Object-Model Independence via Code Implants

Michał Cierniak[1], Neal Glew[2], Spyridon Triantafyllis[3], Marsha Eng[2], Brian Lewis[2], and James Stichnoth[2]

[1] Microsoft Corporation
[2] Microprocessor Technology Lab, Intel Corporation
[3] Department of Computer Science, Princeton University

Abstract. Managed runtime environments, such as those that execute Java or CLI programs, require close cooperation between the virtual machine (VM) and user code, which is usually compiled by a just-in-time compiler (JIT). Although operations such as field access, virtual method dispatch, and type casting depend on VM data structures, having user code call the VM for these operations is very inefficient. Thus, most current JITs directly generate code for these operations based on assumptions about the VM's implementation. Although this design offers good performance, it locks the VM and the JIT together, which makes modifications difficult. In addition, it is hard to experiment with new algorithms and strategies, which may ultimately hurt performance. Finally, extending a runtime platform to support new programming paradigms may require reimplementing the entire system.

We propose a mechanism that allows VMs to implant code into JIT-compiled code. This mechanism allows a strict interface between the JIT and the VM without sacrificing performance. Among other things, it isolates most programming-paradigm aspects within the VM, thus greatly facilitating multiparadigm support. This paper presents our system for code implants, gives an early evaluation of it, and describes how it could be used to implement several programming-paradigm extensions to Java with relatively little implementation effort.

1 Introduction

The *Open Runtime Platform* (ORP [6, 1]) is a high-performance implementation of a virtual machine for Java[4] [15] and the Common Language Infrastructure (CLI [10]). ORP supports Java and CLI with essentially the same implementation, with only subsets of the implementation being Java or CLI specific. This dual support of both CLI and Java is, as far as we know, unique among virtual machines. It also raises the question of whether we can extend ORP to support other programming paradigms.

ORP uses interfaces to partition its implementation into several well-defined modules: the core virtual machine (VM), the just-in-time compiler (JIT), and the garbage collector (GC). ORP's modular design facilitates experimentation,

[4] Other brands and names are the property of their respective owners.

and allows using multiple JIT and GC implementations with the same core VM. To date we have used seven different JITs (see, *e.g.*, [2, 7, 5, 1]) and five different GCs.

However, portions of ORP's current interfaces sacrifice cleanliness for performance. For instance, the JIT can implement a type-inclusion test either as a call to the VM or as an inlined instruction sequence. While a call-based sequence is independent of the VM's implementation, its performance is slower. An inlined sequence can be very fast (*e.g.*, through the use of Cohen's type displays [8]), but it relies on specific VM data structures and may be incorrect if the VM data structures change. Furthermore, some code sequences are so performance-sensitive that the VM does not provide a function to call, and instead the JIT must rely on specific data structures. Virtual method dispatch is an example. ORP uses vtables for virtual method dispatch, and every JIT must understand this implementation and generate vtable-based code.

Many code sequences assume that ORP only supports single-inheritance languages like Java and CLI, and we never considered abstracting these sequences. For example, an upcast is a no-op in ORP, but common implementations of languages with multiple inheritance (*e.g.*, C++) require that an upcast add an offset to the object pointer. Similarly, for a field access in Java, the JIT can generate the address of the field by adding a compile-time constant to the object pointer. In languages with dependent multiple inheritance, an indirection through an *index table* may be required (see, *e.g.*, Grune *et al.* [13, Section 6.2.9.6]).

For ORP to support other languages such as Python or Scheme, we would have to change ORP's object model to include, for instance, multiple inheritance and possibly multi-methods. This would make its current use of vtables and type displays inappropriate, and would make today's ORP JITs generate incorrect code.

We would also like to experiment with supporting new programming paradigms in ORP. Due to the broad acceptance of the Java language, several Java extensions have been proposed to support additional programming paradigms. These extensions include both standard [14] and dynamic [4, 17] aspect-oriented programming, multiple inheritance, mixin-type inheritance [11], and aspects of functional programming [16, 20]. With the exception of standard aspect-oriented programming, only proof-of-concept implementations exist for the other extensions. More realistic implementations do not exist because of the great difficult of extending most existing Java systems. Even today's ORP would require changes to multiple components including the JITs to support many of these extensions.

