Annotating Java Bytecode [*]

Project Report
308-621 Optimizing Compilers
McGill University
April 2000

Patrice Pominville
9745398
patrice@cs.mcgill.ca

Abstract

The emergence of a new class of highly dynamic and object-oriented programming languages presents new challenges to the established field of Compiler Optimization. With the advent of Java and it's popularity, there is now a great incentive for addressing these issues. This paper describes how the Runtime performance of Java could benefit by annotating Java code by means of classfile attributes and details how support for this feature has been added and implemented in the Soot Framework.

1 Introduction

Java is a clean, dynamic object-oriented language that is compiled into classfiles that contain bytecode. Bytecode is a high-level, platform independent program representation that is interpreted by Java Virtual Machines (JVMs). Performing optimizations at the classfile level has distinct advantages with respect to optimizing JVMs:

- Classfiles are portable across all platforms having JVM implementations; optimizing classfiles has the potential of improving performance across all these JVMs.
- Classfile optimizations are performed statically and need only be done once; dynamic on-the-fly JVM optimizations incur runtime costs and must be repeated for each program run.

Hence optimizing classfiles is highly desirable; unfortunately although significant gains can be achieved by performing such optimizations [1], this approach is inherently limited by the nature of bytecode. Bytecode, as a secure, semantically rich platform independent abstraction, is a much higher-level code representation than native machine code; typically one bytecode instruction will map to several native machine instructions. As a result many traditional compiler optimizations cannot be expressed at the bytecode level and thus cannot be generated statically by an ahead-of-time compiler.

On another thread, because of the nature of the Java Runtime there is an opportunity for significant dynamic, profile based optimizations. However at present such optimizations are only done by high performance JVM implementations as there is no standard way to express such information in bytecode.

As the previous comments should have made clear, there is an information gap between the static analysis phase and the Java runtime which consequently imposes a heavy performance burden on JVM implementations. What is needed is a mechanism to bridge this gap while preserving all the characteristics of the Java platform as well as the heavy software engineering investments made in current JVMs. One solution to this problem is to make use of *classfile attributes.* Classfile attributes are a very flexible mechanism to attach arbitrary information to classfiles. Such attributes can be user defined and attached to classfiles without altering their semantics: a JVM implementation will safely ignore attributes it does not understand .

2 Classfile Attributes

The De facto file format for Java bytecode is the classfile format [11]. Builtin to this format is the no-

[*] A complementary web site to this paper can be found at `http://www.cs.mcgill.ca/~patrice/cs621`

```
attribute_info {
        u2 attribute_name_index;
        u4 attribute_length;
        u1 info[attribute_length];
}
```

Figure 1: Classfile Attribute Data Structure

tion of attributes. Classfile attributes are simple data structures present inside classfiles having the format specified by Figure 2.

In Figure 2, `attribute_name_index` is a 2 byte unsigned integer value corresponding to the index of the attribute's name in the classfile's *Constant Pool*, `attribute_length` is 4 byte unsigned integer specifying the length of the attribute's data and `info` is an array of `attribute_length` bytes that contains the actual uninterpreted raw attribute data.

A handful of attributes are already defined as part of the Java Virtual Machine specification [11]; in particular the actual bytecode for a non-abstract, non-native method is contained in the `Code` classfile attribute. However the power of the attribute mechanism lies in the fact that arbitrary attributes can be user defined, as long as their names to not clash with those of the standard attributes. Adding custom attributes will not break compatibility with exiting JVMs as these will simply ignore unknown attributes, without altering the semantics of the given classfile.

Attributes can be associated with 4 different structures within a classfile. In particular classfiles have one `class_info` structure as well as `method_info` and `field_info` structures for each of the class' methods and fields respectively. Each of these 3 structures contain an *attribute table* which can hold an arbitrary number of `attribute_info` structures. Each non-native, non-abstract method's attribute table will contain a unique Code attribute to hold the method's bytecode. This Code attribute has itself a attribute table of it's own, that can contain a few standard attributes and arbitrary custom attributes.

2.1 Possible Uses for Attributes

As hinted at in the previous section, the nature of performance enhancing classfile attributes could be dual: attributes could convey either static compiler analysis information or execution profile information. We shall examine both these application domains presently.

2.1.1 Static Compiler Analysis Information

As stated previously, many traditional compiler optimizations cannot be fully expressed in bytecode because of it's abstract high-level nature; however by defining custom classfile attributes it is possible to get around this problem. The following is a non exhaustive list of possible applications for such attributes.

