Briki: a Flexible Java Compiler [*]

Michał Cierniak
Wei Li

Technical Report 621
Department of Computer Science
University of Rochester
Rochester, NY 14627
{cierniak,wei}@cs.rochester.edu

May 1996

Abstract

We present a Java compiler architecture which uses a unique combination of front- and back-ends to deliver great flexibility. Our compiler is designed to use the same optimization passes no matter which pair of front- and back-end is used.

The compiler can be configured as a traditional stand-alone compiler (which compiles Java source into Java bytecodes). Another configuration can be used as an on-the-fly optimizer (which optimizes applets as they are loaded from the network). We can also compile Java directly to native code, thus using Java as a replacement for one of the traditional programming languages. Other interesting setups are also possible.

This flexibility is achieved by using a common intermediate representation, *JavaIR* (Java Intermediate Representation). Multiple front-ends convert various input formats (Java source, bytecode) into JavaIR. Once represented as JavaIR an application can be transformed with any of the existing *passes*. The modified JavaIR form of an application can be written out in one of the supported output formats.

Although compilers with multiple front- and back-ends have already existed, our approach is unique in supporting high-level code transformations even on applications which are distributed without the source program. Since Briki is written in Java it can be easily integrated into any Java application (e.g. a WWW browser) which dynamically loads applets from the network to provide capabilities of on-the-fly optimization.

[*]This work was supported in part by an NSF Research Initiation Award (CCR-9409120) and ARPA contract F19628-94-C-0057.

1 Introduction

One of the goals of the design of the Java programming language has been to enable the creation of portable applications which can be distributed over the network. One possible use of such applications (called applets) is by including them in HTML pages which can be accessed with World Wide Web browsers, such as Netscape Navigator, HotJava or Internet Explorer. The definition of Java includes both the definition of the programming language itself [6] and the definition of the virtual machine [7] which can run compiled Java applications distributed in the form of *bytecodes* [4]. An application in the bytecode form can run on any computer with an implementation of the Java virtual machine.

As the technology matures, advanced compiler optimizations are being proposed to speed up execution of Java applications (unless explicitly stated otherwise, by a "Java application" we mean either a stand-alone application or an applet). Many techniques used to optimize programs written in other programming languages can be successfully applied to Java programs, other techniques are unique to Java. This paper does not describe any of those specific optimizations. Rather, we present a new way of applying arbitrary optimization techniques. We are developing an optimizing compiler with multiple front- and back-ends.

The idea of re-using much of the compiler code for different programming languages (by having multiple front-ends) and for different target machines (by having multiple back-ends) is not new. There exist commercial compiler systems which use this approach to create multiple compilers which share the same code base.

Java allows something else. It is possible to write a compiler which could apply the same optimizations to all Java applications whether they are distributed with the source code or as bytecode files (`.class` files). This can be accomplished by having two front-ends: one which would read Java source and another one which would read bytecode. Similarly, the same compiler could generate the target code as Java source, bytecode, or possibly native code for a given machine (either directly or by generating other high-level language, such as C, which can be compiled into efficient native programs). A diagram presenting the structure of our compiler is shown in Figure 1. JavaIR (Java Intermediate Representation) is used by us to represent Java programs.

Java is the first popular programming language which allows high-level optimizations on the applications distributed without the source code. Previously, all the optimizations had to applied before the binary distribution was produced. Once a user purchased an application without source, it would become immutable even if the progress in compiler technology brought new techniques which could significantly speed up this type of an application. Java bytecode made it possible to develop a compiler which could directly optimize applications distributed without source.

Depending on the combination of front- and back-ends the compiler can be used in different ways. The most common combinations that we anticipate are:

- **Java to bytecode** A "conventional" compiler — a replacement for `javac` [8].
- **bytecode to bytecode** This can be used to optimize applications distributed without source. One could for instance integrate such a compiler with a web browser and perform optimizations on-the-fly.

