A Framework for Optimizing Java Using Attributes

Patrice Pominville, Feng Qian, Raja Vallée-Rai, Laurie Hendren, and Clark Verbrugge
{patrice,fqian,kor,hendren}@cs.mcgill.ca
clarkv@ca.ibm.com

Sable Research Group, School of Computer Science, McGill University
IBM Toronto Lab

Abstract. This paper presents a framework for supporting the optimization of Java programs using attributes in Java class files. We show how class file attributes may be used to convey both optimization opportunities and profile information to a variety of Java virtual machines including ahead-of-time compilers and just-in-time compilers. We present our work in the context of Soot, a framework that supports the analysis and transformation of Java bytecode (class files)[21,25,26]. We demonstrate the framework with attributes for elimination of array bounds and null pointer checks, and we provide experimental results for the Kaffe just-in-time compiler, and IBM's High Performance Compiler for Java ahead-of-time compiler.

1 Introduction

Java is a portable, object-oriented language that is gaining widespread acceptance. The target language for Java compilers is Java bytecode which is a platform-independent, stack-based intermediate representation. The bytecode is stored in Java class files, and these files can be be executed by Java virtual machines (JVMs) such as interpreters, just-in-time (JIT) compilers, or adaptive engines that may combine interpretation and compilation techniques, or they can be compiled to native code by ahead-of-time compilers. The widespread availability of JVMs means that Java class files (bytecode) have become a popular intermediate form, and there now exists a wide variety of compilers for other languages that generate Java class files as their output. One of the key challenges over the last few years has been the efficient execution/compilation of Java class files. Most of the work in this area has focused on providing better JVMs and the best performing JVMs now include relatively sophisticated static and dynamic optimization techniques that are performed on the fly, at runtime. However, another source of performance improvement is to optimize the class files before they are executed/compiled. This approach is attractive for the following reasons:

(1) Class files are the target for many compilers, and class files are portable across all JVMs and ahead-of-time compilers. Thus, by optimizing class files,

there is potential for a common optimizer that can give performance improvement over a wide variety of source language and target VM combinations.

(2) Class file optimization can be performed statically and only needs to be performed once. By performing the class file optimization statically we can potentially reduce the burden on JIT optimizers, and can allow for more expensive optimizations than can be reasonably performed at run-time. In general, one would want the combined effect of class file optimization and on-the-fly optimization.

Although optimizing class files is beneficial, there are limits to what can be expressed in bytecode instructions. Some bytecode instructions are relatively high-level, thus they hide details that may be optimizable at lower-level representations. For example, an access into an array is expressed as one bytecode instruction, but at run-time the array reference must be checked to ensure it is not null, the array bounds must be checked to ensure the index is in range, and appropriate exceptions must be raised if these checks fail. Clearly, one would like to avoid generating native code for the checks if a static analysis can guarantee that they are not needed.

We have developed a general mechanism for using class file attributes to encode optimization information that can be determined by static analysis of bytecode, but cannot be expressed directly in bytecode. The basic idea is that a static analysis of Java bytecode is used to determine some program property (such as the fact that an array index expression is in range), and this information is encoded using class file attributes. Any JVM/compiler that is aware of these attributes can use the information to produce better native code. In addition to array bound checks, such optimization attributes could be used for: register allocation[1,13], eliminating useless null pointer checks, stack allocation of non-escaping objects, devirtualization based on run-time conditions, specifying regions of potentially parallel code, or to indicate the expected behaviour of exceptions.

Attributes can also be used to convey profile information. Currently, advanced JVMs use on-the-fly profiling to detect hot methods, which may be optimized or recompiled on the fly. However, ahead-of-time compilers cannot necessarily make use of such dynamic information, and even for dynamic JVMs it may also be beneficial to use static information. For example, one could gather profile information from many executions, use information gathered from trace-based studies, or estimate profile information using static analysis. In these cases, the profile information could be conveyed via attributes.

In this paper we provide on overview of our general approach to supporting attributes in the Soot framework. We provide an infrastructure to support a very general notion of attributes that could be used for both optimization attributes and profile-based attributes. The paper is organized as follows. In Section 2, we briefly summarize the Soot framework, and outline our support for attributes. To demonstrate our approach we show how we applied it to the problem of eliminating array bounds checks, and we show how these attributes are expressed in our framework in Section 3. In order to take advantage of the optimization attributes, the JVM/compiler processing the attributed class files must be aware of the attributes. We have modified both the Kaffe JIT and the IBM HPCJ (High Performance Compiler for Java) ahead-of-time compiler to take advantage of the array bound attributes, and we report experimental results for these two systems in Section 4. A discussion of related work is given in Section 5, and conclusions and future work are given in Section 6.

2 Attributes and Soot

Our work has been done in the context of the Soot optimizing framework[21,25, 26]. Soot is a general and extensible framework to inspect, optimize and transform Java bytecode. It exposes a powerful API that lets users easily implement high-level program analyses and whole program transformations. At its core are three intermediate representations that enable one to perform code transformations at various abstraction levels, from stack code to typed three-address code. In Figure 1 we show the general overview of Soot. Any compiler can be used to generate the Java class files, and the Soot framework reads these as input and produces optimized class files as output. The Soot framework has been successfully used to implement many well known analyses and optimizations on Java bytecode such as common subexpression elimination, virtual method resolution and inlining[23]. All of these transformations can be performed statically and expressed directly in optimized Java bytecode. Until recently, the scope of these transformations was limited by the semantics and expressiveness of the bytecodes themselves. Hence, optimizations such as register allocation and array bounds check elimination could not be performed. The objective of the work in this paper was to extend the framework to support the embedding of custom, user-defined attributes in class files.

