Jaune - An ahead of time compiler for small systems

Michael Hope ¡michaelh@juju.net.nz¿

May 11, 2002

Abstract

Jaune is a compiler and set of libraries that can be used to write programs in the Java language for small devices. Included are a basic set of class libraries, a conservative garbage collector, and set of examples that can be run on the 8-bit Nintendo Gameboy and, with appropriate support, most Z80 based systems. This document covers the installation, usage, and basic architecture of the system.

Contents

1 Quickstart

Impatient, eh? The following instructions will allow you to compile for the GB on a Linux system. The system should work under Windows with cygwin, but this has not been tested.

- Download and install gbdk from http://gbdk.sourceforge.net. Version 2.97a or above is required. The most recent version may be available from the gbdk release staging area available at http://gbdk.sourceforge.net/staging/.
- Download the Jaune release from http://sourceforge.net/projects/jaune
- Extract it somewhere safe, such as `$HOME/src`
- Read `config.mk` and customise it for your system.
- Compile Jaune by changing to the top level directory and running 'make'.
- Compile an example by changing to one of the directories under examples and running 'make'.
- Run the example under an emulator such as gnuboy or on real hardware.

2 Introduction

Jaune is an ahead of time compiler and set of runtime libraries. The compiler translates directly from Java `.class` files into the assembly language of the target processor, sacrificing the portability of Java for faster run time and the ability to run on machines with very little memory that otherwise couldn't support runtime linking or compilation.

The features of Jaune include:

- All of the object-oriented features of the Java language.
- A conservative, compiler directed garbage collector.
- Sufficient base class support for implementing most of the niceties of the Java libraries.
- Native implementations of the Object, String and StringBuffer classes for speed.
- A very straight forward native code interface.

The main tradeoffs are:

- `ints` are sixteen bits instead of thirty-two. Java performs almost all operations using `ints`, which are far too expensive on an eight bit machine.
- No dynamic linking. All linking is done at compile time.
- No dynamic class loading. All classes must be known at compile time.

Some of the features of the Java runtime that are not currently supported are:

- No reflection.
- No security model.
- No exceptions. The finally keyword is supported.
- No longs, floats, or doubles. These could be added.

The licensing of Jaune is similar to that of the GNU C Compiler, gcc. The compiler itself is licensed under the GPL and the libraries are licensed under the LGPL. This allows non-GPL programs to be compiled with Jaune and linked against the runtime libraries without adding any further licensing restrictions.

3 Status and Road Map

Jaune is a research project, and I feel that I have learned all that I want to with the 1.0 release. There are many optimisations and improvements that could be made, and the beauty of open source is that people have that opportunity. I do not plan to provide user support or make enhancements past 1.0, but feel free to ask.

Version 1.1 contains more hooks from Jaune into some of the libraries of gbdk and a more rounded set of examples.

4 Documentation

- High level documentation, including the PDF form of this document are available at http://jaune.sourceforge.net/doc/
- API documentation for the Java-based library, jlib, and the compiler are available at http://jaune.sourceforge.net/doc/api/
- API documentation for the C runtime library, clib, is available at http://jaune.sourceforge.net/doc/clib/
- Screenshots of the examples are available at http://jaune.sourceforge.net/screenshots/

5 Usage

Compiling the source code into the final binary image is relatively complicated, but most cases can be handled through the supplied Makefiles. The steps involved are:

1. Compile each source code file into a class file using a normal Java compiler.
2. Compile all of the class files at once into a set of assembly files using the Jaune compiler.
3. Assemble each assembly file into an object file.
4. Link all of the object files together with the Jaune runtime library and the startup module into the final binary.

It is very important that all of the class files be compiled in the one command. The static linking is done through method numbers, and if the order or number of class files changes then the method numbers will change causing unpredictable behaviour.

See one of the example Makefiles for an example of how to use the provided makefiles to correctly compile. Set `JAVA_SRC` to the list of classes to be compiled, `MAIN` to the mangled form of the static main method, and `OUT` to the base name of the output file.

6 Writing against Jaune

This section contains notes on using some of the special features of Jaune, and how to write code that will run efficiently on the target.