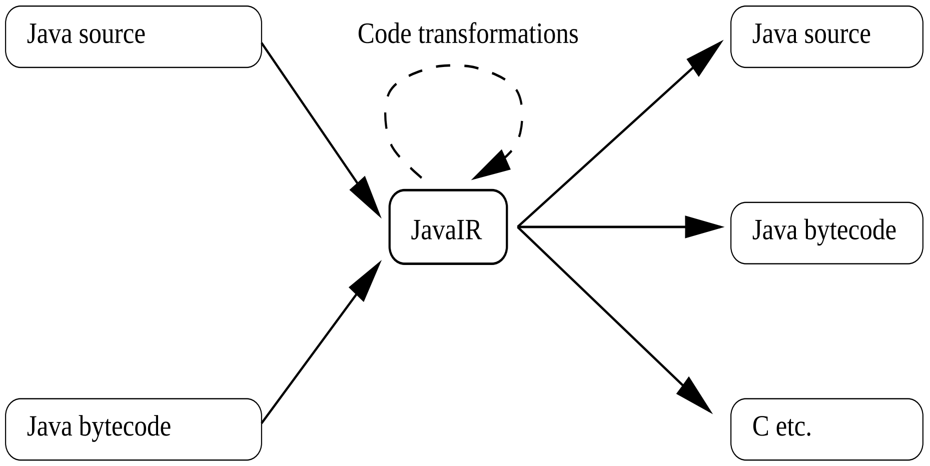


Figure 1: An overview of the compiler

- **Java to Java** Code-cleaning, pretty-printing. Also new optimizations which are just being developed can be debugged much easier if the output is human-readable.

- **bytecode to Java** This allows, for instance, profiling an application distributed without source, yet see the profiling information in a source format. This is an invaluable tool for researchers who want to understand the behavior of existing applications to design new optimization techniques.

- **bytecode to native code** On-the-fly (just-in-time) compilation.

- **Java to native code** Off-line compilation to native code. Off-line compilation to native code would let us use Java as a replacement for one of the traditional languages (C, C++, etc.)

It is also possible to compile other programming languages [9] into JavaIR. This approach has been proposed to allow for distribution of non-Java programs in the bytecode form. It would be of course possible to write a front-end to our compiler for any of the languages which can be compiled into Java bytecode.

All those combinations of front- and back-end can share the same code-transformation passes. One interesting application developed with our compiler uses the bytecode to Java set to instrument `.class` files. We have developed a profiling tool called *NetProf* [5] which instruments Java applications distributed as bytecode and interactively presents the profiling information in a visualization applet. That way we can profile applets over a network without explicit downloading.

The rest of the paper describes our compiler and the status of its implementation. Section 2 describes the compiler at a high-level. Section 3 gives an overview of our universal intermediate representation for Java programs. Selected features of the compiler are demonstrated on examples in Section 4. Restrictions on current implementation are explained in Section 5.

2 Implementation

Our main design goal was to build a compiler that can re-use the same transformation passes while working on different source and target representations of a Java program. There were also other requirements that we had on mind:

- The intermediate representation should be high-level enough that most of the information contained in the source program is easily accessible and reasoning about the program can be done in source-level terms.
- The compiler should be written in Java, so that it can be easily integrated into network applications to perform, for instance, on-the-fly (just-in-time) compilation/optimization or profiling over the network.
- The compiler should be light-weight so that the overhead of integrating it into other applications would be minimized.

After considering the above requirements, we made the following design decisions. We chose to design our intermediate representation to be a syntax tree. Although other representations exist in contemporary compilers, a syntax tree structure is common in the advanced research compilers such as Polaris [1, 3], SUIF [10] or Sage++ [2]. As our intermediate representation for Java, we have designed *JavaIR* (Java Intermediate Representation) which is a syntax tree whose structure follows the source code as close as possible. JavaIR is discussed in more detail in Section 3.

We have designed our front- and back-ends so that they can be used as parts of both applets or stand-alone applications. A compiler driver includes calls to the front-end(s) which read the input from a Java `InputStream` and back-end(s) which can write the output to an `OutputStream`. In addition to using stream as I/O, our compiler can use other data formats if they are more appropriate. For instance, our Java source back-end can generate the source in an array of strings, to make visualization simpler and more efficient.

