# Compiling Java for Embedded Systems

(was: *Developing Java Run-time Applications*)

Per Bothner `<bothner@cygnus.com>`

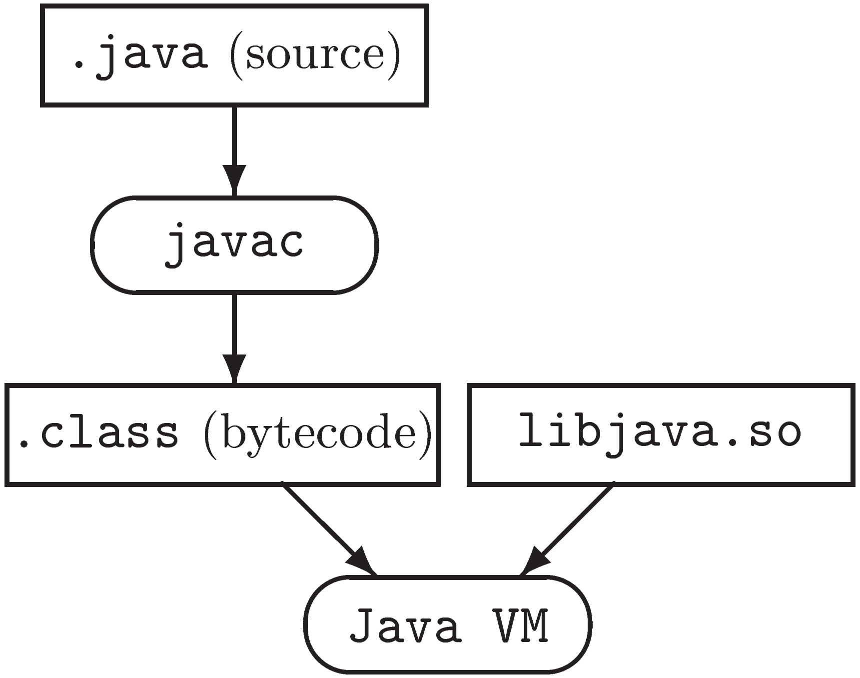
Cygnus Solutions

- Strengths of Java for embedded systems
- Java implementation using interpreters and compilers
- How to translate bytecodes to machine code
- Linking, run-time environment, debugging
- Space usage and other issues wrt embedded systems

# Advantages of Java for embedded systems

- Classes encourage modularity and data abstraction.
- Many useful and portable packages (graphics, simple math and containers, networking, internationaliztion, files, ...).
- Java programs are generally more portable than C or C++, because less is "implementation defined". (Example: `int` size.)
- Safety features (such as array index checking) improve robustness.
- Exceptions make it easier to handle error conditions.
- Powerful, but much easier to master than C++.
- Garbage collection simplifies program design, language definition, and can be faster than manual allocation.

# Usual Java implementation model




This is simple and portable, but is slow, and needs a file system.

# Traditional “ahead-of-time” Compilation

- Many people are more interested in Java as a better C or simpler C++. They are less interested in dynamic loading or the use of bytecodes for portable distribution.
- For them, traditional compilation techniques might provide faster execution.

Even when dynamic loading is needed (*e.g.* a browser), it makes sense to pre-compile the standard libraries (such as `java.lang` and the AWT).

- A number of projects are compiling Java to C or native code.

# Just-In-Time Compilation

A "Just-in-Time" (JIT) compiler translates bytecodes to native code "on-the-fly" *inside* the VM.

This maintains the simple user model (it uses portable bytecodes externally), yet speeds up Java execution substantially.

While bytecodes are great for interchange,
pre-compilation is better for heavily used libraries and applications:

- JIT imposes a start-up penalty each time the application is run.
- JIT cannot do optimizations that take much time or memory.
- The JIT machinery uses memory that may be costly in some environments.
- Porting a good JIT is a substantial effort. Gcc has already been widely ported.

