Using Coffi

Clark Verbrugge*

October 21, 1996

Abstract

This document gives technical information on the structure, installation, capabilities and use of the Java class file parser, Coffi.

1 Introduction

Java [GJS96] is an interpreted language; by design the language is only compiled down to an intermediate form called *bytecode*, and not down to hardware instructions. In the case of standalone Java applications the program is then run by passing the bytecode to a program implementing the Java Virtual Machine (JVM) description; in the case of Applets, the bytecode is transmitted along with a Web page for intepretation by an implementation of the JVM in a Web Browser.

Optimizations in this sort of organization can thus occur in at least two places. Either we optimize the bytecode itself, or we optimize the interpretation of it. The former approach has the distinct advantage of being architecture neutral—the bytecode is intended to be highly portable, and so optimizations we perform on it should be applicable to almost every implementation of the JVM. Optimizations on the interpreter, on the other hand, are necessarily specific to the machine on which the interpreter runs. For portability and generality of impact we obviously prefer the former, though significant gains with the latter strategy are certainly possible.

The Coffi class file parser is meant to form the basis of a bytecode optimizer. It is a program which reads in a class file, parses the associated bytecode, builds the control flow graph, and compiles parsed instructions back into bytecode. This program is intended to serve as a starting point for Java bytecode optimization system; by augmenting Coffi with various optimizations and analyses it is hoped that bytecode can be made considerably more efficient, both in space and in time. It should also be noted that by concentrating on bytecode rather than Java source code we will be able to optimize code for which the source is unavailable, including code transmitted as Applets.

This document does not assume a detailed knowledge of Java bytecode or the Java class file format; it is presumed, however, that the reader has some familiarity with these topics, and has at least skimmed the documents describing JVM bytecode [Cor95, LY96].

*Contact: `clump@cs.mcgill.ca`

2 Overview

The first step to optimizing bytecode is to read the bytecode in a Java class file. Coffi parses the class file structure on-line, reading in fields and constructing internal representations as soon as they are identified. This method follows quite precisely the specification given in the official Java bytecode description. This step is accomplished primarily by the methods in the `ClassFile` class; a single `ClassFile` object is constructed, and the contents of a `DataInputStream` (class file) are read in and stored within local variables of the `ClassFile` object. This process requires some parsing of the input to form the correct objects,[1] but up to this point no interpretation of the bytecode has occurred.

Once the entire class file has been read in, the bytecode for each method is parsed. This process is done through `ByteCode`[2] class methods.

2.1 Instructions

The bytecode for a method in a class file is contained as an attribute of that method. Each method is supposed to have exactly one `Code_attribute`, which is maintained as a simple array of bytes. Parsing these bytes consists of a loop, switching on the actual bytecode and parsing the instruction and its arguments as appropriate. The `Instruction` objects used to represent the parsed bytecode are gathered into a singly-linked list, and attached to the method as the `instructions` field of the `method_info` structure.

Instructions themselves are all subclasses of the generic `Instruction` class. In fact, to keep the parsing code small, most instructions are divided into categories depending on the number and type of arguments they expect—thus the following subclasses of `Instruction` form the superclasses of most instructions:

`Instruction_noargs` for instructions that have no arguments.

`Instruction_byte` if a 1-byte constant argument is required.

`Instruction_byteindex` if a 1-byte index into the constant pool is required.

`Instruction_bytevar` if a 1-byte reference to a local variable is required.

`Instruction_int` if a 2-byte[3] constant argument is required.

`Instruction_intindex` if a 2-byte index into the constant pool is required.

`Instruction_intvar` if a 2-byte reference to a local variable is required.

`Instruction_intbranch` if a 2-byte relative offset is required.

`Instruction_longbranch` if a 4-byte[4] relative offset is required.

