate that the overhead introduced by adding the simple check for multi-dispatch is essentially zero. Please note that in our implementation, table-based JVMs do not construct a dispatch table until the first multi-dispatchable method is inserted.

4.2 Simple Multi-Dispatch

In this example, we show that multi-dispatch is occurring, and roughly measure its overhead. The testing code is short, and is shown in Figure 6. The compiler uses static dispatch to code all four calls to `MMDriver.m(`*X*`,`*X*`)` to execute the method for two arguments of type `A`, because that is the static type of both `anA` and `aB`. Multi-dispatch actually selects among the four methods based upon the dynamic types of the arguments. Therefore, correct output consists of 100,000 repetitions of four consecutive lines: `AA`, `AB`, `BA`, and `BB`. For timing purposes, all output was redirected to `/dev/null` to reduce the impact of input/output. Our results are summarized in Table 3.

The table-based techniques suffers from a substantial startup time, whereas the MSA technique primarily uses existing data structures found in the Java Virtual Machine, and lazily computes any additional values. This reduces the cost of program startup.

```
class A { }
class B extends A { }

class MMDriver implements MultiDispatchable {
 String m(A a1, A a2) { return ‘‘AA’’; }
 String m(A a1, B b2) { return ‘‘AB’’; }
 String m(B b1, A a2) { return ‘‘BA’’; }
 String m(B b1, B b2) { return ‘‘BB’’; }

 static public void main(String args[]) {
  final int LOOPSIZE = 100000;
  A anA = new A();
  A aB = new B();
  MMDriver d = new MMDriver();

  for( int i=0; i<LOOPSIZE; i++) {
   System.out.println(d.m(anA, anA));
   System.out.println(d.m(anA, aB));
   System.out.println(d.m(aB, anA));
   System.out.println(d.m(aB, aB));
  }
 }
}
```

Figure 6: Simple Multi-Dispatch Testing Code

JVM	Time in sec.	(σ)	Norm.	Correct
jdk	26.40 + 0.68	(0.07)	1.00	No
jdk-MSA	28.88 + 0.83	(0.22)	1.10	Yes
jdk-SRP	31.53 + 0.91	(0.11)	1.20	Yes
jdk-NSRP	29.48 + 0.84	(0.17)	1.12	Yes

Table 3: Simple Multi-Dispatch (User+System Execution Time in seconds)

4.3 Double Dispatch

Our third experiment involves computing the performance differences between double dispatch, and the two multi-dispatch implementations of the example given in Figure 2. We constructed a type hierarchy of `AWTEvent` classes, to match those in Figure 2.

We also constructed three different component types:

1. `DD` uses double dispatch to implement `processEvent(AWTEvent)` as shown in Figure 2.
2. `MD` uses Multi-Dispatch to implement `processEvent(AWTEvent)` as shown in Figure 2(b).
3. `FMD` uses Full Multi-Dispatch to implement `processEvent(AWTEvent)` as in Section 1.1.

[10]One warning, noting that 8 files used deprecated APIs.

It separates `MouseEvent` into two different classes: `MouseButtonEvent` and `MouseMotionEvent`. FMD avoids the `switch` statement entirely.

To avoid inlining effects, we added code for updating an instance variable to the body of each `processEvent(AWTEvent)`. This test consists of dispatching a total of one million events through `processEvent(AWTEvent)`. Each event type appears equally often, as we iterate over an array containing one of each event. We compute the loop overhead using an empty loop, and subtract it, and then divide the total elapsed time by the number of events dispatched. The timing results are shown in Table 4.

Dispatch	DD		MD		FMD	
JVM	Time	(σ)	Time	(σ)	Time	(σ)
jdk	0.91	(0.00)	—		—	
jdk-MSA	0.95	(0.00)	3.26	(0.03)	2.90	(0.02)
jdk-SRP	0.96	(0.01)	3.12	(0.08)	2.52	(0.05)
jdk-NSRP	0.95	(0.00)	1.13	(0.02)	0.90	(0.01)
nolock	0.95	(0.00)	0.85	(0.01)	0.60	(0.00)

Table 4: Event Dispatch Comparison
(Call-Site Dispatch Times in microseconds)

Also, we give an additional timing value for our custom SRP implementation, where we disabled mutual exclusion in the dispatcher. Currently our implementation uses a costly monitor to ensure that no other thread is updating the dispatch tables during a multi-dispatch. High-performance concurrent-reader exclusive-writer protocols can eliminate this overhead; the nolock value approximates multi-dispatch in this highest-performance case.

