Supporting Binary Compatibility with Static Compilation*

Dachuan Yu Zhong Shao Valery Trifonov
Department of Computer Science, Yale University
New Haven, CT 06520-8285, U.S.A.
{yu, shao, trifonov}@cs.yale.edu

Abstract

There is an ongoing debate in the Java community on whether statically compiled implementations can meet the Java specification on dynamic features such as binary compatibility. Static compilation is sometimes desirable because it provides better code optimization, smaller memory footprint, more robustness, and better intellectual property protection. Unfortunately, none of the existing static Java compilers support binary compatibility, because it incurs unacceptable performance overhead. In this paper, we propose a simple yet effective solution which handles all of the binary-compatibility cases specified by the Java Language Specification. Our experimental results using an implementation in the GNU Java compiler shows that the performance penalty is on average less than 2%. Besides solving the problem for static compilers, it is also possible to use this technique in JIT compilers to achieve an optimal balance point between static and dynamic compilation.

1 Introduction

Modern software applications are often built up by combining many components. Some of these components are shared libraries which allow multiple applications to share large amounts of system software.

Shared libraries evolve over time so that new functionality can be added, bugs can be fixed, algorithms and efficiency can be improved, and deprecated functions can be removed. Evolving or modifying these libraries can affect applications that depend on them, thus library evolution may cause compatibility problems.

However, it is usually undesirable to recompile a whole application just to accommodate the changes in a single component. In the case of widely distributed libraries, used by many unknown applications, it is often impractical or impossible to recompile even only the importing units. A popular current approach is to try to guarantee that binaries can be directly replaced by compatible binaries without compromising a working system.

Binary compatibility is a concept introduced to address this problem. It was initially referred to as *release-to-release binary compatibility* [11], and later defined in the Java Language Specification (JLS) [12], which describes the changes that developers are permitted to make to a package or to a class or interface type while preserving compatibility with existing binaries. Thus the Java binary compatibility prescribes conditions under which modification and recompilation of classes do not necessitate recompilation of other classes depending on them.

In the Java Virtual Machine [20], support for binary compatibility is primarily due to the use of symbolic references to look up fields and methods at run-time. However, in some cases a native compiler for Java is needed that compiles Java (or bytecode) programs directly into native code in the same manner as compilers for C/C++. This ahead-of-time compilation is desirable because it yields better optimized code, more robust deployed applications, and offers better intellectual property protection [3, 5, 7]. We will elaborate on this later.

Nevertheless, supporting binary compatibility with ahead-of-time compilation is a hard problem because of the seemingly contradictory requirements. When certain changes are allowed due to binary compatibility, the contents of a class cannot be completely determined until the class is loaded. However, ahead-of-time compilers usually generate hard-coded offsets based on the layout information of other classes at compile time.

A well-known problem is that the standard compilation techniques for virtual methods in object-oriented languages preclude binary compatibility (cf. the fragile base class problem [13, 26]). For example, the documen-

*This work is supported in part by DARPA OASIS grant F30602-99-1-0519, NSF grant CCR-9901011, and NSF ITR grant CCR-0081590. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

tation on binary compatibility [30] in the EPOC C++ System says:

> ... virtual member functions are for life—you can't add or remove virtual member functions, or change the virtuality of a function, or change their declaration order, or even override an existing function that was previously inherited, ...

For compliance with the binary-compatibility requirements of Java some existing native compilers solve this problem by generating (at least) some of the code at run time, which unavoidably negates some of the benefits of pre-compilation. Other existing native compilers simply have no support for binary compatibility, because the obvious solutions (e.g. method lookup by name at run time) seem to incur high performance overhead.

This paper presents a simple yet effective solution using static compilation, which meets all Java binary compatibility requirements with little performance penalty. The contributions are:

- In our solution, the compilation is fully static, which allows the compiler to take advantage of the well-developed static compilation techniques for better code optimization.
- Our solution covers all the cases specified in the JLS. Different features—including methods, fields, interfaces, and modifiers—are supported by the same set of simple core techniques.
- Our solution also detects all binary-incompatible changes and gracefully raise proper exceptions at load or run time.
- Our solution is efficient. We describe an implementation in the GNU Java compiler (GCJ). The performance test shows that the performance penalty of our new technique is on average less than 2%.

In the remainder of this introduction, we briefly describe the benefits of static compilation.

1.1 Why Static Compilation?

Two popular approaches for compiling Java programs are Just-In-Time (JIT) compilation (e.g. Sun Hotspot [29], Cacao [18], OpenJIT [24], shuJIT [28], vanilla Jalapeno [1]) and static compilation (e.g. BulletTrain [22], Excelsior JET [10], GCJ [32], IBM VisualAge for Java [14], JOVE [15]). It would be wrong to say one approach is definitely better than the other, since they are suited for different situations [7]. In fact, current research on "quasi-static compilation" [27] shows that combining these two may yield excellent results.

In practice, static Java compilers are sometimes desirable over JIT compilers because they have many advantages [3, 5, 7]:

- Static compilation yields more robust deployed applications. On the one hand, a deployment JIT may be different from the development JIT, which can cause problems due to even slight differences in the virtual machine or library code. With static compilation, programs are compiled into native code allowing the developer to test exactly what is deployed. On the other hand, compilers have bugs. Crashes caused by static compiler bugs sometimes happen at compile time (unless the bug is the kind that generates bad code silently), while bugs in the JIT may cause crashes at program execution time, and some of them may only surface after a portion of the program has been executed many times. Moreover, if the program crashes due to a bug in either the compiler or the program itself, statically compiled code is much easier to debug because the run-time trace is more predictable.
- Static compilation provides better intellectual property protection. Native code is much harder to reverse-engineer than Java bytecode.
- Static Java compilers can perform resource intensive optimization before the execution of the program. In contrast, JIT compilers must perform analysis at execution time, thus are limited to simple optimizations that can be done without a large impact on the combined compile and execute time.
- Static compilation achieves greatly reduced start-up cost, reduced memory usage, automatic sharing of code by the OS between applications, and easier linking with native code.
- Last but not least, static compilation is better suited for code certification than JIT compilation. It is significantly easier to achieve higher safety assurance by removing the compiler from the trusted computing base. There has been a lot of work done in this area [23, 21, 19] which mostly focuses on static compilation.

Regardless of the above advantages, there is an ongoing debate in the Java community on whether statically compiled implementations can meet the Java specification on dynamic features such as binary compatibility. Our paper presents a scheme that accommodates the seemingly contradictory goal of full Java compliance and static compilation, thus showing that binary compatibility can indeed be supported using static compilers. Following the inspiration of "quasi-static compilation" [27], this technique in practice can also be used together with other JIT compilation techniques to achieve an optimal balance point between static and dynamic compilation. Thus we believe this result is of interest to the general audience in the JVM community.

2 Background

Java binary compatibility requires that changes in certain aspects of a class `C` from version to version must not entail the recompilation of other classes that are clients of `C`. (Client classes of `C` are those that reference `C` in some way, such as by accessing members of objects of `C`, or by extending `C`.) For example, changing the order of methods and fields of a class or adding methods and fields to a class must not force recompilation of its clients.

Java virtual machines allow these kinds of changes to occur between releases of a class because references from one class to the fields and methods of other classes are made by symbolic names embedded in the class file. These references are transformed into addresses and offsets during the process of resolution.

However, static compilers that do not consider binary compatibility usually generate these offsets hard-coded, ahead of (link) time. This implies that changes to a class that affect the layout of fields and methods in the class could require all of its clients to be recompiled, since they contain hard-coded addresses and offsets based on the old layout. Failure to recompile all clients of a modified class can result in unexpected run-time behavior.