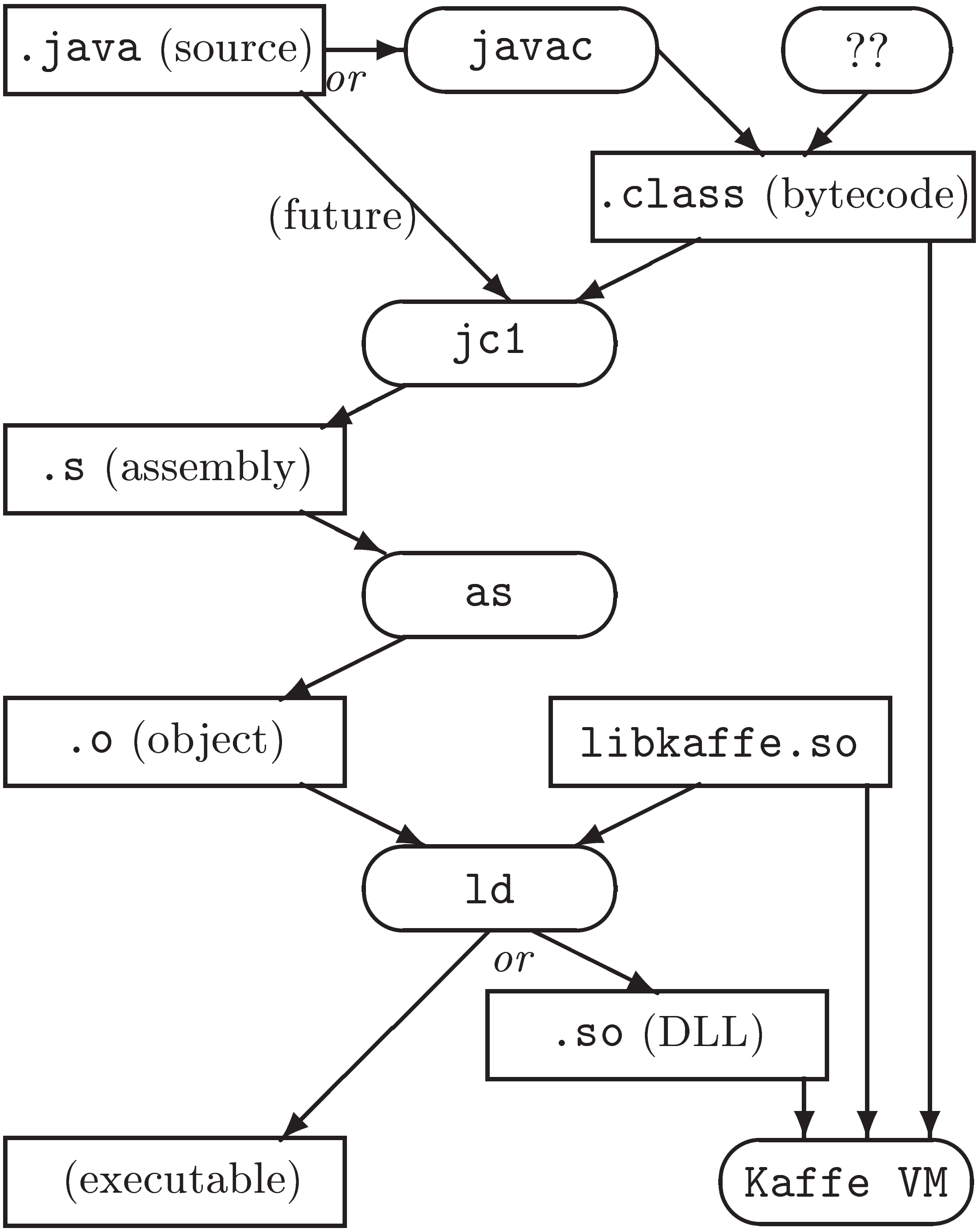


CYGNUS

# The Cygnus/GNU ahead-of-time Compiler

- Cygnus is writing a new Java ahead-of-time compiler.
- This is a new front-end (`jc1`) for `gcc`, the GNU compiler, which already supports C, C++, Objective-C, Fortran77, Ada, Pascal, Chill, on dozens of architectures.
- It uses the existing `gcc` framework, and shares most of its code, including the standard `gcc` optimizer and code generator.
- `jc1` reads bytecodes (`.class` files) and produces assembly code. A future version of `jc1` will be able to read either Java source or bytecodes.
- The run-time environment is based on Kaffe, a free Java VM.