As `DD` does not declare itself multi-dispatchable, the similarity of the results in column 2 of Table 4 again shows that our multi-dispatchable virtual machines do not penalize uni-dispatch code. Further, we see that the cost of interpreting numerous JVM bytecodes followed by another `invokevirtual` (which is `DD`'s strategy) is almost as costly as our multi-dispatch techniques. The full multi-dispatch implementation (`FMD`) is faster than the partial multi-dispatch (`MD`) and is just as fast as double-dispatch (`DD`). This is reasonable, because `MD` ends up double dispatching two of every six events.

Again, we see that the framework-based SRP technique suffers from considerable initial overhead. We hypothesize that it is a result of the object-oriented nature of our implementation of the table-based techniques. In each dispatch, several C++ objects are created and destroyed. Our native SRP implementation removes this overhead, and provides dispatch performance equal to programmer-coded double dispatch.

4.4 Arity Effects

Our final micro-benchmark explores the time penalties as the number of dispatchable arguments and applicable methods grow. To do this, we built a simple hierarchy of five classes (one root class `A`, with three subclasses `B`, `C`, and `D`, and finally class `E` as a subclass of `C`) and constructed methods of different arities against that hierarchy. We defined the following methods:

- classes `A`, `B`, `C`, `D`, and `E` contain unary methods *R*`.m()` (where R represents the receiver argument class).
- classes `A`, `B`, `C`, `D`, and `E` also implement five binary methods, *R*`.m(`*X*`)` where X can be any of `A`, `B`, `C`, `D`, or `E`.
- classes `A`, `B`, `C`, `D`, and `E` implement 25 ternary methods, *R*`.m(`*X*`,`*Y*`)` where X and Y can be any of `A`, `B`, `C`, `D`, or `E`.
- classes `A`, `B`, `C`, `D`, and `E` implement 125 quaternary methods, *R*`.m(`*X*`,`*Y*`,`*Z*`)` where X, Y, and Z can be any of `A`, `B`, `C`, `D`, or `E`.

MSA looks at one fewer dispatchable arguments than the table-based techniques. This is because the receiver argument has already been dispatched by the JVM. For instance, given a unary method, MSA makes no widening conversions for dispatchable arguments. A binary method requires MSA to check only one widening conversion. The table-based techniques dispatch on all arguments, and gain no benefit from the dispatch done by the JVM.

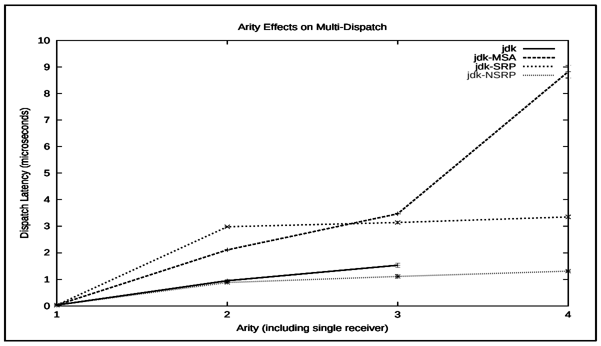


Figure 7: Impact of Arity on Dispatch Latency

Stage	Uni-Swing Methods	Multi-Swing Uni-Methods	Multi-Swing Multi-methods	
warm-up	901,938	901,795	160	(0.02%)
event loop	32,543,684	27,807,327	2,350,172	(7.7%)

Table 5: Swing Application Method Invocations

We invoke one million methods for each arity. This means that each of the unary methods is executed 200,000 times, However each of the quaternary methods is executed only 1,600 times. After computing the loop overhead via an empty loop, we determine the elapsed time to millisecond accuracy, and determine the time taken for each dispatch. Our results are shown in Figure 7.

We simulate the arity effects in the uni-dispatch case by coding a third level of double-dispatch. Already the overhead of constructing a third activation record exceeds the dispatch time of our native SRP implementation. Also, our SRP implementations suffer only linear growth in time-penalties as arity increases, whereas MSA suffers quadratic effects.

