# A Gcc-based Java Implementation

Per Bothner
Cygnus Solutions
<bothner@cygnus.com>

## Abstract

*While a major factor in Java's success is its use of portable bytecodes, we believe it cannot become a mainstream programming language without mainstream implementation techniques, specifically an optimizing ahead-of-time compiler. This allows much better optimization, and much faster application start-up times than with JIT translators. Cygnus is writing a Java front-end for the GNU compiler (`gcc`) to translate Java bytecodes to machine code. This uses proved and widely used technology. The meta-data representing a class a run-time will be laid out by the compiler in static data memory, saving more startup time. We will use the GNU linker to link compiled class files into standard shared or static libraries. For the run-time environment, we are enhancing the existing Kaffe free Java VM to make it full-strength and to support linking with pre-compiled class libraries. We will enhance the GNU debugger (`gdb`) to understand Java, which will provide a familiar and multi-language debugging environment.*

## 1. Introduction

Java [2] has taken off because it is a decent programming language, is buzzword-compliant (object-oriented and web-enabled), and because it is implemented by compiling to portable bytecodes [3]. However, interpreting bytecodes makes Java program many times slower than comparable C or C++ programs. One approach to improving this situation is "Just-In-Time" (JIT) compilers. These dynamically translate bytecodes to machine code just before a method is first executed. This can provide substantial speed-up, but it is still slower than C or C++. There are two main drawbacks with the JIT approach compared to conventional compilers:

- The compilation is done every time the application is executed, which means start-up times are much worse than pre-compiled code.
- Since the JIT compiler has to run fast (it is run every time the application is run), it cannot do any non-trivial optimization. Only simple register allocation and peep-optimizations are practical. (Some have suggested that JIT could potentially produce faster code, since it can dynamically adjust to specific inputs and hardware. But the need for quick re-compilation will make it very difficult to make JIT faster in practice.)

While JIT compilers have an important place in a Java system, for frequently used applications it is better to use a more traditional "ahead-of-time" or batch compiler. While Java has been primarily touted as an internet/web language, many people are interested in using Java as an alternative to traditional languages such as C++, if the performance can be made adequate. For embedded applications it makes much more sense to pre-compile the Java program, especially if the program is to be in ROM.

It has been argued that even for ROM-based applications (where portability is not an issue), it still makes sense to use a bytecode-based implementation of Java, because the bytecodes are more compact than native code. However, the difference is more-or-less negated by the space taken up by the constant pool, though it may be possible for an "optimizing ROMizer" to reduce the space overhead. In any case the space advantage of bytecodes is minor at best, whereas the speed penalty is major.

So Cygnus is building a Java programming environment that is based on a conventional compiler, linker, and debugger, using Java-enhanced versions of the existing GNU programming tools. Since these been ported to just about every chip in use, we will have a uniquely portable Java system.

## 2. The compiler

The core tool is of course the compiler. This is `cc1java`, a new `gcc` [4] front-end. This has similar structure as existing front-ends (such as `cc1plus`, the C++ front-end), and shares most of the code with them. The most unusual aspect of `cc1java` is that its "parser" reads *either* Java source files or Java bytecode files. (The first release will only directly support bytecodes; parsing Java source will be done by invoking Sun's `javac`. A future

version will provide an integrated Java parser, largely for the sake of compilation speed.) In any case, it is important that `cc1java` can read bytecodes, for the following reasons:

- It is the natural way to get declarations of external classes (in this respect a Java bytecode file is like a C++ pre-compiled header file).
- It is needed so we can support code produced from other tools that produce Java bytecodes (such as the Kawa [1] Scheme-to-Java-bytecode compiler).
- Some libraries are (unfortunately) distributed as Java bytecodes without source.

Much of the work of the compiler is the same whether we are compiling from source code or from byte codes. For example emitting code to handle method invocation is the same either way. When we compiling from source, we need a parser, semantic analysis, and error-checking. This is fairly conventional, so I will not go into it.[1] On the other hand: when parsing from bytecodes, we need to extract higher-level information from the lower-level bytecodes. That is the topic of the next section.

## 3. Transforming bytecodes

(This section can be skipped if you are not interested in the internals of `cc1java`.)

The executable content of a bytecode file contains a vector of bytecode instructions for each (non-native) method. The Java bytecodes manipulate a set of local variables and a stack used for expression evaluation. (The first few local variables are initialized with the actual parameters.) Each local variable or stack "slot" is a word big enough for either a 32-bit integer, a float, or an object reference (pointer). (Two slots are used for 64-bits doubles and longs.) The slots are untyped: *e.g.* at one point a slot might contain an integer value, and at another point the same slot might contain an object reference. However, you cannot store an integer in a slot, and then retrieve the same bits re-interpreted as an object reference. Moreover, at any given program point, each slot has a unique type can be determined using static data flow. (The type may be "unassigned", in which case you are not allowed to read the slot's value.) These restrictions are part of Java security model, and are enforced by the Java bytecode verifier. We do a similar analysis in `cc1java`, which lets us know for every program point the current stack pointer, and the type of every local variable and stack slot.