Note that the security models for Java applets in existing WWW browsers does not allow for dynamic bytecode transformations, but we anticipate that this situation will change as the technology matures. From a technical point of view, it is trivial to modify a browser so that every class is being passed to our compiler before it being loaded.

3 Intermediate Representation

The intermediate representation is a syntax tree with a node for every Java class defined in the input. Every class node contains references to nodes for all interfaces, fields and methods defined for that class as well as some other information (access flags, a reference to the super class etc.). *JavaIR* is our implementation of this intermediate representation. An overview of JavaIR is presented in this section.

Figure 2 shows a class hierarchy for a subset of the classes used in JavaIR. For simplicity of the discussion assume that the compiler reads and stores only one class. The extension to handle multiple classes is straightforward.

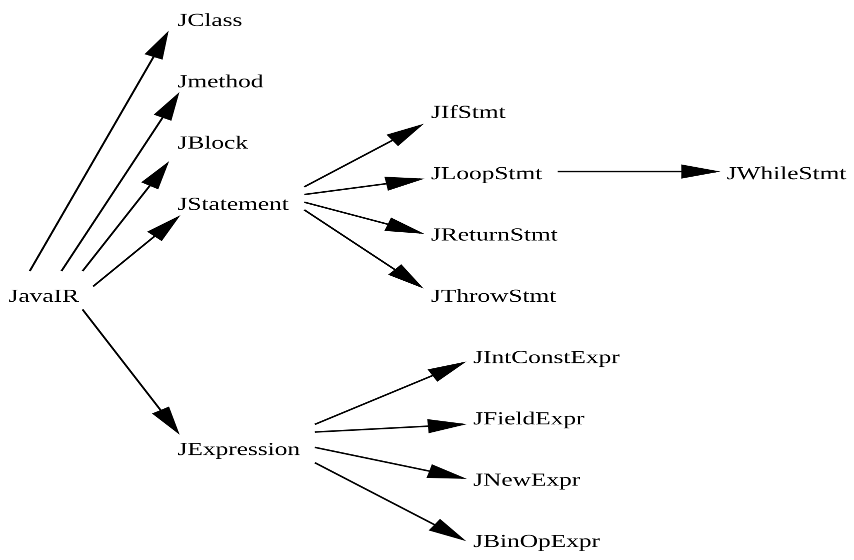


Figure 2: Class hierarchy of the nodes in the JavaIR intermediate representation (only selected expression and statement types are shown)

We will use the following, very simple example to illustrate how JavaIR represents Java programs.

```
class simple {
   int i;
   public simple() {
      i = 0;
   }
} //simple
```

A complete JavaIR representation of class `simple` takes care of many details that are required to maintain the meaning of the program. The image gets even more complicated by the fact that some information is replicated so that it can be accessed quickly. We show a subset of the JavaIR form of `simple` in Figure 3. Not all nodes are shown and not all fields are present. Each box represents a Java object which implements a node in JavaIR. The name in the top left corner is the name of the class which is the type of a given node. All other lines represent fields with field name on the left side (in italics) and the value on the right side. Whenever a value is a reference, it is shown as an arrow pointing to an object. To keep the size of Figure 3 maintainable, we have omitted the symbol table for the constructor and the representation of the statement `super()` and the expression `this`.

The JavaIR representation is not *exactly* the same as the original source. Instead, JavaIR corresponds exactly to the code generated by the compiler. In our example, the code for the constructor contains *implicitly* two additional statements:

```
   public simple() {
      super();      // implicit constructor call
```