4.5 Swing and AWT

Our final test is to apply multi-dispatch to AWT and Swing applications. To do this, we needed to rewrite AWT and Swing to take advantage of multi-dispatch. We modified 11% (92 out of 846) of the classes in the AWT and Swing hierarchies. We eliminated 171 decision points, but needed to duplicate 123 into the specialized methods. Within the modified classes, we removed 5% of the conditionals, and reduced the average number of choice points per method from 3.8 to 2.0 per method. This reduction illustrates the value of multi-dispatch in reducing code complexity.

In all, 57 classes were added, all of them new event types to replace those previously recognized only by a special type id (as in the AWT examples described previously). Our multiple-dispatch libraries are a drop-in replacement that executes a total of 7.7% fewer method invocations, and gives virtually identical performance with applications such as `SwingSet`. In our sample application, we found that the number of multi-dispatches executed almost exactly equaled the total reduction in method invocations. This suggests that every multi-dispatch replaced a double-dispatch in the original Swing and AWT libraries.

We verified operation of the entire unmodified `SwingSet` application with our replacement libraries. Finally to measure performance, we timed a simple Swing application that handles 200,000 `AWTEvents` of different types. The timing results are given in Table 6. (jdk-SRP values are not given because the framework-based system does not support all of the dispatch features listed above).

The Swing and AWT conversion also demonstrates the robustness of our approach. We needed to support multi-dispatch on instance and static methods. Swing and AWT expect to dispatch differently on `Object` and array types. In modifying the libraries, we found numerous situations to apply multi-dispatch to private, protected, and super method invocations. In addition, several multimethods required the JVM to accept co-variant return types from multimethods. All of these features are required for a mainstream programming language.

Dispatch	Uni-Swing		Multi-Swing	
JVM	Time	(σ)	Time	(σ)
jdk	28.03	(0.35)	—	
jdk-MSA	28.69	(0.31)	70.09	(0.15)
jdk-NSRP	29.33	(0.42)	28.30	(0.36)

Table 6: Swing Application Execution Time
(Event loop times in seconds)

5 Multi-Dispatch Issues

Besides performance and correctness, multi-dispatch must contend with a number of serious difficulties which the `javac` compiler cannot recognize. They are: ambiguous method invocations caused by inheritance conflicts, incompatible return type changes, masking of methods by primitive widening operations, and null arguments. Each of these is illustrated in Figure 8. We have developed a tool called `MDLint` that can identify these problems and warn the programmer.

The first difficulty is that multi-dispatch, even in a single-inheritance language, can suffer from ambiguous methods. The two examples using the `m1` methods illustrate this. For the first method invocation, the compiler knows that `A.m1(B)` and `B.m1(A)` are candidates. Neither one is more specific than the other, so the compiler aborts with an error. We can fix that by statically typing the receiver argument to `A`, but multi-dispatch sees exactly the same conflict at runtime. Our `MDLint` program warns about the problem. If the programmer disregards the warning, our JVM detects the error and throws an `AmbiguousMethodException`.

The second difficulty centers around the fact that `javac` considers methods with different argument types as distinct. This means that they can have different return types. Multi-dispatch forges additional connections based on the additional dispatchable arguments. This means that methods which `javac` considered distinct are now overriding each other. In the example, we see that the two `m2(...)` methods override each other for multi-dispatch. Our multi-dispatch implementations throw an `IllegalReturnTypeChange` exception, unless the more specific method returns a subtype of the original returned value..

Throwing a runtime exception may not appear as an elegant, nor acceptable solution. But, one of the key attributes of Java is to maintain security. A malicious programmer can separately compile each class so that the errors are not evident until execution. The Java virtual machine must protect itself from these unruly possibilities, and throwing an exception remains as the only option. As we noted, our `MDLint` tool can recognize and report potential ambiguities and return type conflicts at compile time.

The third difficulty involves the use of literal null as an argument. If null is typed, as in the first example, then `javac` performs its static multi-dispatch with that type. This restricts its set of applicable methods. In our example, ordinary Java can avoid loading class `C`. Multi-dispatch Java loads class `C` and dispatches to the `m3(C)` method.

The null argument problem is an example of a more general problem in Java. Inconsistent invocations can occur when expressions are substituted in place of variables. This is because `javac` can obtain and use more precise type information. As an example, compare the execution of the second and third invocations of `m3(...)`. By replacing `Ab` with its value, we have altered the execution of a program.