Internally `gcc` uses two main representations: The `tree` representation is at the level of an abstract syntax tree, and is used to represent high-level (fully-typed) expressions, declarations, and types. The `rtl` (Register Transform Language) form is used to represent instructions, instruction patterns, and machine-level calculations in general. Constant folding is done using `tree`s, but otherwise most optimizations are done at the `rtl` level.

The basic strategy for translating Java stack-oriented bytecodes is that we create a dummy local variable for each Java local variable or stack slot. These are mapped to `gcc` "virtual registers," and standard `gcc` register allocation later assigns each virtual register to a hard register or a stack location. This makes it easy to map each opcode into `tree`-nodes or `rtl` to manipulate the virtual registers.

As an example, consider how to compile `iadd`, which adds the top two `int`s on the stack. For illustration, assume the stack pointer is 3, and virtual registers 50, 51, and 52 are associated with stack slots 0, 1, and 2. The generated code is:

```
vreg51 := vreg51 + vreg52
```

Notice that the stack exists only at compile-time. There is no stack, stack pointer, or stack operations in the emitted code.

This simple model has some problems:

- We would like to do constant folding, which is done at the `tree` level. However, `tree` nodes are typed.
- The simple-minded approach uses lots of virtual registers, and the code generated is very bad. Running the optimizer (with the `-O` flag) improves the generated code, but you still get a lot of useless stack slots. It would be nice to not *have* to run the optimizer, and if you do, not make so much unnecessary work for it.
- Even the `rtl` representation is semi-typed, since it distinguishes the various machine "modes", (such as pointer, and differently-sized integers and floats).

The last problem we solve by using a separate virtual register for each machine mode. For example, for local variable slot 5 we might use `vreg40` when it contains an integer, and `vreg41` when it points an object reference. This is safe, because the Java verifier does not allow storing an integer in an object slot and later reusing the same value as an object reference or float.

The other two problems we solve by modeling the stack as a stack of `tree` nodes, and not storing the results in their "home" virtual registers unless we have to. Thus we actually do the following for `iadd`:

```
tree arg1 = pop_value (int_type_node);
tree arg2 = pop_value (int_type_node);
push_value (fold (build (PLUS_EXPR,
        int_type_node, arg1, arg2)));
```

[1] Also, we have not started work on it!

The `build` function is the standard `gcc` function for creating `tree`-nodes, while `fold` is the standard function for doing constant folding. The function `pop_value` and `push_value` are specific to `cc1java`, and keeps track of which stack location corresponds to which `tree` node. No code is actually generated – yet.

This works for straight-line code (*i.e.* within a basic block). When we come to a branch or a branch target, we have to flush the stack of tree nodes, and make sure the value of each stack slot gets saved in its "home" virtual register. The stack is usually empty at branches and branch targets, so this does not happen very often. Otherwise, we only emit actual `rtl` instructions to evaluate the expressions when we get to a side-effecting operation (such as a store or a method call).

Since we only allocate a virtual register when we need one, we are now using fewer virtual registers, which leads to better code. We also get the benefits of `gcc` constant folding, plus the existing machinery for selecting the right instructions for addition and other expressions.

The end result is that we end up generating code using the same mechanisms that the `gcc` C and C++ front-ends do, and therefore we can expect similar quality code.

## 4. Class meta-data

Compiling the executable content is only part of the problem. The Java run-time also needs an extensive data structure that describe each class with its fields and methods. This is the "meta-data" or "reflective" data for the class. The compiler has to somehow make it possible for the run-time system to correctly initialize the data structures before the compiled classes can be used.

If we are compiling from bytecodes, the compiler can just pass through the meta-data as they are encoded in the class file. (The VM already needs to know how to read meta-data from a class file.)

It is inconvenient if the meta-data and the compiled code are in different files. The VM should be able to create its representation of the meta-data without having to go beyond its address space. For example reading in the meta-data from an external file causes undesirable consistency headaches, and it may not even be possible for embedded systems.