This paper proposes a system for the VM to specify, in a CPU-independent fashion, low-level instruction sequences (*stubs*) that the JIT can then inline into the code it emits. This system, which includes the LIL language, simplifies the generation of the intricate code sequences needed to implement various language operations like a virtual method dispatch, a type-inclusion test, or a type cast. We argue that it can also support new object models and programming paradigms. It abstracts away details that depend on the object model or programming paradigm, and make JITs oblivious to the details of their implemen-

tation. We further argue that these benefits will be possible without sacrificing the performance of today's ORP. It has the additional benefit of simplifying maintenance for multiple CPU architectures.

This paper presents our results to date. We describe the LIL language in Section 2. To make our proposal concrete, we describe in some detail how to implement one operation, downcasts, using LIL in Section 3. We discuss how the VM and JIT interoperate to inline this code sequence into JIT-generated code, and present the results of an experiment to evaluate the performance impact from inlining this operation. Section 4 gives an overview of using this system for adding new programming paradigms to ORP with only modest implementation effort.

2 LIL

We designed a language called LIL[5] to express low-level code to be inlined in a CPU-independent way. This section gives a brief overview of LIL.

Here is an example of a LIL stub that invokes a virtual method on an object.

```
entry 0:managed:ref,g4,f4:g1;
locals 1;
ld l0,[i0+0:pint];
ld l0,[l0+32:pint];
in2out managed:g1;
call l0;
ret;
```

This stub is compiled into code that acts like a function. The stub's `entry` line states that it is called using the managed-code calling convention (i.e., the VM-specified convention for calling JIT-generated code) with three arguments, and that it returns a result. The arguments are of type `ref` (reference to an object in the heap), `g4` (32-bit general-purpose value), and `f4` (32-bit floating-point value), and the result is of type `g1` (8-bit general-purpose value). (The "0" reflects a low-level implementation detail that is beyond the scope of this paper.) The rest of the stub consists of the instructions that are to be executed when the stub is called.

- The `locals 1;` instruction declares a single local variable.
- The `ld l0,[i0+0:pint]` instruction loads a `pint` (pointer-sized general purpose value, often used for pointers that are not objects in the heap) from the address given by `i0` (the first argument) into `l0` (the first local)—in this example, this pointer points to the vtable for the object whose method is being invoked.
- The second `ld` instruction loads a `pint` from the address given by `l0` plus 32—in this example, this is the entry in the vtable for the method being invoked.

[5] LIL stands for Low-level Intermediate Language, and its pronunciation suggests its "little"ness or lightweight nature.

- The third instruction (`in2out managed:g1`) sets up for a call; in particular, it copies the arguments into an output area, and declares that the call will use the managed-code calling convention and return a `g1` into the implicit variable `r`.
- The `call l0` instruction calls the address in `l0` and sets the variable `r` to the value returned.
- The final `ret` instruction returns. The current value of `r` is the value returned by the stub.

Notice that the stub implicitly makes a tail call. LIL has an explicit way to make a tail call that is optimized by the code generator. The above stub could also be expressed as:

```
entry 0:managed:ref,g4,f4:g1;
locals 1;
ld l0,[i0:pint+0:pint];
ld l0,[l0+32:pint];
tailcall l0;
```

All LIL variables and operations are typed by a simple type system. The type system makes just enough distinctions to know the width of values and where they should be placed in a given calling convention. For example, the type system distinguishes between floating-point and general-purpose values but not between signed and unsigned. In addition, the type system distinguishes various kinds of pointers (e.g., pointers to heap objects versus pointers to fixed VM data structures), because in the future we may want the LIL code generator to be able to enumerate GC roots on LIL activation frames.