2.1 Class File Attributes

The de facto file format for Java bytecode is the class file format [17]. Built into this format is the notion of attributes that allows one to associate information with certain class file structures. Some attributes are defined as part of the Java Virtual Machine Specification and are essential to the correct interpretation of class files. In fact, all of a class's bytecode is contained in attributes. Attributes can also be user-defined and Java virtual machine implementations are required to silently ignore attributes they do not recognize.

The format of class file attributes is very simple and flexible: attributes consist of a name and arbitrary data. As shown in Figure 2, `attribute_name_index` is a 2 byte unsigned integer value corresponding to the index of the attribute's name in the class file's *Constant Pool*, `attribute_length` is a 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. This simplistic model conveys great freedom and flexibility to those that wish to create custom attributes as they are unhindered by format constraints.

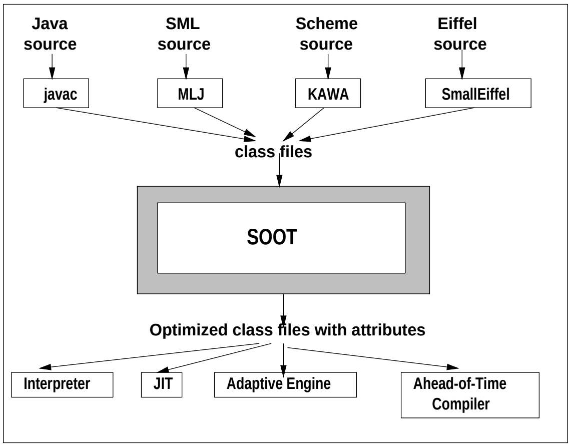

Fig. 1. General Overview

The only binding requirement is for a custom attribute's name not to clash with those of standard attributes defined by the Java Virtual Machine Specification.

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

Fig. 2. Class File Attribute Data Structure

Attributes can be associated with four different structures within a class file. In particular class files 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 three structures contains an *attribute table* which can hold an arbitrary number of `attribute_info` structures. Each non-native, non-abstract method's attribute table contains a unique Code attribute to hold the method's bytecode. This Code attribute has an attribute table of its own, which can contain standard attributes used by debuggers and arbitrary custom attributes.

2.2 Adding Attributes to Soot

An Overview: Figure 3 provides a high-level view of the internal structure of Soot, with support for attributes. The first phase of Soot is used to convert the input class files into a typed three-address intermediate code called Jimple[6,25]. In Jimple, each class is represented as a `SootClass`, and within each `SootClass` there is a collection of `SootField`s and `SootMethod`s. Each method has a method body which is represented as a collection of instructions, with each instruction represented as a `Unit`.

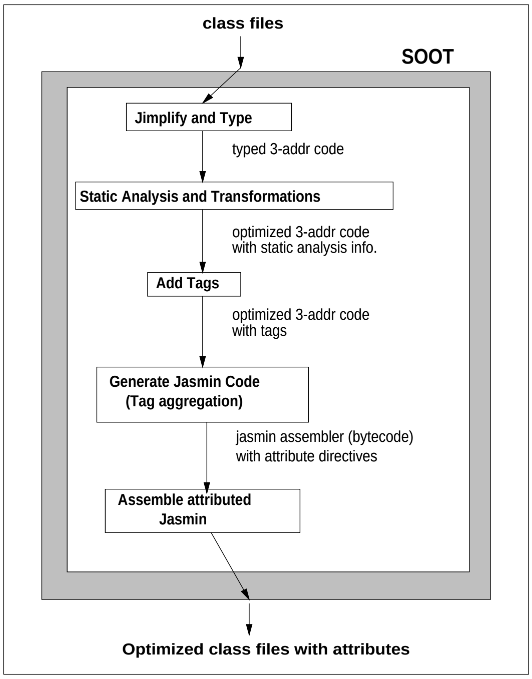
class files
SOOT
Jimplify and Type
typed 3-addr code
Static Analysis and Transformations
optimized 3-addr code with static analysis info.
Add Tags
optimized 3-addr code with tags
Generate Jasmin Code (Tag aggregation)
jasmin assembler (bytecode) with attribute directives
Assemble attributed Jasmin
Optimized class files with attributes

Fig. 3. Internal Structure of Soot

Jimple was designed to be a very convenient intermediate form for compiler analyses, and the second phase of Soot, as shown in Figure 3, is to analyze and transform the Jimple intermediate representation. There already exist many analyses in the Soot framework, but a compiler writer can also add new analyses to capture information that will be eventually output as class file attributes. Soot includes an infrastructure for intraprocedural flow-sensitive analyses, and implementing new analyses is quite straightforward. In Section 3 we discuss our example analysis for array bounds elimination.

After the analysis has been completed, analysis information has been computed, but one requires some method of transferring that information to attributes. In our approach this is done by attaching tags to the Jimple representation (third phase in Figure 3).

After tagging Jimple, the fourth phase of Soot automatically translates the tagged Jimple back to bytecode. During this phase the tags may be aggregated using an aggregation method specified by the compiler writer. Our system does not directly produce class files, but rather it produces a form of assembly code used by the Jasmin bytecode assembler[11]. We have modified the Jasmin assembler language so that during this phase Jimple tags are converted to Jasmin attribute directives.

Finally, the fifth phase is a modified Jasmin assembler that can read the attribute directives and produce binary class files with attributes.

