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

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

ABSTRACT

Mainstream object-oriented languages, such as C++ and Java, provide only a restricted form of polymorphic methods, namely single-receiver dispatch. In common programming situations, programmers must work-around this limitation. We detail 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, two table-based dispatchers (*Multiple Row Displacement* (MRD) and *Single Receiver Projections* (SRP)), and a tuned SRP dispatcher. Our general-purpose technique provides smaller dispatch latency than equivalent programmer-written double-dispatch code.

1. INTRODUCTION

Multiple-dispatch, where dynamic method selection depends on the types of more than one (and potentially all) arguments, is an area of considerable interest and current research. Dynamic method selection on multiple arguments arises naturally in common programming situations. Two examples are binary operations such as equality testing and event-based programming where actions depend on the dynamic type of both the component and event. The latter exemplifies the container problem where static method overloading (e.g., single-receiver dispatch in Java[1], and C++) is inadequate. 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.

Despite being studied for more than a decade, multi-methods have been limited to research object-oriented languages such as CLOS, Dylan, and Cecil [2]. Experience with these languages has shown that programs utilizing multi-methods are shorter, less error-prone to code, and more extensible. Unfortunately, multiple-dispatch has suffered from two drawbacks: dispatch efficiency and potential ambiguities. Modern multiple-dispatch techniques are more efficient and the ambiguities can be recognized during compilation [3]. Consequently, we re-examine the design and implementation of multi-methods within a production language (i.e., Java) and their application in multi-method versions of `Swing` and `AWT`.

Others have extended the Java language in order to support multi-methods [1]. A preprocessor accepts an extended version of Java with new multiple-dispatch keywords, and emits the double-dispatch equivalent in standard Java. The key advantage is that the system generates standard bytecodes which can execute on any virtual machine. But, the overhead of executing the double-dispatch bytecodes remain.

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.

2. AN EXTENDED JAVA VM

Our approach is to extend the Java Virtual Machine to perform multiple-dispatch directly. The programmer labels classes that require multiple-dispatch with a marker interface, `MultiDispatchable`. This technique of using an empty interface to mark special properties is accepted as part of the Java programming language – the `Cloneable` interface operates in the same way. We have not changed the syntax of Java in any way. Indeed, our multiple-dispatch programs are compiled by the existing `javac` compiler included with the Java Development Kit.

*{dutchyn,paullu,duane,bromling}@cs.ualberta.ca, Department of Computing Science, University of Alberta, Edmonton, AB, Canada, T6G 2E8

†wade@csd.uwo.ca, Department of Computer Science, The University of Western Ontario, MiddleSex College, London, ON, Canada, N6A 5B7

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

OOPSLA 2000 Companion Minneapolis, Minnesota

Within the Sun Microsystems Research VM[2] we intercept any multi-method definitions when classes are loaded, and replace the standard *invoker* function with a multi-invoker. The invoker is a VM routine which assists the interpreter to begin a new method by constructing the new activation record and acquiring any needed locks for synchronized methods. When a multi-method is called, our custom multi-invoker examines the method arguments on the stack, locates an alternate method that is specific to the arguments, and begins execution of this alternate method.

Single-dispatch classes and methods do not have their invoker changed, and therefore we impose no penalty on their operation. The only impact our multiple-dispatch virtual machine applies to single-dispatch programs is to check for the marker interface at class load time.

We implemented three different method lookup techniques for the multi-invoker. First, *Most Specific Applicable* (MSA) is a dynamic version of the existing static method selection for Java. It serves as a reference platform to ensure we maintain compatibility with existing Java semantics. Next, Multiple Row Displacement (MRD) and Single Receiver Projections (SRP) [3] are implemented using a general-purpose dispatch table framework. Since there are overheads associated with using the framework, we also re-implemented SRP as a hand-coded dispatcher. In micro-benchmarks, this custom dispatcher reduces the latency of a multi-method dispatch to less than the equivalent double-dispatch code. Therefore, programmers can write multiple-dispatch programs and execute them *faster* than programmer-coded double-dispatch.