Every instruction is a subclass of one of the above subclasses of `Instruction`, with the exception of `Instruction_Newarray`, `Instruction_Tableswitch`, and `Instruction_Lookupswitch`. These

[1] Most of the data stored in class files has variable length, and so needs to be parsed to be read in in its entirety

[2] In fact only static methods are used, so no actual `ByteCode` object need be created. However, due to perceived bugs in the initialization of static references, a `ByteCode` object is created to make sure the class is properly loaded.

[3] Yes, "int" is a misnomer, since int's are really 4-bytes in Java. This should really be `Instruction_short`.

[4] Yes, "long" is also a misnomer, since long's are 8-bytes in Java.

three instructions have non-trivial data which cannot be parsed as a simple application or extension of one of the base argument types, and so are distinct subclass of `Instruction` itself.

Each `Instruction` maintains information about its self. As well as the actual bytecode and arguments, there is also a field for the index from the start of the bytecode (`label`) and a `String` representing the name of the instruction. The `label` is of particular importance, since it marks the actual location of the start of the given instruction. If anything is moved or changed within the instruction stream, this label will have to be recalculated (and some instructions are aligned on dword boundaries, so this is not entirely trivial); thus, this value should not be relied upon except right after loading or right after a recompile. Each `Instruction` also has a method `nextOffset(int)` which returns the offset of the next `Instruction` assuming this one is placed at the given location—this information is used when recalculating the `label` fields of `Instructions`. Boolean flags include:

`labelled` The instruction is the target of a branch.

`branches` The instruction can change control flow.

`calls` The instruction can cause a method invocation.

`returns` The instruction terminates a method invocation.

Flag information is used in the construction of the control flow graph.

2.2 Conversion to Pointers

Once the bytecode has been parsed and converted into a singly-linked list of `Instructions`, offsets are converted into pointer references. This lets instructions be inserted and deleted with considerably less effort. This process is done through the `build` method of the `ByteCode` class.

First, an array of `Instructions` is created and elements of the linked list of Instructions are inserted sequentially into this array. This is of course quite wasteful in space, but if we desire to convert offsets to pointers, it is considerably more efficient to perform a binary search on an array to locate the instruction corresponding to a given offset rather than searching through the entire linked list each time. We could also have converted the bytecode directly into an array, but this would not only require two passes (one to calculate size, one to build the array), but would not give us the flexibility to insert and delete instructions later on (for which a linked list is much preferred).

Each `Instruction` has a method called `offsetToPointer.` This method is called from the `ByteCode` object with that same `ByteCode` object as an argument. If the `Instruction` needs to convert an offset to a pointer, it calls the binary search routine of the `ByteCode` object to locate within the array of `Instructions` the `Instruction` with the appropriate label/offset.

After this process is complete the `ByteCode` object is no longer needed.

2.3 The Control Flow Graph

The CFG is currently built on demand and then discarded when it is no longer needed; this is only done in the `showCFG` method of the `ClassFile` class. Constructing the CFG involves creating a new object of class `CFG` and calling the method `buildCFG` for each method in the class file.

Constructing a new `CFG` object does most of the work. Here the list of `Instructions` is examined, and basic blocks extracted. Each time a basic block is identified (and leaders are easy to identify given the boolean flags associated with each `Instruction`), it is pulled off the front of the `Instruction` list and a `BasicBlock` object is allocated to hold those instructions. These `BasicBlock` objects are then collected into a singly-linked list. Thus, once the CFG has been built, the list of `Instructions` is no longer a single list, but a collection of small lists stored in a list of `BasicBlocks`.

Within the `CFG` object, as the `BasicBlocks` are constructed, they are also stored in a `Hashtable`, indexed by the first `Instruction` of the `BasicBlock`. This will allow us to locate the `BasicBlock` corresponding to the target `Instruction` of any change in control flow (since all such `Instructions` are necessarily leaders).