# The Big Picture



# Example

Java Source:

```
class Fib {
  static int fib (int i)
  {
    int x0 = 1, x1 = 1;
    while (--i >= 0) {
      int x2 = x0 + x1;
      x0 = x1;
      x1 = x2;
    }
    return x0;
  }
}
```

Gcc -O2 code (Sparc):

```
Fib_fib_0:
        mov 1,%o1
        mov 1,%g3
.LL3:   addcc %o0,-1,%o0
        bneg .LL4
        add %o1,%g3,%g2
        mov %g3,%o1
        b .LL3
        mov %g2,%g3
.LL4:   retl
        mov %o1,%o0
```

(10 instructions)

Kaffe JIT code:

```
    save  %sp,-136,%sp
    mov   1,%l0
    mov   %l0,%l1
    mov   1,%l0
    mov   %l0,%l2
    st  %i1,[%fp+0x48]
    st  %l1,[%fp-40]
    st  %l2,[%fp-36]
    b  L1
    nop
L2: ld   [%fp-40],%l3
    mov   %l3,%l4
    ld   [%fp-36],%l5
    mov   %l5,%l6
    addcc  %l4,%l6,%l4
    mov   %l4,%l7
    mov   %l5,%l4
    mov   %l4,%l3
    mov   %l7,%l4
    mov   %l4,%l5
    st  %l3,[%fp-40]
    st  %l5,[%fp-36]
    st  %l7,[%fp-32]
L1: ld   [%fp+0x48],%i0
    addcc  %i0,-1,%i0
    mov   %i0,%i2
    st  %i0,[%fp + 0x48]
    cmp   %i2,0
    bge   L2
    nop
    ld   [%fp-40],%i3
    mov   %i3,%i4
    mov   %i4,%i0
    ret
    restore
```

(35 instructions)

# Bytecodes

- A Java `.class` file contains information about a class, its fields, and its methods. The body of each method is represented using reverse-polish bytecode instructions for a stack-based virtual Java machine.
- Example: "`iadd`" adds two `int`s on top of the stack.
- Each method also has some "registers," used for parameters and local variables.
- The value of a stack slot or local variable slot is an untyped "word."
- However, at any given PC, each slot has a fixed type. The type can be recovered using simple data-flow analysis. This is done by the Java byte-code verifier as part of the Java security model.

# Translating bytecodes to machine code

- Most real machines use registers to evaluate expressions, not a stack.

A straight-forward translation of stack operations yields bad code.

- What saves us is that we can determine at each point the value of the stack pointer, and the types of each slot.

So we map each slot into a `gcc` virtual register, except we use different virtual registers for different machine types.

- Example: To compile "`iadd`", assume we determine that SP is 2; (stack-slot 0, `int`) is in virtual register 50; and (stack-slot 1, `int`) is in virtual register 51; then the generated code is:

```
vreg50 := vreg50 + vreg51
```

Register allocation assigns virtual registers to hardware registers or stack frame slots. The Java stack operations are compiled away.

# Bytecodes to abstract tree nodes

A refinement is to only allocate virtual registers *between* basic blocks. Within a basic block, we maintain a "local" (compile-time) stack of `tree` (abstract syntax) nodes.

Thus the `iadd` becomes:

```
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 local stack is maintained by `push_value` and `pop_value`. We construct `tree` nodes using `build`, and do constant folding using `fold`; both are standard `gcc` functions.

At the end of a basic block, the local stack (usually empty) is copied to the home virtual registers.

This optimization allows constant folding, and generates fewer unneeded temporaries.

# Code compactness

■ Byte code instructions are more compact than machine code.

Example: `Fib.java` takes 134 bytes (bytecode) vs 373 (i86 machine code) or 484 (Sparc).

■ However, Java needs a "constant pool" containing symbols and literals.

All calls, and all field and variable accesses are indirect using names in the constant pool.

■ Thus the size of the `Fib.class` file is 1227 bytes. This largely negates the compactness advantage of bytecode, and makes it difficult to compare.

Loaded into memory it might use more space (unpacking a compact encoding), or less (common strings can be shared).

# Class Reflective Data

- A `.class` contains descriptions of the superclass, fields, and methods of a class.
- Such data is represented in the Java run-time using various run-time data-structures.
- This "meta-" or reflective data includes name and position of each field and method.
- Useful for dynamic loading, serialization, RMI, debugging, ... .
- However, there is a substantial space cost. In an embedded environment one might want to leave out some or all of this symbol information.

# Compiling reflective data

The most efficient way for the compiler to make such data available is to generate initialized static data that follow the VM internal representation. That makes startup fast, without requiring dynamic memory allocation.

For example:

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

compiles into assembly code equivalent to:

```
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");
}
```

# Total code size

- Sun's Java Runtime Environment is 5MB compressed.
- Includes many libraries (native code and Java bytecodes); run-time (interpreter; memory, object, and thread management; etc).
- Classes are down-loaded "when needed" – but that may be impractical in embedded environment.
- Different embedded systems have different resources and needs. Hence need to be able to configure the Java run-time for different environments. Best if user can re-compile Java implementation to suit.

# Linking

- A traditional linker can be used to link pre-compiled Java code.

No Java-specific support is needed – though some may be desirable.

- Conventional static linking can leave out classes and methods that are never referenced. Assumes application does not load classes by name – *i.e.* no use of `Class.forName("myClass")`.

Virtual (non-final non-static) methods are more difficult for the linker to elide. It is more difficult to tell if they are never used, since they can be called indirectly through a dispatch table.

- Various space-saving optimization can be done with a smart linker, such as combining string literals.

Linker could generate stub procedures, *e.g.* between pre-compiled code and uncompiled bytecode methods.

# Run-time

- Compiled Java code depends on the specific layouts and data structures of a Java runtime environment.

(Could compile to standard Native Method Interface, but that has major overheads.)

- Because Java code can dynamically download `.class` files, we essentially need a complete VM.

If the VM has a JIT, we can use the same calling convention for statically compiled and dynamically loaded code.

- We are using Kaffe, a free VM developed by Tim Wilkinson. This has a JIT and has been ported to many architectures.

We are making various changes to improve Kaffe, including adding support for pre-compiled statically allocated classes.

# Garbage collection

- Manual memory management in the presence of threads or complex data structures is difficult, error-prone, and leads to excess copying.
- Automatic techniques ("garbage collection") have been studied for decades.
- Which method to use depands on many factors: Compiler support; multi-thread requirements; real-time requirements; availability and speed of manipulating page protection; application usage patterns; tightness of memory; etc.
- No one method is best in general. Should be able to plug in different algorithms, especially in embedded systems, which vary widely in needs and resources. Because GC ties in with object management, it may be desirable to tune and re-compile the Java system for best performance.

# Mixing languages

- Sometimes Java methods need to call lower-level C or C++ functions. Such code can be wrapped in a `native` method. There is a standard "Java Native Interface" (JNI) for communicating between Java and C/C++.

- JNI defines a *binary* interface (ABI) that should work on any Java VM. The cost is substantial overhead: Instead of C/C++ code being able to access a field directly, it has to call at least one function on each access.

- JNI depends on the class reflective data, and thus removes the option of leaving that data out to save space.

- Cygnus will offer two native interfaces: The standard JNI, and a lower-level faster interface that allows direct access. We plan an interface that will provide very natural access to Java objects and methods from C++. (We are making the Java and C++ ABIs the same whenever possible.)

# Debugging

- Cygnus is enhancing `gdb`, the GNU debugger, to understand Java. This allows debugging mixed-language applications.

The `jc1` compiler knows how to emit debugging information (`stabs`, `coff`, `dwarf`) that `gdb` can read. Thus we get basic debugging "for free."

- Debugging Java raises similar issues as debugging C++: `Gdb` needs a Java expression parser; needs to know how object and classes are laid out; know how to call methods; etc.

- Debugging dynamically loaded code is a complication. In that case, `gdb` needs to be able to extract debugging information from the run-time data structures. Some changes in the VM will be needed to ease this.

# Status

- The compiler mostly works. Most classes compile and run correctly. Exception handling needs work; the compiler sometimes crashes; polishing and optimization are still needed.
- The Kaffe VM is available for many platforms. We have made numerous improvements to simplify the data structures and interfaces, and to make it more suitable for pre-compiled code.
- The `gdb` debugger has most of the support needed to debug pre-compiled Java code. Some features are missing, but it is already useful for debugging real code, and can give `jdb` a run for its money.