To complete our development environment, we created a simple ambiguity testing tool, `MDLint`, that reports on potentially ambiguous multi-methods. In keeping with the defensive and security-conscious nature of Java, we also ensure that our Java Virtual Machine recognizes and reports ambiguous dispatches by throwing a runtime exception.

3. RESULTS

A number of micro-benchmarks validate the correctness and performance of our system.Also, our experience with implementing multi-method versions of the `Swing` and `AWT` libraries show the robustness and benefits of full-featured multiple-dispatch in Java.

For example, we looked at an example double-dispatch from the `java.AWT.Component.processEvent()` method. Our custom SRP implementation requires only 0.90 μs to dispatch a binary multi-method (including 0.40 μs cconcurrency overhead), whereas the original double-dispatch requires 0.96 μs. As the parameter-arity increases, this disparity increases in favor of multiple-dispatch.

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. We modified 11% of the classes; we removed 5% of the conditionals and reduced the average number of choice points per method from 3.8 to 2.0 in the changed code. This reduction illustrates how multiple-dispatch reduces code complexity. Our multiple-dispatch libraries are a drop-in replacement that executes 7.7% fewer method invocations, and gives identical performance with applications such as `SwingSet`.

Our practical experience with `Swing` shows that an implementation of multiple-dispatch must be compatible with Java-specific language features. In particular, programs need to perform multiple-dispatch on literal `null` and array arguments, as well as accept primitive values as arguments to multi-methods. Object arguments alone are insufficient. Next, multiple-dispatch must support `instance`, `static`, and private multi-methods; `Swing` even applies `super` multi-methods. Finally, our implementation relaxes Java's rigid no-variant return type limitation and supports covariant specialization on return types within a behaviour.

[2]also known as the *classic* VM

4. FUTURE WORK

One limitation of our existing system is that it does not support JIT compilers. However, we have examined the OpenJIT system and it appears to be a straight-forward process to implement a multi-dispatch invoker using their JIT framework. A second key improvement is to remove locking from concurrent multiple dispatch. Benchmark timings suggest that we can reduce our dispatch latency by 40% with this one alteration. Third, we have identified additional places where our multiple-dispatch `Swing/AWT` implementations can be further re-factored. This will reduce the total amount of code, and may increase performance as well.

Last, our custom SRP implementation provides for *lazy type numbering* where new types are not placed into the dispatch tables until they are required for multiple-dispatch. We propose to extend this facility to class-unloading, so that methods revert to lower-arity multiple-dispatch (or even single-dispatch) whenever possible. We see great potential for this technique in long-lived Java server applications.

5. CONCLUDING REMARKS

The primary research contribution of this work is the design and implementation of an extended Java Virtual Machine that supports general-purpose multiple-dispatch with better performance than double-dispatch. Single-dispatch performance and semantics, as well as source and binary compatibility with existing class libraries, is maintained.

In contrast to other approaches, our implementation requires no changes to the Java syntax or compiler. Our multiple-dispatch also provides the full range of functionality required of systems such as `Swing` and `AWT`, including support for primitive values, `null`, and arrays. Also, multiple-dispatch with co-variant return types is permitted on instance, `static`, and `private` methods, and `super` method invocations.

Efficient, full-featured, compatible multiple-dispatch in a mainstream language such as Java reduces the barriers to adopting multi-methods in practice.

6. REFERENCES

[1] J. Boyland and G. Castagna. Parasitic methods: An implementation of multi-methods for Java. In *OOPSLA '97 Conference Proceedings*, pages 66–76. Association for Computing Machinery, November 1997.

[2] C. Chambers. Object-oriented multi-methods in Cecil. In *ECOOP '92 Conference Proceedings*, pages 33–56. Springer-Verlag, June 1992.

[3] C. Pang, W. Holst, Y. Leontiev, and D. Szafron. Multiple method dispatch using multiple row displacement. In *ECOOP '99 Conference Proceedings*, pages 304–328. Springer-Verlag, June 1999.