Engineering a Common Intermediate Representation for the Ovm Framework

K. Palacz J. Baker C. Flack C. Grothoff H. Yamauchi J. Vitek

S^3 lab, Department of Computer Sciences, Purdue University

Abstract

The Ovm framework is a set of tools and components for building language runtimes. We present the common intermediate representation of this framework and software design patterns used throughout Ovm. One of the main themes in this work has been to support experimentation with new linguistic constructs and implementation techniques. To this end, Ovm components were designed to be parametric with respect to the instruction set on which they operate and its semantics. We argue that our approach eases the task of writing new components without sacrificing efficiency.

Key words: Virtual machines, object-oriented languages, Java.

1 Introduction

The goal of the Ovm project is to deliver an open source framework for the purpose of building language runtimes. Ovm is a toolkit that provides the basic components of a virtual machine. These components can be specialized and assembled into an Ovm configuration customized for a particular problem domain. The framework is designed so as to be able to support different object models, while our first emphasis is on supporting a Java personality, we have plans to provide a C# personality. In Ovm, a configuration defines a new runtime environment, for example we have developed a Real-Time Java configuration which yields a virtual machine implementing part of the Real-time Specification for Java.

The framework is composed of a number of tools and subsystems (interpreter, verifier, rewriters, compilers) which must be kept operational whenever the instruction set on which they operate is modified or extended. One of the earliest design decisions in the project was to attempt to parameterize Ovm components by their instruction set using a common and intermediate representation, OvmIR, which can be easily customized to the specific needs of a particular configuration. We consider components to be parametric if they can be applied to different instruction sets with minimal changes to their implementation. In Ovm this form of parameterization is achieved by a combination of a reflective instruction set specification and consistent use of software design patterns [6]. Instructions in the OvmIR are represented by data structures that detail their semantics and that can be inspected introspectively by the framework components. The OvmIR specification uses a type hierarchy to classify instructions according to their behavior. This classification is used by the tools to reduce the amount of redundant code written for processing OvmIR.

In this paper we report on the design of the Ovm intermediate representation and architecture of software components that manipulate it. The OvmIR embodies a number of software engineering tradeoffs made to meet the specific requirements of customizability. The OvmIR is a common code format to the entire Ovm framework as illustrated in Figure 1. This representation is flexible as it allows to define new operations and modify the semantics of existing ones. In order to write tools that may adjust to changes in the instruction set, we use a single semantic specification for the entire Ovm tool chain, from static analysis to interpreter and code generation. While the OvmIR is instrumental in generating an interpreter it is not unique in that respect [11,3].

The paper is structured as follows. Sec. 2 introduces the goals and architecture of the Ovm framework. Sec. 3 describes how the OvmIR is specified. Sec. 4 introduces the design patterns that are used to operate on the IR. Sec. 5 describes components that have been build using the OvmIR. Sec. 6 describes the use of OvmIR-to-OvmIR transformations in allowing necessary non-Java semantics to be expressed in Java source. Sec. 7 gives a simple example of extending the IR. We conclude with related and future work.

2 The Ovm Framework

This paper documents work on the Ovm project developed at Purdue and intended as an open source framework for building language runtimes. The framework provides basic components of a virtual machine, which can be specialized and combined as an Ovm configuration specific to a problem domain. The Ovm components are written almost entirely in Java, which is also the first language to be targeted by Ovm, an approach that has been referred to as Java-in-Java. The framework contains more than 150K lines of code and over 2000 classes, including an interpreter, a just-in-time compiler and an ahead-of-time compiler. The configurations currently supported by Ovm include a plain

Java configuration and a configuration implementing the Real-Time Specification for Java. Several garbage collection algorithms are supported.

To obtain a virtual machine with the Ovm framework, a number of steps have to be followed. First a personality must be chosen, such as Java, and a particular configuration, *e.g.* Real-time Java. The first stage of VM construction involves loading Ovm within a host VM. All classes within the kernel of the virtual machine (as opposed to user code) are loaded, along with their code. The objects used to implement the kernel are also instantiated at this time. Ovm then proceeds to analyze its own bytecode and recognize idioms. The third stage involves creation of a binary image. At this point all objects belonging to the Ovm kernel are transformed into the Ovm object model. This staged VM generation process is shown in Figure 2.

Ovm transforms incoming bytecode to an intermediate representation called OvmIR that is the common input of all execution modes (interpret/compile). An idiom whose effect can be stated as a transformation on this IR can be added without touching an interpreter or compiler and will have well-defined semantics independent of the execution mode. The transformations are effect by a simple peep-hole rewriting tool. The architecture of Ovm is illustrated in Figure 1.

Idiom recognition is implemented by an iterative process, target code fragments are processed until a fixed point is reached. The effects of idioms as well as replacement of some high-level Java operations with lower-level sequences, are performed iteratively until a fixed point is reached. This allows

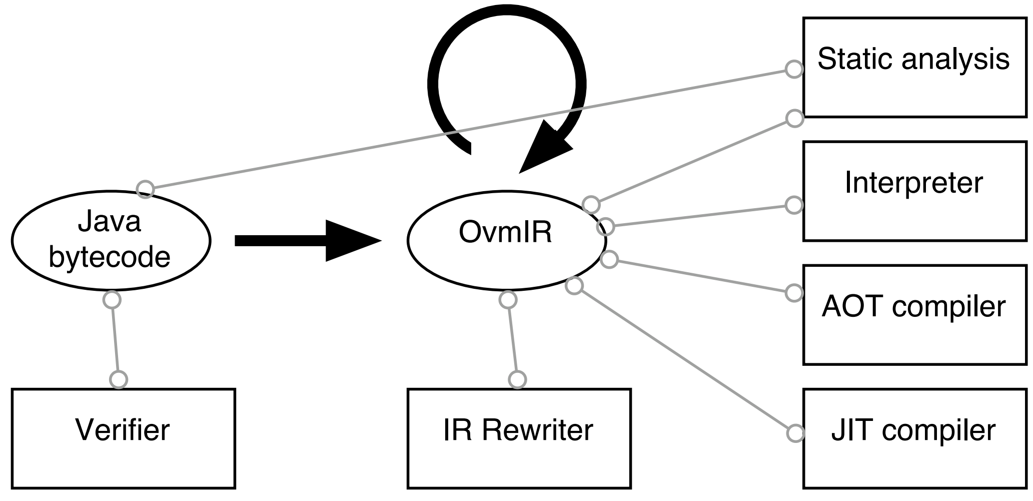

Figure 1: OvmIR is the common input to all tools in the Ovm framework.