Discovering the basic blocks is of course only part of building the control flow graph. It is also necessary to generate predecessor and successor lists for each basic block, in order to support forward and backward analyses. This is done through the `buildCFG` method of the `CFG` class. Each `BasicBlock` has two `Vectors`, one of predecessors and one of successors; building the complete control flow graph is then just a matter of running through the basic blocks, and for each `BasicBlock` b adding the list of possible successors to b's successor `Vector`, and for each such successor adding b to its predecessor `Vector`.

There is one difficulty with this process: exceptions. Because the exception handling code is contained within the normal code, extra care must be taken to ensure each time an exception might be thrown that the exception handling code is marked as a successor. Currently this is correctly done for the `athrow` bytecode, but is not done for method invocations[5] yet.

3 Modifications

Once the internal representation of instructions has been constructed, or by working on the control flow graph if that has been constructed, the code can be arbitrarily changed.

3.1 Modifying the List of Instructions

Given the linked list form representing the bytecode, inserting and deleting instructions is reasonably easy; one just performs list surgery as appropriate. Naturally, care must be taken to avoid deleting the targets of branches or inserting code prior to targets without updating the pointer of the source instruction. Instructions themselves are created by instantiating one of the leaf subclasses of `Instruction` and filling in the required arguments.

If the CFG has been created and the bytecode is being modified, not only is it necessary to ensure basic blocks remain basic blocks, but the `pred` and `succ Vectors` must be updated as well. It is not necessary to update the `Hashtable` since it is only used in the initial construction.

Some functions are available to assist in modifications to the class file. These are described below; note that this is a very incomplete set of utility functions. More will eventually be written, but this all that is available right now.

[5] The difficulty here is that we have to locate and load the class file for the given method in order to find out what exceptions might be thrown.

3.2 Modification to the Constant Pool

The class file maintains an array of variable length structures called the `constant pool`. Many instructions have read-only arguments indirectly referred to by an index into the constant pool; certainly all references to methods, classes or fields are performed in this way.

Thus, to insert an instruction or to delete one may require modifications to the constant pool. Since the constant pool maintains only unique entries (ie, entries are shared), this requires searching through the constant pool to see if the required entry is there or not. At the moment the following functions (in the `ClassFile` class) are available to assist in modifications; note that these are not efficient for batch modifications—if there are many changes to be made it might be faster to put the constant pool into some sort of search structure.

`increaseConstantPool(short)` This increases the size of the constant pool larger by the given amount. It ensures the reference to the constant pool maintained as a `static` member of the `Instruction` class is up to date.

`locateUtf8(String)` This scans through the constant pool looking for a `CONSTANT_Utf8_info` string matching the given one; returns the index if found, or 0 if not.

`locateNameAndType(int,int)` This scans through the constant pool looking for a `CONSTANT_NameAndType_info` entry with name and type indices matching the given ones; returns the index if found, or 0 if not.

`locateClass(int)` This scans through the constant pool looking for a `CONSTANT_Class_info` entry with name index matching the given one; returns the index if found, or 0 if not.

`newUtf8(String)` Returns a newly created `CONSTANT_Utf8_info` structure for the given string.

`addFieldRef(String,String,String)` Creates a new `CONSTANT_FieldRef_info` object according to the given class name, name and type descriptor strings. Since a field reference requires the indices of a `CONSTANT_Class_info` object and a `CONSTANT_NameAndType_info`, and these require a few `CONSTANT_Utf8_info`s, all these are created as well. All objects except for the field reference itself are searched for within the constant pool before being created, and only created if they don't already exist, in which case they are inserted into the constant pool.

`addMethodRef(String,String,String)` Does the same task as with `addFieldRef`, but creates a `CONSTANT_MethodRef_info` instead.

`insertCode(method_info,Instruction,Instruction)` This inserts the given code into the given method, after the given `Instruction`. The appropriate list surgery is performed, and the size of the `Code_attribute` is adjusted. This does not adjust any labels within the code, change the code size[6] or perform checks for changes in jump targets (most of this is done during the recompile anyway). Some debugging information stored as attributes may no longer be valid after this since line numbers and code offsets might no longer correspond. This method is really meant to be instructional more than functional.