Register Allocation For each method, register allocation can be abstracted and computed statically. This is definitely a promising avenue for attributes and has already been show effective by 2 research groups [2] [3].

Array Bound Checks The Java Language mandates that each array access be checked to be valid (within the array's bounds). This is done by JVMs at runtime; however as previous work has shown, many such accesses can be shown valid at compile time. These could be annotated by means of attributes.

Null Pointer Checks The Java Language mandates that each object reference be determined non-null before it is dereferenced. Although this is often achieved by hardware traps which incur no runtime performance overhead, some architectures could benefit from a nullness attribute.

Stack Allocation of Objects In bytecode there is only one instruction for allocating memory for objects, and the memory is always allocated on the heap. However certain static analyses such as *escape analysis* can determine that some object can be safely allocated on the stack, thus potentially reducing some memory management runtime overhead [4].

Runtime Static Method Binding Virtual call sites that cannot be safely resolved statically often can be resolved dynamically if certain conditions holds. These conditions could be supplied as annotations.

Parallel Computations In a method, if two regions of code, or method calls are deemed to be independent then this could be annotated and the JVM would be given a chance to execute them in parallel.

Exception Handling as Control Flow Some application use exception handling intensively as a control flow mechanism. Providing such a hint could allow a JVM to potentially use a possibly different exception handling mechanism tailored to this type of situation. The expected target of an exception handler could be specified, and given certain constraints, the exception handler could be called directly from the catch clause.

2.1.2 Profile Information

The most successful paradigm to date for squeezing speed out of Java, has been for JVMs to tune their execution according to the execution profile of a program by natively compiling the most heavily used pieces of code. Hence JIT enabled JVMs typically compile only the most executed methods based on their dynamic appraisal of a specific run. This has yielded significant speed improvements over purely interpreted bytecode and much effort is still being invested in this approach as the *Java HotSpot Project* gives testimony [12].

The drawback with this approach is that an overly heavy burden is placed on JVMs that must dynamically profile the execution of the code they run and make optimization decisions on-the-fly. All this incurs a runtime cost and can lead to highly complex and buggy JITs. A solution would be to use attributes to provide such profiling information gathered on previous, ahead-of-time runs of a program. Given that most programs present a similar execution profile from run to run, such information could ease the burden of JIT implementors and reduce runtime profiling cost. The following is a non-exhaustive list of possible applications for such attributes.

Hot Methods Methods could be given a *hotness* rating. This could provide useful hints for JITs in helping them decide which methods to compile.

Persistent Objects Allocation sites could be given a persistence rating based on the expected lifetime of an allocated object. This could provide a useful hint to high-performance garbage collectors on how to efficiently manage object allocation.

Garbage Collection There is no garbage collector that is optimal for all programs. For some programs a generational garbage collection is a big plus, while for others it is overkill or the generational allocation assumption simply does not hold. Based on profiling, annotations could be produced to specify which type of GC would be best for a program. A JVM could then use this hint to select an appropriate GC at runtime.

Branch Prediction Annotation
Annotating which bytecode branches are most frequently taken could help a JVM produce more efficient native code.

Hot Data Based on profiling, if certain objects are often accessed as a group, this information could be conveyed to the JVM for better memory allocation and data locality. This attribute could provide a virtual memory map for the JVM.

2.2 Annotation Issues

In deciding what to annotate and in designing specifications for custom attributes many issues must be considered. First it is highly desirable that the annotations be compatible with Java's execution model. That is, annotations should be platform independent and ideally should not compromise the verifiability of classfiles. The latter requirement might mandate special considerations while designing an attribute scheme. For example in [3]'s virtual register allocation scheme, virtual registers are monotyped to ensure speedy verifiability at runtime. Luckily for many of the annotation ideas exposed previously, verifiability is a non-issue: for example most profile annotations can simply be viewed as 'hints' given to the runtime.

In designing annotations it is also important to take into account the classfile bloat that will result from these. This could have negative performance affects due to network bandwidth and caches; furthermore, the time needed to process the annotations themselves could overshadow any possible gains that could be obtained from these. Finally annotations should be general enough so that they provide benefits for most programs on most architectures.