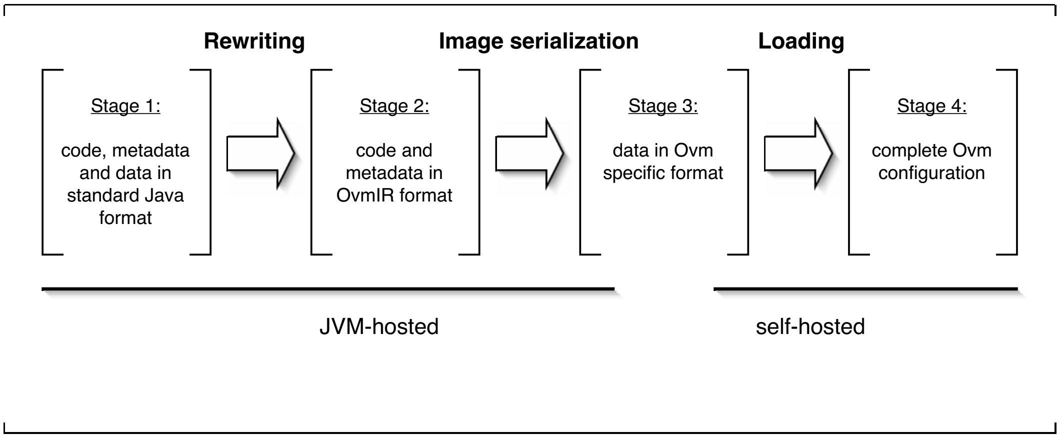
Rewriting
Image serialization
Loading
Stage 1:
code, metadata and data in standard Java format
Stage 2:
code and metadata in OvmIR format
Stage 3:
data in Ovm specific format
Stage 4:
complete Ovm configuration
JVM-hosted
self-hosted

Figure 2: The staged image construction process. For any configuration, the Ovm data structures and code are first loaded on standard JVM. Idiom recognition is applied in the first transformation pass which yields OvmIR code and metadata for the Ovm kernel. In the second stage, the objects that compose the Ovm kernel are transformed into the Ovm object model and serialized into a binary image. Finally, the image is loaded and the VM become executable.

the effect of a pragma to be expressed in high-level IR constructs that may even include other idioms as long as cycles are avoided.

3 A Reflective IR Specification

The intermediate representation currently in use in the Ovm framework is a stack-based high-level intermediate language that closely mirrors Java bytecode [12]. The choice to base OvmIR on bytecode was made for pragmatic reasons. Java bytecode is a compact and executable representation with a well-specified semantics. This entails a shorter learning curve for working with the framework components and a natural path towards an interpreter. Of course, there are also drawbacks. Bytecode is not fully typed; thus, it is necessary to perform static analysis to recover type information. Other drawbacks include relative addressing and the complexity of dealing with the stack when performing code transformations. For our purposes, these drawbacks have turned out to be minor.

The OvmIR specification is expressed as a set of classes, one class per instruction in the IR. This has the advantage of a close correspondence between the textual form of the specification and the internal data structures used within the VM and tools operating on this IR. Furthermore, this permits writing code that operates on the IR by reflection. For example, the abstract interpreter handles many instructions using introspection on their definition. It is possible to programatically alter the instruction set to add new operations or modify the semantics of existing ones. In many cases, such changes require minimal changes on tools operating on the IR.

3.1 Abstract machine model

The semantics of instructions are defined with respect to an implicit abstract machine. Currently, we use a stack-based machine similar to the one expected by Java bytecode. Thus, every operation may read and write values from local variables, push and pop variables from the stack, consume values from the instruction stream, jump to a set of offsets, throw exceptions, have evaluation side effects or access compile-time constant values contained in a constant pool. The OvmIR specification consists of the definition of a number of *instructions*, *values* and so-called *value sources*. While instructions are self-explanatory, it is worth discussing values and value sources. A value object is the representation of a concrete value that will be manipulated at runtime by the virtual machine. The `Value` class is used to represent runtime values and their types. Typing constraints on the values consumed or produced by an instruction are expressed by using subclasses of the `Value` class (*e.g.* integer values are represented by instances of `IntValue`). Since it is not always possible or practical to model all the constraints on values using inheritance, further typing constraints can be expressed as annotations on the value type. For example, `CPIndexValue` is a subclass of `IntValue` used to denote constant pool indices and declares a field indicating the type of constant pool entry, such as field reference, type name reference etc.

3.1.1 Instructions

Each IR instruction corresponds to a subclass of the `Instruction` class. Analyses written for the framework use the flyweight pattern [6] to avoid the need for multiple instances of the same instruction class. The semantic specification of every instruction is provided in the parameterless constructor of the respective class. Each instruction must at least implement the methods `size()`, which returns the size of the instruction in bytes, and `getOpcode()`, which returns a numeric instruction identifier. Further behavior is added only by subclasses for which the behavior is meaningful. Thus, code manipulating the OvmIR can rely on the type checker to prevent errors such as trying to use a constant pool index as a jump target.

The semantics of instructions are characterized by the following arrays of abstract values that are fields of `Instruction` (see Figure 3). Furthermore, instructions that implement the Throwing interface define an array of exceptions that may be thrown as a side effect of executing the instruction.

streamIns	values consumed from instruction stream
stackIns	values consumed from the stack
stackOuts	values produced on the stack
evals	expressions evaluated for side effects
jumpTarget	target program counter value (subtypes of FlowChange)
controlValue	value used to control branching (subtpyes of ConditionalJump)

Figure 3: Fields that specify the input-output behavior of each instruction.

As an example of one of simple instruction specifications, consider the definition of the SWAP instruction which exchanges the two topmost stack values. The code for SWAP is shown in Figure 4. SWAP is a *concrete instruction* in that it corresponds to an actual operation in the instruction set. It is defined as a subclass of the abstract class StackManipulation which in turn extends Instruction. The hierarchy permits us to reuse code between related instructions by inheritance. Extensive examples for this are given in Sections 4 and 5. The instruction hierarchy can also be used to classify instructions into various categories.

```
class SWAP extends StackManipulation {{
    stackIns = new Value[]{ new Value(), new Value()};
    stackOuts = new Value[]{ stackIns[1], stackIns[0]};
}}
```

Figure 4: Specification of the SWAP instruction.