Current run-time compilers for Java have encountered similar problems. Taking the virtual method invocation as an example, binary compatibility is usually accomplished using run-time compilation techniques: Just-in-time compilers generate code for classes at run-time. During the run-time compilation, a virtual method invocation on an object of a loaded class can be safely compiled based on the determined *vtable* (virtual method table) of that class. However, a virtual method invocation on an object of a class which is not loaded yet cannot be handled in the same manner. In this case, the compiler emits special code which "stitches" the actual method invocation code lazily when it is executed.

For optimization purposes, JIT compilers often use guarded inlining (where the guard checks for the object type at run-time) to handle the scenario where the inlining is invalidated by further class loading. When such a scenario occurs, run-time compilation has to be performed. There has been also work on techniques for inlining virtual methods more efficiently [6, 16, 25].

Typically, static compilers use global or whole-program analysis [4] to do inlining or devirtualization. However, without the ability to perform compilation at run-time, they assume that no changes will be made to classes referred to by the compiled code. Hence, they do not comply with the JLS. The trade-off between binary compatibility (for full compliance with the JLS) and cross-class inlining is obviously an issue for static Java compilers. While our solution for binary compatibility does not directly support cross-class inlining, in cases when dynamic compilation may not be desirable due to various reasons discussed in section 1.1, an implementation would employ our solution together with other schemes that support inlining (but not binary compatibility) to achieve optimal results. For instance with quasi-static compilation [27] both pre-compiled native code and bytecode of classes are shipped together. When binary changes invalidate the pre-compiled native code, the VM falls back to compiling or even interpreting the bytecode. Similarly, a version of native code compiled with cross-class inlining can be shipped together with native code compiled using our approach. In the common case, when no changes to other classes are made, the inlined version of native code would be used for maximum efficiency. In cases when binary changes are detected, the system would fall back to running the version compiled without inlining but with support for binary compatibility, thus avoiding run-time compilation.

In summary, run-time compilers support binary compatibility with various run-time compilation techniques. However none of the existing static Java compilers provide support for binary compatibility, primarily because the high overhead negates much of the advantages of static compilation.

3 Static Compilation vs Binary Compatibility

In this section, we give examples to illustrate the concept of Java binary compatibility. We also show how the naïve application of the standard vtable approach for static compilation fails.

Consider the following program. Class `Programmer` defines two virtual methods, `eat` and `hack`. Class `JavaProgrammer` extends class `Programmer`, overrides those two methods, and defines a new virtual method `study`. The `main` method of class `Manager` creates instances of both the above classes and does some virtual method calls. Note that at run-time the variable `Whoami` contains an object of class `JavaProgrammer`, although its static class is `Programmer`.

```
public class Programmer {
  void eat  () { ... };
  void hack () { ... };
}
public class JavaProgrammer
     extends Programmer{
  void eat  () { ... };
  void hack () { ... };
  void study() { ... };
}
public class Manager {
  public static void main (String args[]) {
    // some code that runs for days...
    ...
    Programmer Tom = new Programmer();
    JavaProgrammer Jerry = new JavaProgrammer();
    Programmer Whoami = new JavaProgrammer();
    ...
    Tom.eat();
    Tom.hack();
    Jerry.eat();
    Jerry.hack();
    Jerry.study();
    ...
}}
```

The standard technique used in object-oriented programming language implementations supports virtual method dispatch by collecting the virtual methods of a class in a record called a *vtable*, and providing a pointer to this record in each object of the class. When the three classes above are compiled, the layout of the vtables of classes `Programmer` and `JavaProgrammer` is determined statically (Figure 1), and the code in class `Manager` is compiled to invoke virtual methods by accessing the corresponding entries in the vtables of these classes, reachable through the respective objects. Since the variable `Whoami`, declared of class `Programmer`, can be bound to an object of class `JavaProgrammer`, the layout of the vtable of `JavaProgrammer` must be consistent with that of `Programmer`, so that virtual method invocations can be compiled to use the same offset in the vtable for a given method of `Programmer`, regardless of the dynamic class of the object.

However, this vtable approach cannot be directly applied if we want to support binary compatibility. When

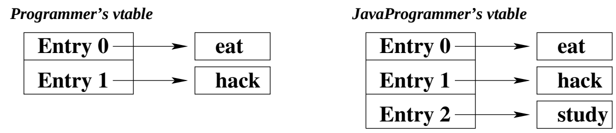

Figure 1: The vtables.

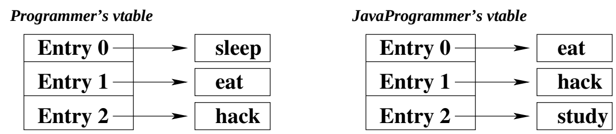

Figure 2: Scenario A: adding a method.

changes are made to the binary of a class, the locations of method pointers in the vtable may change, which invalidates the offset information used to compile other classes. Even worse, vtables reachable through objects of the same static class may now have different layout, as illustrated in the following subsections.

3.1 Scenario A: Adding a Method

Here we make a binary-compatible change to class `Programmer` by adding a new method `sleep` at the very beginning.

```
public class Programmer {
  void sleep() { ... }; // New Method!
  void eat  () { ... };
  void hack () { ... };
}
```

Now we recompile class `Programmer` only. The vtables for `Programmer` and `JavaProgrammer` are shown in Figure 2. The vtable layout of class `Programmer` has changed, and it is no longer consistent with the vtable layout of class `JavaProgrammer`. When these classes are loaded and `Manager.main` invoked, the code for `Tom.eat` will access the wrong entry in the vtable and end up calling method `sleep`. A similar problem occurs with `Tom.hack`. This is exactly the behavior shown by the current GCJ, which uses the standard vtable approach, and thus does not support binary compatibility.

Note that even if we had added the method `sleep` at the end of class `Programmer`, the problem still exists, because when we recompile class `Programmer`, method `sleep` will use entry 2 of the vtable. However the entry 2 of the vtable of class `JavaProgrammer` is already occupied by method `study`, and the invocation `Jerry.study` of `Manager` was compiled based on this.

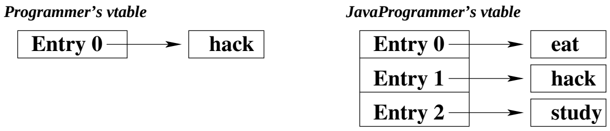

Figure 3: Scenario B: removing a method.

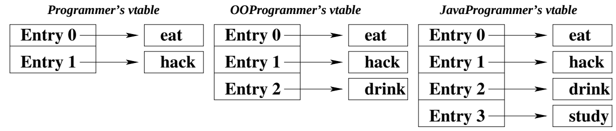

Figure 4: Scenario C: changing class hierarchy.

The observation here is that the vtable layout may change due to changes in the class. So we really should not have made any assumptions about the vtable layouts of `Programmer` and `JavaProgrammer` in `Manager`. Moreover, the information available at compile time is not sufficient for building the vtables, since classes in the same hierarchy may change, yet we still need to maintain consistency between a subclass and its superclass.

3.2 Scenario B: Removing a Method

Some source code modifications, such as removing a method from a class, are binary incompatible changes in the sense that other programs which work fine with the old binary may cease to function when linked with the evolved new binary due to the removal of the method. However, the safety of modern software systems demands that under no circumstances may an application crash. The JLS requires that, under the incompatible change in which a method is removed, the program should still run as long as the missing method is not used, and that an exception should be raised if code tries to invoke the missing method at run time.