A LIL stub executes with an activation frame on the stack. Conceptually, this activation frame is divided into a number of areas that can vary in size and type across the execution of the stub. For our purposes, the areas are inputs, locals, outputs, and return. The inputs initially hold the arguments parsed by the caller, but they can change by assignment. Their number and type is fixed across the stub. The locals hold values local to the stub. Their number is determined by `locals` instructions, and their types are determined by a simple flow analysis. The outputs hold values passed to functions called by the stub. Their number and types are determined by `in2out` and `out` instructions. These instructions set up an output area and assign to the outputs, and then a `call` instruction performs the actual call. The return is a single location that is present following a `call` instruction or whenever an assignment is made to it; its type is determined by a flow analysis. Each input, local, output, and return is a LIL variable, and are referred to using the names `i0`, `i1`, ..., `l0`, `l1`, ..., `o0`, `o1`, ..., and `r`, respectively.

LIL's instructions include arithmetic operations, loads, stores, conditional and unconditional jumps, calls, and returns. They are summarized in Table 1. An operand `o` is either a LIL variable or an immediate value. The address part of load, store, and increment instructions can include a base variable, a scaled index variable, and an immediate offset; the scale can be one, two, four, or eight.

This format was chosen to easily take advantage of instructions and addressing modes of the IA32 architecture, the Itanium® Processor Family (IPF) architecture, and other architectures. The address also includes the type of the value being loaded, stored, or incremented. The conditions in a conditional jump are standard comparisons (equal, not equal, less, etc.) between two operands.

Table 1. LIL General-Purpose Instructions

Category	LIL syntax	Description
Declarations	`:label;`	
	`locals n;`	
	`in2out cc:rettype;`	
	`out sig;`	
Arithmetic	`v = o;`	Move
	`v = uop o;`	Unary
	`v = o1 op o2;`	Binary
Memory access	`ld v, addr;`	Load
	`st addr, o;`	Store
	`inc addr;`	Increment
Calls	`call o;`	Normal call
	`tailcall o;`	Tail call
	`call.noret o;`	Call that does not return
	`ret;`	
Branches	`jc cond, label;`	Conditional branch
	`j label;`	Unconditional branch

LIL also includes some constructs specific to our virtual machine, such as accessing thread-local data for synchronization or object allocation. However, these do not concern this paper and will not be discussed further.

The LIL system has two parts: a parser and a code generator. The parser takes a C string as input and produces an intermediate representation (IR) of LIL instructions. The parser includes a `printf`-like mechanism for injecting runtime constants such as addresses of functions or fixed VM data structures. The code generator takes the LIL IR as input and produces machine instructions for a particular architecture.

3 Subtype Tests

To illustrate how the VM can use LIL to insulate JITs from details of the object model and type data structures, this section considers subtype tests. Both Java and CLI include bytecodes like `checkcast` and `instanceof` that test whether an object is in a particular type. The typical implementation dereferences the object to obtain its class's data structure, and then tests whether this class is a subtype of the target type. The naive implementation of this subtype test is

to traverse superclass, interface, and array element-type pointers searching for the target type. In contrast, ORP uses Cohen's type display technique [8]. In practice, this implementation is much faster than the naive implementation, and improves overall application performance [3]—even more so when the JIT inlines the fast path into its generated code. However, the code to be inlined is heavily dependent upon the details of Cohen's techniques and the details of the VM's data structures. This example is an ideal case for showing how knowledge can be isolated in the VM without sacrificing performance.

3.1 Cohen's Type Displays

The basic idea of Cohen's type displays is to store a table of ancestors in the type data structures. In ORP, we store a fixed sized (`MAX_DEPTH` − 1) table in the vtable of each class. For a class at depth d, if $d \leq$ `MAX_DEPTH`, then the table contains the class's ancestors at level two through d and then `NULL`s; otherwise, the table contains the class's ancestors at level two through `MAX_DEPTH`. Note that every class's level-one ancestor is `java.lang.Object` (or `System.Object` in CLI), so we do not need to store this class. Each entry points to the class data structure for the corresponding class (not to the vtable of that class). To test if a class represented by vtable v is a subtype of another class represented by class data structure c, ORP does the following. If c's depth is one (meaning c is `java.lang.Object`), the test succeeds. Otherwise if c's depth d is less than or equal to `MAX_DEPTH`, ORP compares entry $d - 1$ of v's ancestor table with c, and this is the result of the subtype test. Otherwise, ORP falls back to the naive implementation, which is also used if the target type is an interface or array type. Since the performance-critical cases are most often class types within the maximum depth, most of the time a short sequence of instructions is executed.

