Compiling Java for Embedded Systems

By Per Bothner (*`bothner@cygnus.com`*)

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 allows much better optimization along with much faster application start-up times than with JIT translators. Cygnus Solutions is writing a Java front-end for the GNU compiler, gcc, in order to translate Java bytecodes to machine code. This uses a widely used, proven technology. In this paper, we discuss issues in implementing Java using traditional compiler, linker, and debugging technology; particular emphasis is given to using Java in embedded and limited memory environments.

1 JAVA IMPLEMENTATION

Java (see Bibliography for [**JavaSpec**] on page 15) has become popular 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 (see Bibliography for [**JavaVMSpec**] on page 15). The traditional Java implementation model is to compile Java source code into `.class` files containing machine-independent byte-code instructions. These `.class` files are downloaded and interpreted by a browser or some other Java “Virtual Machine” (VM). See the diagram in Figure 1 (below) for a model of the traditional Java implementation.

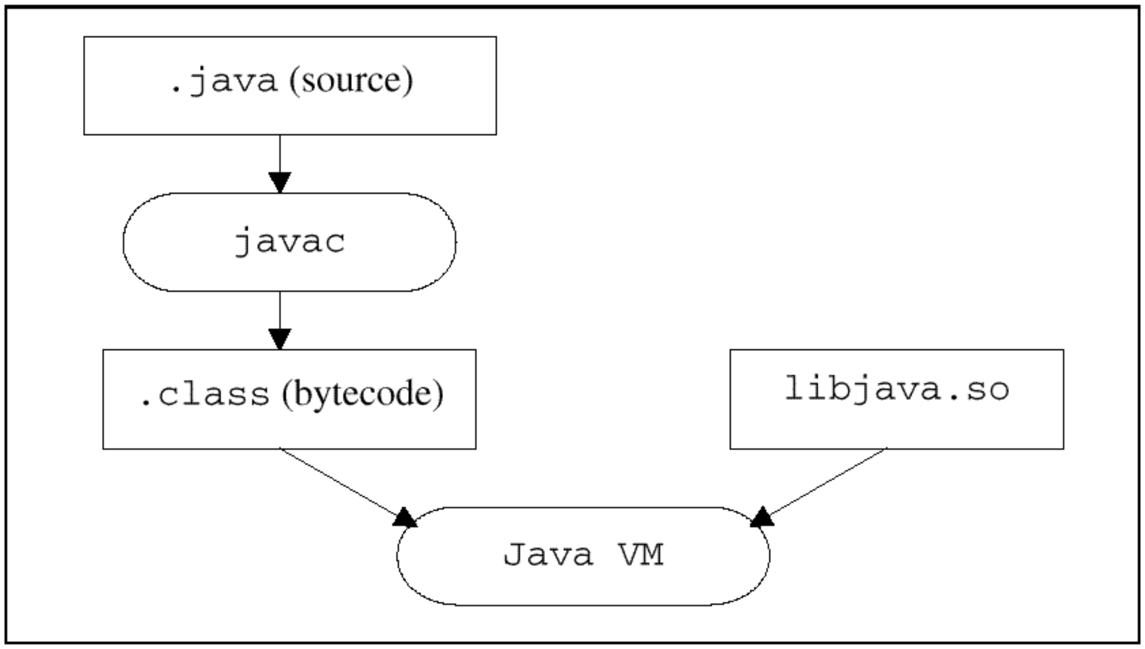

Figure 1: Typical Java implementation

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 three 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 than a stand-alone compiler, 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.)
- The JIT compiler must remain in (virtual) memory while the application is executing. This memory may be quite costly in an embedded application.

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 many think of Java as primarily an internet/web language, others are interested in using Java as an alternative to traditional languages such as C++, provided performance is acceptable. For embedded applications, it makes much more sense to pre-compile the Java program, especially if the program is to be in ROM.

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. These have been ported to just about every chip in use (including many chips only used in embedded systems), and so we will have a uniquely portable Java system. See Figure 2 on page 3 for a model of the compilation process.