Hosts, Tags and Attributes: 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 Figure 4. There are five Soot classes that implement the Host interface; these are `SootClass`, `SootField`, `SootMethod`, `Body` and `Unit`, the latter of which is Soot's abstract notion of a bytecode instruction.

```
public interface Host {
  public List getTags(); /* gets list of tags associated with the host.*/
  public Tag getTag(String aName); /* gets a tag by name. */
  public void addTag(Tag t); /* adds a tag to the host. */
  public void removeTag(String name); /* removes a tag by name. */
  public boolean hasTag(String aName); /* checks if a tag exists.*/
}

public interface  Tag {
  public String getName();
  public byte[] getValue();
}
```

Fig. 4. The Host and Tag Interfaces

Tags are meant to be a generic mechanism to associate name-value pairs to Host objects in Soot; they are not necessarily mapped into class file attributes. For this purpose, we have introduced the `Attribute` interface, which extends the `Tag` interface. Soot objects that are subtypes of Attribute are meant to be mapped into class file attributes; however, because the Soot framework uses the Jasmin tool to output bytecode, an Attribute object must actually be an instance of `JasminAttribute` for the translation to take place (see Section 2.2 for more information).

Compiler implementors can create application-specific subclasses of `Jasmin-Attribute` and attach these to Hosts. There is a natural mapping between the aforementioned Soot classes that implement the Host interface and the attribute architecture present in class files as described in Section 2.1. `JasminAttributes` attached to a `SootClass` will be compiled into an entry in the attribute table of the corresponding class. `SootMethod` and `SootField` attributes are dealt with similarly. Dealing with `JasminAttributes` attached to Soot `Units` is a bit trickier and is addressed in the following section.

Mapping Unit Attributes into a Method's Code Attribute Table: Soot `Attributes` attached to `Units` do not map trivially to a given class file structure as was the case for `SootClass`, `SootMethod` and `SootField` attributes, because `Units` naturally map to bytecode instructions, which do not have associated attribute tables. The obvious solution is to map all of a method's Unit attributes into entries in the method's Code Attribute's attribute table in the generated class file. Each entry will then contain the bytecode program counter (PC) of the specific instruction it indexes. This is what is done automatically by the Soot framework at code generation time. However, generating one `Code` attribute per `Unit` attribute can lead to undue class file bloat and increased processing and memory requirements by virtual machine's runtime attribute interpretation module. Often different instances of identical Code Attribute attributes should be expressed in a tabular format. For example instead of creating 10 null pointer check attributes for a method, it is more efficient to create a single redundant null pointer table as an attribute for these in the class file. The Soot framework allows an analysis implementor to easily create this table by providing the `TagAggregator` interface as outlined in Figure 5.

By implementing this interface and registering it in the class `CodeAttributeGenerator`, it is possible to selectively aggregate Tags attached to different Unit instances into a single Tag. A user can aggregate all Attributes generated by his/her analysis by iterating over a method's Units and calling the `aggregateTag` method on each of the Tags attached to a given Unit. The `produceAggregateTag` method is then called to produce a single aggregate attribute to be mapped into single attribute in the method's Code Attribute's attribute table.

```
public interface TagAggregator {
  public void aggregateTag(Tag t, Unit u);
  public Tag produceAggregateTag();
}
```

Fig. 5. The TagAggregator Interface

Extending Jasmin for Attribute Support : The Soot framework does not directly generate bytecode; instead it uses the Jasmin tool to do so. Jasmin specifies a textual assembler-like grammar for class files and transforms conforming input into binary class files. Because the Jasmin grammar does not provide constructs for expressing generic class file attributes, we have augmented it to accept and correctly process the added language constructs for attributes.

Informally, an attribute is encoded in Jasmin as a triple consisting of an attribute directive, the attribute's name and the attribute value in Base64.

The attribute directive is one of `.class_attribute`, `.method_attribute`, `.field_attribute` and `.code_attribute`. 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 *before* the end of the method they relate to. Code attributes that correspond to instructions with specific bytecode PC values must express this symbolically. This is done by outputting a Jasmin assembler label before each bytecode that is indexed by some attribute. This label is then used as proxy for the PC of the bytecode it indexes in a Jasmin `.code_attribute` attribute. Labels are encoded inside an attribute's Base64 value data stream by surrounding the label name with the `%` symbol. When our modified Jasmin actually creates the class file attribute, it will replace the labels found in an attribute's data stream by the corresponding 16-bit bigendian PC value.

Figure 6(a) gives an example of a Java method and Figure 6(b) gives an extract of the attributed Jasmin assembler generated, showing the labeled bytecode instructions corresponding to the two array accesses in the program. For this example the generated class file attribute will be 6 bytes: 2 bytes for the PC represented by `label2`, followed by 1 byte for the Base64 encoded value `AA==`(no check needed), followed by 2 bytes for the PC represented by `label3` and finally 1 byte for the Base64 encoded value `Aw==` (array bounds checks needed).

```
public void sum(int[] a) {
  int total=0;
  int i=0;
  for (i=0; i<a.length; i++)
    total += a[i];
  int c = a[i];
}
```

(a) Java source

```
.method public sum([I)V
    ...
label2:
    iaload
    ...
label3:
    iaload
    return
.code_attribute ArrayNullCheckAttribute
    "%label2%AA==%label3%Aw=="
.end method
```

(b) attributed Jasmin

Fig. 6. Attributed Jasmin

Summary: With all of these features the Soot framework is now well endowed with a simple-to-use, generic-attribute generation feature that is tightly integrated into its overall optimization support facilities. This enables analysis implementors to seamlessly augment their analysis with custom attribute support.

3 Attributes for Array Bounds Checks

In the previous section we outlined how we have integrated attributes into the Soot optimizing framework. In this section we illustrate the framework using an example of eliminating array bounds checks. We briefly describe the array bounds check problem, the analyses we use to find unneeded checks, and how to create bounds check tags and convert them into class file attributes by using the Soot framework. Finally, we show how to modify a JVM to take advantage of our attributes.