Consider what happens when we remove the method `eat` from class `Programmer` in the original program and recompile it. The vtables for `Programmer` and `JavaProgrammer` are shown in Figure 3. Obviously, the vtable layout of class `Programmer` has changed, and it is no longer consistent with the vtable layout of class `JavaProgrammer`.

```
public class Programmer {
  // void eat  () { ... }; // Removed!
  void hack () { ... };
}
```

In this case, the correct behavior of a virtual method invocation `obj.eat` in any old binary depends on the static class of the object `obj`. If the static class of `obj` is `JavaProgrammer`, the method invocation works fine, as if no change had been made. However, if the static class of `obj` is `Programmer`, a `NoSuchMethodError` exception should be thrown when the method is invoked, even if `obj` actually contains an object of class `JavaProgrammer` which defines method `eat`.

The standard vtable approach fails in this case as well. The invocation `Tom.eat` in `Manager` will call the wrong method `hack`, while `Tom.hack` will have implementation-dependent results, since it uses a pointer located outside of the actual vtable.

The observation here is that we need to gracefully handle incompatible changes by raising exceptions at run time. Of course, we still need to keep in mind the consistency of vtables.

3.3 Scenario C: Binary Change at Run Time

Some static compilers (e.g. BulletTrain) perform dependency analysis before executing a Java program, and attempt to recompile if inconsistency is detected. In the cases of scenarios A and B, these compilers would have refused to run `Manager`, or attempted to recompile it automatically. The problem is that this behavior is not only non-compliant with the binary compatibility requirements of the JLS (which intends to solve these issues without recompilation of the client classes of the changed class), but also that this dependency analysis cannot always be done statically.

A simple example which presents a challenge to the static dependency analysis scheme is reflection. Using reflection, a program can load arbitrary class files which are not known at compile time. This means that the static analysis may not work out all the dependencies. Even if reflection is not a concern, binary changes may occur at run time after some classes are already loaded and executed. In these cases, recompilation is not possible. Here we use another binary-compatible change, namely the insertion of a new class into the class hierarchy, to demonstrate the problem.

```
public class OOProgrammer
     extends Programmer { // New Class!
  void drink() { ... };
}
public class JavaProgrammer
     extends OOProgrammer{ // New Hierarchy!
  void eat  () { ... };
  void hack () { ... };
  void study() { ... };
}
```

Based on the original program, before we make any changes, suppose class `Manager` (but not `JavaProgrammer`) is already loaded and being executed. During the execution of the first line of `main`, we insert a new class `OOProgrammer` between `Programmer` and `JavaProgrammer`. We compile `OOProgramer` and recompile `JavaProgrammer`. The vtables are shown in Figure 4. Clearly, the vtable layout of `JavaProgrammer` has changed. The line for `Jerry.study` in `Manager` is going to call method `drink` which happens to reside in the entry 2 of `Jerry`'s vtable after the change.

The lesson is, binary changes may occur after some classes are loaded. Static dependency analysis and recompilation are sometimes not only undesirable, but unaffordable. Reflection is an additional complication.

4 Our Approach

In this section we present our solution for static compilers to support Java binary compatibility. Table 1 and Table 2 summarize all the binary changes to classes and interfaces specified in Chapter 13 of the Java language specification [12]. Compatible changes are marked with "+" and incompatible changes are marked with "−". We present the solutions for these changes in the coming subsections. In these tables, SEC x means the solution is presented in Section x. The symbol $\surd$ is only used for binary compatible changes. It means that the solution is trivial and requires no change to the existing implementation. The numbers associated with each binary change are used for cross-referencing in later sections.

4.1 Virtual Methods

The major difficulty in supporting Java binary compatibility is the handling of virtual methods. In this section, we present our idea in a simplified setting where only virtual methods under single inheritance are considered. Temporarily putting aside other features (e.g. modifiers) makes it easier to understand our solution. Extending this solution to work for static methods and constructors is trivial. All the other language features can be supported with simple extensions which we present later.

4.1.1 Idea

From the lessons we learned in Section 3, we know that even though we want to compile classes ahead of time, we cannot afford to build the vtables statically. The information we get during the ahead-of-time compilation

No.	+/−	Binary Change to Class	Solution
1	+	adding(overriding) method/constructor (without modifier change)	SEC 4.1
2	+	changing hierarchy preserving super(s)	SEC 4.1
3	+	adding field (without modifier change)	SEC 4.2
4	+	abstract → nonabstract	$\surd$
5	+	final → nonfinal	$\surd$
6	+	nonpublic → public	$\surd$
7	+	allowing more access to member	$\surd$
8	+	final field → nonfinal field	$\surd$
9	+	adding/deleting transient modifier	$\surd$
10	+	changing formal parameter name of method/constructor	$\surd$
11	+	abstract method → nonabstract	$\surd$
12	+	final method → nonfinal; nonfinal static meth → final static	$\surd$
13	+	changing synchronized modifier	$\surd$
14	+	changing throws clause	$\surd$
15	+	changing method/constructor body	$\surd$
16	+	adding method/constructor that overloads existing one	$\surd$
17	+	changing static initializer	$\surd$
18	−	removing method/constructor	SEC 4.1
19	−	removing field	SEC 4.2
20	−	changing hierarchy without preserving super(s)	SEC 4.4.1
21	−	noncircular hierarchy → circular hierarchy	SEC 4.4.1
22	−	nonfinal → final	SEC 4.4.1
23	−	nonabstract → abstract	SEC 4.4.1
24	−	nonfinal virtual method → final virtual	SEC 4.4.2
25	−	restricting access to member	SEC 4.4.2
26	−	nonfinal field → final field	SEC 4.4.2
27	−	static ↔ instance member	SEC 4.4.2
28	−	nonabstract method → abstract	SEC 4.4.2
29	−	public → nonpublic	SEC 4.4.2
30	+/−	adding(overriding) method (with modifier change)	SEC 4.4.3
31	+/−	adding field (with modifier change)	SEC 4.4.3
32	+/−	changing signature	SEC 4.4.3
33	+/−	about native methods	SEC 4.4.3

Table 1: Java binary compatibility summary: classes.

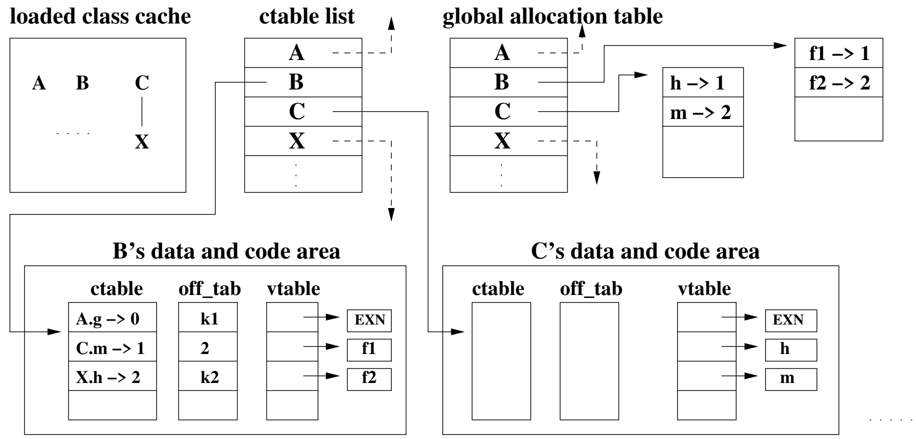

Figure 5: Our solution.

No.	+/−	Binary Change to Interface	Solution
34	+	changing hierarchy preserving super(s)	SEC 4.3
35	+	nonpublic → public	√
36	+	adding/deleting transient modifier	√
37	+	changing formal parameter name of method	√
38	+	changing synchronized modifier	√
39	+	adding method that overloads existing one	√
40	−	removing member	SEC 4.3
41	−	changing hierarchy without preserving super(s)	SEC 4.4.1
42	−	public → nonpublic	SEC 4.4.2
43	+/−	adding field	SEC 4.2 SEC 4.4.3
44	+/−	adding method	SEC 4.3 SEC 4.4.3
45	+/−	changing signature	SEC 4.4.3

Table 2: Java binary compatibility summary: interfaces.

is not sufficient to determine the vtable layout. Besides, we need to handle our compilation carefully so that we can detect binary-incompatible changes and emit error messages gracefully.