3.2 Implementation

ORP offers runtime helper functions for subtype test operations, and the JIT may generate calls to the appropriate helper. For `checkcast`, the helper function takes two arguments: the object, and the class data structure for the target type. To call this helper function, the JIT emits instructions corresponding to the following simple LIL stub:

```
entry 0:managed:ref,pint:ref;
tailcall checkcast_helper;
```

Since the JIT knows the target type at compile time, the JIT could obtain better performance by inlining a fast path sequence customized to the target type. To achieve this inlining in ORP without LIL, the JIT would need to know that Cohen's type displays are being used, the location of the vtable within objects, and the location of the ancestor table in vtables. If any of these details change, then the JIT must be changed. If the object model is changed or Cohen's algorithm is replaced by another, perhaps to accommodate multiple inheritance,

then the JIT must change accordingly. Prior to developing LIL, ORP did have a JIT that did this customized inlining, so we are able to compare its performance against the LIL version.

To address both the performance issue and the software engineering issue, we modified ORP to use LIL to communicate information from the VM to the JIT without making the JIT dependent on this information. For `checkcast`, this works as follows: The JIT requests from the VM the runtime helper function for `checkcast`, and passes at JIT time the class data structure for the target type. The VM can optionally return a LIL stub, consisting of a customized sequence for that type, to inline into the JIT-generated code. If the type is a class type of less than maximal depth, the LIL stub will be the following (where `d` is the depth, `c` is the class data structure for the type, `ato` is the offset of the ancestor table within a vtable, and `throw_class_cast_exception` is a helper function that throws an appropriate exception):

```
entry 0:managed:ref,pint:ref;
jc i0!=0,nonnull;
r=i0;
ret;
:nonnull;
locals 1;
ld l0,[i0+0:pint];
ld l0,[l0+ato+4*(d-1):pint];
jc l0!=c,failed;
r=i0;
ret;
:failed;
out managed::void;
call.noret throw_class_cast_exception;
```

(Note that the stub is a two-argument function even though it specialized on its second argument, and that `ato+4*(d-1)` is actually a constant computed by the VM and is not a LIL expression.) The JIT inlines and converts the LIL stub into its own internal IR. For typical compiler IRs, this should be straightforward.

In this new implementation, the JIT is not aware that the type inclusion tests are implemented with type displays. Therefore, if the VM were modified to support multiple inheritance, the same JIT would still work for Java programs without any modifications and with no performance penalty. A new implementation could use a technique more suitable for multiple inheritance like the one described by Vitek *et al.* [19].

The scheme achieves our goal: it allows us to make performance optimizations without making the JIT dependent upon any VM information. All the JIT needs to understand is LIL and how to inline it. The next section evaluates the resulting performance, showing both that the LIL version performs similarly to the customized JIT version and that both versions of the JIT achieve speedup over the unoptimized call version.

3.3 Performance evaluation

As an experiment, we modified ORP's high-performance O3 JIT to include an ad-hoc runtime helper inlining system. This includes both a LIL inliner and custom versions of `checkcast` and `instanceof`. The custom version requires the JIT to have specific knowledge of the type display scheme including details of the VM data structures; the LIL version insulates the JIT from all such details. This section compares the performance of both versions of inlining with doing no inlining. We use the SPEC JVM98 [18] benchmark to perform the comparison.[6] The measurements are taken on a four processor 2.0 GHz Intel® Xeon™ machine with HyperThreading disabled, with 4 GB of RAM, and a heap size of 96 MB.