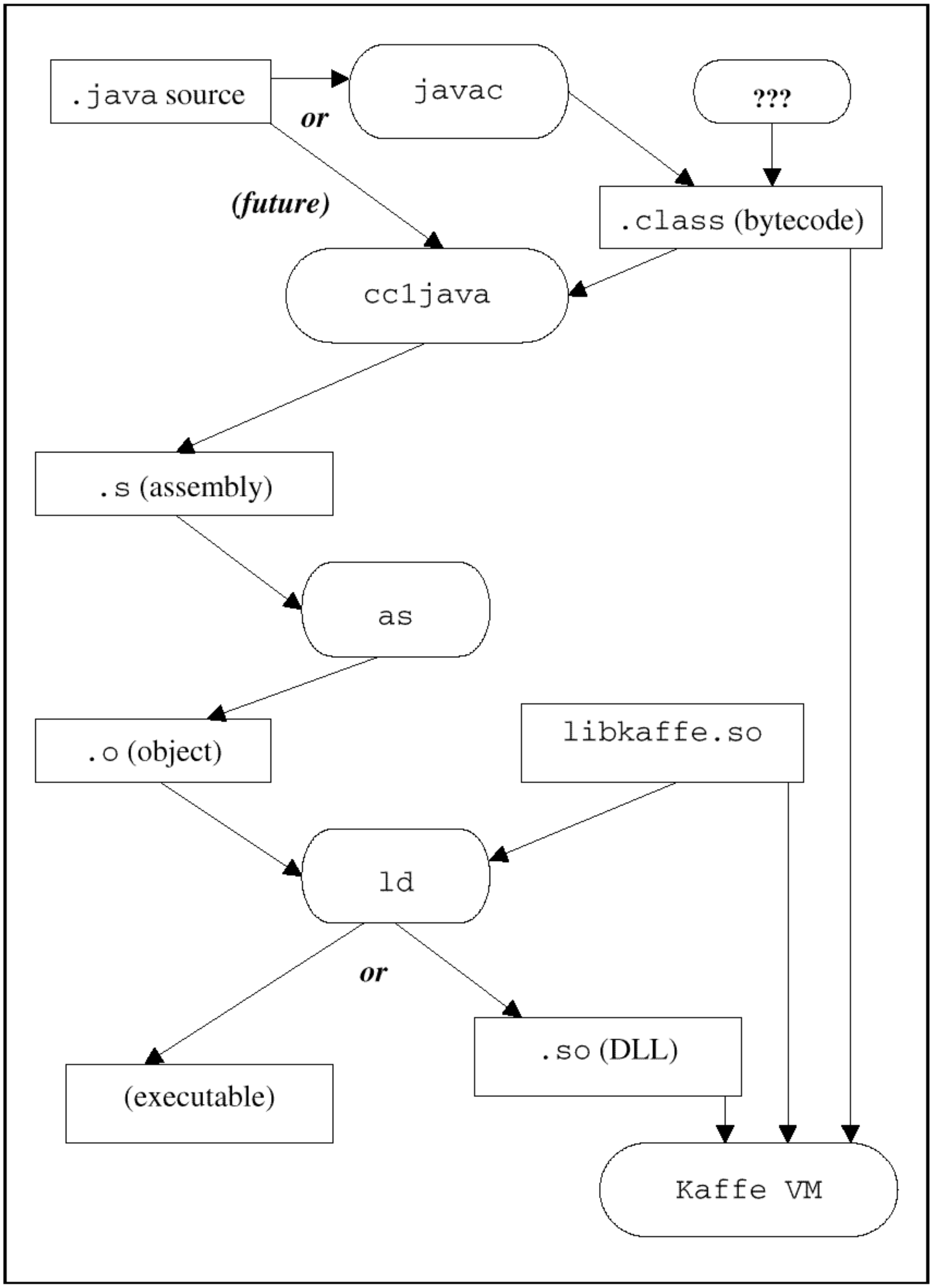

Figure 2: Compiled Java

2 ISSUES WITH EMBEDDED JAVA

Sun’s motto for Java is “Write once, run anywhere”. As part of that, Sun has been pushing Java as also suitable for embedded systems, announcing specifications for “Embedded Java” and “Personal Java” specifications. The latter (just published at the time of writing) is primarily a restricted library that a Java application can expect.