We solve this problem by building vtables during class loading. Once loaded, a class is considered fixed. Further changes to this class can be ignored, according to the JLS. Thus we can safely determine the layout of the vtables.

A minor complication is that vtable layouts have to be consistent between a superclass and a subclass. In other words, a method m is located at the same position in the vtable of a subclass as in the vtable of a superclass. Luckily, the loading of a superclass precedes the loading of a subclass, which makes it possible to construct the vtable of the subclass based on the vtable layout of the superclass. In our solution, we maintain this consistency with the help of a global allocation table which reflects the layout of the vtables of all the loaded classes. During the loading of a class, we check the global allocation table to learn the vtable layout of the superclass. Then we follow the layout of the superclass and construct the class's vtable by appending fresh entries at the end. We also record the newly determined layout in the global allocation table so that any subclasses can access it.

The problem now is how to statically compile a virtual method invocation when the vtable layout is not determined statically. We handle this by introducing an extra level of indirection by compiling virtual method invocations to fetch an offset table entry before accessing the vtable. The offset table maps a virtual method to the offset of the method in the vtable. Its entries are filled in at run time when the corresponding class is loaded.

The idea of our approach is shown in Figure 5. To enable this approach, we need to make changes to both the compiler and the class loader.

Compiler Every class is statically compiled to contain a customizing table (ctable) and an offset table (off_tab). The size of the ctable is proportional to the number of distinct external method invocations in the class. For each distinct external method referenced in the code of the class, there is a corresponding entry in the ctable. In a class C, if an external method f is invoked on both an object of a class A and an object of its subclass B, then both $A.f$ and $B.f$ will appear in C's ctable. A ctable entry maps an external method to a unique natural number. This natural number is the offset of the entry for the external method in the offset table. The offset table entries are filled in incrementally at run time according to the information in the global allocation table. A virtual method invocation is compiled to go through the corresponding offset table entry before accessing the vtable.

Class loader The class loader has to maintain the global allocation table, the offset tables, and the vtables during class loading. When a class X is loaded, the class loader constructs the vtable for X based on the vtable layout of the superclass of X, which is specified in the global allocation table. Here we reserve the entry 0 to point to some special exception code. Once the vtable is constructed, the class loader registers the vtable layout in the global allocation table. This information is also propagated to the offset tables of the loaded client classes of X. In this step it is possible that the offset table of a certain class A contains an entry for a method m of class X, while m does not actually exist in this newly loaded class X. This means that there must have been some binary incompatible changes (e.g. m was removed from X, while A was compiled with an old version of X). In this case, the class loader puts the special offset 0 in the corresponding offset table entry. The entry 0 of a vtable always points to some special code that would raise proper exceptions when the method m is invoked.

Example Consider what happens at run time when executing a virtual method invocation `o.m`, where `o` is of static class `C`. Suppose this method invocation appears in the body of class `B`. The statically determined ctable of class `B` designates an offset k for the method `m` of class `C`. In our scheme, `o.m` is compiled to access the entry k of class `B`'s offset table for a new offset k'. This new offset k' is copied from the global allocation table during class loading. It is the actual offset of the method `m` in the vtable of class `C`. Although the dynamic class of object `o` could be a subclass of `C`, it is safe to use the offset k' to access the vtable of object `o` for invoking the method, because we have arranged the vtables of a superclass and its subclasses to be consistent.

Correctness The correctness of our solution for virtual methods is based on the following observations: the global allocation table provides a correct view of all the vtables of the loaded classes; the vtable of a subclass is consistent with the vtable of its superclass; all offset tables are consistent with the global allocation table; and a virtual method invocation cannot be executed before the class of the receiver object is loaded. We refer interested readers to our internal report [31] for a formal development and its soundness proof.

4.2 Fields

The support for fields can be separated into three categories: support for private (or even protected) fields, support for non-private fields, and support for various kinds of access modifiers.

Private fields can only be referenced from within the defining class. Thus they do not require any special care for binary compatibility. Protected fields can only be referenced from the defining class or a subclass. In this case, the loading of the referencing class cannot precede the loading of the defining class. We can completely patch the references to these fields during class loading because by that time the layout of these fields is already determined. By doing this, we do not have to go through the extra indirection of the offset tables.

Changing non-private fields may affect other classes that depend on them. A similar technique as we used for methods can be used here, though it may be relatively less efficient. However, it is generally good software engineering practice to limit the use of non-private fields. Using non-private fields is also discouraged in the Binary Compatibility chapter of the JLS; to quote from Section 13.4.7 of JLS [12], “Widely distributed programs should not expose any fields to their clients.” In fact, non-private fields (especially as part of public APIs) seem to be quite rare. To the authors' knowledge, they are almost non-existent in the standard Java libraries. We believe that the inefficiency here will not have much impact in practice.

Nevertheless, the handling of removed fields is tricky. Unlike calling a method, the trick with reserving the 0-offset entry will not work in this case because accessing it as a field will not raise any exceptions. Using a run-time check for every field access to determine whether the offset for a field is valid has too great a cost in performance. Our solution is, instead of detecting missing fields lazily, to raise exceptions at class loading time when trying to fill in an offset table entry for a field, if the corresponding information is not in the global allocation table. Note that this solution does not obey the JLS on the particular aspect that exceptions of a missing field should be raised lazily. For full compliance with the JLS, one possible solution is to fill in the offset table entry of the missing field with some special offset which triggers an OS trap when accessed. A similar technique is introduced by Joisha *et al.* [17] for the IBM Quicksilver quasi-static compiler [27] for a different purpose, namely to trigger the "stitching" (or linking) operations.

More surprisingly, adding a field is not always a compatible change if changes of modifiers are involved. We discuss this peculiarity in Section 4.4.3 together with other modifier changes.

4.3 Interfaces

The common practice in supporting Java interfaces is to use interface tables (itables); we refer the reader to the work of Alpern *et al.* [2] for a discussion of the prior implementation techniques for interface dispatch and an efficient implementation of Java interfaces. For each interface that a class implements, there is a corresponding itable which contains all the methods declared in the interface. At run time, an interface method invocation would involve looking up the itable by interface name, fetching the address of the interface method from a fixed offset, determined at compile time, and invoking it. The itable look-up mechanism provides natural support for binary compatibility; however, if the method layout of an itable may change, it would be wrong to use a fixed offset to access an interface method.

Fortunately, this is exactly the same problem that we solved for virtual methods and vtable dispatch. All we have to do is make sure interface method invocations go through the offset table, and fill in the offset table incrementally at class loading time once the itable layout is determined.

4.4 Other Changes

All the other binary changes specified by the JLS can be supported by making simple extensions to the techniques discussed so far. They happen to all be incompatible changes, and fall into the following categories. (The numbers at the beginning of the bullets are used for cross-referencing with the entries in Table 1 and Table 2.)

4.4.1 Checking constraints

The incompatible changes in this category are handled by maintaining constraints either explicitly or implicitly, and checking them against the loaded classes during class loading. When any of the constraints are violated, exceptions are raised (`VerifyError`, `ClassCircularityError`, `InstantiationError`, etc).