The last difficulty is more complex; and, at this time, unsolved. The compiler selects a method based upon widening operations, and may change the type of primitive arguments. In the example, the compiler inserts instructions to convert `b` from a `byte` to an `int`. At runtime, we have lost all traces that `b` was originally specified as a `byte`. Indeed, the programmer might have wanted to force that exact conversion; the bytecodes would be identical to compiler-generated conversions.

```
class A {
 void m1(B b1) {...}
 void m4(int i) {...}}
class B extends A {
 void m1(A a1) {...}
 void m4(byte b) {...}}

class C extends B { }

class Driver {
 int m2(A a1, A a2) {...}
 String m2(B b1, B b2) {...}
 void m3(A a1) {...}
 void m3(B b1) {...}
 void m3(C c1) {...}

 public static void main(String args[]) {
  A Ab = newB(); // static:  A, dynamic:  B
  B Bb = newB():  // static:  B, dynamic:  B

  // multi-dispatch difficulties
  Bb.m1(Bb); // javac:  ambiguous method
  Ab.m1(Bb); // javac:  OK, MDJ: ambiguous

  // incompatible return type change
  int i = m2(Bb, Bb); // javac:  bad return type
  int i = m2(Ab, Ab); // javac:  no error
  int i = m2(Ab, Ab); // javac:  OK, MDJ: runtime error

  // null arguments are more consistent
  A a = null;
  m3.(a); // regular Java executes m3(A)
       // MDJ loads C, executes m3(C)
  m3.(null); // both execute m3(C)
  m3(Ab); // executes m3(A)
  m3(new B()); //replace variable with value

  // primitive widening hides correct method
  byte b = 7;
  Ab.m4(b); // javac:  widens, calls A.m4(int)
       // MDJ: ignores B.m4(byte)
  Ab.m4(int(b)); // programmer widening }
```

Figure 8: Examples of Multi-Dispatch Issues

6 Implementation

In this section, we describe how the Java Virtual Machine was extended to support dynamic multi-dispatch. We begin by examining how to indicate which classes are multi-dispatchable by the JVM. We then examine where multi-dispatch must occur, and finally we review the three different multi-dispatch implementations.

6.1 Marking Classes as Multi-Dispatchable

Before we begin examining an actual multi-dispatch, we need to tell the system that multi-dispatch is required. We do this on a class-by-class basis, by implementing an empty interface, `MultiDispatchable` in each class that is multi-dispatchable. The Java programming language has already leveraged this idea for marking class capabilities with the `Cloneable` interface. We use the `MultiDispatchable` interface to denote that any method sent to a multi-dispatch receiver should be handled by the multi-dispatcher. For efficiency, we add a flag to the internal class representation to indicate that a class is multi-dispatchable, rather than searching its list of interfaces at each method invocation. The value of this flag is set once, at class load time.

Our selection of `MultiDispatchable` as the marker clearly requires us to support multi-dispatch on a class-by-class basis, not on a method-by-method or argument-by-argument basis. That is, every method invocation where the uni-dispatch receiver is a member of a multi-dispatchable class goes through our multi-dispatcher. Furthermore, because interfaces are inherited, this approach requires any subclass of a multi-dispatchable class to also be multi-dispatchable. Most importantly, any method invocation where the receiver argument is not marked for multi-dispatch continues unchanged through the uni-dispatcher. The syntax of Java programs is unchanged, and the performance and semantics of uni-dispatch remains intact.

6.2 Adding Multi-Dispatch

Uni-dispatch of an `invoke` bytecode provides us with a method pointer from the array of methods in the receiver argument class. At this point, the interpreter loop is about to build a new frame to execute the found method. The interpreter loop (and classic VM JIT compilers) proceed to call a special function, called the `invoker` that handles the details of building the new frame and starting the new method. The Research JVM uses different invokers for native, bytecode, synchronized, JIT-compiled, and other method types. Similar to the OpenJIT system[22], we replace this invoker function with our own custom one that computes the correct multi-dispatch method. Once the more precise method is known, we simply invoke it directly.

The multi-invoker is installed at class-load time. The interpreter loop and invoker for uni-dispatch are unchanged. This supports our claim that single-dispatch programs and libraries suffer no execution time penalties.

We have experimented with three different multi-dispatch techniques; they are examined in the succeeding sections. For each technique, we also describe our solution for the implementation issues described in section 5.

6.3 Reference Implementation: *MSA*

Our reference implementation is an extension of the Most Specific Applicable algorithm described in section 15.11 of *The Java Language Specification* and in section 2.2 of this paper. In particular, we reexamine the steps described in section 2.2 in light of the dynamic types being available.

