A Gcc-based Java Implementation

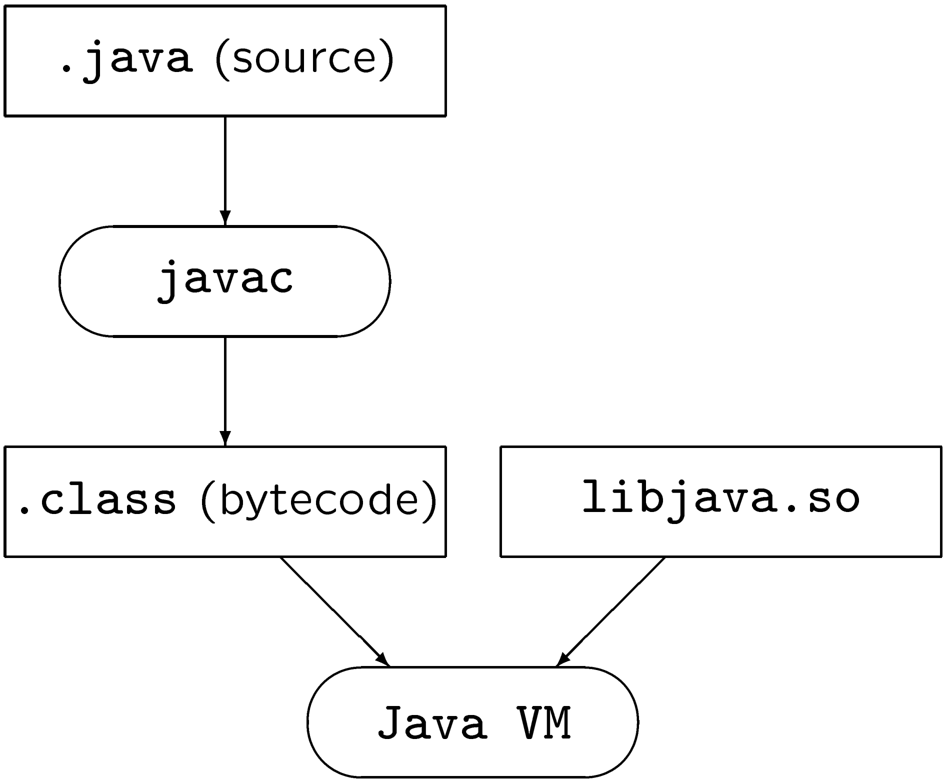
Implementing Java as just another language

Per Bothner

<bothner@cygnus.com>

Cygnus Solutions

Usual Java implementation model



This is simple, portable, slow.