- (20,41) When compiling a class, we add a constraint for every upward cast indicating the expected inheritance relationship. During execution, we have the system maintain all the constraints specified by the currently loaded classes. When a class is loaded, we check its constraints against the loaded class hierarchy. We also check the newly loaded class against the constraints maintained by the system.
- (21) If the class hierarchy becomes circular due to incompatible changes, we can detect it during class loading.
- (22) If a class Sub inherits a nonfinal class $Super$, and $Super$ is changed to be final, we can detect it during the loading of class Sub.
- (23) Abstract classes cannot be used to create instances. Similar to what we did for upward casts, we add a constraint for every instance creation indicating that the class being instantiated cannot be an abstract class. These constraints are checked during class loading.

4.4.2 Tagging and exception handling

Most modifier-incompatible changes can be handled by tagging the global allocation table entries with modifiers (e.g. access control, readable/writable, instance/static, etc.). In the offset tables, the entries are tagged with the expected modifiers, too. During class loading, the class loader decides whether the modifiers are compatible, and fills in an offset table entry with the registered offset only if they are.

While we could use offset 0 for all kinds of error handling, it is usually preferred to raise different exceptions on different incompatible changes. To achieve this, we can reserve more entries in the vtables and other data structures (e.g. itables) for various exception code. If a certain access is denied according to the global allocation table, the offset of the corresponding exception entry is used.

- (24) Final virtual methods cannot be overridden. During class loading, a subclass studies the vtable layout of its superclass. A tag in the global allocation table indicates whether a virtual method is final. If a final virtual method is overridden, `VerifyError` exception is raised immediately.
- (25) If a binary change restricts access to a member, the tag in the corresponding global allocation

table entry can be used to decide whether an access is granted. If an access attempt to a field is denied, `IllegalAccessError` exception is immediately raised. If an access attempt to a method or constructor is denied, the offset of the exception entry is used.

- (26) If a field that was not final is changed to be final, then it can break compatibility with pre-existing binaries that attempt to assign new values to the field. Our solution is to tag the final field with read-only access in the global allocation table. If some offset table is expecting the field to be tagged with writable access, `IllegalAccessError` exception is raised.

- (27) Changing the `static` modifier of members could raise exceptions when the member is accessed. Static members and instance members are tagged differently in the global allocation table and the offset tables. In the case of field modifier mismatch, `IncompatibleClassChangeError` exception is raised; while in the case of modifier mismatch of other members, the entry in the table is filled with the offset for code raising the exception.

- (28) An abstract method cannot be invoked. If a subclass S inherits a method f defined in the old binary of a superclass C, and C is changed by declaring f as an abstract method, invoking f on an object of S is going to raise an exception (`AbstractMethodError`). Our solution is, when constructing the vtable of a class C that declares an abstract method f, to put a pointer to the exception code in the entry of f. A subclass that does not define f inherits the exception code, while a subclass that defines f works as if no change is made.

- (29,42) The members of nonpublic classes cannot be accessed from outside the package. Having access tags in the corresponding global allocation table entries solves this problem. Similar observations apply to interfaces.

4.4.3 Miscellaneous

- (32,45) Changing a signature has the combined effect of removing the old member and adding a new one.

- (33) Adding or deleting a native modifier is considered compatible. The support for this is trivial. Other changes related to native methods are beyond the scope of the JLS.

Manager's ctable	*Manager's offset table*
Tom.eat –> 0	
Tom.hack –> 1	
Jerry.eat –> 2	
Jerry.hack –> 3	
Jerry.study –> 4	

Figure 6: The ctable and offset table of `Manager`

- (30,31,43) Adding a field/method is a compatible change, except in the following cases.

  1. The new field/method shadows/overrides an old one, and the new field/method is less accessible than the old one.
  2. The new field/method shadows/overrides an old one, and the new field/method is a static (instance) member but the old one is an instance (static) member.
  3. The new field in an interface may shadow a field in other classes.

  These cases are handled as (25), (27) and (26).

- (44) Adding methods to interfaces is listed as a binary compatible change in the JLS. However, it MAY break compatibility with pre-existing binaries [8]. Based on our support for interfaces in Section 4.3, together with the technique described for (28), we can build the itable of an interface with pointers to specific exception code. If a class which implements the interface does not define all the interface methods, the exception code is inherited.

5 Example Revisited: Programmer & Manager

In our solution, the class `Manager` is compiled to contain a ctable and an offset table (Figure 6). The ctable maps distinct external methods used in the class to unique offsets. The offset table is initially empty, and is filled in incrementally at run time with the offsets of these external methods.

5.1 Scenario A Revisited: Adding a Method

In Scenario A we added a new method `sleep` at the very beginning of `Programmer`. Our solution does not require recompilation of `Manager`, because `Manager` does not make any assumptions about the vtable layouts of `Programmer` and `JavaProgrammer`.

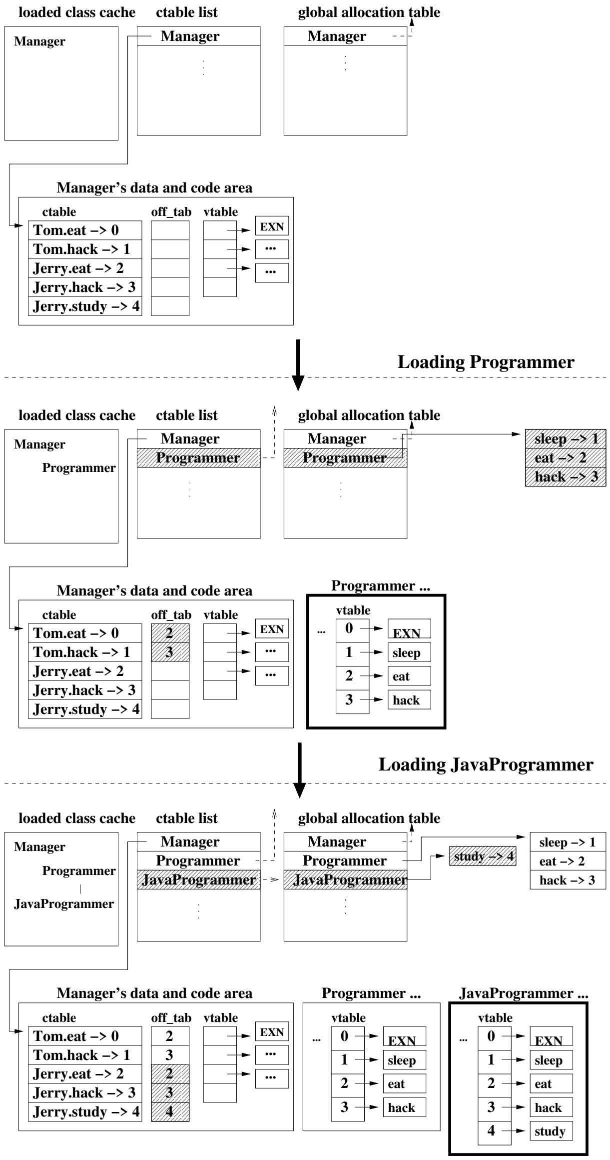

Figure 7: Scenario A revisited: adding a method.