A possible solution is to emit something like:

```
static const char FooClassData[]
= "\xCa\xfe\xba\xbe...";

static {
  LoadClassFromByteArray(FooClassData);
  Patch up method to point to code;
}
```

The code marked `static` is compiled into a dummy function that gets executed at program startup. This can be handled using whatever mechanism is used to execute C++ static initializers. This dummy function reads the meta-data in external format from `FooClassData`, creates the internal representation, and enters the class into the class table. It then patches up the method descriptors so that they point to the compiled code.

This works, but it is rather wasteful in terms of memory and startup time. We need space for both the external representation (in `FooClassData`) and the internal representation, and we have to spend time creating the latter from the former. It is much better if we can have the compiler directly create the internal representation. If we use initialized static data, we can have the compiler statically allocate and initialize the internal data structures. This means the actual initialization needed at run is very little – most of it is just entering the meta-data into a global symbol table.

For example:

```
public class Foo extends Bar {
  public int a;
  public int f(int j) { return a+j; }
};
```

compiles into something like:

```
int Foo_f (Foo* this, int j)
{ return this->a + j; }

struct Method Foo_methods[1] = {{
   /* name: */   "f";
   /* code: */   (Code) &Foo_f;
   /* access: */ PUBLIC,
   /* etc */     ...
}};

struct Class Foo_class = {
   /* name: */         "Foo",
   /* num_methods: */ 1,
   /* methods: */     Foo_methods,
   /* super: */       &Bar_class,
   /* num_fields: */  1,
   /* etc */          ...
};

static {
  RegisterClass (&Foo_class, "Foo");
}
```

Thus startup is fast, and does not require any dynamic allocation.

## 5. Linking

The `cc1java` program creates standard assembler files that are processed by the standard unmodified GNU assembler. The resulting object files can be placed in a dynamic or static library, or linked (together with the run-time system) into an executable, using a standard linker. The only linker extension we really need is support for static initializers, but we already have that (since `gcc` supports C++).

While we do not *need* any linker modification, there are some that may be desirable. Here are some ideas.

The VM uses a lot of names – for classes, files, local variables, methods, type signatures, and source files. These are represented in the constant pool of `.class` files as `CONSTANT_Utf8` values. Compilation removes the need for many of these names (because it resolves many symbolic field and method references). However, names are still needed for the meta-data. Two different class files may generate two references the same name. It may be desirable to combine them to save space (and to speed up equality tests). That requires linker support. The compiler would place these names in a special section of the object file, and the linker will combine identical names.

The VM maintains a global table of all loaded classes, so it can find a class given its name. When most of the classes are statically compiled and allocated, it is reasonable to pre-compute (the static part of) the global class table. One idea is that the linker could build a perfect hash table of the classes in a library or program.

When we compile a class that makes a reference to some other class, we assume that class is also statically compiled. For example, if a method in class `B` calls a static method `f` in class `A`, we emit a direct function call to `A.f`. This is fast, but does not work if the other class is loaded at run-time. In other words while dynamically compiled code can call pre-compiled code, we do not support calling the other way (except for virtual or interface calls to overridden methods). I think this is a reasonable tradeoff, since I cannot think any reason why you would want to compile class `B` that directly references class `A`, without you also wanting to compile class `A`. But if you really wanted that ability, we might be able to do it with some extra indirection. The indirect references might be patched either at run-time or at link-time.

## 6. Run-time

Running a compiled Java program will need a suitable Java run-time environment. This is in principle no different from C++ (which requires an extensive library, as well as support for memory allocation, exceptions, and so on). However, Java requires more run-time support than "traditional" languages: It needs support for threads, garbage collection, type reflection (meta-data), and all the primitive Java methods. Full Java support also means being able to dynamically load new bytecoded classes, though this may not be required in some embedded environments. Basically, the appropriate Java run-time environment is a Java Virtual Machine.

It is possible to have `cc1java` produce code compatible with the proposed portable "Java Native Method ABI." Such code could run under any Java VM that implements the ABI. However, the portable ABI has relatively high overhead, so if you are not concerned about binary portability it is better to use a more low-level ABI, at the level of the JDK 1.0 native methods. While we plan to support the portable ABI (if it becomes popular), we will also support the lower-level ABI. Certainly the standard Java functionality (such as that in `java.lang`) will be compiled to the lower-level ABI.

A low-level ABI is inherently dependent on a specific VM. We are using Kaffe [5], a free Java VM written by Tim Wilkinson with help from volunteers around the "Net." Kaffe uses either a JIT compiler on selected platforms (Intel x86 and soon also Sun Sparc), or a conventional bytecode interpreter which is quite portable (except for thread support). Using a JIT compiler makes it easy to call between pre-compiled and dynamically loaded methods (since both use the same calling convention).

We are making many enhancements to make Kaffe a more suitable target for pre-compiled code. One required change is to add a hook so that pre-compiled and pre-allocated classes can be added to the global table of loaded classes. This means implementing the `RegisterClass` function used in section 4.