Java using traditional techniques

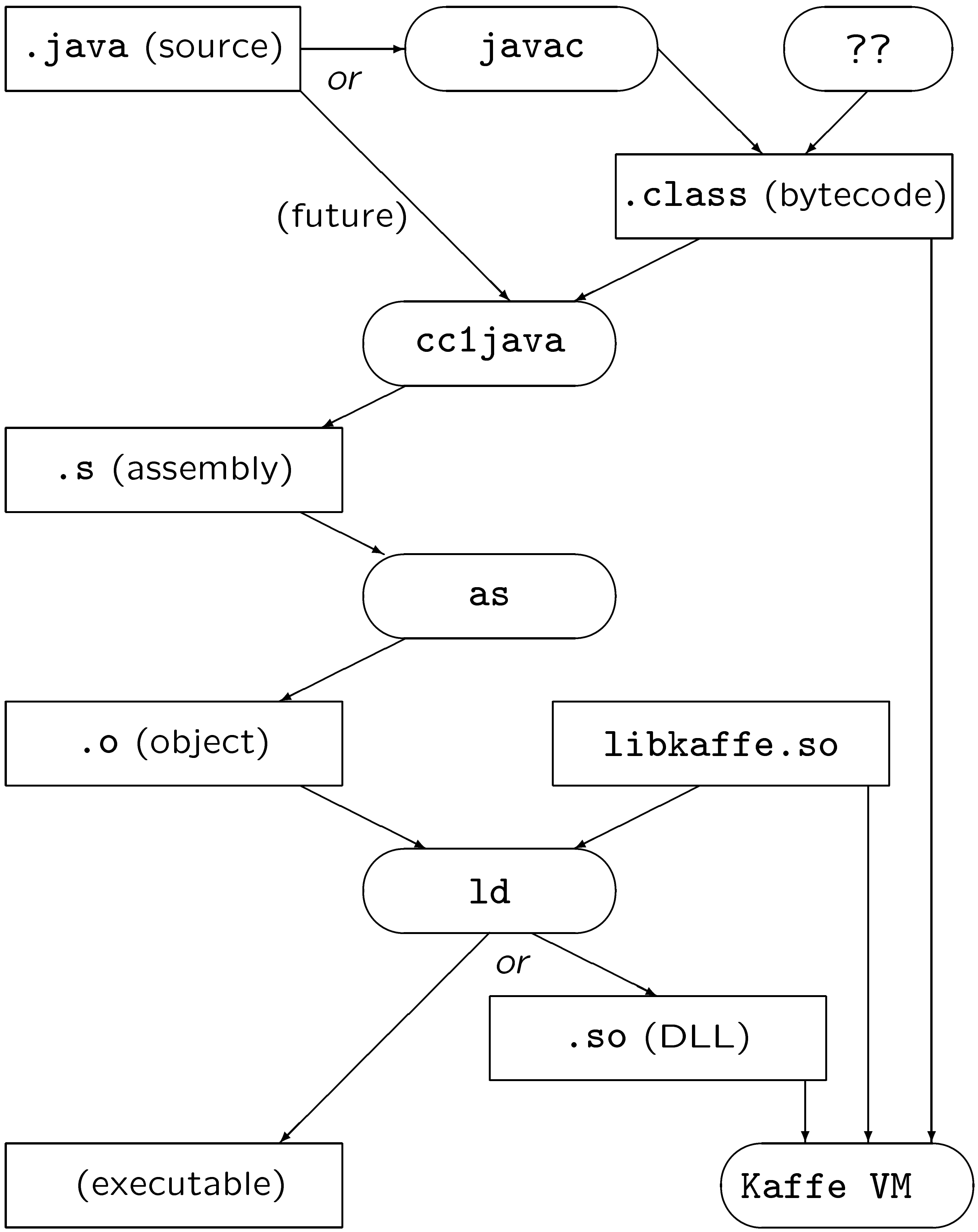
Many people are more interested in Java as a better C or simpler C++. They are less interested in dynamic loading or the use of bytecodes for portable distribution.

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).

Cygnus is writing a new Java front-end for `gcc`, the GNU compiler, which already supports C, C++, Fortran77, Ada, Pascal, Chill, on dozens of architectures.

The run-time environment is based on Kaffe, a free Java VM.

The Big Picture



What about Just-In-Time Compilers?

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 an good JIT is a substantial effort. Gcc has already been widely ported.

Example

Java Source:

```
class Fib {
  static int fib (int i)
  {
    int x0 = 1;
    int 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)

The Compiler

`cc1java` is a new language front-end for `gcc`. It uses the existing `gcc` framework, and shares most of its code, including the standard `gcc` optimizer and code generator.

`cc1java` reads bytecodes (`.class` files) and produces assembly code.

A future version of `cc1java` will be able to read either Java source or bytecodes. For now, we use Sun's `javac` to create bytecodes.

Transforming bytecodes

Bytecode abstract instructions manipulate untyped local variable slots and stack slots.

Data-flow analysis recovers for a given PC the SP and type in each slot.

For each slot and low-level type we associate a `gcc` virtual register.

Example: “`iadd`” adds two `int`s on top of the stack.

If (stack-slot 0, `int`) is in virtual register 50; (stack-slot 1, `int`) is in virtual register 51; and SP is 2; 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" 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.

Class reflective data

A `.class` contains descriptions of the superclass, fields, and methods of a class.

Such data is represented in the Java runtime using various run-time data-structures.

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");
}
```

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.

Debugging

We will enhance `gdb`, the GNU debugger, to understand Java. This allows debugging mixed-language applications.

The `cc1java` compiler already 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

Basic structure of compiler works, including parsing bytecodes, generating trees, emitting assembler and debugging information. Not all bytecodes are implemented yet, and only some of the reflective data is being emitted.

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.