The specification of SWAP describes that the instruction consumes two values from the stack and pushes two values back onto the stack. The fact that the output values are identical to the input values, with the exception of their order, is represented by pointer equality of the Value objects.

The core of the instruction hierarchy used within Ovm is shown in Figure 5.

The given hierarchy reflects properties of the Java bytecode instruction set and is based on pragmatic considerations, not on any systematic analysis of features of all conceivable instructions of stack machines. It turns out to be impossible to model all the features of bytecode using single inheritance. For example, `FlowChange` is an interface and is implemented by the concrete instruction `RET` (return) that inherits from `LocalAccess`. Another example is the `Throwing` interface which is implemented by all instructions that can throw exceptions.

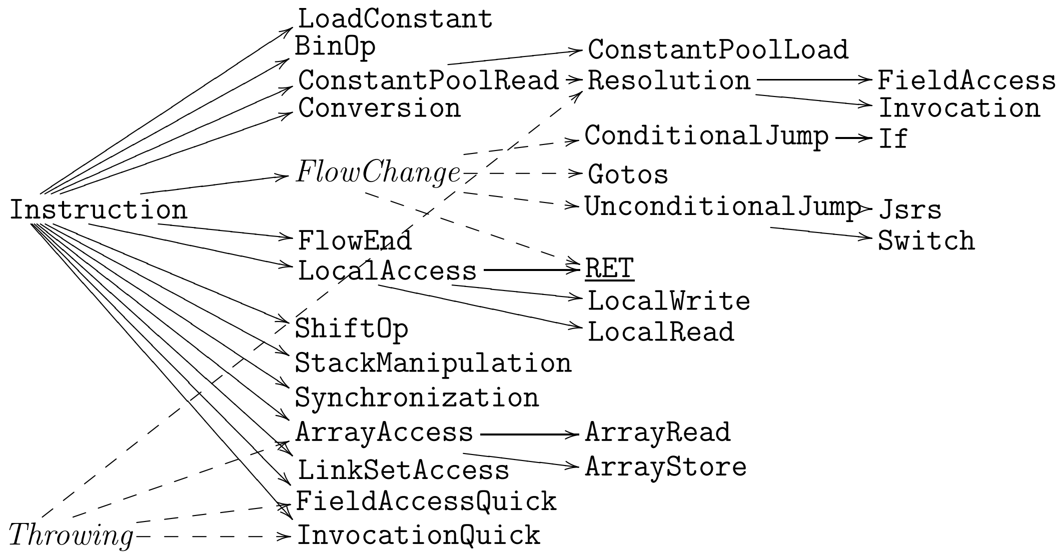

Figure 5: The instruction hierarchy. The OvmIR consists of 237 concrete instructions (`RET` is the only one shown here) that account for 3400 lines of code.

Figure 6 gives a simple example of reflective programming. The method initializes a lookup table, the static variable `instructionSet`, with instances of every concrete instruction. The code assumes that all instructions are inner classes of the `Instruction` class and that they have a constructor with no arguments. Only concrete instructions are instantiated.

3.1.2 Values and Value Sources

Another way to associate output values with input values in the specification is to add *value sources* to values. Each `Value` object contains a reference to an instance of the `ValueSource` interface. A variety of value source implementations are provided, such as binary arithmetic and logical expressions, and routine invocation expressions. Since value sources are also represented as Java data structures, they can be analyzed introspectively. Value sources can

```
void makeInstructionSet() {
   Class[] inr = Instruction.class.getDeclaredClasses();
   for ( int i = 0; i < inr.length; i++)
      if ( Instruction.class.isAssignableFrom( inner[i])
      && !Modifier.isAbstract( inr[i].getModifiers())) {
        Instruction inst =
            (Instruction) inr[i].newInstance();
        instructionSet[inst.getOpcode()] = inst; } }
```

Figure 6: Reflective instruction set initialization.

reference existing values, and hence the instruction specification forms an expression tree consisting of a sequence of expressions represented by the `evals` array followed by a sequence of expressions in the `stackOuts` array. The evaluation of both sequences can introduce side effects. For example, `IADD`, which has no side effects and pushes the sum of two integers on the stack could be defined as given in Figure 7[1] .

```
class IADD extends BinOp {{
    super(IADD);
    stackIns = new Value[] {
        new IntValue(), new IntValue()};
    evals = new Value[] {};
    stackOuts = new Value[] {
        new IntValue(new BinExp(stackIns[0],
            "+",
            stackIns[1])) }; }}
```

Figure 7: Specification of the `IADD` instruction.

Individual instruction definitions are assumed to be fragments of code with single input and single output *modulo exceptions*.

The IR supports flow control private to an instruction definition, as shown in Figure 8.

[1] In Ovm the specification of `IADD` is further abstracted to:
class `IADD` **extends** `BinOp{IADD(){super(IADD,"+",intFactory);}}`

```
class FCMP extends Instruction {{
  stackIns = new Value[] {
                new FloatValue(),
                new FloatValue()};
  IfExp e = new IfExp(
                new CondExp( stackIns[1], ">",
                              stackIns[0]),
                ONE,
                new IntValue( new IfExp(
                    new CondExp( stackIns[1],
                                 "==", stackIns[0]),
                ZERO, MINUSONE)));
  stackOuts = new Value[]  new IntValue(e) ; }}
```

Figure 8: Specification of the floating point comparison instruction.

The IR can also specify control flow instructions that affect the program counter. In this case instructions declare their control value and jump targets, as shown in Figure 9.

```
class IFNE extends ConditionalJump {{
    stackIns = new Value[] { new IntValue(), new IntValue() };
    streamIns = new IntValue[]{ new PCValue() };
    jumpTarget = streamIns[0];
    controlValue = new IntValue(
        new CondExp( stackIns[0], "!=", stackIns[1]));}}
```

Figure 9: Specification of the `IFNE` branch instruction.

Note that instead of using pointer identity to bind stack output values to input values, one can use a separate kind of value source to denote that a value comes from a stack slot. However, the presented approach results in more concise definitions.

The `InstructionBuffer` class maintains a notion of the current program counter as well as the definitions of constants referenced from the bytecode stream. Because of this design, instruction objects can retrieve and interpret their immediate operands without any state of their own. For example, the concrete instruction GETFIELD subclasses the abstract class FieldAccess. FieldAccess provides a method getSelector(InstructionBuffer) to return information about the name and type of the field being accessed. The state required by the method is encapsulated in the instruction buffer argument. This design follows the flyweight pattern [6], allowing the InstructionSet class to hold a single instance of each concrete instruction.