When the multi-invoker is called, we already have a `methodblock` that the compiler specified and the uni-dispatch resolution mechanism has found. We also have the top of the operand stack, so we can peek at each of the arguments as well. Last, we have the actual receiver, which can provide the list of methods (including inherited ones) that it implements.

Every method is represented by a `methodblock` containing many useful pieces of information. First, it holds the name of the method. Next, it contains a handle to the class that contains this method[11]. Third, it contains the signature which we can parse to get the arity and types of the dispatchable arguments. For performance, we parse the signature only once. We add two fields to the `methodblock`: `int arity` to cache the arity, and `ClassClass **argClass` to hold the class handles for the dispatchable arguments.

With these three pieces of information, we implement a dynamic version of the MSA algorithm directly. Wherever the original algorithm would use the static type of an argument, we apply the known dynamic type instead. In step 2(b) from section 2.2, the compiler would compare the static type of each argument with the corresponding declared type for the candidate method. In the dynamic case, we have the arguments on the stack, so we can find their dynamic types. We compare each argument's dynamic type against the declared type of the corresponding argument of the method. We discard any method whose declared types do not match the arguments on the stack. The remaining methods are *dynamically applicable*.

The issue of null-valued arguments becomes significant at this point. JLS chapter 4 recognizes the need for a *null type* to represent (untyped) null values. It further declares in section 4.1 that the null type can be coerced to any non-primitive type. Fur-

[11]Recall that methods might be inherited; this class handle is the original implementing class.

ther, section 5.1.4 allows null types to be widened to any object, array or interface type. Statically, this means that an (untyped) null argument can be widened to any class. In the dynamic case, we want to do the same. Therefore, whenever we encounter a null argument we accept the conversion of that null to a method argument of type class, array, or interface.

Unfortunately, if we have a null argument, we may retain a method which accepts arguments of classes that are not yet loaded. We need to force these classes to be loaded to ensure that the next step operates correctly.

Given the list of applicable methods, step 2(d) finds the unique most specific method. Again the operation is identical to the process the `javac` compiler executes. One applicable method is selected as tentatively the most specific. Each other applicable method is tested by comparing argument by argument (including the receiver argument) against the tentatively most specific. At each step, we discard any methods that are less specific. We continue this process until only one candidate method remains, or two or more equally specific methods remain. In the latter case, we have an ambiguous method invocation, and we throw an `AmbiguousMethodException` to advertise this fact.

Next, we verify that the return type for our more specific method is compatible with the compiler-selected method. This check relaxes JLS 8.4.6.3, where we must reject any invocation that has a different return type, yet ensures type-safety. If the return type is different, we throw an `IllegalReturnTypeChange` exception at runtime.

6.4 Table-based Dispatch

Our SRP framework-based techniques is taken from the Dispatch Table Framework (DTF) [23]. This is a toolkit of many different uni-dispatch and multi-dispatch techniques. In order to call the DTF to dispatch a call-site, we need to inform the DTF of the various classes and methods present in our Java program. Our interface consists of a number of straight-forward routines to perform this registration.

The JVM maintains in-memory structures for each loaded `.class` file. We have extended that `ClassClass` structure to contain a `DTFType` field. It contains a pointer to the C++ object generated by the DTF. Once a class is dynamically loaded by the JVM, we check to see if we must register it with the dispatcher. If the dispatcher has already been instantiated, we register the class via `javaAddClass(...)` and store away the returned `DTFType` pointer.

If a dispatcher has not been instantiated, and the just-loaded class is uni-dispatch only, we defer the registration in order to reduce the overhead to uni-dispatch programs. If the just-loaded class is marked for multi-dispatch and the dispatcher has not been instantiated, the process is more complex. First, we instantiate a new dispatcher. Then, we register each class that has already been loaded, ensuring that its superclasses and superinterfaces are registered first.

Finally, as the last part of registering a class with the dispatcher, we need to see whether any methods from other classes were held in abeyance until this class was loaded. This can occur if the methods from other classes expect dispatchable arguments of the class we're just now loading. As we shall see below, we deferred registering these methods until the class was loaded.

Java's facility for dynamically reloading classes forces us to ensure that two classes with the same name are assigned different `DTFTypes`. Java ensures that two classes with the same name are treated as distinct by insisting that each one is loaded by a different classloader [19]. We apply the same technique by supplying the DTF framework with a name consisting of the classloader name, followed by "`::`" and followed by the class name. They system classloader is given the empty name " ".