3.1 The Array Bounds Check Problem in Java

Java requires array reference range checks at runtime to guarantee a program's safe execution. If the array index exceeds the range, the runtime environment must throw an `IndexOutOfBoundsException` at the precise program point where the array reference occurs. For array-based computations, array bounds checks may cause a heavy runtime overhead, and thus it is beneficial to eliminate all checks which a static analysis can prove to be unneeded. In fact, several Java virtual machines implement array bounds check elimination algorithms in their JIT compilers[22,4]. In these systems the optimization is done at runtime as part of the translation of bytecode to native code. However, since the optimization is done at runtime, this approach has two limitations.

(1) Only relatively simple algorithms can be applied because of time constraints.
(2) They lack global information, such as field information and whole-program information. Usually a JIT compiler can not afford the expense of these analyses.

We have developed an algorithm that works at the bytecode level. By statically proving that some array references are safe and annotating these using class file attributes, an attribute-aware JIT can avoid generating instructions for array bounds checks without performing the analysis itself. The attributed class files can also be used by an ahead-of-time compiler, such as IBM's High Performance Compiler for Java.

Java requires two bounds checks, a lower bound check and upper bound check. The lower bound of an array is always a constant zero. The upper bound check compares the index with the array length. On popular architectures, such as IA-32 and PowerPC, both checks can be implemented by just doing an upper bound check with an unsigned compare instruction, since a negative integer is always larger than a positive one when it is interpreted as an unsigned integer.

Thus, in order to be really beneficial, one must eliminate both the upper and lower bound checks.

Another subtle point is that eliminating array bounds checks is often also related to eliminating null pointer accesses. Each array access, for example `x[i]`, must first check that the array `x` is not-null. In many modern compilers null pointer checks are performed by handling the associated hardware trap if a null pointer is dereferenced. In this case the machine architecture guarantees a hardware exception if any very low memory addresses are read or written. In order to do the upper array bounds check the length of the array must be accessed, and since the length of the array is usually stored at a small offset from `x`, this access will trap if `x` is null. Thus, the array bounds check gives a null pointer check for free. If the array bounds check is eliminated, then it may be necessary to insert an explicit null pointer check (since the address of `x[i]` may be sufficiently large to avoid the null pointer trap, even if `x` is null).

3.2 Static Analyses

We have developed two static analyses, *nullness analysis* and *array bounds check analysis*, using the Soot framework. Each of these analyses are implemented using the Jimple typed 3-address representation.

Nullness Analysis: Our nullness analysis is a fairly straightforward flow-sensitive intraprocedural analysis that is implemented as an extension of the `BranchedForwardFlowAnalysis` class that is part of the Soot API. The basic idea is that variable `x` is non-null after statements of the form `x = new();` and statements that refer to `x.f` or `x[i]`. We also infer nullness information from condition checks of the form `if (x == null)`. Since the nullness analysis is intraprocedural, we make conservative assumptions about the effect of method calls.

Array Bounds Check Analysis: The core of the array bounds check algorithm is an intraprocedural analysis. For each method, it constructs an inequality constraint graph of local variables, integer constants, and other symbolic nodes (i.e. class fields, array references, and common subexpressions) similar in spirit to the work by Bodik et. al. [2]. The algorithm collects constraints of nodes and propagates them along the control flow paths until a fixed point is reached. For each array reference, the shortest distance from an array variable to the index indicates whether the upper bound check is safe or not, and the shortest distance from the index to the constant 0 determines the safety of lower bound check.

We have extended this basic analysis in two ways. The first handles the case where an array is assigned to a field in a class. Fields with `final` or `private` modifiers are analyzed first. Often these fields can be summarized by simply scanning all methods within the current class. Using these simple techniques we can determine whether a field is assigned a constant length array object that never changes and never escapes.

The second extension is used to find rectangular arrays. Multidimensional arrays in Java can be ragged (i.e. different rows in an array may have different lengths), and this makes it more difficult to get good array bounds analysis. However, in scientific programs arrays are most often rectangular. Thus, we have also developed a whole-program analysis using the call graph to identify rectangular arrays that are passed to methods as parameters. Rectangular arrays can have two different meanings, *shape rectangular* (each dimension has the same size, but subdimensions can be sparse in memory or aliased), or *memory-shape rectangular* (the object is created by 'multianewarray' bytecode and no subdimensions are ever assigned other array objects). The second type is stricter than the first. For bounds check analysis, shape rectangular is good enough.

3.3 From Analysis to Attributes

After the analysis phase the flow information is associated with Jimple statements. The next step is to propagate this information so that it will be embedded in the class file attributes. This is done by first tagging the Jimple statements, and then specifying a tag aggregator which packs all the tags for a method into one aggregated tag.

Format of Attribute: We first outline the attribute as it eventually appears in the generated class file. The structure of the array bounds attribute is quite straightforward. It has the name `"ArrayNullCheckAttribute"`. Figure 7 shows the format of the array bounds check attribute as it will be generated for the class files.

```
array_null_check_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u3 attribute[attribute_length/3];
}
```

Fig. 7. Array Bounds Check Attribute

The value of `attribute_name_index` is an index into the class file's constant pool. The corresponding entry at that index is a `CONSTANT_Utf8` string representing the name `"ArrayNullCheckAttribute"`. The value of `attribute_length` is the length of the attribute data, excluding the initial 6 bytes. The `attribute[]` field is the table that holds the array bound check information. The `attribute_length` is 3 times larger than the table size. Each entry consists of a PC (the first 2 bytes) and the attribute data (last 1 byte), totalling 3 bytes. These pairs are sorted in the table by ascending PC value.