The OvmIR specification contains approximately 3400 lines of code and 237 concrete instruction definitions which results in about 14 lines of code per definition. The line count includes abstract classes and supporting methods. Easily recognizable syntactic conventions (class and constructor declarations) account for approximately 4 lines per instruction hence it requires about 10 lines of nontrivial code to specify an instruction.

Example. To demonstrate the expressiveness of a reflective specification, consider the implementation of ovmp, the Ovm counterpart to the javap class file disassambler. Ovmp prints the mnemonic of each opcode and, for instructions such as branch or loads that have immediate arguments, the value of those argument in the format specified by the instruction semantics. The main loop is written as shown in Figure 10.

```
while ( ib.remaining() > 0 ) {
    Instruction i = ib.get();
    print( i.getName());
    if ( isStreamReader( i) )
        iprinter.visitAppropriate( ib.currentOpcode()); }
```

Figure 10: Main loop of the Ovmp disassambler.

A runabout dispatches to the proper print method. Because the instruction hierarchy lacks a type to characterize all iStream readers, the implementation takes advantage of the reflective IR specification. It filters calls to the runabout based on the characteristics of instructions. This is done by the method iStreamReader() which return true if an instruction has immediate arguments (see Figure 11).

The argument printer runabout has sixteen methods and is a total of 120 lines long. The entire ovmp has 520 lines (out of which 60 are comments).

```
iStreamReader( Instruction i) {
    return i.istreamIns.length != 0; }
```

Figure 11: A method that selects instructions that have immediate arguments.

4 Design patterns

The software architecture of the tools and components of the Ovm framework has evolved over time. This section describes this evolution and the motivations behind the changes. Originally, the instruction objects used dedicated methods to perform abstract execution of bytecode. The dedicated methods accessed the state information of the current method via a helper object which was provided to the constructor of every instruction object and stored in a field of the instruction. This created problems in that the instruction objects could not be used concurrently by multiple threads operating on different methods.

The use of dedicated methods also had the disadvantage that every additional analysis or processing required changes to each instruction class. Thus, each of the instructions was extended with an accept method and various analyses were written as visitors operating on the instructions. In order to make code-factoring easier, the instructions were arranged in a hierarchy (see Figure 5) that reflects commonalities between the instructions. Convenience methods implemented by the instructions were factored into common superclasses. An example of such a method is `getCPIndex`, a method that returns the index into the constant pool for every instruction that accesses the constant pool.

The visitors that implement the various analyses are also able to take advantage of the instruction hierarchy; the visit methods can be refactored using the hierarchical visitor pattern [10]. For example, our access modifier inference tool does not need to distinguish between Java's four field access operations (`GETFIELD`, `PUTFIELD`, `GETSTATIC`, `PUTSTATIC`). Using the hierarchical visitor pattern, Jamit only needs to implement the visit method for the abstract `FieldAccess` instruction class. In order to make the hierarchical visitor pattern work, a helper method that indirects calls from `visit(PUTSTATIC i)` to `visit(FieldAccess i)` is required (see code in Figure 12).

Writing this indirection code, while conceptually trivial, turns out to be cumbersome over time. Each time the instruction hierarchy evolved, the base-classes of the visitors needed to be rewritten. With over 200 instruction classes

```
class FieldAccess {
    void accept( Visitor v) {
        v.visit( this); } }

class GETFIELD extends FieldAccess {
    void accept( Visitor v) {
        v.visit( this); } }

class HierarchicalVisitor extends Visitor {
    void visit( FieldAccess i) {
        ...
    }
    void visit( GETFIELD i) {
        visit( (FieldAccess) i); } }

class Main {
    static void main() {
        Visitor v = new HierarchicalVisitor();
        Instruction i = new GETFIELD();
        inst.accept( v); } }
```

Figure 12: Snippet of the Hierarchical Visitor design pattern used in previous versions of Ovm.

it became difficult to track changes in the hierarchy. Probably worse, the instruction set needed to be expanded to support operations that are not part of the JVM standard. The use of the visitor pattern required that every analysis supplied visit methods for *all* instructions. Thus, every change in the hierarchy of the instruction set required updates to several visitors. Considering that one of the requirements for OvmIR is that the instruction set is customizable, this was not practical.

The problem was solved by replacing the use of visitors with the Runabout pattern [7]. Runabouts declare visit methods just like visitors, but instead of doing double-dispatch with `accept` methods in the instruction objects, the appropriate visit methods are found by reflection and invoked by dynamically generated and loaded helper classes. The result of this final refactoring was that hundreds of `accept` methods were removed from the instruction objects and hundreds of `visit` methods that were either abstract (visitor interface), empty (default base class) or indirecting to other visit methods (hierarchical visitor) became obsolete. The Runabout code in Figure 13 is the equivalent

```
class FieldAccess { }

class GETFIELD extends FieldAccess { }

class MyRunabout extends Runabout {
    void visit( FieldAccess i) {
        ... } }

class Main {
    static void main() {
        Runabout r = new MyRunabout();
        Instruction i = new GETFIELD();
        r.visitAppropriate( inst); } }
```

Figure 13: Snippet of Runabout about code with equivalent functionality to the hierarchical visitor example.

to the visitor code in Figure 12. As the example shows, using the Runabout eliminates the need for the `accept` methods and the code for the code for the hierarchical indirection. The performance impact of the dispatch with the Runabout on the framework was minor (a few percent depending on the analysis), confirming expectations from the microbenchmarks (Figure 14). Figure 15 shows the various classes in the Ovm framework that are Runabouts visiting the instructions. The table also lists the number of visit methods that each of these classes implements. The total number of instruction classes in Ovm is 273 (237 of these are concrete instructions, but some tools only support the 201 instructions of the Java VM specification [12]).

5 Components built around Ovm

Various tools have been built around the OvmIR. Tools that are used by the VM itself include a bytecode verifier, a simple JIT compiler, an interpreter generator, a bytecode-to-C++ translator, and the OvmIR transformation infrastructure. The analysis framework has been used to implement Kacheck/J [8], a tool to infer confined types, Jamit, an access modifier inference tool and Hitsuji, a tool that performs control-flow analysis (for example, 0-CFA [15] or type-safe method inlining [14]).