For a class marked for multi-dispatch, we need to register its methods along with their types, via `javaAddMethod(...)`. If this class implements `MultiDispatchable` directly, then we register all of its methods, including inherited ones. Alternately, if `MultiDispatchable` is an inherited interface for this class, then we know that its superclass has already registered its methods. Therefore, we don't need to register them, we only need to register the methods that we directly implement.

This method registration process is complicated by our desire to lazily-load classes. If a method accepts an argument with a class not yet seen by the JVM, we know that we could never dispatch to it until that class is loaded[12]. Therefore, we set aside that method for future registration.

If all of the argument types for the method are al-

[12] As mentioned above, our DTF-based systems do not permit null as a dispatchable argument. Therefore, this guarantee holds.

ready registered with the DTF, then we proceed to register the method. We provide a `methodblock` pointer that we want the framework to return if this method is the dispatched target. We bundle up the `DTF_Type` values found in the `ClassClass` structures for each argument class (including the receiver argument) and pass them to the framework. The framework replies with a `DTF_Behavior` pointer that we store in the `methodblock`.

Dispatch becomes a very simple operation. We build an array of the `DTF_Type` pointers from the arguments on the Java stack. If we encounter a null argument, we throw a `NullPointerException`. The `DTF_Type` array, along with the `DTF_Behavior` pointer from the compiler-selected method allow the framework to locate the `methodblock` pointer that we had previously registered.

We expect that the returned `methodblock` pointer is the method for multi-dispatch. We validate it against the compiler-selected method. If the return type has changed, we abort the dispatch and throw an `IllegalReturnTypeChange` exception. Otherwise, we call the found method's original invoker and return its value as the result of the interpreter's call to a method invoker.

Single Receiver Projections Single Receiver Projections [16] is a technique that considers a multi-dispatch as a request for the joint most specific method available on each argument. For a given argument position and type, an ordered (most-specific to least-specific) vector of potential methods is maintained. The vectors for all the argument positions are intersected to provide an ordered vector of all applicable methods. Because of the ordering, this vector can be quickly searched for the most applicable method.

SRP uses a uni-dispatch technique to determine the vector of potential methods for each individual argument. Many different techniques are known: row displacement, selector coloring [2], and compressed selector table indexing [26]. Our implementation uses selector coloring, because timing experiments [17] indicates that technique provides the fastest dispatch times.

7 Future Work

Our MSA implementation is the most complete. It supports

1. `null` as a dispatchable argument,
2. multi-dispatch on static methods[13].
3. primitive dispatchable arguments and widening thereof,
4. multi-threaded dispatch.

Our table-framework-based dispatchers do not currently support these facilities. Adding them would provide additional flexibility, and allow them to fully support the Java programming language semantics. In particular, we have a two-table design that will allow one thread to dispatch through an existing table, while we register additional methods and/or classes to a new one.

Our custom SRP system implements multi-dispatch as a critical section, protected by a mutual-exclusion lock. We have a technique which would eliminate this overhead (approximately 0.38 μs.for every multi-dispatch) and allow concurrent multi-dispatch. The penalty is that every thread would need to halt while the multi-dispatch tables are being updated.

Other multi-dispatch techniques exist, including compressed n-dimensional tables [1, 12], look-up automata [10, 11], and efficient multiple and predicate dispatch [7]. A comprehensive exploration applying these techniques to Java is incomplete at this time.

Another significant improvement for multi-dispatch is to incorporate our code testing tool into the `javac` compiler. At this time, `MDLint` exists as a separate executable which will recognize and warn the programmer about common ambiguities and difficulties. It analyzes a complete application and identifies the code sections where the programmer could invoke an ambiguous method, or have a conflicting return type.

We support multi-dispatch on all method types (`instance`, `static`, `interface`, `private`, etc.), except constructors. Because the same bytecode is used to invoke a constructor in the superclass and a constructor with a wider type, we cannot distinguish the two possibilities. This issue is a specific instance of the need to apply a `super` to an argument other than the single-receiver. Fortunately, in our experience, this requirement does not arise in common programming practice (except for constructors).

Our native SRP implementation allows our dispatch tables to identify only those types that are are multi-dispatched. This *lazy type numbering* is reversible, allowing the tables to shrink as classes are unloaded.

[13]signaled by implementing `StaticMultiDispatchable`.