“Embedded systems” covers a large spectrum from 4-bit chips with tens of bytes of RAM, to 32- and 64-bit systems with megabytes of RAM. I believe it will be very difficult to squeeze a reasonably complete implementation of Java into less than one MB. (However, people have managed to squeeze a much reduced Java into credit-card-sized “smart cards” with about 100kB.) In general, there is less memory available than in the typical desktop environment where Java usually runs. Those designers that have been leery of C++ because of performance concerns (perceived or real) will not embrace Java. On the other hand, those that have been leery of C++ because of its complexity will find Java easier to master.

2.1 Advantages of Java

Java has a number of advantages for embedded systems. Using classes to organize the code enforces modularity, and encourages data abstraction. Java has many useful and standardized classes (for graphics, networking, simple math and containers, internationalization, files, and much more). This means a designer can count on having these libraries on any (full) implementation of Java.

Java programs are generally more portable than C or C++ programs:

- The size of integer, floats, and character is defined by the language.
- The order and precision of expression evaluation is defined.
- Initial values of fields are defined, and the languages require that local variables be set before use in a way that the compiler can check.

In fact, the only major non-determinacy in Java is due to time-dependencies between interacting threads. Safety-critical applications will find the following features very useful:

- More disciplined use of pointers, called “references” instead, provides “pointer safety” (no dangling references; all references are either null or point to an actual object; de-referencing a null pointer raises an exception).
- Array indexing is checked, and an index out of bounds raises an exception.
- Using exceptions makes it easier to separate out the normal case from error cases, and to handle the error in a disciplined manner.

The portability of Java (including a portable binary format) means that an application can run on many hardware platforms, with no porting effort (at least that’s the theory).

2.2 Code compactness

It has been argued that even for ROM-based embedded systems (where portability is not an issue), it still makes sense to use a bytecode-based implementation of Java, since the bytecodes are supposedly more compact than native code. However, it is not at all clear if that is really the case. The actual bytecode instructions of the Fib class (a program to calculate Fibonacci numbers, which is discussed in the section, **5 Status**, on page 14) only take 134 bytes, while the compiled instructions for i86 and Sparc take 373 and 484 bytes respectively. (This is if we assume external classes are also pre-compiled; otherwise, 417 and 540 bytes are needed, respectively.) However, there is quite a bit of symbol table information necessary, bringing the actual size of the `Fib.class` file up to 1227 bytes. How much space will actually be used at run-time depends on how the symbolic (reflective) information is represented - but it does take a fair bit of space. (Pre-compiled code also needs space for the reflective data structure –about 520 bytes for this example.) Out tentative conclusion is that the space advantage of bytecodes is minor at best, whereas the speed penalty is major.

2.3 Space for standard run-time

In addition to the space needed for the user code, there is also a large chunk of fixed code for the run-time environment. This includes code for the standard libraries (such as `java.lang`), code for loading new classes, the garbage collector, and an interpreter or just-in-time compiler.

In a memory-tight environment, it is desirable to be able to leave out some of this support code. For example, if there is no need for loading new classes at run-time, we can leave out the code for reading class files, and interpreting (or JIT-compiling) bytecodes. (If you have a dynamic loader, you could still down-load new classes, if you compile them off-line.) Similarly, some embedded applications might not need support for Java's Abstract Windowing Toolkit, or networking, or decryption, while another might need some or all of these.

Depending on a conventional (static) linker to select only the code that is actually needed does not work, since a Java class can be referenced using a run-time String expression passed to the `Class.forName` method. If that feature is not used, then a static linker can be used.