The Ovm framework includes a Java bytecode interpreter written in C. The interpreter is automatically generated from the instruction specification which ensures that the part of the system written in C is synchronized with the rest of the codebase. In particular, this ensures that the C interpreter actually operates on the same IR that is produced by tools written in Java without forcing the interpreter to consult `Instruction` objects at runtime.

We chose to implement only a subset of Java bytecode instructions in C code. The more involved instructions that require cooperation with the JVM runtime such as, for example, the resolution of symbolic member references or memory allocation, are reduced to invocations of methods of the runtime. Because instruction specifications can be viewed as an abstract syntax tree format for a subset of C expressions, the interpreter generator is fairly simple. Conceptually, each instruction is compiled to a `case` clause that copies immediate and stack operands into local variables, computes the result, and pushes it on the stack. For instance, `IADD` would be compiled to the following code (Figure 16).

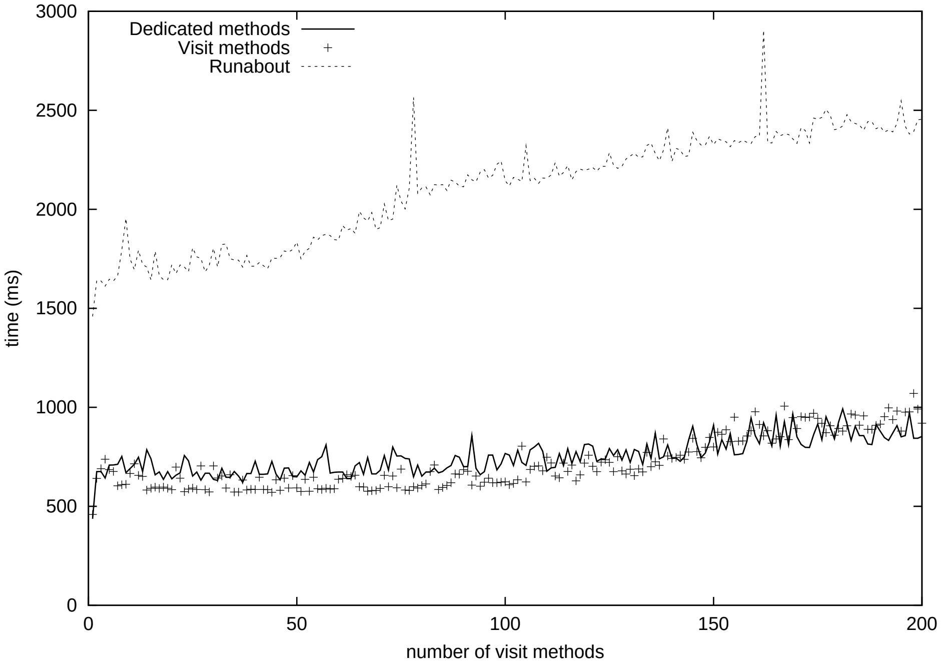

Figure 14: Runtime comparison of dedicated methods, visitors and the Runabout in Micro-benchmark for 10.000.000 iterations on SUN JDK 1.4.1, P-III 1 GHZ, Linux 2.4.18. The time shown is for the dispatch only.

Runabout	# visit methods
CloneInstructionVisitor	202
ClassCleaner	2
LinearPassController	1
AbstractInterpreter	27
ControlFlowInterpreter	8
BasicBlockJ2CTranslator	11
TypeNameClosure	10
SimpleJitVisitor	134
JamitConstraintGenerator	8
ZeroCFA	18
ConstraintGenerator	9
MaxStackHeightInference	30

Figure 15: List of the Runabouts working on the instructions.

```
case INSTR_IADD: {
    jint stack_in_0 = POP().jint;
    jint stack_in_1 = POP().jint;
    jint stack_out_0 = stack_in_1 + stack_in_0;
    PUSH_P( stack_out_0);
    INCPC( 1);
    NEXT_INSTRUCTION;
}
```

Figure 16: Interpreter generation.

Currently, our interpreter loop is not using a switch statement; instead, we generate a threaded interpreter [3] using language extensions of the GCC compiler. We had to change very few lines of code in the interpreter generator to switch between these two techniques.

5.2 Ahead of Time Compilation

We are currently using the same instruction specification to implement a bytecode to C++ compiler, J2c. This compiler does not use high-level C++ features such as member functions and run-time type identification, but treats C++ as a portable assembly language with exception-handling support. The compiler may be configured for either conservative or precise garbage collection.

Although the instruction specification forms the basis of J2c's intermediate representation, J2c extends the specification in a few ways. For instance, the OvmIR differentiates between explicit method invocations and traps into the runtime. J2c translates both types of method calls into a single form so that they can be devirtualized and inlined uniformly. J2c also defines a `ValueSource` type corresponding to `MULTIANEWARRAY`'s operands. Such a type makes sense for occurrences of the `MULTIANEWARRAY` instruction, but does not fit into the singleton framework, since `MULTIANEWARRAY` takes a variable number of arguments.

5.3 Just in Time Compilation

We have implemented a just-in-time compiler, SimpleJIT, which converts the OvmIR into Intel x86 native code in one linear pass. SimpleJIT is not intended to be an optimizing compiler, but rather a basic fast compiler like the Jikes RVM baseline compiler [2]. SimpleJIT emulates the operand stack in the native stack frame. Each concrete instruction is covered by a visit method in the compiler. Figure 17 shows the visit method for the family of instructions pushing an integer on the stack. The code exploits the instruction hierarchy to cover nine concrete instructions.

```
public void visit( IConstantLoad instruction) {
    asm.pushI32( instruction.getValue( this));
}
```

Figure 17: The visit method for `ICONST_0`, `ICONST_1`, `ICONST_2`, `ICONST_3`, `ICONST_4`, `ICONST_5`, `ICONST_M1`, `BIPUSH`, and `SIPUSH`.

5.4 Static Analysis

Ovm includes a number of classes for performing static program analysis. Implementing a static analysis requires the implementation of a set of visit methods for the instructions that are relevant for the analysis. Furthermore, the analysis needs to select an iterator that specifies the traversal over the code. Ovm provides two basic templates. The first one performs a linear pass and that will visit every instruction once. The second runs a fixpoint iteration that is coupled with a customizable abstract interpreter. For customization, the analysis defines the level of abstraction by providing the set of abstract values. Defining the abstract values requires code that provides tests for value equality and merging of abstract values at join points. The default set of abstract values in Ovm corresponds to the set used by a Java bytecode verifier and distinguishes between four basic primitive types (int, float, double, long), the null reference, jump targets (JSR), initialized objects and uninitialized objects. The default execution model also corresponds to the abstract execution performed by a bytecode verifier.