Other changes are not strictly needed, but are highly desirable. The original Kaffe meta-data had some redundant data. Sometimes redundancy can increase run-time efficiency (*e.g.* caching, or a method dispatch table for virtual method calls). But the gain has to be balanced against the extra complication and space. Space is especially critical for embedded applications, which is an important target for us. But I believe the simplifications made so far have been generally advantageous. We are also reducing the use of linked lists. In many cases, the meta-data structures has sets whose size are known *a priori*. In those cases it is usually better to use an array. This saves the space of the link fields, and improves locality. It also reduces the need for relocation of shared libraries. Fixed-size arrays also make it easier to implement the new reflection APIs in JDK 1.1.

## 7. Debugging

Our debugging strategy for Java is to enhance `gdb` (the GNU debugger) so it can understand Java-compiled code. This follows from our philosophy of treating Java like other

programming languages. This also makes it easier to debug multi-language applications (C *and* Java).

Adding support for Java requires adding a Java expression parser, and routines to print values and types in Java syntax. It should be easy to modify the existing C and C++ language support routines, since Java expressions and primitive types are very similar to those of C++.

Adding support for Java objects is somewhat more work. Getting, setting, and printing of `Object` fields is basically the same as for C++. Printing an `Object` reference can be done using a format similar to that used by the default `toString` method – the class followed by the address *e.g.* `java.io.DataInput@cf3408`. Sometimes you instead want to print the *contents* of the object, rather than its address (or identity). Strings should by default be printed using their contents, rather than their address. For other objects `gdb` can invoke the `toString` method to get a printable representation, and print that. However, there should be different options to get different styles of output.

`Gdb` can evaluate a general user-supplied expression, including a function call. For Java, this means we must add support for invoking a method in the program we are debugging. Thus `gdb` has to be able to know the structure of the Java meta-data so it can find the right method. Alternatively, `gdb` could invoke functions in the VM to do the job on its behalf.

`Gdb` has an internal representation of the types of the variables and functions in the program being debugged. Those are read from the symbol-table section of the executable file. To some extent this information duplicates the meta-data that we already need in the program's address space. We can save some file space if we avoid putting duplicate meta-data in the symbol table section, and instead extend `gdb` so it can get the information it needs from the running process. This also makes `gdb` start-up faster, since it makes it easier to only create type information when needed.

Potentially duplicated meta-data includes the source line numbers. This is because a Java program needs to be able to do a stack trace, even without an external debugger. Ideally, the stack trace should include source line numbers. Therefore, it is best to put the line numbers in a special read-only section of the executable. This would be pointed to by the method meta-data, where both `gdb` and the internal Java stack dumper can get at it.

Extracting symbolic information from the process rather than from the executable file is also more flexible, since it makes it easier to also support new classes that are loaded in at run-time. While the first releases will concentrate on debugging pre-compiled Java code, we will want to debug bytecodes that have been dynamically loaded into the VM. This problem is eased if the VM uses JIT (as Kaffe does), since in that case the representation of dynamically-(JIT-)compiled Java code is the same as pre-compiled code. However, we still need to provide hooks so that `gdb` knows when a new class is loaded into the VM.

Long-term, it might be useful to download Java code into `gdb` itself (so we can extend `gdb` using Java), but that requires integrating a Java evaluator into `gdb`.

## 8. Profiling

One problem with Java is the lack of profiling tools. This makes it difficult to track down the "hot-spots" in an application. Using `gcc` to compile Java to native code lets us use existing profiling tools, such as `gprof`.

## 9. Status

As of December 1996, the basic structure of the compiler (parsing bytecodes, generating trees, and emitting assembler) works. However, not all the bytecode operations have been implemented. Also, the code for emitting the meta-data has not been written.

Jeffrey Hsu (<hsu@cygnus.com>) is also working on this project.

The Kaffe VM works on most (non-graphical) Java code. We have enhanced it in various ways, and modified the data structures to be simpler and more suitable for being emitted by the compiler.

In other words: Many of the more difficult pieces have been written, but they do not quite fit together yet. Therefore we have no performance figures yet, but we expect efficiency similar to C++.

## References


[1] P. Bothner. Kawa, the java-based scheme system. `http://www.cygnus.com/~bothner/kawa.html`.

[2] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[3] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[4] R. M. Stallman. *Using and Porting GNU CC*. Free Software Foundation, Boston, MA, 1996.

[5] T. Wilkinson. Kaffe - a free virtual machine to run java code. `http://www.tjwassoc.demon.co.uk/kaffe/kaffe.htm`.