In turn, multimethods can revert to lower arity multi-dispatch (or even single-dispatch). We see great promise in this technique for long-lived Java server applications.

The DTF framework contains another dispatcher, *Multiple Row Displacement* [23] (MRD) that operates 15% faster than SRP. Therefore, we expect that dispatch could be enhanced to provide even lower latency by applying this technique. Unfortunately, MRD currently does not support incremental dispatch table updates in the same way that SRP does. In a dynamic environment such as Java, incremental updating of dispatch tables is desirable. Enhancing MRD to support incremental updates is another research priority.

8 Related Work

Others have attempted to add multi-dispatch to Java through language preprocessors. Boyland and Castagna [3] provide an additional keyword *parasite* to mark methods which should have multi-dispatch properties. They effectively translate these methods into equivalent double dispatch Java code. By translating directly into compiled code, they apply a textual priority to avoid the thorny issue of ambiguous methods. Unfortunately, the parasitic method selection process is a sequence of several dispatches to search over a potentially exponential tree of overriding methods.

The language extension and preprocessor approach has other limitations. First, existing tools do not support the extensions; for example, debuggers do not elide the automatically generated double-dispatch routines. Second, instance methods appear to take objects only, which is too limiting. Our experience with Swing shows that existing programs often double-dispatch on literal `null` and array arguments and pass primitive types as arguments; multi-methods need to support these non-object types. Third, preprocessors limit code reuse and extensibility; adding multi-methods to an existing behaviour requires either access to the original source code or additional double-dispatch layers.

David Chatterton [8, 9] examines two different multi-dispatch techniques in mainstream languages: C++ and Java.. First, he considers providing a specialized dispatcher class. Each class that participates as a method receiver must register itself with the dispatcher. To relieve the programmer of this repetitive coding process, he provides a preprocessor that rewrites the Java source to include the appropriate calls. Each method, marked with the keyword *multi* is also expanded by the preprocessor into many individual methods, one for each combination of classes (and superclasses). A method invocation is replaced by a call to the dispatcher which searches via reflection for an exact match. That method is then invoked. This system also suffers from exponential blowup of methods.

His second approach examines the performance of various double dispatch enhancements. He provides a modified C++ preprocessor which analyses the entire Java program. It can construct a number of different double dispatch structures, including cascaded and nested `if-elseif-else` statements, inline `switch` statements, and simple two-dimensional tables. Again, he expands every possible argument type combination, in order to apply fast equality tests, rather than slow subtype tests. A significant restriction is that full program analysis is required. This defeats the ability to use existing Java libraries, and eliminates Java's dynamic class loading features.

One interesting language for multi-dispatch is Leavens and Millstein's *Tuple* [18]. They describe a language "similar in spirit to C++ and Java" that permits the programmer to designate the individual arguments that will be considered for multi-dispatch, on a callsite-by-callsite basis. His paper does not describe an implementation; it appears to be a model of potential syntax and semantics only. A future project might be to implement his syntax specifically into the Java environment. In particular, a simple syntax extension would allow `super` method invocations on arbitrary multi-dispatch arguments.

9 Concluding Remarks

We have presented the design and implementation of an extended Java Virtual Machine that supports multi-dispatch. This is the first published description of how to implement arbitrary-arity multi-dispatch in Java. In contrast to the more verbose and error-prone double dispatch technique, currently found in the AWT (Figure 2), multi-dispatch typically reduces the amount of programmer-written code and generally improves the readability and level of abstraction of the code.

Our approach preserves both the performance and semantics of the existing dynamic uni-dispatch in Java while allowing the programmer to select dynamic multi-dispatch on a class-by-class basis without any language or compiler extensions. The changes to the JVM itself are small and highly-localized. Existing Java compilers, libraries, and programs are not affected by our JVM modifications and the programs can achieve performance comparable to the original JVM (Table 2).

In a series of micro-benchmarks, we showed that our prototype implementation adds no performance overhead to dispatch if only uni-dispatch is used (Table 2) and the overhead of multi-dispatch can be competitive with explicit double dispatch (Table 4).

We have also introduced and implemented an extension of the Java Most Specific Applicable (MSA) static multi-dispatch algorithm for dynamic multi-dispatch. In addition, we have performed the first head-to-head comparison of table-based multi-dispatch techniques implemented in a mainstream language. In particular, we implemented Single Receiver Projections (SRP). Overall, our tuned SRP implementation performs as well (or better) than programmer-targeted multi-dispatch. With performance improvements in concurrency, we expect our tuned system to out-perform double dispatch.