A typical analysis uses this basic form of abstract execution and interposes calls to analysis specific visitors that inspect the state of the abstract interpreter for information relevant to the particular analysis. If the value abstractions are extended to better match the different abstract domain of a given analysis, visit methods of the abstract interpreter must be overriden to ensure proper handling of the new values. Examples of existing extensions of the abstract value set in Ovm include the addition of a special value for the `this` reference in Kacheck/J [8] and the use of a flow sensitive type sets for the implementation of 0-CFA [15].

The instruction specification isolates the analysis code from irrelevant changes in the OvmIR. Often a single visit method covers the behavior of multiple instructions that are equivalent from the point of view of the analysis. For example, the default abstract interpreter has generic code for instructions that merely perform basic operations such as moves or arithmetic. Thus, instructions that fall into these categories can be added trivially without changing the abstract interpreter.

Example. Figure 18 gives a simple example of a `visit` method that overrides the default behavior of the abstract interpreter for the `NEW` instruction. In the context of 0-CFA, a `NEW` instruction pushes a flow set on the stack that contains the type of the object being constructed. The visit method is located within the body of a runabout called within the fixpoint iteration. The method starts by querying the `NEW` instruction for the name of the class under construction. Since the instruction is a flyweight object, the runabout passes the current instruction buffer to the instruction so that it can retrieve the type name.

```
void visit( NEW instruction) {
    TypeName type = instruction.getClassName( ibuf);
    getFrame().push( valueFactory.makeSet( type));
}
```

Figure 18: The visit method for the `NEW` instruction.

The `valueFactory` object is an abstract value factory that creates the flow set which is then pushed onto the operand stack. The implementation of `getClassName` in the `NEW` instruction class is shown in Figure 19. The method is written in terms of operation on the state of the instruction buffer and an auxiliary `getCPIndex` method which returns the constant pool index immediately following the opcode of the current instruction.

```
TypeName getClassName( InstructionBuffer ibuf) {
    int index = getCPIndex( ibuf));
    return ibuf.getConstantPool().getTypeNameAt( index);
}

int getCPIndex( InstructionBuffer ibuf) {
    return ibuf.getCode().getChar( ibuf.getPC() + 1);
}
```

Figure 19: The `NEW` flyweight instruction object extracts context dependent information from the instruction buffer.

Performance. The code needed to extract information from the abstract execution is typically small. For instance, the addition of the `this` pointer to the abstract value set for our confinement checker Kacheck/J [8] is specified in 160 lines of code. Flow sensitive types for the 0-CFA algorithm are implemented in 370 lines. The code for constraint generation in Kacheck/J is merely 660 lines; obtaining type set information for the 0-CFA takes 530 lines.

The use of the flyweight pattern for instructions is key for achieving high throughput. Kacheck/J, which can be run as a standalone application and performs what amounts to a slightly extended variant of bytecode verification, has competitive running times. Figure 20 shows the running time of the analysis on a set of large benchmark programs. The graph plots the number of MB of bytecode (size of class files inclusive of constant pools) in the benchmark against the time required for the analysis proper (we did not include the time it takes to load the bytecode and parse the constant pools). This amounts to roughly 10MB per second, which appears competitive with tools written in C.

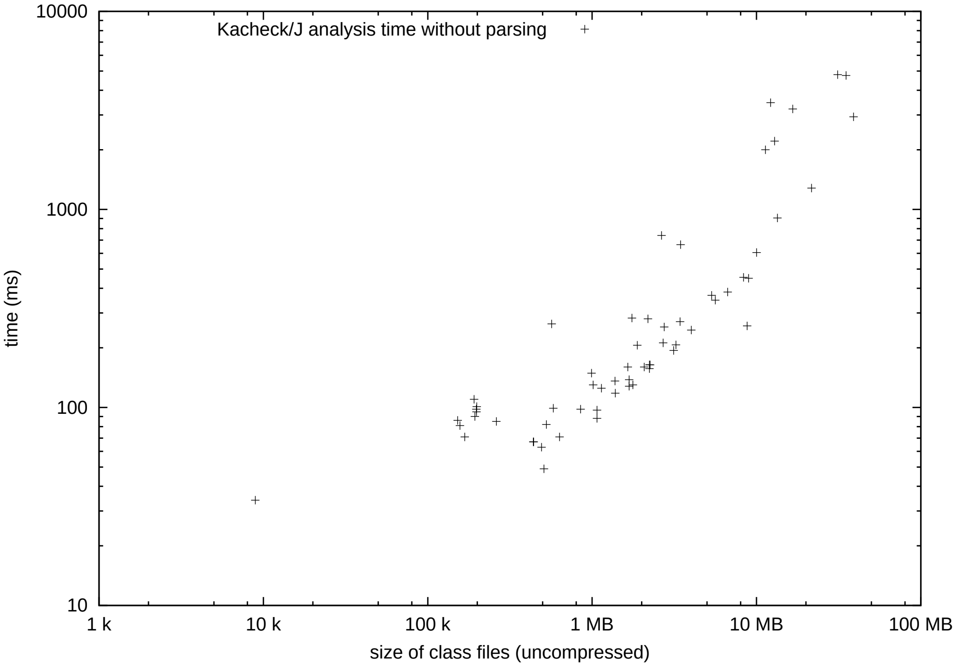

Figure 20: Performance of the analysis framework for Kacheck/J on a PIII-800 running Sun JDK 1.4.1.

5.5 Code Manipulation

Code manipulation of OvmIR is performed by `Editor` objects. Editors operate on instruction buffers and provide two abstractions, `Cursor`s and `Marker`s. Cursors are used for inserting instructions. Markers act as symbolic jump targets. Like the analyses, editing is typically performed by a visitor that iterates over the code in some application specific order. A transformation consists of a sequence of edit operations followed by a commit. The original code remains visible until the commit is performed.

Cursors have methods to create all of the concrete instructions. These methods often provide slightly higher-level abstractions than the actual instructions of the IR. For example, the cursor will emit appropriate code for a branch instruction, *i.e.* either a short branch or a sequence of instructions that use a combination of short branch and long jump. Similarly, an insertion operation like load constant will automatically choose the best instruction for the given value (such as `ICONST3` for 3, or `BIPUSH(42)` for 42). The required constant pool entries are also automatically generated.