[6] Which is distinct from the size of the code *attribute*.

3.3 Modifying Fields

Fields of a class are maintained within an array of `field_info` structures. These structures only describe the field itself; if a field is to be used within the bytecode a `CONSTANT_FieldRef_info` will have to be constructed as well, for use by the field load/store bytecode instructions. The following functions are available for assisting in the creation of new fields:

`increaseFields(short)` This increases the size of the array of `field_info` structures by the given amount.

`addField(short,String,String)` Creates a new `field_info` object according to the given access bitmask (see the `ACC_*` constants of the `ClassFile` class), name and type descriptor for the field. A `field_info` object requires constant pool indices for the name and type, so the appropriate objects are created and put in the constant pool if they don't exist already. Note that no check is made that the field is distinct from other fields.

4 Recompiling

Once parsed, and perhaps modified, the internal representation of the class file can be written out again. The most critical part of this process is the conversion of the list of `Instruction`s for each method back into an array of bytecode. This process if done through the `ClassFile` method `recompile(method_info)` or `recompiledAll()` (which as you may expect recompiles all methods). Note that constructing the control flow graph dismembers the list of `Instruction`s; the list will have to be reconstructed in this case before recompilation can be done (a simple process since each `BasicBlock` maintains a `tail` as well as `head` for its list of `Instruction`s, but it is not currently done).

First the offset from the start of code is recalculated for each `Instruction`; the list is scanned and labels adjusted through the `relabel(Instruction)` method, which itself calls `nextOffset(int)` for every instruction asuming instructions start with offset `0`. An array of the correct size is then allocated, and filled in with the bytecodes for each `Instruction` and its associated arguments. This latter step is accomplished by calling the `compile(byte[],int)` method of each `Instruction`. As well as writing out the bytes for the `Instruction`, this also converts references to targets of branch instructions back into relative bytecode offsets. The exception table is also converted back to using code offsets.

5 Software

5.1 Installation

The software itself comes in a single zip-file. Make a directory and use info-zip to unpack the files into it. One of the files included is a `Makefile`—run `make` to complete the installation.

A makefile is used to simplify the installation, even though recompiling the whole thing is just a matter of running `javac Coffi.java`. Before compiling, however, it is necessary to extract the

`Instruction` heirarchy, all of which is contained in the one file `Instructions.java`. A small program called `Explode` is therefore included, which does a very crude parse of the `Instructions.java` file to extract each subclass and create a distinct file for it. This step is necessary to accommodate Java compilers with small input file capacities. Note that changes to `Instruction` subclasses should be done to the code within the `Instructions.java` file and the command `java Explode Instructions.java` rerun, rather than making direct modifications to the individual `Instruction` subclass files.

The command `make clean` can be run to delete all `.class` files and the exploded `Instruction` subclasses. The command `make zip` builds a zip-file `Coffi.zip` containing all source code.

5.2 Usage

To compile the entire program, run `javac Coffi.java`. Run `java Coffi -h` for help about command-line arguments and switches.

5.3 License

Coffi is covered under the *GNU Public License* [Fou91].

6 Known Bugs

The following bugs/features are known to exist. They will be eradicated as time permits.

- The control flow graph does not take into account possible control jumps for exception handling from method calls.

References

[Cor95] Sun Microsystems Computer Corporation. The Java virtual machine specification: Release 1.0 beta, draft. http://java.sun.com/doc/language_vm_specification.html, August 1995.

[Fou91] Free Software Foundation. GNU general public license, version 2. http://www.gnu.org/copyleft/gpl.html, June 1991.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification, version 1.0*. Addison-Wesley, 1996. See: http://java.sun.com/doc/language_specification.html.

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification: Release 1.0.2*, chapter 4: The class File Format. Addison-Wesley, 1996. Draft available at: http://java.sun.com/doc/language_vm_specification.html.