6.1 Name Mangling

Mangling is the process of changing an identifier from one form to another in a predictable way to get around limitations of the target form. Jaune uses mangling to translate the potentially long, potentially overloaded Java class and method names into a more C like form. Specifically, Jaune uses mangling on class names and method names to compact the fully qualified name of a class and to give unique names to overloaded methods. For example, the class name `jaune.lang.JObject` is mangled to `JLJObject`, and the overloaded method `append(int i)` is mangled into `append1`.

The mangling rules are needed to determine the name of the main method and to write the function names for any native methods.

The mangling rules for class names are:

1. Start with the fully qualified class name eg `jaune.lang.JString`.
2. Split out the package name `jaune.lang` from the class name `JString`.
3. Take the first letter of each part of the package name, capitalise it, and join them together.
4. The mangled form is the mangled form of the package plus the class name. `J + L + JObject -> JLJObject`.

It is possible for the mangled names of two different classes to collide. The compiler contains a way of avoiding this but it has not been tested.

The mangling rules for method names are:

1. The base name is the method's name eg `append` from `append(int i)`.
2. If this name is already used, increase the count, append the count number to the end of the base name, and try again. eg `append` -¿ `append1` -¿ `append2`.

The method names are assigned in the order that the methods are defined in the source file. If you are implementing a native class that has overloaded methods make sure to add new methods after the current ones to preserve the ordering.

The method label is the mangled class name plus two underscores (__) plus the mangled method name. The method label for special methods such as the class initialiser or static initialiser is the mangled class name plus one underscore (_) plus the cleaned up form of the method name.

For example, in this code

```
package tests.graphics;

public class DrawCircles {
    ...
    public static void main(String[] args) {
        ...
    }
}
```

the mangled class name is `TGDrawCircles`, the mangled method name of main is `main`, and the method label of main is `TGDrawCircles__main`.

6.2 Native Methods

Jaune has been designed to interface easily with native methods. No special coding is required for native static method that takes primitive types, Strings, StringBuffers, or basic Objects as parameters and returns one of the same. The garbage collector correctly interfaces with native methods, so no special considerations need to be made when operating with or allocating objects.

To implemented a native method in C,

- Include `clib.h` from the device directory.
- Start a function whos name is the mangled method name.
- Mark the return type using one of the J (JBOOLEAN, JINT, etc) typedefs.
- Mark the parameters in reverse order using the J typedefs. Include the this pointer as the last argument if the method is not static.
- Implement the body of the function.

The String, StringBuffer, and Instance structure definitions may be used if one of the parameters is one of those types. The macros `TRYTHROW`, `ASSERT`, and `VALIDATE` may be useful. See the runtime library code in `device/clib` for more information.

6.3 Coding considerations

It is even more important than normal to write efficient code when running on a small system. The following hints should be used in moderation - it is more important to write maintainable, reusable code than to save a few percent in the execution time.

- Prefer final methods. If a method is marked as final then the compiler can make a direct reference to it instead of going through the virtual method table. Marking a class as final marks every method as final.

- Make helper methods static. A static method doesn't have the extra overhead of fetching and pushing the `this` pointer.
- Define constants as static final. Constants defined this way can be inlined instead of fetched from the field.
- Minimise the use of class fields. In the current implementation a field access is done through an expensive virtual table lookup. Local variables and method arguments are accessed through the faster stack pointer.
- Minimise heap usage. Heap allocation and garbage collection are expensive.
- Re-use constant instances. If a temporary instance of a class is often used in or returned from a method, consider turning it into a constant.
- Consider native implementations for time critical or expensive methods.

7 Implementation

This section contains notes about the implementation of Jaune.

7.1 Virtual Machine Translation

The Java Virtual Machine is a relatively compact stack based processor with many specialised instructions. Due to the limited number of registers the GB has and limited scope of this project, Jaune takes the easy way out and translates the stack based operations of the JVM into stack operations on the target processor. A peephole optimiser removes many of the redundant stack operations and provides limited register packing.

The registers and their uses are:

scratch Register used for any intermediate values. Most operations store one of their operands in scratch. DE is used on the GB and Z80.

lptr Local variables pointer. Points to the top of the local variables area on the stack. BC is used on the GB and Z80. The lptr must be preserved across native function calls.