The editor uses stateful instruction objects. The reason for this is that code generation requires context information, such as the location of a jump target, that is not always available before a commit. Correctness of the resulting code is not enforced. In fact, the cursor API allows the insertion of arbitrary sequences of bytes into the instruction stream.

Complex modifications such as code motion are possible within the framework, but can quickly become complicated to express since the client code is not provided with any automated mechanisms to deal with data-flow. Moving a single instruction such as `IADD` will always be difficult because the surrounding instructions that first prepare the operand stack and later process the result must also be moved. An SSA-based IR would make these dependencies more explicit. We are planning to provide a higher-level abstraction of the IR that will give client code a more SSA-like view.

6 Idiom Recognition

Because OvmIR is the input accepted by the interpreter, ahead-of-time compiler, and just-in-time compiler, a uniform approach is possible to the various unsafe or “magic” operations (*e.g.* memory dereferencing) that must be available to a VM implementation but cannot be expressed in pure Java. Most such operations are very short sequences of code, often heavily executed, so that it would not be practical to implement them with a heavyweight mechanism such as the Java Native Interface; in any case a single native library implementing the operations would not suffice, as the interpreter and compilers will use different techniques to realize the same operations.

We include in OvmIR a small set of opcodes for non-Java primitive operations that can, either singly or in combination, express the special operations Ovm needs. The interpreter and compilers simply need to support these opcodes as well. To make the extended semantics available at the level of Ovm’s Java source, we do *idiom recognition*[9]: we attach special semantics to certain idiomatic uses of standard Java constructs and, in an OvmIR-to-OvmIR transformation pass, replace those idioms with appropriate IR sequences. The transformation pass can be iterated to a fixed point, allowing new idioms to be defined in terms of existing ones. A report[4] more fully describes this aspect of Ovm; this section provides a brief overview. Figure 1 illustrates the architecture.

By *tag idiom*, we denote an idiom that can be recognized unequivocally, such as implementing a specific marker interface or including a marker exception type in a throws clause. By *heuristic idiom*, we mean one that is detectable with some finite risk of false negatives or false positives, such as a sequence of related operations in some order (perhaps altered by javac before the recognizer sees it). We use *guarded idiom* to name the combination of those techniques, a heuristic idiom that is associated with a tag and is only recognized where the tag is used. Guarded idioms can be elaborate enough to express special operations at a usefully high level, while eliminating the risk of false positives and mitigating false negatives: if the tag is seen and the rest of the idiom is not recognized a warning can be given.

```
public static ObjectModel getObjectModel()
    throws InvisibleStitcher.PragmaStitchSingleton {
    return (ObjectModel) InvisibleStitcher
        .singletonFor( "ovm.core.domain.ObjectModel"); }
```

Figure 21: A stitched Abstract Factory method. It returns singleton of an abstract implementing class. The stitcher idiom transforms all call sites to constant-loads of the singleton.

Ovm provides a small set of primitive tag idioms, based on marker interfaces and exceptions, with which other tag and guarded idioms can be defined. We define a new idiom by subclassing an existing tag and writing its IR transformation in terms of the Editor and Cursor interfaces. No other code—in particular, no compiler or interpreter code—needs to be touched to implement a new idiom as long as its semantics can be expressed with some sequence of existing OvmIR operations. The idiom in the following example has a definition that occupies 32 lines in a single source file.

Figure 21 illustrates an idiom we use throughout to achieve configurability using the familiar Abstract Factory design pattern[6] without paying the indirection cost that might otherwise weigh against heavy use of this pattern in a VM.

7 Customizing the Instruction Set

The Ovm framework allows us to experiment with many aspects of the virtual machine design space, some of which require changes to the instruction set. Consider, for instance, a change to the VM internal data structures that

```
static class LDC_W extends ConstantPoolLoad { {
    CPIndexValue val = new CPIndexValue(
                    CPIndexValue.CONSTANT_Any,
                    TypeCodes.USHORT);
    streamIns = new IntValue[]  val;
    stackOuts = new Value[] { new Value(
        new CPAccessExp( val)) };
    exceptions_ = new TypeName.Scalar[] {
        VIRTUALMACHINE_ERROR };
  }
  final TypeName.Scalar[] exceptions_;
  TypeName.Scalar[] getThrowables() {
        return exceptions_; }
}

static class LDC_DW extends LDC_W { {
    CPIndexValue val = new CPIndexValue(
            CPIndexValue.CONSTANT_Any, TypeCodes.UINT);
    streamIns = new IntValue[] val;
    stackOuts = new Value[] {
            new Value( new CPAccessExp( val)) }
}
```

Figure 22: The specification of Java's LDC_W instruction, which loads a constant using a 16 bit index, and LDC_DW, which loads a constant using a 32 bit index.

merges per-class constant pools into a global shared data structure (as done in Jikes [2]). The current limit on constant pool indices (65535) must be overcome by extending the OvmIR with new instructions to load values from 32 bit constant pool indices.

To define the new instructions LDC_DW and LDC_2D, one must define compile-time constants for the new opcodes, define the new instructions, and add these new instructions to the instruction set. This amounts to adding roughly 20 lines of code. One must also teach the Cursor class used in bytecode rewriting how to generate the new instructions.

As shown in Figure 22, the specification for LDC_DW is similar to that of LDC_W. The only difference between the two is the size of the integer constant in streamIns. This value is used by ConstantPoolRead.getCPIndex to decode the immediate operand statically and by the interpreter generator to choose the decoding macro to invoke at runtime.

Finally, one must check that the new instruction is supported by all visitors that may encounter the new opcode. For LDC_DW, the only place that requires a change is the CloneInstructionVisitor that needs to use the new method in the Cursor to clone the LDC_DW instruction. More elaborate additions of new instructions, especially if they can not simply be modelled as subtypes of existing instructions, typically require slightly more extensive additions to visitors that need to handle the new instruction.

8 Related work

Interpreter generation has been employed by many systems, among them Java Virtual Machines (*e.g.* Hotspot) and Scheme interpreters ([11]). Recent refinements in interpreter implementation techniques [5] suggest that a sophisticated interpreter may be a valid alternative to an unsophisticated dynamic compiler. The Virtual Virtual Machine project aims to build a *dynamically* reconfigurable virtual machine capable of running a variety of bytecode based languages [1]. vmgen [3] is a tool capable of generating interpreters for a variety of virtual machines. vmgen accepts instruction definitions in form of textual description in a special purpose programming language. We decided to express the instruction definitions directly as Java data structures and hence removed the necessity of a parser for the special purpose language. In effect, we have sped up the development process (since no parser is needed) as well as the development cycle (since the textual definition does not have to be processed to generate a representation the Java tools can work on).