The least 2 bits of the attribute data are used to flag the safety for the two array bounds checks. The bit is set to 1 if the check is needed. The null check information is incorporated into the array bounds check attribute. The third lowest bit is used to represent the null check information. Other bits are unused and are set to zero. The array reference is non-null and the bounds checks are safe only when the value of the attribute is zero.

Add Attributes: It takes two steps to add attributes to class files when using the Soot annotation scheme. The attribute is represented as a `Tag` in the Soot framework. For the array bounds check problem we proceed as follows:

Step 1: Create an `ArrayCheckTag` class which implements the `Tag` interface. The new class has its own internal representation of the attribute data. In our implementation the `ArrayCheckTag` uses 1 byte to represent bounds checks as explained above. For each array reference, we create an `ArrayCheckTag` object. The tag is attached to a Jimple statement which acts as a `Host` for the array check tag.

Step 2: Create a class called `ArrayCheckTagAggregator` which implements the `TagAggregator` interface.
The aggregator will aggregate all array check tags for one method body. We then register the aggregator to the `CodeAttributeGenerator` class, and specify the aggregator as active. The aggregator generates a `CodeAttribute` tag when it is required to produce the aggregated tag. The `CodeAttribute` tag has the name `"ArrayNullCheckAttribute"`, which is the attribute name in the class file.

Soot manages all `CodeAttribute` tags and produces a Jasmin file with the appropriate attribute directives, and finally Soot calls the extended Jasmin assembler to generate the class file with attributes.

3.4 Making a JVM Aware of Attributes

After generating the annotated class file, we need to make a JVM aware of attributes and have it use them to improve its generated native code. We modified both Kaffe's OpenVM 1.0.5 JIT and the IBM HPCJ ahead-of-time compiler to take advantage of the array bound attributes. Below we describe the modifications needed for Kaffe; the modifications to HPCJ are similar.

The KaffeVM JIT reads in class files, verifies them, and produces native code on demand. It uses the `_methods` structure to hold method information. We added a field to the `_methods` structure to hold the array bounds check attribute. Figure 8 shows the data structure.

When the VM reads in the array bounds check attribute of the Code attribute, it allocates memory for the attribute. The <PC, data> pairs are then stored in the attribute table. The pairs were already sorted by PC when written into the class file, so no sorting has to be done now.

The Kaffe JIT uses a large switch statement to generate native code for bytecodes. It goes through the bytecodes sequentially. We use the current PC as the key to look up the array bounds check attribute in the table before generating

```
                                                        typedef struct
typedef struct _methods   typedef struct _soot_attr       _soot_attr_entry
{ ....                    { u2 size;                    { u2 pc;
  soot_attr attrTable;      soot_attr_entry *entries;     u1 attribute;
} methods;                } soot_attr;                  } soot_attr_entry;
```

Fig. 8. Modified Kaffe Internal Structure

code for array references. Because attribute pairs are sorted by ascending PC, and bytecodes are processed sequentially, we can use an index to keep the current entry in the attribute table and use it to find the next entry instead of searching the whole table. Figure 9 gives the pseudocode.

```
idx = 0;
...
case IALOAD:
  ...
  if (attr_table_size > 0) { /* attributes exist. */
    attr = entries[idx].attribute;
    idx++;
    if (attr & 0x03)  /* generates bounds checks. */
       check_array_index(..);
    else
    if (attr & 0x04) /* generates null pointer check. */
        explicit_check_null(..);
  } else             /* normal path */
    check_array_index(..);
```

Fig. 9. Using attributes in KaffeVM

Here, we turn off bounds check instructions when the array reference is non-null and both bounds are safe. We also insert null check instructions at the place where bounds check instructions can be removed but the null check is still needed.

4 Experimental Results

We measured four array-intensive benchmarks to show the effectiveness of the array bounds check attribute. The benchmarks are characterized by their array reference density, and the results of bounds check and null check analysis. These data are runtime measurements from the application programs, and do not include JDK libraries. We also measured the class file size increase due to attributes. Finally we report the performance improvement of benchmarks on attribute-aware versions of KaffeVM and the IBM High Performance Compiler for Java.

4.1 Benchmarks

Table 1(1) shows benchmark characteristics.[1] The third column describes the array type used in the benchmark: `s` represents single dimensional and `m` represents multidimensional.[2] The last column shows array reference density of the benchmark, which is a count of how many array references per second occur in the benchmark. It is a rough estimate of the potential benefit of array bounds check elimination.

Table 1. Benchmarks

(1) Benchmarks				(2) Analysis results					(3) File size increase		
Name	Source	Type	Density	low	up	both	not null	all	Soot -W	with attr.	incr.
mpeg	specJVM98	s/m	19966466/s	89%	51%	50%	58%	26%	256349	276874	8.0%
FFT	scimark2	s	10262085/s	77%	61%	59%	97%	59%	2380	2556	7.4%
LU	scimark2	m	14718027/s	97%	64%	64%	68%	31%	1641	1907	16.2%
SOR	scimark2	m	13683052/s	99%	99%	99%	51%	50%	445	507	13.9%

4.2 Result of Analysis

Table 1(2) shows the percentage of checks that can be eliminated. This is measured by instrumenting the benchmark class file and getting dynamic numbers.

The first and second columns show the percentages of lower and upper bounds checks that could be safely removed. Usually we see that a much higher percentage of lower bounds can be proved safe than upper bounds. The third column shows the percentage of bounds checks where both upper and lower can be safely removed. The forth column shows the percentage of safe null pointer checks, and the fifth column shows the percentage of array references that are not null and with both bounds checks safe. Clearly we are most interested in the last column, when we determine that we can eliminate both bounds and we know the array is not null.