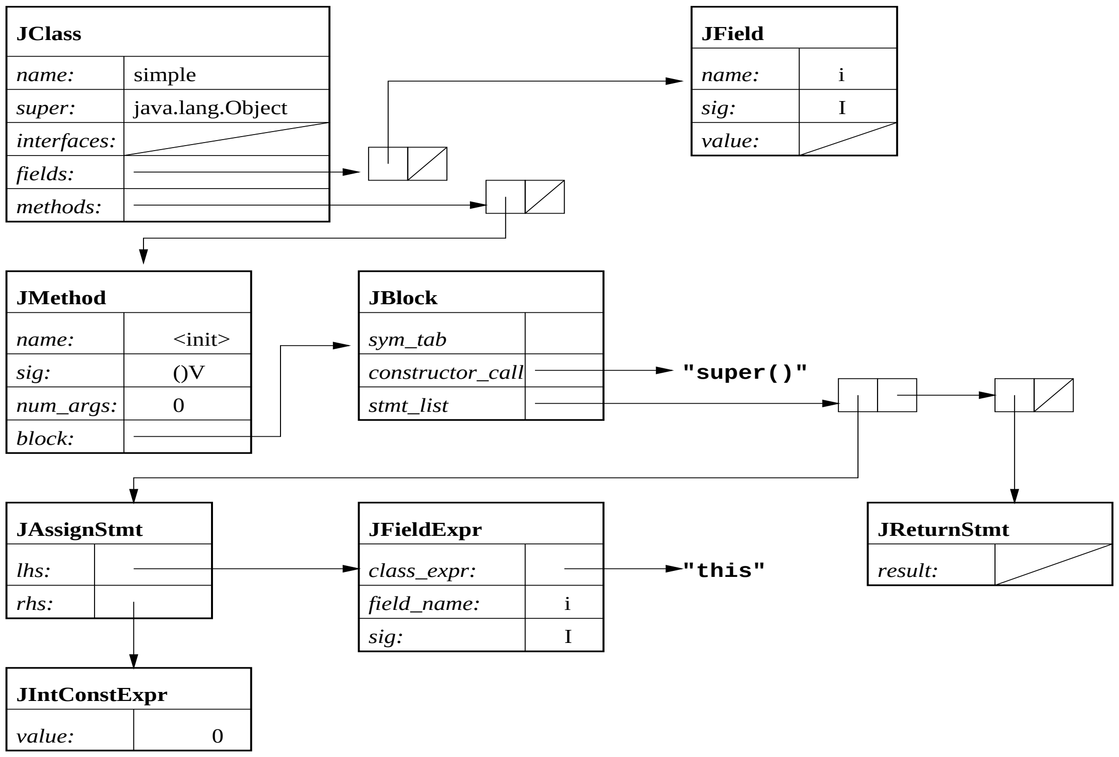


Figure 3: A simplified view of the JavaIR representation of the class `simple`

```
        i = 0;
        return;        // implicit return statement
    }
```

All implicit methods, statements and expressions are represented explicitly in JavaIR. Note that the constructor call is not inserted in the same list as all other statements. The reason is that the constructor call is sufficiently different in its properties that we decided to keep a reference to it in separate field

All nodes in the syntax tree are objects of some class derived from the class `JavaIR` (see Figure 2). The intermediate representation defines classes which are not derived from `JavaIR`, but none of them can be used as a node in the tree. A class is represented with an object of a class `JClass` which contains lists of objects representing fields, methods and interfaces. Let us discuss the representation of methods (fields and interfaces follow the similar idea, but are much simpler). As can be seen in Figure 3, a method is represented with an object of the class `JMethod`. If the method is not native, this object contains a reference to an object of type `JBlock` which in turn contains a reference to the symbol table and a list of statements. A statement may contain other statements or expressions.

4 Code Transformations

This section explains on some examples what kind of transformations are performed by our compiler. For lack of space, we do not include technical details about how the transformations are being performed. We mostly follow known techniques which we customized for the specific need of the Java language.

4.1 Recovering High-level Structure

One of the important features of our compiler is the ability to recover high-level structure of the source Java program given its bytecode representation. Consider the following code fragment of a `.class` file as disassembled by `javap` [8]. It is not apparent what this sequence of bytecodes does.

```
 0 iload_1
 1 iconst_1
 2 if_icmpeq 10
 5 iload_1
 6 iconst_3
 7 if_icmpne 18
10 aload_0
11 iconst_1
12 putfield #4 <Field cond1.i I>
15 goto 23
18 aload_0
19 iconst_2
20 putfield #4 <Field cond1.i I>
23 aload_0
24 dup
25 getfield #4 <Field cond1.i I>
28 iconst_1
29 iadd
30 putfield #4 <Field cond1.i I>
```