The Java run-time needs to have access to the name and type of every field and method of every class that is loaded. This is not needed for normal operation (especially not when using precompiled methods); however, a program can examine this information using the `java.lang.reflect` package. Furthermore, the Java Native Interface (JNI, a standard ABI for interfacing between C/C++ and Java) works by looking up fields and methods by name at run-time. Using the JNI thus requires extra run-time memory for field and method information. Since the JNI is also quite slow, an embedded application may prefer to use a lower-level (but less portable) Native Interface.

Because applications and resources vary so widely, it is important to have a Java VM/run-time than can be easily configured. Some applications may need access to dynamically loaded classes, the Java Native Interface, reflection, and a large class library. Other applications may need none of these, and cannot afford the space requirements of those features. Different clients may also want different algorithms for garbage collection or different thread implementations. This implies the need for a configuration utility, so one can select the features one wants in a VM, much like one might need to configure kernel options before building an operating system.

2.4 Garbage collection

Programmers used to traditional malloc/free-style heap management tend to be skeptical about the efficiency of garbage collection. It is true that garbage collection usually takes a significant toll on execution time, and can lead to large unpredictable pauses. However, it is important to remember that is also an issue for manual heap allocations using malloc and free. There are many very poorly written `malloc/free` implementations in common use, just as there are inefficient implementations of garbage collection.

There are a number of incremental, parallel, or generational garbage collection algorithms that provide performance as good or better than `malloc/free`. What is difficult, however, is ensuring that pause times are bounded—*i.e.*, a small limit on the amount of time a new can take, even if garbage collection is triggered. The solution is to make sure to do a little piece of garbage collection on each allocation. Unfortunately, the only known algorithms that can handle hard-real time either require hardware assistance or are very inefficient. However, "soft" real-time can be implemented at reasonable cost.

3 COMPILING JAVA

The core tool of Cygnus's Java implementation is the compiler. This is `jc1`, a new `gcc` front-end (see Bibliography for [**GCC**] on page 15). 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 `jc1` 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 `jc1` 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 Scheme-to-Java-bytecode compiler; see Bibliography for [**Kawa**] on page 15).

- 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 compile from source, we need a parser, semantic analysis, and error-checking. On the other hand, when parsing from bytecodes, we need to extract higher-level information from the lower-level bytecodes, which we will discuss, below, in **3.1 Transforming bytecodes**.

3.1 Transforming bytecodes

This section describes how `jc1` works.

The executable content of a bytecode file contains a vector of bytecode instructions for each (non-native) method. The bytecodes are instructions for an abstract machine with some local variable registers 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 not typed; *i.e.*, 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 `jc1`, 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 `trees`, 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 ints 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 code generated by `jc1` is the following:

```
reg51 := vreg51 + vreg52
```

Note 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, compared to conventional compilers:

- 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) fixes the generated code, but you still get a lot of useless stack slots. It would be nice not to *have to* run the optimizer, and if you do, not to make unnecessary work for it.
- The `rtl` representation is semi-typed, since it distinguishes the various machine "modes" such as pointer, and different-sized integers and floats. This causes problems because a given Java stack slot may have different types at different points in the code.

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 jc1actually does 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)));
```

The build function is the standard `gcc` function for creating `tree`-nodes, while fold is the standard function for doing constant folding. The functions `pop_value` and `push_value` are specific to `jc1`, and keep 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 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 code quality.

3.2 Class meta-data

Compiling the executable content is only part of the problem. The Java run-time also needs an extensive data structure that describes 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. (If the run-time supports dynamically loading new classes, it already knows 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 run-time 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 may cause consistency problems, 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 is 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 Ð most of it is just entering the meta-data into a global symbol table.

Consider the following example class:

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

That class compiles into something like the following:

```
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.

3.3 Static references

A class may make references to `static` fields and methods of another class. If we can assume that the other class will also be `jc1`-compiled, then `jc1` can emit a direct reference to the external `static` field or method (just like a C++ compiler would). That is, a call to a `static` method can be compiled as a direct function call. If you want to make a static call from a pre-compiled class to a known class that you do *not* know is pre-compiled, you can try using extra indirection and a "trampoline" stub that jumps to the correct method. (Not that this feature is very useful: it makes little sense to pre-compile a class without also pre-compiling the other classes that it statically depends on.)