4.3 Class File Increase

The increase in file size due to adding attribute information is shown in Table 1(3).

[1] Note that we use the abbreviation `mpeg` for the benchmark `mpegaudio`.

[2] Multidimensional arrays are harder to analyze due to their more complex aliasing, as well as the possibility of being non-rectangular.

The first column shows the file size in bytes after Soot whole program optimization. The second column shows file size of the optimized class file including array bounds check attributes, and the relative increase in file size is listed in the last column. The file size increase depends primarily on the static count of array references—each array reference needs 3 bytes of attribute information, and the class itself needs a constant pool entry to store the attribute name. Note that in this prototype work we have made no effort to reduce or compress this information; significant improvements should be possible.

4.4 Kaffe Performance Improvement

We measured the KaffeVM (ver 1.0.5 with JIT3 engine) modified to take advantage of array bounds check attributes. It runs on a dual Pentium II 400MHz PC with 384Mb memory, Linux OS kernel 2.2.8, and glibc-2.1.3. The modified JIT compiler generates code for an array reference depending on the attribute. If no attribute is present, it generates bounds check instructions as in Figure 10(a). If the attribute shows safe bounds check and unsafe null check, it inserts null check code in place of the bounds check (Figure 10(b)). If both bounds checks and null check are safe, no instructions are added.

```
cmp reg1, [reg2+off]
jge outofboundserror
```

(a) Array Bounds Check Instructions

```
cmp reg1, 0
je  nullpointerexception
```

(b) Explicit Null Check Instructions

Fig. 10. Check Instructions

Table 2 gives the benchmark results for the attribute-aware KaffeVM. The "nocheck" column shows the running time without any bounds or null checks for the application classes, while the "with attr" and "normal" columns show the execution times of each benchmark with and without attributes respectively. Each benchmark gets some improvement roughly scaled according to the percentage of safe checks. Note that LU without checks actually has a performance degradation; this anomaly is discussed in the next section.

Table 2. KaffeVM Runtime

name	normal	nocheck	with attr
mpeg	80.83s	62.83s(22.3%)	72.57s(10.2%)
FFT	51.44s	48.84s(5.1%)	50.01s(2.8%)
LU	81.10s	81.88s(-0.9%)	78.15s(3.6%)
SOR	46.46s	41.23s(11.3%)	43.19s(7.0%)

4.5 High Performance Compiler Performance Improvement

The High Performance Compiler for Java runs on a Pentium III 500MHz PC with the Windows NT operating system. The structure of the High Performance Compiler is such that safe checks could be removed prior to its internal optimization phase, allowing subsequent optimizations to take advantage of the reduced code; this has resulted in a speed increase that does not correlate as well with the relative number of checks removed. Tables 3 and 4 show the benchmark times with and without internal optimizations—the last two columns in each table give the performance improvement when either just array bounds checks or just null pointer checks are completely removed; note that as with KaffeVM, there are some unexpected performance decreases.

Table 3. IBM High Performance Compiler without Optimizations

name	normal	nocheck	with attr	noarray	nonull
mpeg	50.88s	29.96s(41.1%)	39.14s(23.1%)	30.64s(39.8%)	47.94s(5.8%)
FFT	28.22s	25.09s(11.1%)	26.59s(5.8%)	25.15s(10.9%)	28.85s(-2.2%)
LU	39.99s	28.83s(27.9%)	32.33s(19.2%)	28.92s(27.7%)	38.39s(4.0%)
SOR	24.16s	15.46s(36.0%)	15.55s(35.6%)	15.18s(37.2%)	23.96s(0.8%)

Table 4. IBM High Performance Compiler with Optimizations On

name	normal	nocheck	with attr	noarray	nonull
mpeg	21.27s	15.93s(25.1%)	20.33s(4.4%)	17.12s(19.5%)	20.82s(2.1%)
FFT	17.39s	15.34s(11.8%)	19.45s(-11.8%)	16.08s(7.5%)	18.58s(-6.8%)
LU	21.50s	14.84s(30.8%)	21.27s(1.1%)	15.03s(30.1%)	21.49s(0.0%)
SOR	11.93s	8.88s(25.6%)	8.88s(25.6%)	8.88s(25.6%)	11.92s(0.1%)

Generally with the High Performance Compiler results, we see a very slight improvement in the running time due to null pointer check elimination ("normal" column and "nonull" column respectively), and a significantly larger improvement due to array bounds check elimination ("noarray" column). This reflects the relative cost of the two operations—where possible the High Performance Compiler implements null pointer checks by handling the associated hardware trap if a null pointer is dereferenced. The machine architecture guarantees a hardware exception if any very low memory addresses (e.g. zero) are read or written. Thus, since most null pointer checks are required because of an impending dereference or write anyway, the null pointer check can be implemented as an implicit byproduct of the subsequent code (see Figure 11(a)). The result is that the check itself has relatively little apparent cost. Array bounds checks, alternatively, require an explicit test and branch, and so eliminating them has a noticeable impact on the code being executed.

Surprisingly, the combination of eliminating both kinds of checks together is significantly more effective than the sum of eliminating both individually. This is a consequence of both optimization and the way null pointer checks have been implemented through the hardware exception mechanism. An array element reference in Java needs to be guarded by both a null pointer check and an array bounds check on the array index ([17]). In the High Performance Compiler, the code generated for an array bounds check naturally consists of a load of the array size followed by a comparison of the size with the index in question. The array size field, however, is offset only a small distance in memory from the start of the array object; the hardware supports trapping on a range of low addresses, and so a dereference of the array size field is as effective as dereferencing the object itself at generating the required hardware trap if the original object is null. Subsequent optimizations easily recognize this situation and combine the two checks into the same actual code; the code output for an array load or store is thus often identical whether a null pointer check is performed or not (see Figure 11(a)).