The joeq virtual machine [16] is a compiler infrastructure that provides an intermediate representation modeling Java bytecode. The latest source distribution of joeq uses two variations of the visitor pattern to access the bytecode IR. The first variant is a nonhierarchical visitor that uses a switch statement on the opcode to dispatch to visit methods for the concrete instruction classes (e.g., GETFIELD). The visit methods take opcode-specific arguments that specify the parameterization of the instruction (for example, the selector of a field). These arguments are retrieved from the instruction stream by the code in the switch statement. The other style of visitor is a hierarchical visitor that dispatches over an instruction hierarchy with multiple inheritance, similar to the one used in Ovm. The major difference is that joeq uses accept methods that always sequentially call all applicable visit methods in abstract-to-concrete order (e.g., ExceptionThrower, StackConsumer, StackProducer, TypedInstruction, LoadClass, CPInstruction, FieldOrMethod, FieldInstruction, and GETFIELD for GETFIELD.accept()). For this visitor, information from the instruction stream is passed to the visitor using state in the instruction objects, preventing the use of singleton instruction objects for all parametrized opcodes. In Ovm, *all* instructions are flyweight objects and the required state is passed as an argument to the methods of the instructions that decode the bytecode stream. Notably, joeq's bytecode-related analyses use the switch-based nonhierarchical visitor almost exclusively in favor of the more object-oriented hierarchical visitor. Joeq also has a hierarchical visitor for its quad-based IR, which is similar to the hierarchical visitor for the bytecode IR.

9 Future Work

The instruction hierarchy is mostly ad hoc and tied to the Java bytecode instruction set. However, instead of static categorization in terms of the type system, one can envision dynamic inference of bytecode specification properties since most of the necessary information is present in the IR definition of instructions. For example, it is possible to infer that `IINC` both reads and writes local variables because of the way that `LocalAccess` code sources are used in its definition. Instructions could be dynamically annotated with appropriate properties in the inference phase. This would remove the burden of manual categorization from the programmer and ensure that categorization is consistent with the semantics given by the IR. This new approach assumes a static set of properties according to which the inference would proceed. As an extension, one could provide a way to specify new properties outside of the standard set supplied by the framework. Tools interested in such properties would have a way to specify them, and the inference engine would be able to discover these properties. For example, a tool performing stack height inference might be interested in identifying all the instructions that do not change the stack height. The tool would register a predicate, for example.

```
boolean hasInvariantStackHeight(Instruction instr) {
    return instr.stackIns.length == instr.stackOut.length;
}
```

The instruction property inference engine would annotate the appropriate instructions with the `InvariantStackHeight` property. The stack height inference tool would then ignore all instructions that have this property.

Currently, expression trees are given for each individual instruction. Optimizing compilers, however, typically operate on intermediate representations of entire functions. It is conceivable to convert the expression trees forming instruction definitions to standard three address code form and obtain an intermediate representation of any function by combining the expression trees from the sequence of bytecodes constituting the definition of this function.

10 Conclusion

We have presented how certain design patterns can be used to build an extensive set of VM components around the specification of an intermediate representation. The resulting IR is customizable and the components using the IR can be easily adapted to changes in the configuration. Static analysis tools that use our implementation of a bytecode-based IR show competitive performance.

Acknowledgments. This work is supported by grants from DARPA (PCES), and NSF (CCR–9734265). The authors are grateful to Tony Hosking, Doug Lea, David Holmes and Krista Bennett for their comments and Jacques Thomas for implementing ovmp. We thank the anonymous reviewers of IVME'03 as well as the attendees for valuable comments.

References

[1] B. Baillarguet and I. Piumarta. A highly-configurable, modular system for mobility, interoperability, specialization, and reuse. In *2nd ECOOP Workshop on Object-Orientation and Operating Systems*, June 1999.

[2] M. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño dynamic optimizing compiler for Java. In *Proceedings ACM 1999 Java Grande Conference*, pages 129–141. ACM, June 1999.

[3] M. Anton Ertl and David Gregg. Building an interpreter with vmgen. In *Compiler Construction (CC'02)*, pages 5–8. Springer LNCS 2304, 2002.

[4] C. Flack, T. Hosking, and J. Vitek. Idioms in Ovm. Technical Report CSD-TR-03-017, Department of Computer Sciences, Purdue University, 2003.

[5] E. Gagnon and L. Hendren. Effective inline-threaded interpretation of java bytecode using preparation sequences. In *Compiler Construction, 12th International Conference*, Jan 2003.

[6] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[7] Christian Grothoff. Walkabout revisited: The runabout. In *ECOOP 2003 - Object-Oriented Programming*, Berlin, Heidelberg, New York, 2003. Springer-Verlag.

[8] Christian Grothoff, Jens Palsberg, and Jan Vitek. Encapsulating objects with confined types. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 241–253. ACM SIGPLAN, 2001.

[9] David Hovemeyer, William Pugh, and Jaime Spacco. Atomic instructions in java. In Magnusson [13], pages 133–154.

[10] Martin E. Nordberg III. Variations of the Visitor Pattern. 1996.

[11] Richard A. Kelsey and Jonathan A. Rees. A tractable Scheme implementation. *Lisp and Symbolic Computation*, 7(4):315–335, 1994.

[12] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, Reading, Massachusetts, 1996.

[13] Boris Magnusson, editor. *ECOOP 2002 — Object-Oriented Programming: 16th European Conference, Málaga, Spain: proceedings*, volume 2374 of *Lecture Notes in Computer Science*, Berlin, Heidelberg, New York, June 2002. Springer-Verlag.

[14] Jens Palsberg and Neal Glew. Type-safe method inlining. In Magnusson [13], pages 525–544.

[15] Olin Shivers. *Control-flow analysis of higher-order languages.* PhD thesis, Carnegie Mellon University, 1991.

[16] John Whaley. Joeq: A Virtual Machine and Compiler Infrastructure. In *ACM SIGPLAN 2003 Workshop on Interpreters, Virtual Machines and Emulators (IVME 2003).* ACM SIGPLAN, 2003.