One can of course understand this code with some effort. Nevertheless, many compiler algorithms are best expressed in terms of high-level constructs like a loop or an assignment. Our compiler parses the above sequence of Java VM instructions and generates an equivalent JavaIR representation which can be printed as the following Java source fragment (as described earlier the internal JavaIR representation is not directly printable since it is a collection of Java objects with references to each other).

```
if(((a1 == 1) || (a1 == 3))) {
   this.i = 1;
} else {
    this.i = 2;
} //if
this.i = (this.i + 1);
```

Note that `a1` is used to represent the first argument to the method containing this sequence of bytecodes. The identifier used by the programmer in the source cannot be recovered. This makes a difference only if we intend to de-compile the source for human use. For the compiler the identifier is not relevant. What is important is the type of the parameter and this can be easily recovered from the bytecode representation of a class.

4.2 Instrumentation

Instrumentation is an example of a compiler transformation which can be easily implemented with our compiler. To visualize the performance of a program, we would like to instrument source level constructs, so that we are able to say how many times a loop was executed, how often was the else-branch of an if-statement taken, etc. Instrumenting the code with method invocations at the beginning and end of all interesting control-flow blocks is trivial if we do it at the JavaIR level.

NetProf [5] is a set of tools which can visualize performance information over a network. NetProf first instruments a class that is to be profiled, then the application is executed, and after it terminates a visualization tool presents the performance in the terms of the recovered Java source. The instrumentation is done with our Java compiler configured as shown in Figure 4.

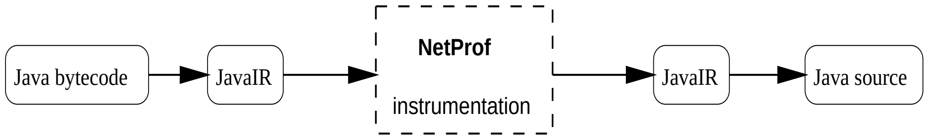


Figure 4: NetProf instrumentation

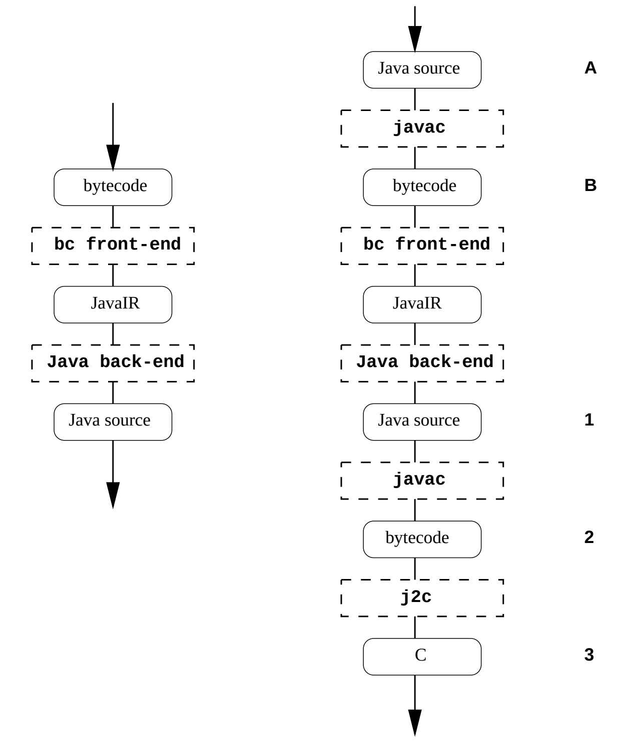
The dashed box in Figure 4 shows the instrumentation pass which had to be written for NetProf. All the other actions shown the diagram are handled by the compiler: bytecode is read from a Java stream (e.g. a file), it is converted to JavaIR and after the instrumentation takes place the transformed JavaIR is written out as Java source. The instrumentation pass was created with relatively little effort and all code needed to insert calls to the instrumentation library and adding new fields to the instrumented class is contained in a file less than 400 lines long.

4.3 Generating Java Source

Given that the structure of JavaIR follows closely the structure of a Java program, the Java source back-end is very simple. It walks the JavaIR tree and recursively prints its subtrees. The Java source in Section 4.1 was generated automatically with the Java source back-end.