A related problem has to do with string constants. The language specification requires that string literals that are equal will compile into the same String object at run-time. This complicates separate compilation, and it makes it difficult to statically allocate the strings (as in done for C), unless you have a truly unusual linker. So, we compile references to string literals as indirect references to a pointer that gets initialized at class initialization time.

3.4 Linking

The `jc1` 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.

Java needs a lot of names at run-time –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 could place these names in a special section of the object file, and the linker could then combine identical names.

The run-time 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.

4 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 needed in some embedded environments. Basically, the appropriate Java run-time environment is a Java Virtual Machine.

It is possible to have `jc1` produce code compatible with the Java Native Interface ABI. Such code could run under any Java VM that implements the JNI. However, the JNI has relatively high overhead, so if you are not concerned about binary portability it is better to use a more low-level ABI, similar to the VM's internal calling convention. (If you are concerned about portability, use `.class` files.) While we plan to support the portable JNI, we will also support such a 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—a free Java VM***, written by Tim Wilkinson (see Bibliography for [**Kaffe**] on page 15), with help from volunteers around the "Net." Kaffe uses either a JIT compiler on many common platforms, 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.

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). However, 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. Therefore, we have put some effort into streamlining the Kaffe data structures, such as replacing linked lists by arrays.

4.1 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 such as `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. (For embedded systems one would probably leave out line numbers in the run-time, and only keep it in the debug section of the executable file.)

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`.

4.2 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`, and the `gcov` coverage analyzer.

5 STATUS

As of early July 1997, `jc1` was able to compile a simple test program, which calculates the Fibonacci numbers (up to 36), both iteratively and recursively (which is slow!), and prints the results. No manual tinkering was needed with the assembly code generated, which was assembled and linked in with `kaffe` (a modified pre-0.9.1 snapshot) as the run-time engine. On a SparcStation 5 running Solaris2, it took 16 seconds to execute. In comparison, the same program dynamically compiled by Kaffe’s JIT takes 26 seconds, and Sun’s JDK 1.1 takes 88 seconds to run the same program.

These numbers are encouraging, but they need some context. Start-up times (for class loading and just-in-time compilation) should in all cases be fairly minor, since the execution time is dominated by recursive calls. In the `jc1`-compiled case, only the actually test class (calculating Fibonacci plus the main methods that prints the result) is compiled by jc1; all the other classes loaded (including the classes for I/O) are compiled by `kaffe`’s JIT-compiler. This means there would be some slight extra speed up if all the classes were `jc1`-compiled. Do note that the test-program uses simple C-like features that are easy to compile: integer arithmetic, simple control structures, and direct recursion. The results cannot be directly generalized to larger programs that use object-oriented features more heavily.

The basic structure of the compiler works, but there is still quite a bit of work to do. Many of the byte-codes are not supported yet, and neither is exception handling. Only some very simple Java-specific optimizations are implemented. (Of course, `jc1` benefits from the existing language-independent optimizations in `gcc`.)

The Kaffe VM works on most 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.

The Java support in the `gdb` debugger is partially written, but there is still quite a bit to go before it is user-friendly. We have not started work on our own library implementation or source-code compiler.

ACKNOWLEDGEMENTS

This paper is partly based on a document called “A GCC-based Java Implementation” (© 1997, IEEE), which was presented at IEEE Compcon, Spring 1997.

BIBLIOGRAPHY

[GCC] ***Using GNU CC***
Richard Stallman
Free Software Foundation, 1996.

[JavaSpec] ***The Java Language Specification***
James Gosling, Bill Joy, Guy Steele
Addison-Wesley, 1996.

[JavaVMSpec] ***The Java Virtual Machine Specification***
Tim Lindholm, Frank Yellin
Addison-Wesley, 1996.

[Kaffe] ***Kaffe—a virtual machine to run Java code***
Tim Wilkinson
`http://www.kaffe.org/`

[Kawa] ***Kawa, the Java-based Scheme System***
Per Bothner
`http://www.cygnus.com/~bothner/kawa.html`