3 Attribute Support in Soot

Soot [8] is an object-oriented bytecode analysis and optimization framework implemented in Java and developed by the Sable Compiler Research Group at McGill University [7]. In the context of this project, we have extended the framework to support the embedding of custom, user defined attributes in classfiles. The Soot framework enables one to easily define and implement various compiler analyses. The added attribute support as is presented in this section, enables the implementation of a wider range of such analyses, encompassing those whose results cannot be expressed directly in bytecode.

```
public interface Host
{
    public List getTags();
    public Tag getTag(String aName);
    public void addTag(Tag t);
    public void removeTag(String name);
    public boolean hasTag(String aName);
}
```

Figure 2: The Host Interface

```
public interface Tag
{
    public  String getName();
    public  byte[] getEncoding();
    public  String toString();
    public  void setValue(byte[] value);
}
```

Figure 3: The Tag Interface

3.1 The Host and Tag Interfaces

Attribute support in Soot has been achieved by adding two key interfaces: `Host` and `Tag`. Hosts are objects that can hold Tags; conversely Tags are objects that can be attached to Hosts. These interfaces are listed in Figures 2 and 3. There are 4 Soot classes that implement the Host interface; these are `SootClass`, `SootField`, `SootMethod` and `Unit`, the latter of which is Soot's abstract notion of a bytecode instruction. Application specific subclasses of Tag can be created and attached to these Hosts by implementors of Soot based analyses. As can be easily inferred, there is a natural mapping between these 4 Soot classes and the attribute architecture present in classfiles as described in Section 2. Tags attached to a SootClasses will be compiled into an entry in the attribute table of the corresponding class and similarly for methods and fields. Tags attached to Soot Units will compiled into entries of the Unit's method's Code attribute table, along with the bytecode Program Counter (PC) of the specific instruction they index.

3.2 Producing Annotated Classfiles

The process of translating Tags held by Soot Hosts into actual classfile attributes is now detailed. In the current state of affairs Soot produces Jasmin code [9] for it's processed classfiles. The Jasmin code is then transformed by Jasmin into actual classfiles. To enable attribute support, we have extended the Jasmin syntax with 4 new directives: `.class_attribute`, `.method_attribute`, `.field_attribute` and `.code_attribute`. These all have the following format:

```
attribute directive attribute_name attribute_value
```

where `attribute_name` corresponds to the attribute's name that will be stored in the classfile's ConstantPool and `attribute_value` is the actual value of the raw byte array for the attribute which is encoded in Base64 in order to maintain the textual format of Jasmin code. Our custom Jasmin version will compile the appropriate attribute in the resulting classfile from these triples, translating the attribute values in Base64 back to a raw byte array. There is a peculiarity to this scheme: for `.code_attribute`, Jasmin will replace the first 2 bytes of the attribute's data by the PC of the instruction it is referring to, as Soot currently lacks a mechanism for abstracting the Program Counter for a method's bytecode (see Section 5.1). Hence at present, all Soot generated Code attributes in classfiles start with a 2 byte PC index that specify an instruction context.

As expected each Tag attached to a SootClass will generate a corresponding `.class_attribute` in the Jasmin code, and similarly SootField attributes translate to `.field_attribute` directives, SootMethod attributes to `.method_attribute` directives and Unit attributes to `.code_attribute` directives.

These directives must be produced in Jasmin code at specific locations:

.class_attribute These must be found immediately *before* the class' field declarations.

.field_attribute These must be found immediately *after* the field declaration they relate to.

.method_attributes These must be found immmediately *after* the method declaration they relate to.

.code_attribute These must be found immediately *after* the instruction they relate to.

Sample Jasmin code embedding `.code_attributes` is given in Figure 4.

3.3 Auxiliary Support in Soot for Tags

Several utility classes and interfaces have been added to Soot to provide additional support for Tags. An overview of these is now given.

```
iadd
daload
dastore
.code_attribute ArrayCheckTag AAAB
dastore
.code_attribute ArrayCheckTag AAAB
aload_0
```

Figure 4: Sample Annotated Jasmin Code

3.3.1 The TagManager class

This class is meant to contain static methods to provide Tag related functionality. At present it provides a flexible facility for printing out Tags: a `TagPrinter` can be registered and will subsequently be used for printing calls made through it's interface.

`TagManger` also currently provides a lookup mechanism for mapping an attribute name onto the proper Soot class (if any) corresponding to the attribute. This is useful to decode Soot attributes found when reading classfiles.

