Reprinted from the

Proceedings of the GCC Developers' Summit

June 2nd–4th, 2004
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Review Committee

Eric Christopher, *Red Hat, Inc.*
Janis Johnson, *IBM*
Toshi Morita, *Renesas Technologies*
Zack Weinberg, *CodeSourcery*
Al Stone, *Hewlett-Packard*
Richard Henderson, *Red Hat, Inc.*
Andrew Hutton, *Steamballoon, Inc.*
Gerald Pfeifer, *SuSE, GmbH*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

Gcj: the new ABI and its implications

Tom Tromey
Red Hat, Inc.
tromey@redhat.com

Andrew Haley
Red Hat, Inc.
aph@redhat.com

What is binary compatibility?

The Java Language Specification [2] has an entire chapter, Chapter 13, dedicated to binary compatibility. This chapter lays out rules for writing binary compatible programs: programs can be changed in these ways without requiring the recompilation of dependent modules. This covers some simple, obvious things, such as the fact that adding or removing the `synchronized` keyword from a method won't affect binary compatibility. It also covers more complex rules, so for instance it is possible to override an inherited method or rearrange fields in a class without affecting compatibility.

Note that binary compatibility and source compatibility differ. For instance, it is binary compatible to change a field's access from `protected` to `public`. This is not source compatible in some situations.

Binary Compatibility has a great promise: with a few restrictions, you will never have to recompile libraries again.

1 Why we want it

Initially, the gcj project paid no attention to Chapter 13. In practice we implemented a more static language than Java, and it looked as if it would be difficult to get good performance from pre-compiled code that adhered to the binary compatibility rules.

This led to one important restriction on gcj-compiled code, namely that two classes with the same name could not both be loaded at once: this is PR 6819 [4]. Over time, this has proved to be more and more difficult to work around. For instance, in 2003 we split out some libraries from libgcj because some programs shipped their own copies; this in turn caused other problems.

Another important problem we tried to solve in 2003 was the proper operation of class loaders. As it turned out, class loading and binary compatibility are related, and we realized we could solve both problems with the same implementation.

In particular, sophisticated applications such as Eclipse rely on Java's lazy loading and linking capabilities to control class loading and visibility. It isn't possible both to satisfy the proper semantics of a class loader and to have ordinary ELF-style linking.

The Java language gives programmers facilities that go far beyond what is possible in more conventional programming languages. For example, you may define a class loader to load your own classes into the virtual machine. Your class loader will have its own name space and it will inherit classes from the base Java class loader but its own loaded classes will not be externally visible. You can define your own scheme for resolving symbols.

It is quite possible for the same Java class to be loaded several times by several different class

loaders, and in each case its references will be resolved differently.

When a class is loaded, references it makes to other classes are not immediately resolved. This allows mutually dependent classes to be loaded, and later fixed up by calling `resolveClass`.

All of this is a very long way from what can be achieved by using conventional ELF linkage.

2 Implementation

The implementation of a new binary compatibility ABI for gcj began several years ago with the work of Bryce McKinlay, and the paper Yu [1].

The basic idea behind our implementation approach is to put all references made by a class into two special tables, called the `atable` and the `otable`.

The `otable`, or Offset Table, is a table of offsets from some base pointer. The `atable`, or Address Table, is a table of absolute addresses. Every class has an `atable` and an `otable`. Initially these tables are filled with symbolic references. Later, when the class is linked, these symbolic references are turned into offsets or addresses, as appropriate.

2.1 Class references

Class references are handled via the constant pool, a table that already existed in the old ABI. Entries in the constant pool are resolved when a class is prepared; an operation like `new` or `instanceof` refers to an entry in the pool.

2.2 Static methods and fields

A static method or field is referred to via the `atable`. Each symbolic entry in the `atable` consists of three parts: a class name, a member name, and a type signature. At class preparation time, the appropriate class and member are found, access checks are done, and then the address of the member is written into the `atable` slot. So, code in Class A that refers to a static member of Class B does so via an index into the `atable` belonging to Class A.

If a static method is not found, we simply write the address of a function which will throw the appropriate exception.

If a static field is not found, we throw an `IncompatibleClassChangeError` at class preparation time. In Yu [1] this is mentioned as a bug in the design; however, we believe that this behavior is specifically allowed by the linking rules in section 12.3 of the Java Language Specification [2].

2.3 Instance methods

Instance methods are handled via the `otable`, not the `atable`. Like the `atable`, the `otable` holds class names, member names, and type signatures. However, instead of mapping these to addresses, it instead maps them to offsets.

When computing the value of an `otable` slot for an instance method, we load and lay out the target class and all its superclasses as well. As part of this process, we compute the target class's vtable; from this we find the correct value to put in the `otable` slot.