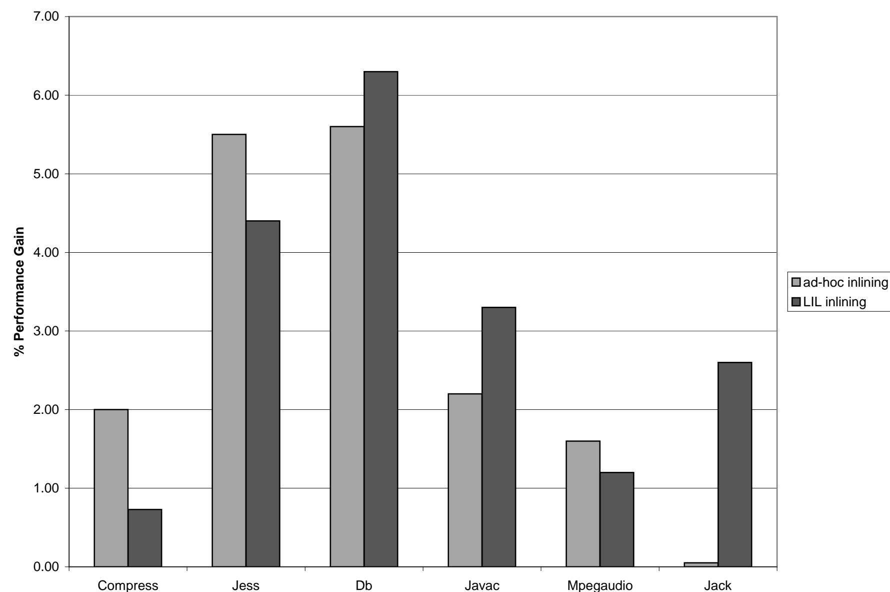

Fig. 1. Performance comparison of inlining on SPEC JVM98.

The results appear in Figure 1. The baseline does no inlining of helper functions; that is, JIT-generated code calls helper functions in the VM (however, these VM helper functions do have a fast path using the type displays). The graph shows the performance improvement of the two versions of inlining over this baseline.

[6] We use the SPEC benchmarks only as benchmarks to compare the performance of the various techniques within our own VM. We are not using them to compare our VM to any other VM and are not publishing SPEC metrics of any kind.

For the most part, there are small but significant performance improvements from inlining the helper functions. The gains from both schemes are in the same order of magnitude, ranging from no improvement on Jack and 5.6% on Db for ad-hoc inlining, and less than 1% on Compress and 6.3% on Db for LIL inlining. These results are encouraging and suggest that code implants offer good performance along with their other benefits.

4 Multiparadigm Support

This section shows how LIL code implants can support extensions to ORP's programming paradigm. As ORP stands today it is not suited for such extensions because some operations are implemented through ad-hoc JIT-generated sequences. In particular, field access, method invocation, downcasts, and upcasts are implemented by the JIT. The first step towards extending ORP's programming paradigms is to require these operations be implemented with LIL code implants. Once this is in place, the modifications required for the extensions discussed in this section are to the VM data structures and to the LIL that the VM provides to the JITs. The JIT implementation is largely unaffected. In our experience, the JIT is often the most complex part of the system and any changes to it are difficult. Therefore, limiting the changes needed to support a new programming paradigm to the core VM greatly simplifies the extension's implementation.

4.1 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) has been gaining popularity in recent years. Typically AOP extensions to Java are implemented without any changes to the JVM implementation and generate standard class files. AspectJ [9] is a popular example of such a system.

A new class of AOP applications is emerging that require dynamic AOP [4, 17]: the ability to add, modify and remove aspects at runtime. In a typical scenario, the source code is not available and and it cannot be rewritten in an AOP version of Java. Instead, the system must modify an existing application. The point cuts needed for dynamic AOP usually include method invocations and (sometimes) field accesses. The approaches used in prior work can be grouped into three categories:

- Use the debugger interface to intercept method invocations. The downside of this approach is poor performance and a limited set of point cuts available (*e.g.*, field accesses may not be available).
- Modify the bytecodes via a custom class loader. While this implementation is flexible, its performance is a problem since each potential point cut must be instrumented and a check must be performed at runtime to determine if a given point cut is active. Bytecode instrumentation of every point cut cannot be made as cheap as custom solutions.

– Modify the JIT to emit appropriate code. This approach offers the best performance but it requires potentially cumbersome modifications to the JIT.