3.3.2 The TagPrinter Interface and the StdTagPrinter class

The TagPrinter interface is meant to be implemented by classes that can print Tags. For example a PreatyPrinter class or a XML printer class could implement this interface formating tags in a distinct fashion. As previously noted, a `TagPrinter` is registered with the `TagManager` to configure the latter's Tag printing behavior.

One such class that has been implemented and is now available in Soot is the `StdTagPrinter` that prints out attributes in a easily parsible format. This facility is used by the PrintAttributes utility (see Section 6).

3.4 A Sample Attribute

A first Soot attribute is already being developed and is currently successfully supported by the framework. This attribute has been tentatively named `ArrayCheckTag` and can be used to annotate array accesses that have been proven by some analysis to be within bounds, thus indicating to a JVM that it can safely omit corresponding runtime array bound checks. This attribute is currently being used in Feng Qian's work at McGill University in implementing a Soot based analysis for unnecessary array bound checks elimination. Figure 5 exhibits the salient point in his

```
 if (maxValueMap.containsKey(index)) {
   AbstractValue indexV =
     (AbstractValue)maxValueMap.get(index);

   if (indexV.lessThan(arrayLength))
     upCheck = false;
  }
 else if (index instanceof IntConstant) {
   AbstractValue tmpAv =
      AbstractValue.newConstantValue(index);
   if (tmpAv.lessThan(arrayLength))
     upCheck = false;
 }
 Tag checkTag = new ArrayCheckTag(lowCheck, upCheck);
 if (!lowCheck || !upCheck) {
>    s.addTag(checkTag);
 }
```

Figure 5: Adding an ArrayCheckTag to a Unit

analysis. In the given code, having determined that either an upper array bound or lower array bound need not be checked, he creates an `ArrayCheckTag` and attaches it to the Jimple statement that contains the array reference. Soot then automatically takes care of propagating these tags to the appropriate bytecode array access instruction at code generation time.

The current encoding of the ArrayCheckTag attribute's data at the bytecode level compromises 3 bytes. First like currently all Soot generated Code level attributes the first 2 bytes are the PC of the bytecode instruction it references. The remaining byte is used to encode which of the upper and lower bound checks can be omitted. This in fact requires only 2 bits of the byte. If the first bit is on, then an the upper bound check can be omitted and if the second bit is on then the lower bound check can be omitted.

Thus the total size of an ArrayCheckTag attribute in a classfile is 9 bytes (6 bytes for the header and 3 bytes for the data), plus the cost of the ArrayCheckTag's ConstantPool entry which is shared by all such attributes in a given class. Hence annotating array bound checks can be done effectively in terms of code size. Note however that the size and format of this attribute are likely to grow somewhat as we standardize the encoding of Soot attributes. In particular we plan on introducing major/minor version numbering of attributes for future scalability.

```
> java PrintAttributes FFT.class
<FFT:public void <init>()>+61/ArrayCheckTag AA==
<FFT:public void <init>()>+73/ArrayCheckTag AA==
<FFT:public void <init>()>+74/ArrayCheckTag AA==
```

Figure 6: Sample Output of the PrintAttributes Utility

4 Tools to Visualize Annotated Classfiles

4.1 PrintAttributes Utility

This is a simple utility to print out custom attributes in a easily parsible format. It uses JavaClass API [10] to extract the attributes from the specified classfile and uses Soot's `StdTagPrinter` class to print them out. The utility currently only takes one argument, the filename of the class to print out. Sample output for a classfile that has been annotated by the array bounds check analysis is given in Figure 6.

4.2 Hypertext Browsing of Attributes

We have extended and modified JavaClass' `class2HTML` utility in support of custom attributes. The resulting utility processes classfiles and produces corresponding HTML files that have hyperlinks to attributes. The utility uses Soot to format the attributes it finds. If an attribute is understood by Soot it will be formatted in a human friendly fashion, usually by instantiating a class for the attribute and calling upon it's toString method. Sample output produced by this utility can be found at `http://www.cs.mcgill.ca/~patrice/cs621/FFT.html`

5 What's Missing for more Complete Attribute Support

5.1 Abstracting the PC

It is at times useful to produce attributes whose data contains references to the explicit value of the PC (Program Counter) of certain bytecode instructions. This cannot be done easy in Soot since Soot works in terms of Units and not absolute bytecodes. Although it would be possible to let Tags refer to Units and then translate these references into PCs at the Jasmin level, this scheme still presents many problems as Units in Soot can be edited, reordered or otherwise deleted by various optimizations, the effect of which would have to be reflected in the referencing Tags. We are currently still evaluation how abstracting the PC could be best achieved in the Soot framework.