Old-ABI code calls virtual methods like:

```
(((vtable *) obj)[index]) (obj, ...)
```

With the new ABI, this is transformed to:

```
(((vtable *) obj)[otable[index]])
                 (obj, ...)
```

If an instance method is not found, we put a special value into the `otable` slot which, when the vtable lookup is done, results in a call to a method that throws `IncompatibleClassChangeError`.

2.4 Instance fields

Instance fields are handled similarly to instance methods. Where old ABI code compiles a field reference:

```
*((type *) (obj + offsetof (field)))
```

the new ABI produces the equivalent of:

```
*((type *) (obj +
            otable[field_index]))
```

Although this is an extra memory reference, it is less painful than might first appear: the `otable` and `atable` have good locality, typically being referred to many times in a method.

Note that because all class layout is done dynamically, even references to one's own private fields must go through the `otable`, as one's superclass might add or remove fields and this will change the offsets of all subclass fields.

2.5 Interfaces

Interface dispatch also requires an extra indirection via the `otable`, and it requires us to compute interface dispatch tables at runtime, much as we compute the vtables and class layout at runtime.

2.6 Exception handlers

For `catch` clauses we write a class name (instead of a reference to a Class object) into the DWARF-2 exception table. The class name is suitably mangled so that the type matching function for a catch block can distinguish between old and new ABI code.

When an exception is thrown, these class names are looked up by the appropriate class loader and turned into references to the corresponding classes.

2.7 Versioning

gcj still statically generates an instance of Class for each class that is compiled. In the future we plan instead to generate a class descriptor, which will be instantiated as a Class at runtime. This will insulate compiled code from changes to java.lang.Class, and it will also make it slightly easier for us to handle ABI versioning. We intend to add an ABI version number to the class descriptor, and then let the runtime library handle compatibility as desired.

2.8 libgcj API

Compiled code must still make references to symbols exported from libgcj. For instance, operations such as `new` or `instanceof` are implemented by means of exported `_Jv_` functions; the compiler generates direct calls to these functions.

We have considered redirecting calls to these functions via the `atable` as well, but as there are only twenty or so it seems simpler to handle these according to the usual versioning rules for shared libraries.

Compiled code continues to know the layout of array types. We don't anticipate arrays changing incompatibly.

We plan to continue to compile parts of the core library—in all likelihood at least `java.lang` and `java.io`—using the old ABI. Ap-

plication code cannot portably replace these classes, so there is no drawback to compiling them old-style.

2.9 Bytecode Verification

One related problem is that of bytecode verification with an ahead-of-time compiler.

In Java, the compile-time and runtime environments might be very different. In order to handle this and still ensure runtime type safety, a typical JVM will perform bytecode verification in the runtime environment.

gcj includes a bytecode verifier as part of its compilation, when compiling from bytecode to object code. However, this is insufficient when the bytecode can be loaded into an arbitrary runtime environment. In particular it would be possible to construct an environment where all the requirements of the compiled code (names of types and methods) are met, but where the result allows subversion of the type system.

For example, a class f might be defined:

```
class f implements B
{
  ...
```

and a user could write an initializer

```
B thing = new f();
```

but if an incompatible change were made to f

```
class f
{
  ...
```

the variable `thing` would now refer to an object that did not implement B. This is a violation of the type system.

The solution to this is to perform bytecode verification in two steps. The first step, still in gcj, works much like an ordinary verifier. All the "static" properties of bytecode, such as whether the declared stack depth is sufficient, can be verified once. Now, when the verifier is asked to verify a fact about a type or method, it always yields `true`, and adds a "verification assertion" to the generated code.

At runtime, these assertions are verified when the class is linked. This process is much quicker than ordinary bytecode verification, which requires modeling the control flow of the code. These assertions are of the form 'A implements B' or 'A extends B', which are very easy to check.

2.10 Type assertions for source code

A similar problem occurs when compiling from Java source to native code. In this situation, there is no verification step to split. Instead, the assertion table is filled based on any implicit upcasts that appear in the source; each such cast represents a constraint on the type hierarchy that must remain true at runtime.

2.11 CNI

CNI, the Compiled Native Interface, is a way to write Java `native` methods in C++ with zero overhead. With CNI, Java classes are used to generate C++ header files, which then enable relatively ordinary C++ code to make calls on Java objects.

CNI is also going to require some changes. In essence this will involve duplicating some of the `atable` and `otable` logic from gcj in g++ and arranging for these references to be resolved at runtime when appropriate. We anticipate accomplishing this by emitting static initializers which will register table contributions from the current compilation unit with the libgcj runtime.