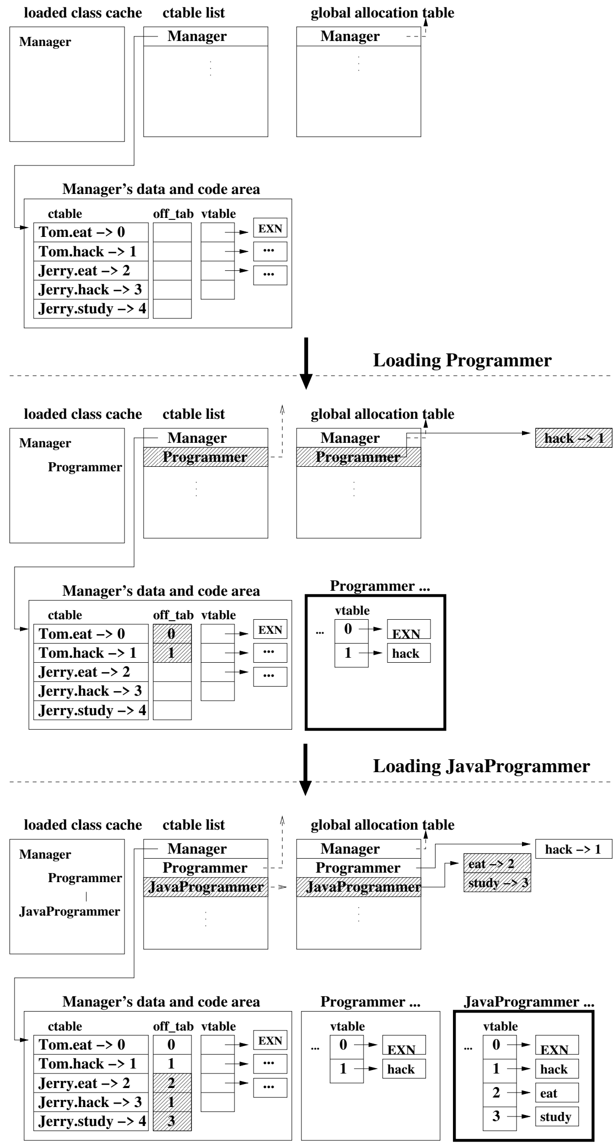

Figure 8: Scenario B revisited: removing a method.

Figure 7 illustrates what happens during the class loading of `Programmer` and `JavaProgrammer`. When `Programmer` is being loaded, its vtable is constructed on the fly so that any binary changes before class loading can be taken into account. The layout of this newly constructed vtable is registered in the global allocation table entry of `Programmer`. This layout information is used to fill in the relevant offset table entries of `Manager`.

When `JavaProgrammer` is loaded later (being a subclass of `Programmer`, its loading cannot precede that of `Programmer`), its vtable is constructed following the layout requirements specified by the global allocation table entry of `Programmer`. By doing this, the consistency of the vtable layouts between a subclass and its superclass is maintained. Once this is done, the layout information is propagated accordingly to the offset table entries of `Manager`. Using the offset specified in the corresponding offset table entry, a virtual method invocation will successfully get through.

5.2 Scenario B Revisited: Removing a Method

In Scenario B, we removed the method `eat` from class `Programmer`. Although this is an incompatible change, the program is still supposed to run as long as the removed method is not invoked. In class `Manager`, if we execute a method invocation `Jerry.eat`, the program will execute as usual. However, if we execute `Tom.eat`, an exception should be raised.

In our solution (Figure 8), after class `Programmer` is loaded, there would be no information about method `eat` in the global allocation table. However in `Manager`, which expected a method `eat` from class `Programmer`, there is an offset table entry which expects an offset for method `eat`. In this case we put 0 in the offset table entry. In the actual vtable of any class, the entry at offset 0 is always set to point to some specific code that would raise proper exceptions when invoking the `Tom.eat` method at run time. However, things will be as usual if `Jerry.eat` is executed, because `eat` does occur in the global allocation table entry of `JavaProgrammer`.

5.3 Scenario C Revisited: Binary Change at Run Time

Our solution works even if binary changes happen at run time (Figure 9). Here we illustrate why this is true using the example in scenario C, in which we changed the class hierarchy compatibly by inserting a new class. In scenario C, after class `Manager` is loaded and being executed, we add a new class `OOProgrammer` into the old class hierarchy. When object `Tom` is being created, class `Programmer` is loaded. Then, when object `Jerry` is being created, both class `OOProgrammer` and class `JavaProgrammer` are loaded. The vtable layouts of `Programmer`, `OOProgrammer` and `JavaProgrammer` are determined during class loading. All of this information is registered in the global allocation table. Using this information, the offset table of `Manager` is filled in incrementally. Eventually, the virtual method invocations will get through.

6 Algorithm

We have separately talked about how to support various binary compatibility issues in Section 4. In this section, we take a different view and present roughly the overall algorithm of our solution. The algorithm consists of two parts: the *compiler* part and the *class loader* part.

6.1 Compiler

To support binary compatibility, the major difference in the compilation is that the metadata structures (e.g. vtable, itable, field record, etc) of classes and interfaces are not fixed.

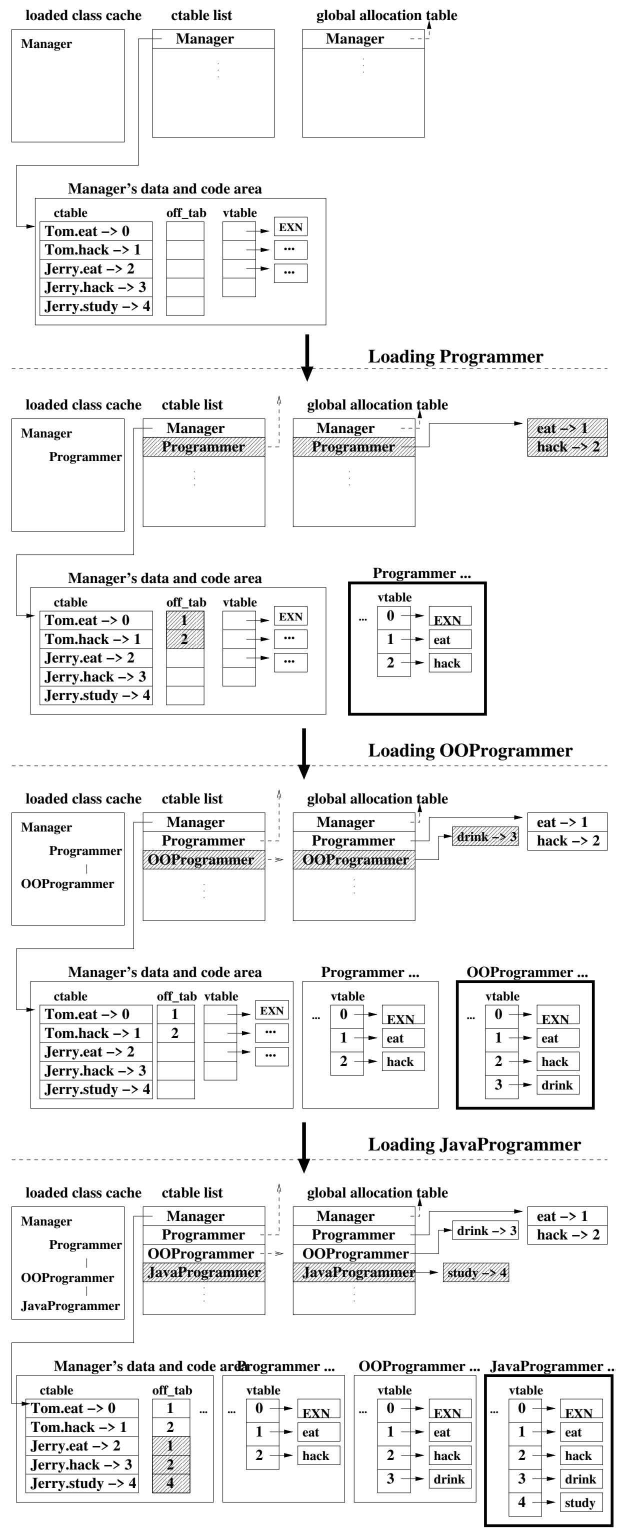

Figure 9: Scenario C revisited: changing class hierarchy.

1. *Create ctable and offset table for every class being compiled.* The ctable maps external references

(including references to various kinds of members) to unique offsets to the offset table. The offset table entries are tagged with the expected modifiers of the members. The contents of the offset table entries are blank. They are to be filled in incrementally at run time when the corresponding class is loaded. It is guaranteed that an offset table entry will be filled in before it is used, because no access to a class can be made before the class is loaded.