The other registers used and their symbolic names are:

ret Alias used for the registers that contain the return value from a method. To integrate better with C and sdcc, this is defined as HL on the Z80 and DE on the GB.

param Alias used for the register used to push parameters to a method call. HL is used on the GB and Z80.

7.2 Garbage Collector

This section is copied from the garbage collector source in `device/clib/gc.c`.

Heap implementation and associated garbage collector. Both are very simple and are tied closely together and moderately to the vm they run with. Waba gets rid of the vm dependency by providing a per object scan callback, but this design uses hard coded links to avoid the overhead.

The Heap is a doubly linked non-circular list of free and allocated blocks. Each block consists of a header which references the next and previous block, a block size, and a magic number. Block sizes are always a multiple of two. The LSB of the size is used to mark if this block is free or not.

The garbage collector is a simple conservitave mark-sweep vm guided collector. Conservitive means that it can't always distinguish between references and numbers, meaning that it has to everything as a references. VM guided means that the VM can give the GC hints as to what are references and what aren't, such as marking reference arrays and marking references in instances. This is primarily done to cut the GC time instead of to increase the amount of reclaimed memory. The marking is done at compile time, and the check operation is much cheaper than checking if a value is a reference or not.

Some future ideas:

- Perhaps a circular list is better. It would remove the start/end of list checks but could need a one element in list check.
- The next pointer and size can be derrived from each other, so one is redundant. Computation is easier this way though.
- The magic number could be removed.
- Separate the list into a free and allocated lists. Makes the gc search and alloc search faster.

7.3 Peephole Optimiser

The peephole optimiser scans the generated assembly code and changes small sections into more efficient forms based on an input rule set. The word peephole comes from how the optimiser only looks ('peeps') at a small section of the code at once. This means that there is not enough context to perform global optimisations, but it does allow simple optimisations across instruction boundaries and allows the author to write a simpler, more maintainable code generator.

The syntax is based on `copt`. A rule consists of a set of lines to match, and a set of lines to replace them with if they match. The matching lines may include the wild cards tokens `%1, %2, %3` up to `%9`. A wild card token will match any single token. If a wild card is used more than once in a match set then each match must match the same value. The replacement lines may use wild cards to pull values in from the match.

The rules must be conservative. As the optimiser cannot tell if a register is later in the code, certain assignments must not be optimised out.

Some example rules are:
Change a push/pop into a more efficient form

```
replace {
push hl
pop bc
} by {
ld b,h
ld c,l
}
```

Remove redundant pushes

```
replace {
push %1
pop %1
} by {
}
```

Change a generic form of a shift into a more specialised, efficient form

```
replace {
ld h,b
ld l,c
add hl,hl
ld b,h
ld c,l
} by {
sla c
rl b
}
```

The optimiser puts many restrictions on the input rule set for speeds sake. See the source code and javadoc for more information.

7.4 Potential Optimisations

As mentioned earlier, Jaune takes a very direct approach to code generation. Some optimisations that could be performed are:

Dead code elimination Dead code occurs when a computation is performed but the result is never used. It is very easy to generate these in Java as the current compilers don't warn or perform their own elimination.

Copy propagation Copy propagation changes more expensive variable to variable assignments when the value of the incoming variable is known.

Constant operand detection This is a specialised form of Copy propagation especially for arithmetic operations such as add, subtract, multiply, divide, and shift. If the value of one of the operands is known, such as a multiplier or the shift count, then specialised code can be generated instead of calling the generic methods.

Method prologue removal Each method contains a prologue and epilogue that save and set up the lptr. In certain small methods this could be removed.

Optimised virtual tables The current method and field virtual tables are made up of global method ID and method address pairs. This is straight forward to generate, but requires a relatively expensive binary search at runtime. It should be possible due to Java's single inheritance model to collapse the entries down into linear table where the method ID is the index in that table. Interfaces will require special consideration.

8 Re-targeting

Jaune was not designed to be easily re-targeted to other assemblers or other processors, but the required parts could abstracted out or much of the framework re-used. To find what must change, search for `RETARGET` across the compiler source code. Little support is required from the target C compiler or startup libraries.