We plan to make several other CNI changes

now, while we're changing the ABI, in order to postpone any other needed ABI changes. In particular we plan to introduce smart pointers to allow seamless `NULL`-pointer checking on all platforms, and we plan to tighten the rules about what parts of memory can be assumed to be scanned by the garbage collector.

3 Consequences

This approach to binary compatibility has some very interesting consequences for gcj and gcj-compiled code.

3.1 gcj as JIT

Due to the new runtime linkage model and the new approach to bytecode verification, gcj can now compile a single .class file in complete isolation. That is, compiling a class file doesn't require gcj to read any other classes, not even `java.lang.Object`. This works because a class file has complete symbolic information about its dependencies—just what the `atable` and `otable` require—and because verification will answer "yes" to any type-related question without actually examining any other types until runtime.

This property in turn lets us use gcj itself as a caching JIT. Conventionally, `ClassLoader.defineClass()` takes an array of bytes that is the binary code for a class and loads it into memory. Instead, we compute a cryptographic checksum of the bytes and use it as a key into a cache of shared libraries. If the class is found, we simply `dlopen()` it. If not, we invoke gcj (which is possible and relatively efficient because we only need the class file in isolation) to put a new shared library in the cache.

We're also considering the possibility of making it easy to prime the libgcj cache. To make existing Java applications run with decent performance, you would then only need to compile each .jar file and copy the resulting .so into the cache. No application changes would be needed. Another approach we're investigating is to change `URLClassLoader` to transparently find shared libraries corresponding to `.jar` files on its class path.

3.2 VM independence

The code generated by gcj is also surprisingly VM-independent. It refers to the various tables (`otable`, `atable`, assertion table), and to the small number of libgcj builtin functions known to gcj. This means that gcj-compiled code could easily be loaded into any VM implementing this interface; the biggest assumption is that the runtime includes a conservative garbage collector. Even that may not necessarily be true in the future: a few garbage collection hooks would remove even that requirement.

The generated code is also quite independent of other aspects of the runtime environment, for instance the kernel or libc. It should be possible to compile Java code once, and then simply never recompile it even as the rest of the system, including libgcj, is upgraded.

We're hoping other free Java implementations will adopt this same approach as the basis of a "pluggable JIT" interface.

3.3 Performance and Size

It is too early to know the precise impact of the new ABI. For some cases, we know that the penalty will be small: for instance, the cost of a static method invocation via the `atable` is similar to the cost of indirection via the PLT.

On the other hand, we expect some costs to be larger: for example, instance field references

will be more expensive.

The Yu [1] paper quotes an average performance penalty of less than 2%; however, their implementation did not implement field indirection.

4 Problems and gotchas

It is possible that important Java programs may rely on the precise link-time behavior of existing VMs. In case it becomes necessary to change our approach, we believe we can emulate the more lazy behavior of other VMs in one of two ways. On machines with the required support, we can map a special, unwriteable memory segment, and then fill `atable` slots with pointers into this area. This approach will let us differentiate between `NULL` pointer traps and invalid field traps, and then throw the appropriate exception. For other platforms, we can add extra instrumentation to the compiled code, at some performance cost.

5 Today and Tomorrow

As of this writing, Andrew is still finishing the implementation of the core parts of the new ABI. His work builds on some earlier patches from Bryce, and is checked in on `gcj-abi-2-dev-branch`. Tom hopes to begin work on the verification problem soon.

Andrew has built a demo version of gcj-as-JIT and posted some results to the gcj list; see his post [3]. The results are surprisingly good—a longer startup delay, as would be expected, but performance falling between that of Sun's and IBM's JITs on Linux. We anticipate some useful performance gains from tree-ssa as well, eventually—in particular smarter array bounds checking.

Ideally we would like to see a supported, but perhaps still preliminary, version of this ABI in GCC 3.5, with real compatibility promised starting with 3.6.

6 Acknowledgements

We would like to thank Bryce McKinlay for his initial and ongoing work in this area, Jeff Sturm for his contributions to the new ABI project, and Sarah Woodall for editing.

References

[1] Zhong Shao Dachuan Yu and Valery Trifonov. Supporting binary compatibility with static compilation. In *2nd USENIX Java™ Virtual Machine Research and Technology Symposium (JVM'02)*. Usenix, August 2002. `http://www.usenix.org/publications/library/proceedings/javavm02/yu/yu_html/`.

[2] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass., 2000.

[3] Andrew Haley. gcj-jit, 2003. `http://gcc.gnu.org/ml/java/2003-01/msg00022.html`.

[4] Oskar Liljeblad. Pr 6819, 2002. `http://gcc.gnu.org/PR6819`.