2. *Compiling external references.* Accesses to external references are compiled to go through the offset table. The object code fetches an offset from the offset table, and uses it to access the corresponding metadata structure.
   Taking virtual method invocation as an example, if an object `o` is of static class X, then a virtual method invocation `o.m()` that appears inside class C would be compiled as follows (where the final `o` is the self pointer):

   ```
   let off_m = lookup(ctable, "X", "m")
    in o.vtable [off_tab[off_m]] (o)
   ```

   Here `ctable` is the ctable of class C, `off_tab` is the offset table of class C. Class C's ctable entry for the virtual method m of class X is fixed. The lookup can be performed at compile time, so that at run time we can fetch the vtable offset directly from a certain offset table entry.

6.2 Class Loader

The class loader needs to maintain the related data structures such as the global allocation table, offset tables, and metadata structures like the vtable. It also has to check for various constraints during class loading. Here is what happens when a class C is loaded by the class loader. Loading interfaces is handled similarly.

1. *Loading superclasses.* Recursively load all the superclasses of C, if they are not loaded already.

2. *Check for constraints related to the class hierarchy.* In this step we only need to check things related to class C. Refer to Section 4.4.1 for details.

3. *Create a global allocation table entry for class C (T_c).* This T_c maps every member to its position information in the corresponding metadata structure (e.g. vtable), together with the modifier tags. Static members are easy to deal with, but special attention must be paid when mapping instance members, because we have to do it in such a way that it is consistent with the global allocation table entries of C's superclasses.
   In order to do this, we have to check the global allocation table entries of all the superclasses of C (recursively, they are already consistent with each other). Note that this data is not copied from the global allocation table entries of C's superclasses to the entry of C, because that would be space-inefficient. This is also when we make sure final methods are not overridden. After that, we construct the mappings of C's fresh instance members (those members defined in C but not in any superclasses of C). Here we can only use those offsets which are not yet used in any superclasses of C. In particular some offsets (e.g. 0) are reserved for incompatible change exception handling and cannot be used to map members to.

4. *Create the class data structures (e.g. vtable) of class C.* We do this according to the layout specified in the global allocation table entry C and the entries of C's superclasses. If the global allocation table maps a member to position k, then the data for the actual member is put at position k of the corresponding data structure. Beside those specified by the global allocation table, we also need to fill in the reserved entries with pointers to particular exception code.

5. *Fill in currently loaded classes' offset table entries that correspond to the members in C.* We do this with the help of the global allocation table entries of C and its superclasses. This step is done by iterating over the ctable list. An inverse ctable list would probably help to improve efficiency.
   Here we may find out that other classes might be expecting a non-existent member from C, or the expected access is not granted by comparing the modifier tags. In these cases we put offsets of exception code (e.g. 0) in the offset tables of those classes. However, as we discussed in Section 4.2, removed fields cannot be handled in the same manner. We raise an exception immediately when trying to fill in the offset table entry for that removed field.

6. *Fill in the offset table of class C.* This is done in the same manner as the last step. For C's offset table, we fill in those entries that correspond to members expected from the loaded classes. The entries for members expected from not-yet-loaded classes are left to be filled in later.

7. *Add the information of class C to the loaded class cache.* Also add the class hierarchy constraints demanded by class C to the set of constraints maintained by the system.

```
movl _ZN4java4lang6System3outE, %eax ;object
movl (%eax), %edx                    ;vtable
movl %eax, (%esp)                    ;this
movl _CD_C+4, %eax                   ;argument
movl %eax, 4(%esp)
call *116(%edx)
```

direct dispatch (vtable)

```
movl _ZN4java4lang6System3outE, %ecx ;object
movl _CD_C+4, %eax                   ;argument
movl (%ecx), %edx                    ;vtable
movl %ecx, (%esp)                    ;this
movl %eax, 4(%esp)
movl otable+4, %eax                  ;offset
call *(%eax,%edx)
```

indirect dispatch (offset table)

Figure 10: Sample method invocation code

8. *Extend the ctable list with a pointer to the ctable of class C.*

7 Implementation and Performance Evaluation

For ease of presentation, we have used vtable entry numbers in the offset table entries in the examples. In an actual implementation, a "processed offset" can be used instead. This "processed offset" is the vtable entry number multiplied by the size of a vtable entry (size of a pointer to code). Thus we transferred some of the burden from run time to compile time.

Bryce McKinlay implemented part of our solution for GCJ. His implementation so far provides full support for virtual methods and partial support for interface methods. The support for fields is still work in progress. This implementation is to be included in the future GCC release 3.1.

When compiling with the `-O2` optimization flag, it turns out that our new scheme generates one more assembly instruction for each virtual method invocation. Figure 10 shows the result of compiling a virtual method invocation. In our indirect dispatch scheme with the offset table involved, the code fetches the offset from the corresponding offset table (`otable`) entry, and adds it to the vtable pointer before calling the method. In contrast, the original direct vtable dispatch scheme generates code which adds the offset of the method to the vtable pointer and calls it. When the same call occurs in a loop (or in succession), the compiler moves the otable load out of the loop, so the overhead is reduced.

Benchmark	Direct	Indirect	Unit	Ratio (%)
Same:Instance	34.59	34.84	ns/call	99.27
Other:Instance	38.55	37.29	ns/call	103.39
Crypt (A)	7.26	7.27	s	99.82
HeapSort (A)	2.14	2.15	s	99.67
Series (A)	46.90	47.62	s	98.49
Crypt (B)	48.41	48.49	s	99.82
HeapSort (B)	14.69	14.67	s	100.14
Series (B)	485.87	493.12	s	98.53
AlphaBeta	24.76	25.04	s	98.89
MonteCarlo	32.89	32.64	s	100.77
Euler	327.55	328.65	s	99.66
RayTracer	45.21	44.81	s	100.90
MolDyn	518.09	529.13	s	97.91

Table 3: Java Grande 2.0 benchmarks

Our tests are based on the Java Grande 2.0 benchmarks [9] (the current version of GCJ cannot compile the SPECjvm98 benchmark suite). All results were obtained on a DELL Precision 410 workstation running Red Hat Linux 7.1. The machine has 512MB of main memory and 500MHz Pentium III processor with 512KB of cache. The average results over 3 rounds of tests, using dynamic linking with `-O2` flag turned on for both the direct vtable dispatch scheme of GCJ (Direct) and our indirect offset table dispatch scheme (Indirect), are shown in Table 3. In the "Ratio (%)" column, numbers less than 100 indicate performance slowdown using our scheme, while numbers greater than 100 indicate performance speedup.

The first two benchmarks are taken from benchmark suite section 1 (Low Level Operations). They test the performance of invoking virtual (instance) methods on an object of the same class and of another class. These two benchmarks perform a large number of iterations over 17 method invocations; every invoked method simply increases a global static counter. The result indicates that much of the offset table overhead was optimized away in these cases. Somewhat surprisingly the performance on "Other:Instance" was improved, possibly due to the different instruction scheduling.

The rest of the benchmarks (IDEA Encryption, Heap Sort, Fourier Coefficient Analysis, Alpha Beta Search, Monte Carlo Simulation, Computational Fluid Dynamics, 3D Ray Tracer, and Molecular Dynamics Simulation) are chosen from Sections 2 (Kernel) and 3 (Large Scale Applications) of the benchmark suite. We chose those with the most method invocations involved. Some of them are run on different data sizes (A/B). The performance penalty is on average less than 2%. Again we see some performance speedup in the test cases.