The use of LIL allows dynamic AOP to be implemented with the same performance as the latter solution, but without the need to modify the JIT. The VM can use data structures to keep track of dynamic aspects. These data structures can then be consulted to generate the appropriate LIL sequences for each method invocation and field access.

4.2 Multiple Inheritance and Mixins

Changing the type inheritance model affects type instantiation, type casts, field accesses, and method invocation. Since all these actions are implemented through implanted LIL sequences, any changes needed to support a new inheritance model are confined within the VM. The rest of this section describes how multiple inheritance, mixins, and extensible objects can be supported using this system.

Classic Multiple Inheritance The easiest way to implement multiple inheritance is the one followed by most C++ compilers. That is, each base class instance begins at a certain offset from the start of the object. Each base class instance also has its own vtable pointer, which differs from the object's vtable pointer by a certain offset.[7] If this implementation is followed, then only the stubs for type casts need to change. Instead of just returning its argument, an upcast stub would now look as follows:

```
entry 0:managed:ref:ref;
r=i0+BASE_OFF;
ret;
```

where `BASE_OFF` is the offset of the requested base class's instance within the object. Downcast stubs must change in a similar but opposite way.

Mixins As explained by Flatt *et al.* [11], implementing mixin-style inheritance may require "boxing" some references. That is, a mixin-typed reference is implemented as a pointer to a structure that contains a pointer to an actual object as well as field and method translation tables (and possibly other information). These tables map field and method names into the offsets within the object and vtable respectively. Casts from normal class types to a mixin type must create the boxed reference, whereas casts from mixin types to a normal class type must retrieve the underlying object and discard the outer structure. Field accesses and method invocations through mixin references require looking up the field or

[7] This implementation slightly complicates garbage collection, since references now do not necessarily point to the start of an allocated object. This problem can be solved, although the details are too technical to include here.

method name in the translation tables. Clearly the VM can make LIL stubs for casts, field access, and method invocation that match this scheme; we omit the details.

Extensible Objects Some object-oriented languages such as Python allow the user to dynamically add fields to an object at runtime. A simple implementation has a list of dynamic field names and values in each object. If a JIT requests a field not statically declared in a type, then the VM generates a stub to search the dynamic field list of the object. If a requested field is not found in the dynamic field list, the stub can either throw an exception or create the field depending upon the desired language semantics. The VM can also provide the JIT with LIL stubs for dynamic field addition operations. Clearly, optimizations of this simple scheme are also expressible using LIL stubs.

4.3 Functional Programming

It is well known that functions, especially first-class functions in functional programming languages, are equivalent to objects with a single method [12]. Function application becomes invocation of the single method. The VM can present functions and function types to a Java JIT as if they were objects. The method invocation sequence shown in Section 2 results in two loads and a call for a function application. Typical implementations of functional programming languages have only one load and a call. The extra load could degrade performance. To avoid it, the VM could store the code address directly in the function, say immediately after the vtable.[8] Then if a Java JIT requests a virtual dispatch on a function type, the VM can generate a LIL sequence that loads the code pointer directly from the function, such as the following (for a function of no arguments or returns):

```
entry 0:managed:ref:void;
locals 1;
ld l0,[i0+4:pint];
tailcall l0;
```

Functional programming languages also have features like polymorphism, discriminated unions, and lazy evaluation that are quite different from typical object-oriented features. For example, polymorphism in these functional programming languages is related to generics in object-oriented languages. These features might need different object models, field access sequences, and method invocation sequences. As an example of the latter, thunk creation and forcing for lazy evaluation could be hidden in method invocation code that crosses from Java to lazy functional code. We speculate that these differences could be hidden from the JIT using LIL.

[8] In typical implementations of functional programming, there is no vtable, but instead there is a header word used by the garbage collector. In ORP, there is no header word, and instead the information contained in the header word is stored in the vtable. Thus the space requirements are the same.