References

[1] E. Amiel, O. Gruber, and E. Simon. Optimizing multi-method dispatch using compressed tables. In *OOPSLA 1994 Conference Proceedings*, pages 244–258. Association for Computing Machinery, October 1994.

[2] P. Andre and J. Royer. Optimizing method search with lookup caches and incremental coloring. In *OOPSLA 1992 Conference Proceedings*. Association for Computing Machinery, 1992.

[3] J. Boyland and G. Castagna. Parasitic methods: An implementation of multi-methods for Java. In *OOPSLA 1997 Conference Proceedings*, pages 66–76. Association for Computing Machinery, November 1997.

[4] K. Bruce, L. Cardelli, G. Castagna, The Hopkins Object Group, G. T. Leavens, and B. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995.

[5] T. Budd. *An Introduction to Object Oriented Programming, Second Edition.* Addison-Wesley Publishing Co., Reading, Massachusetts, 1997.

[6] C. Chambers. Object-oriented multi-methods in Cecil. In *ECOOP 1992 Conference Proceedings*, pages 33–56. Springer-Verlag, June 1992.

[7] C. Chambers and W. Chen. Efficient multiple and predicate dispatching. In *OOPSLA 1999 Conference Proceedings*, pages 238–255. Association for Computing Machinery, November 1999.

[8] D. Chatterton. *Dynamic Dispatch in Existing Strongly Typed Languages.* PhD thesis, School of Computing, Monash University, Monash, Australia, 1998.

[9] D. F. Chatterton and D. M. Conway. Multiple dispatch in C++ and Java. In *TOOLS '21 Asia*, pages 75–87, 1996.

[10] W. Chen. Efficient multiple dispatching based on automata. Master's thesis, GMD-ISPSI, Darmstadt, Germany, 1995.

[11] W. Chen, V. Turau, and W. Klas. Efficient dynamic lookup strategy for multi-methods. In *ECOOP 1994 Conference Proceedings*, pages 408–431. Springer-Verlag, July 1994.

[12] E. Dujardin, E. Amiel, and E. Simon. Fast algorithms for compressed multimethod dispatch table generation. *ACM Transactions on Programming Languages and Systems*, 20(1):116–165, January 1998.

[13] A. Goldberg and D. Robson. *Smalltalk-80 The Language and its Implementation.* Addison-Wesley Publishing Co., Reading, Massachusetts., 1983.

[14] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1996.

[15] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, 2nd Edition.* Addison-Wesley Publishing Company, Reading, Massachusetts, 2000.

[16] W. Holst, D. Szafron, Y. Leontiev, and C. Pang. Multi-method dispatch using single-receiver projections. Technical Report 98-03, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, 1998.

[17] W. M. Holst. *The Tension between Expressive Power and Method-Dispatch Efficiency.* PhD thesis, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, 2000.

[18] G. T. Leavens and T. D. Millstein. Multiple dispatch as dispatch on tuples. In *OOPSLA 1998 Conference Proceedings*, pages 244–258. Association for Computing Machinery, October 1994.

[19] S. Liang and G. Bracha. Dynamic class loading in the Java virtual machine. In *OOPSLA 1998 Conference Proceedings*, pages 36–44. Association for Computing Machinery, October 1998.

[20] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1996.

[21] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, 2nd Edition.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1999.

[22] H. Ogawa, K. Shimura, S. Matsuoka, F. Maruyama, Y. Sohda, and Y. Kimura. Openjit: An open-ended, reflective jit compile framework for java. In *ECOOP 2000 Conference Proceedings.* Springer-Verlag, 2000.

[23] C. Pang, W. Holst, Y. Leontiev, and D. Szafron. Multiple method dispatch using multiple row displacement. In *ECOOP 1999 Conference Proceedings*, pages 304–328. Springer-Verlag, June 1999.

[24] G. Steele. *Common Lisp.* Digital Press, Burlington, 1985.

[25] B. Stroustrup. *The C++ Programming Language: Third Edition.* Addison-Wesley Publishing Co., Reading, Massachusetts, 1997.

[26] J. Vitek and R. N. Horspool. Compact dispatch tables for dynamically typed programming languages. In *Proceedings of the International Conference on Compiler Construction*, 1996.