The symmetric situation, eliminating array bounds checks without also eliminating null pointer checks, is also not as effective as one might expect. In order to remove the array bound check while leaving behind the implicit null pointer check, specific code to dereference the array object must be inserted in order to still trigger the hardware trap mechanism if the array object is null (e.g. code in Figure 11(a) is replaced by the code in Figure 11(b)). This means that the benefit of deleting the bounds check code is offset slightly by the code required to explicitly dereference the object as part of a null pointer check.

```
mov eax,[ebx+offset]
     (implicit null ptr check)
cmp eax,edx}
jge outofboundserror
```

(a) Array Bounds Check with Implicit Null Pointer Check.

```
test eax,[eax]
     (explicit null ptr check)
```

(b) Null Pointer Check Inserted if Array Bounds Checks Eliminated.

Fig. 11. HPCJ Check Instructions

It is interesting that anomalous results occur in the FFT runs as well as the LU run of KaffeVM. Here the benchmark runs without some or all runtime checks are actually slower than the versions with checks. Since we are only reducing the checking overhead, and never increasing it, it seems counterintuitive that performance would ever be less than the baseline for any of our runs. However, in certain key benchmark functions the code changes due to eliminating some but not all bounds checks seems to negatively impact instruction cache utilization, and we find our code highly sensitive to the exact sequence of bytes being executed. For instance, if the benchmark is compiled so as to artificially ignore the array bounds attribute information for a specific function (and thus generate normal bounds checks regardless of whether attribute information tells us they are unnecessary), much of the performance degradation is eliminated. The interaction of optimizing compilers with optimizing hardware is clearly a complex issue with many tradeoffs, and we may not be able to benefit all programs equally.

5 Related Work

Work related to this paper falls into three categories: (1) other tools for optimizing class files; (2) related techniques for optimizing array bounds checks; and (3) other uses of class file attributes.

5.1 Class File Tools

The only other Java tool that we are aware of that performs significant optimizations on bytecode and produces new class files is Jax[24]. The main goal of Jax is application compression where, for example, unused methods and fields are removed, and the class hierarchy is compressed. Their system is not focused on low-level optimization and it does not handle attributes.

There are a number of Java tools that provide frameworks for manipulating bytecode: JTrek[14], Joie[5], Bit[16] and JavaClass[12]. These tools are constrained to manipulating Java bytecode in their original form, however. They do not provide convenient intermediate representations such as Baf, Jimple or Grimp for performing analyses or transformations, they do not allow the production of bytecode, nor do they handle attributes.

5.2 Array Bounds Checks

Array bounds check optimization has been performed for other languages for a long time. Value range analysis has been used to remove redundant tests, verify programs, or guide code generation [9]. Further, there are a number of algorithms that use data flow analysis to remove partial redundant bounds checks [8,15].

More recently, the Java language has been the focus of research. Array bounds check elimination has been implemented in JIT compilers [22,4]. Midkiff et. al. [18,19] proposed a Java `Array` package and loop versioning algorithm to overcome the bounds check overhead in Java scientific programs. An algorithm for general applications was presented in [2]. Compared with these intraprocedural algorithms, our algorithm can take advantage of field information and our analysis for finding rectangular arrays.

Field analysis is useful to other optimizations such as object inlining and escape analysis [7]. Knowing an array's shape can also help memory layout of array objects [3].

5.3 Using Attributes

To the best of our knowledge there has been relatively little work done in investigating the possible uses of class file attributes to improve performance. We are aware of only two research groups that have been investigating this topic and both are focused on performance of bytecode.

Hummel et. al. gave an initial study on using attributes to improve performance, where they showed, using hand-simulation, that performance could be improved using attributes [10]. More recently this group has presented a system built on `guavac` and `kaffe` [1]. Their modified `guavac` compiler translates from Java source code to their own intermediate representation, and then converts this intermediate representation to annotated bytecode, which is then executed using their modified `kaffe` JIT. They have concentrated on conveying register allocation information. 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 second group, Jones and Kamin, have also focused on register allocation [13]. Their approach monotypes each virtual register, which allows for efficient runtime verifiability of their attributes; attributes to deal with spills are also presented. Their experimental results also exhibit significant code speedups.

Compared to these groups, our work is more focused on providing a general purpose framework for producing attributed class files. The input to our system can be class files produced by any compiler, and we provide the necessary infrastructure to convert the class files to typed 3-address intermediate code and to perform analysis on this intermediate code. We also provide a simple mechanism for converting the resulting flow analysis information to class file attributes. In this paper we have demonstrated how this scheme can be applied to array bounds checks; however, it can easily be applied to other problems, including the register allocation problem.

6 Conclusions and Future Work

In this paper we have presented an approach to using class file attributes to speed up the execution of Java bytecode. Our approach has been designed and implemented as part of the Soot bytecode optimization framework, a system that takes as input any Java bytecode and produces optimized and attributed bytecode as output. In our system, the compiler writer develops the appropriate flow analysis for the Jimple typed 3-address representation, and then uses tags to attach attribute information to the appropriate hosts (statements, methods, classes or fields). If the tags are attached to statements (`Units`), the compiler writer can also specify a tag aggregator which combines all tags within a method. The Soot system applies this aggregator when producing the attributed class files. As part of our system we have produced an extended version of the Jasmin assembler which can now support attribute directives and produces attributed class files.