5 Conclusion

Extending existing virtual machines to support new object models and programming paradigms is difficult because knowledge about the object model is spread across many components and modules. Such knowledge includes how field accesses, method invocations, down casts, and up casts should be implemented. This paper has shown how to address this problem without sacrificing performance. The solution is to concentrate knowledge of the object model in the core VM component, and to use code implants to inline performance critical operations into JIT-generated code and other components. Then implementing new object models or programming paradigms requires changes to a small part of the system only.

We originally designed LIL to provide CPU-independence and better maintainability of stubs within ORP. We were pleasantly surprised when we realised that it could also be used to implement a code implant system and thus achieve better modularity for ORP with the same performance.

While we have investigated code implants for `checkcast` and `instanceof` in ORP, much more work remains before ORP will be a platform for multiparadigm experiments. Many other opportunities remain for the use of code implants. Future work will explore these possibilities.

References

1. A.-R. Adl-Tabatabai, J. Bharadwaj, D.-Y. Chen, A. Ghuloum, V. Menon, B. Murphy, M. Serrano, and T. Shpeisman. The StarJIT Compiler: A Dynamic Compiler for Managed Runtime Environments. *Intel Technology Journal*, 7(1), February 2003. Available at http://intel.com/technology/itj/2003/volume07issue01/art02_starjit/p01_abstract.htm.
2. A.-R. Adl-Tabatabai, M. Cierniak, G.-Y. Lueh, V. M. Parikh, and J. Stichnoth. Fast, Effective Code Generation in a Just-In-Time Java Compiler. *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, June 1998.
3. B. Alpern, A. Cocchi, and D. Grove. Dynamic Type Checking in Jalapeño. *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM '01)*, April 2001.
4. J. Baker and W. Hsieh. Runtime Aspect Weaving Through Metaprogramming. *Proceedings of the International Conference on Aspect-Oriented Software Development*, April 2002.
5. A. Bik, M. Girkar, and M. Haghighat. Experiences with JAVA JIT Optimization. *International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems*, October 1998.
6. M. Cierniak, M. Eng, N. Glew, B. Lewis, and J. Stichnoth. Open Runtime Platform: A Flexible High-Performance Managed Runtime Environment. *Intel Technology Journal*, 7(1), February 2003. Available at http://intel.com/technology/itj/2003/volume07issue01/art01_orp/p01_abstract.htm.
7. M. Cierniak, G.-Y. Lueh, and J. Stichnoth. Practicing JUDO: Java Under Dynamic Optimizations. *Proceedings of the SIGPLAN '00 Conference on Programming Language Design and Implementation*, June 2000.

8. N. H. Cohen. Type-extension type test can be performed in constant time. *ACM Transactions on Programming Languages and Systems*, 13(4), October 1991.
9. Eclipse.org. AspectJ Project, 2003. Available at http://eclipse.org/aspectj.
10. ECMA. *Common Language Infrastructure.* ECMA, 2002. Available at http://www.ecma-international.org/publications/Standards/ecma-335.htm.
11. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 171–183, New York, NY, 1998.
12. N. Glew. Object closure conversion. In A. Gordon and A. Pitts, editors, *3rd International Workshop on Higher-Order Operational Techniques in Semantics*, volume 26 of *Electronic Notes in Theoretical Computer Science*, Paris, France, Sept. 1999. Elsevier. Available at http://www.elsevier.nl/locate/entcs/volume26.html.
13. D. Grune, H. E. Bal, C. J. H. Jacobs, and K. G. Langendongen. *Modern Compiler Design.* Wiley, 2000.
14. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.
15. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, Second Edition.* Addison-Wesley, 1999.
16. M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, Paris, France, Jan. 1997.
17. A. Popovici, G. Alonso, and T.Gross. Just In Time Aspects: Efficient Dynamic Weaving for Java. *Proceedings of the International Conference on Aspect-Oriented Software Development*, March 2003.
18. Standard Performance Evaluation Corporation. SPEC JVM98, 1998. See http://www.spec.org/jvm98.
19. J. Vitek, R. N. Horspool, and A. Krall. Efficient type inclusion tests. *ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages and Applications*, October 1997.
20. D. Wakeling. Compiling lazy functional programs for the java virtual machine. *Journal of Functional Programming*, 9(6), November 1999.