5.2 Reading Soot Attributes back into Soot

At present we can produce custom Soot annotated classfiles but we cannot read these same attributes, or any other custom attributes, back into Soot. It would also be desirable to parse attributes from text files when processing a set of class files. These issues are presently being addressed..

6 Related Work

To the best of our knowledge there has been little work done in investigating the possible uses of classfile attributes to improve the performance of bytecode. We are aware of only 2 research groups that have been investigating this topic and both are focused on conveying register allocation information through attributes [2] [3]. This involves developing a *Virtual Register allocation* scheme where one assumes an infinite number of registers and then proceeds to statically minimize the number that are actually used. The scheme developed by [3] monotypes each virtual register which allows for efficient runtime verifiability of their attributes; attributes to deal with spills are also presented. Experimental results obtained to date by both groups exhibit significant code speedups.

7 Future Work

Most of the attribute support in Soot is now complete; although some important features still require our attention, work can now be more focussed on developing innovative analyses that make use of this new facility. It will also be crucial that runtime systems understand and make use of the classfile attributes generated by these. In this respect we expect to collaborate with IBM's HPCJ [6] group and investigate how the Kaffe OpenVM [5] could be modified in support of these. Once we have such runtime support we will conduct experimental results to validate the soundness and effectiveness of our annotations.

8 Conclusion

Classfile attributes can be exploited to convey extra information to JVMs and allow for faster code execution speeds. Support for generating custom classfile attributes has now been added to *The Soot bytecode Optimization Framework*. Soot analyses can now make use of a simple API to annotate various Objects in the framework; classfile attributes will automatically be generated from these annotated objects. Two tools have been developed to view custom annotated classfiles. One of these allows for hypertext browsing of the generated annotated classfiles. Although attribute support in the Soot framework is now extensive, there still remains some work to be done in order to have a more complete and flexible implementation; most notably we must find a way to abstract the PC for methods in Soot. Nonetheless Soot is now attribute enabled can be currently be used as an effective tool to generate attributes by those who require such functionality.

Acknowledgements

Thanks are in order for Raja Vallée-Rai, principal author of the Soot framework, for helping conceive the attribute support facilities that have been described in this paper; and Feng Qiang for providing the array bounds check analysis here described, effectively the first analysis to make use of Soot attributes.

References

[1] Raja Vallée-Rai, Etienne Gagnon, Laurie Hendren, Patrick Lam, Patrice Pominville and Vijay Sundaresan, “Optimizing Java Bytecode using the Soot Framework: Is it Feasible?”, *CC/ETAPS 2000*, LNCS 1781, pp. 18-34, 2000.

[2] Ana Azevedo, Alex Nicolau and Joe Hummel. “Annotating the Java Bytecodes in Support of Optimization”, *ACM Java Grande Conference*, San Francisco, CA, June 1999.

[3] Joel Jones and Samuel Kamin, “Annotating Java Bytecodes in Support of Optimization”, *Concurrency: Practice and Experience*, forthcoming.

[4] David Gay, Bjarne Steensgaard “Fast Escape Analsis and Stack Allocation for Object-Based Programs”, *CC/ETAPS 2000*, LNCS 1781, pp. 82-93, 2000.

[5] *The Kaffe OpenVm,* `http://www.kaffe.org`

[6] *IBM’s High Performance Compiler for Java* `http://www.research.ibm.com/topics/popups/innovate/java/html/hpcj.html`

[7] *The Sable Compiler Research Group* `http://www.sable.mcgill.ca`

[8] *The Soot Bytecode Optimization Framework* `http://www.sable.mcgill.ca/soot`

[9] *The Jasmin Bytecode Assembler* `http://mrl.nyu.edu/meyer/jvm/jasmin.html`

[10] *JavaClass Bytecode Engineering Framework* `http://www.inf.fu-berlin.de/ dahm/JavaClass`

[11] Tim Lindholm, Frank Yellin *The Java Virtual Machine Specification, Second Edition,* `http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html`

[12] *Java HotSpot Technology,* `http://java.sun.com/products/hotspot`