Our system is very easy to use, and we showed how to apply it to the problem of eliminating array bound checks. We provided an overview of our array bounds check analysis and we showed how the results of the analysis can be propagated to attributes. We also modified two virtual machines, the Kaffe JIT and the IBM HPCJ ahead-of-time compiler, to take advantage of the attributes, and we presented experimental results to show significant performance improvements due to the array bounds check attributes.

In our current approach our attributes are not necessarily verifiable. Given verifiable class files and a correct analysis, we guarantee to produce correct attributes. Thus, if our system is used as a front-end to an ahead-of-time compiler, there is no problem. However, if our attributed class files are transmitted from our system to an external VM via an insecure link, we need a safety-checking mechanism to ensure the attribute safety. Necula's proof-carrying code[20] could be one solution to this problem.

Based on our work so far, we believe that the attributes supported by our system can be used for a wide variety of tasks as outlined in the introduction, and we plan to work on several of these in the near future. In particular, we wish to examine how escape analysis and side-effect information can be expressed as attributes, and we plan to examine how some of the profile-based attributes can be used. We also would like to see how attributes can be used with other virtual machine implementations.

Acknowledgements. The authors would like to thank IBM CAS for supporting Raja Vallée-Rai with a IBM CAS Fellowship. This work was also supported by NSERC and FCAR. We would also like to thank Patrick Lam and Richard Godard for their work in implementing the first versions of null pointer analysis, which was the foundation for our null pointer attributes.

References

1. Ana Azevedo, Joe Hummel, and Alex Nicolau. Java annotation-aware just-in-time (AJIT) compilation system. In *Proceedings of the ACM 1999 Conference on Java Grande*, pages 142–151, June 1999.
2. R. Bodik, R. Gupta, and V. Sarkar. ABCD: Eliminating Array Bounds Checks on Demand. In *Proceedings of PLDI '00*, pages 321–333, June 2000.
3. M. Cierniak and W. Li. Optimizing Java bytecodes. *Concurrency, Practice and Experience*, 9(6):427–444, 1997.
4. Michal Cierniak, Guei-Yuan Lueh, and James M. Stichnoth. Practicing JUDO: Java under Dynamic Optimizations. In *Proceedings of PLDI '00*, pages 13–26, June 2000.
5. Geoff A. Cohen, Jeffrey S. Chase, and David L. Kaminsky. Automatic program transformation with JOIE. In *Proceedings of the USENIX 1998 Annual Technical Conference*, pages 167–178, Berkeley, USA, June 15–19 1998. USENIX Association.
6. Etienne M. Gagnon, Laurie J. Hendren, and Guillaume Marceau. Efficient inference of static types for Java bytecode. In *Proceedings of SAS 2000*, volume 1824 of *LNCS*, pages 199–219, June 2000.

7. S. Ghemawat, K.H. Randall, and D.J. Scales. Field Analysis: Getting Useful and Low-Cost Interprocedural Information. In *Proceedings of PLDI '00*, pages 334–344, June 2000.
8. R. Gupta. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems, 2(1-4):135–150*, 1993.
9. W. Harrison. Compiler analysis of the value ranges of variables. *IEEE Transactions on Software Engineering, 3(3):243–250*, 1977.
10. Joseph Hummel, Ana Azevedo, David Kolson, and Alexandru Nicolau. Annotating the Java bytecodes in support of optimization. *Concurrency: Practice and Experience*, 9(11):1003–1016, November 1997.
11. The Jasmin Bytecode Assembler. http://mrl.nyu.edu/meyer/jvm/jasmin.html.
12. JavaClass. http://www.inf.fu-berlin.de/ dahm/JavaClass/.
13. Joel Jones and Samuel Kamin. Annotating Java class files with virtual registers for performance. *Concurrency: Practice and Experience*, 12(6):389–406, 2000.
14. Compaq-JTrek. http://www.digital.com/java/download/jtrek.
15. Priyadarshan Kolte and Michael Wolfe. Elimination of redundant array subscript range checks. In *Proceedings of PLDI '95*, pages 270–278, 1995.
16. Han Bok Lee and Benjamin G. Zorn. A tool for instrumenting Java bytecodes. In *The USENIX Symposium on Internet Technologies and Systems*, pages 73–82, 1997.
17. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, Reading, MA, USA, second edition, 1999.
18. S. Midkiff, J. Moreira, and M. Snir. Optimizing bounds checking in Java programs. *IBM Systems Journal*, 37(3):409–453, August 1998.
19. J.E. Moreira, S.P. Midkiff, and M. Gupta. A Standard Java Array Package for Technical Computing. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, TX, March 1999.
20. G. Necula. Proof-carrying code. In *Proceedings of POPL '97*, pages 106–119, January 1997.
21. Soot - a Java Optimization Framework. http://www.sable.mcgill.ca/soot/.
22. T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-in-Time Compiler. *IBM Systems Journal*, 39(1):175–193, 2000.
23. Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, and Étienne Gagnon. Practical virtual method call resolution for Java. In *Proceedings OOPSLA 2000*, pages 264–280, October 2000.
24. Frank Tip, Chris Laffra, Peter F. Sweeney, and David Streeter. Practical experience with an application extractor for Java. In *Proceedings OOPSLA '99*, pages 292–305, October 1999.
25. Raja Vallee-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java Bytecode Optimization Framework. In *Proceedings of CASCON '99*, pages 125–135, 1999.
26. Raja Vallée-Rai, Etienne Gagnon, Laurie Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Proceedings of CC '00*, volume 1781 of *LNCS*, pages 18–34, March 2000.