5 Status of Our Implementation

Currently we have implemented only one front-end, bytecode and one back-end, Java source. Given finite resources the choice was made since it is both most interesting and most general. Figure 5(a) shows the current status of our implementation.



(a) Current implementation (b) Constructing "virtual" front- and back-ends by using publicly available compilers

Figure 5: Current status of our compiler.

Of the two front-ends of interest: bytecode and Java source, we picked the bytecode front-end since it is more difficult to develop and we wanted to make sure that JavaIR is designed so that it can be recovered from bytecode. Since the structure of JavaIR follows closely the syntax of Java programs, developing a front-end for Java source will be relatively easy, especially that there exist grammars for Java in a form which can be used to automatically generate a parser for Java. It is even possible to obtain the source code for the Java compiler developed by Sun Microsystems.

Choosing the bytecode front-end has an additional advantage that we can use any of the publicly available Java to bytecode compilers, such as `javac` which is distributed as part of the Java Developer's Kit [8] to create a "virtual" Java front-end. Furthermore, if compilers from other languages to the Java bytecode are developed, we can also create virtual front-ends for those languages. One such compiler that is available now is AppletMagic [9] developed by Intermetrics. AppletMagic compiles Ada 95 to Java bytecode.

Choosing Java source as the back-end has a similar advantage. We can use `javac` [8] to compile the output of our compiler to bytecode. The bytecode can be further compiled into C using `j2c` [11]. This gives us two additional virtual back-ends. Figure 5(b) shows those possibilities with the front-ends marked with capital letters and the back-ends marked with numbers. All possible letter–number combinations give a valid compiler. There are six such compilers available now (nine if we include AppletMagic).

We plan to implement some of the other front- and back-ends directly in our compiler. The highest priority is for the bytecode back-end which is necessary for on-the-fly optimizations.

6 Conclusion

We have presented a Java compiler infrastructure which can be used to easily build optimizing compilers, code instrumentation tools, de-compilers, pretty-printers, code-verifiers and other programs which transform or analyze Java applications. Our compiler works on Java applications distributed in any form – even if the source program is not available.

Since the compiler itself is written in Java, it is easy to integrate it into most Java software (e.g. WWW browsers) which dynamically loads Java applets from the network.

References

[1] B. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, B. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Polaris: The Next Generation in Parallelizing Compilers. In K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing: 7th International Workshop*, volume 892 of Lecture Notes in Computer Science. Springer-Verlag, Berlin/Heidelberg, 1995.

[2] F. Bodin. Sage++: An Object-oriented Toolkit and Class Library for Building Fortran and C++ Restructuring Tools. In *Proceedings of the Second Annual Object-Oriented Numerics Conference*, 1994.

[3] K. A. Faigin, J. P. Hoeflinger, D. A. Padua, P. M. Petersen, and S. A. Weatherford. The Polaris Internal Representation. Technical Report 1317, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign.

[4] J. Gosling. Java Intermediate Bytecodes. In *Proceedings of the ACM SIGPLAN Workshop on Intermediate Representations (IR '95)*, pages 111–118, San Francisco, CA, January 1995. In *ACM SIGPLAN Notices 30*:3 (March 1995).

[5] S. Parthasarathy, M. Cierniak, and W. Li. NetProf: Network-based High-level Profiling of Java Bytecode. May 1996. submitted for publication.

[6] Sun Microsystems. The Java Language Specification. October 30, 1995. Version 1.0 Beta.

[7] Sun Microsystems. The Java Virtual Machine Specification. August 21, 1995. Release 1.0 Beta.

[8] Sun Microsystems. The Java(tm) Developer's Kit. May 1996. Version 1.0. Available at http://java.sun.com/java.sun.com/products/JDK/index.html.

[9] S. T. Taft. Programming the Internet in Ada 95. March 1996. submitted for publication.

[10] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *ACM SIGPLAN Notices*, 29(12):31–37, December 1994.

[11] J2c/CafeBabe java .class to C translator. January 1996. Available at http://www.webcity.co.jp/info/andoh/java/j2c.html.