Another interesting observation is on the size of the object files. The new indirect dispatch scheme for binary compatibility puts extra offset tables in the object files. However, the vtables are no longer needed. When testing with the Java Grande 2.0 benchmarks, it turned out that the object file size using the new scheme is on average 1% less than using the standard vtable dispatch scheme.

8 Related Work

Joisha *et al.* [17] use an indirection table to enable efficient sharing of executable code for the IBM Quicksilver quasi-static compiler [27]. Besides increasing reusability of binary code, their solution also provides some support for binary compatibility. When binary changes (especial compatible ones) to a class `C` are detected during class loading, the class `C` is recompiled without requiring any changes to the loaded client classes of `C`, because all stitching (or linking operations) are performed on the indirection table. This stitching, or the operation which fills in the indirection table, happens incrementally the first time any single entry is used during the execution of the program. To enable this operation, they use some special offsets to trigger "traps" in the OS. When the program tries to access the memory using these offsets, the trap handler takes care of filling in the indirection table entry and resuming the program execution. Since the major concern of their paper is the sharing of code images, they do not explicitly address the handling of various binary incompatible changes.

In contrast, our solution does not require using any dynamic compilation techniques. Unlike the approach taken by Joisha *et al.*, we handle the problem of binary compatibility by building vtables and other class data structures not during compilation but during class loading. A global allocation table is introduced to help maintain the consistency of table layout between superclasses and subclasses. We also introduce an offset table for every class. These offset tables are filled in with the help of the global allocation table during class loading as soon as the referenced class is loaded. The statically compiled code (e.g. for method invocation) uses the offset tables to access the corresponding class data structures (e.g. vtables). Due to the observation that an external reference cannot be executed before the referenced class is loaded, going through the offset tables is guaranteed to be safe. Thus our approach does not rely on OS-dependent trapping mechanisms to trigger the linking process at runtime. However, similar trapping mechanisms can be used to handle missing fields in cases when full compliance with the JLS is important. Lastly, because we fill in all related offset table entries during the loading of a class, we can check for various binary incompatible changes. Our paper presents a detailed discussion of all the binary changes, including both compatible and incompatible, defined by the JLS.

9 Conclusion

We have presented a scheme which uses static compilation to support Java binary compatibility. All of the binary compatibility requirements in the Java Language Specification are supported with the same set of simple techniques. Binaries changed in a compatible manner can link successfully with pre-existing binaries that previously linked without error. Incompatible changes raise various run-time exceptions accordingly. Our implementation shows that this approach is fairly efficient and has the potential of being applied to real systems.

10 Acknowledgments

We want to specially thank Bryce McKinlay for implementing the indirect dispatching scheme for GCJ and his inspiring discussion on related issues, and Manish Gupta for keeping us informed about the state-of-the-art in dynamic Java compilers. We also thank many posters on the GCJ mailing list (java@gcc.gnu.org) for answering our questions about the GCJ internals, Greg Collins and Andrew McCreight for valuable comments on an early version of our paper, and the anonymous reviewers.

References

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM System Journal*, 39(1), February 2000.

[2] B. Alpern, A. Cocchi, S. Fink, D. Grove, and D. Lieber. Efficient implementation of Java interfaces: Invokeinterface considered harmless. In *Proc. 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 108–124, 2001.

[3] P. Bothner. A GCC-based Java implementation. In *IEEE Compcon 1997 Proceedings*, pages 174–178, February 1997.

[4] D. Chambers, C. Dean, J. Grove. Whole-program optimization of object-oriented languages. Technical Report TR-96-06-02, Dept. of Computer Science and Engineering, University of Washington, 28, 1997.

[5] D. Chase, R. Hoover, and K. Zadeck. BulletTrain technology white paper. `http://www.naturalbridge.com/`, 2001.

[6] D. Detlefs and O. Agesen. Inlining of virtual methods. In *ECOOP'99, LNCS 1628*, pages 258–278, 1999.

[7] O. P. Doederlein. The Java performance report – part IV (static compilers). `http://www.javalobby.org/members/jpr/`, August 2001.

[8] S. Drossopoulou, D. Wragg, and S. Eisenbach. What is Java binary compatibility? *ACM SIGPLAN Notices*, 33(10):341–358, 1998.

[9] Edinburgh Parallel Computing Centre. The Java Grande forum benchmark suite. `http://www.epcc.ed.ac.uk/javagrande/`, 2001.

[10] Excelsior, LLC. Excelsior JET. `http://www.excelsior-usa.com/jet.html`, 1999.

[11] I. R. Forman, M. H. Conner, S. H. Danforth, and L. K. Raper. Release-to-release binary compatibility in SOM. In *Proc. 1995 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 426–438, Oct. 1995.

[12] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification (Second Edition)*. Addison-Wesley, 2000.

[13] IBM Corporation. IBM's system object model (SOM): Making reuse a reality. First Class, a bimonthly publication of the Object Management Group, October 1994.

[14] IBM Corporation. IBM VisualAge for Java. `http://www.software.ibm.com/ad/vajava/`, 1998.

[15] Instantiations, Inc. Jove, optimizing native compiler for Java technology. `http://www.instantiations.com/jove/product/thejovesystem.htm`, 2000.

[16] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani. A study of devirtualization techniques for Java Just-In-Time compiler. In *Proc. 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, 2000.

[17] P. G. Joisha, S. P. Midkiff, M. J. Serrano, and M. Gupta. A framework for efficient reuse of binary code in Java. In *Proc. 15th ACM International Conference on Supercomputing*, pages 440–453, New York, June 2001.

[18] A. Krall and R. Grafl. CACAO — A 64-bit JavaVM just-in-time compiler. *Concurrency: Practice and Experience*, 9(11):1017–1030, 1997.

[19] C. League, V. Trifonov, and Z. Shao. Type-preserving compilation of featherweight Java. In *Proc. 8th Foundations of Object-Oriented Languages Workshop*, London, January 2001.

[20] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification (Second Edition)*. Addison-Wesley, 1999.

[21] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. In *Proc. 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 85–97, Jan. 1998.

[22] NaturalBridge, Inc. BulletTrain optimizing compiler and runtime for JVM bytecodes. `http://www.naturalbridge.com/`, 1996.

[23] G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *Proc. 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 333–344, 1998.

[24] H. Ogawa, K. Shimura, S. Matsuoka, F. Maruyama, Y. Sohda, and Y. Kimura. OpenJIT: an open-ended,reflective JIT compiler framework for Java. In *14th European Conference on Object-Oriented Programming*, pages 362–387, 2000.

[25] M. Paleczny, C. Vick, and C. Click. The Java HotSpot™ server compiler. In *Proceedings of the 1st Java™ Virtual Machine Research and Technology Symposium*, 2001.

[26] P. Potrebic. What's the fragile base class (FBC) problem? Be™ newsletter – the newsletter for BeOS™ developers and customers. Issue 79, June 1997.

[27] M. J. Serrano, R. Bordawekar, S. P. Midkiff, and M. Gupta. Quicksilver: a quasi-static compiler for Java. In *Proc. 2000 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 66–82, Oct. 2000.

[28] K. Shudo. shuJIT—JIT compiler for Sun JVM/x86. `http://www.shudo.net/jit/`, 1998.

[29] Sun Microsystems. The Java HotSpot virtual machine white paper. `http://java.sun.com/products/hotspot/`, 2001.

[30] Symbian Ltd. EPOC C++ system documentation – controlling binary compatibility. `http://www.symbian.com/`, 1999.

[31] The FLINT project. Binary compatibility report. `http://flint.cs.yale.edu/~dachuan/bincomp/main.ps.gz`, 2001.

[32] The Free Software Foundation. The GNU compiler for Java. `http://gcc.gnu.org/java/`, 2000.