Java/C++ integration: Writing native Java methods in natural C++

Per Bothner
Brainfood
<bothner@gnu.org>

Tom Tromey
Red Hat
<tromey@redhat.com>

Abstract

"Native methods" in Java are methods written in some other language, usually C or C++. Sun included in JDK 1.1 the "Java Native Interface" (JNI) for writing such native methods in a portable way, independent of JVM implementation. JNI satisfies that goal, but has two major problems: JNI code has major inherent inefficiencies, because evrything thas to be done as calls through a run-time table of function pointers. Using JNI is also very verbose, tedious and error-prone for the programmer.

The Gnu Compiler for the Java platform (GCJ) is based on compiling Java to machine code using the Gnu Compiler Collection (Gcc) framework. GCJ offers an in addition to JNI an alternative, the Compiled Native Interface (CNI). CNI is based on the idea of making the C++ and Java data representations and calling conventions as close as practical, and using a slightly modified Java-aware C++ compiler to compile native method written in C++ methods. CNI code is both very efficient and is also very easy and natural to write, because it uses standard C++ syntax and idioms to work with Java data. The runtime and class library associated with GCJ is libgcj, which is written in a mix of Java and C++ code using CNI.

1 Background

Not all the code in a Java application can be written in Java. Some must be written in a lower-level language, either for efficiency reasons, or to access low-level facilities not accessible in Java. For this reason, Java methods may be specified as "native". This means that the method has no method body (implementation) in the Java source code. Instead, it has a special flag which tells the Java virtual machine to look for the method using some unspecified lookup mechanism.

1.1 The Java Native Interface

Sun's original Java Development Kit (JDK) version 1.0 defined a programming interface for writing native methods in C. This provided rather direct and efficient access to the underlying VM, but was not officially documented, and was tied to specifics of the VM implementation. There was little attempt to make it an abstract API that could work with any VM.

For JDK 1.1, Sun defined a "Java Native Interface" (JNI) that defines the offical portable programming interface for writing such "native methods" in C or C++. This is a binary interface (ABI), allowing someone to ship a compiled library of JNI-compiled native code, and have it work with any VM implementation(for that platform).

The problem with JNI that it is a rather heavy-weight interface, with major run-time overheads. It is also very tedious to write code using JNI. For example, for native code to access a field in an object, it needs to make two function calls (though the result of the first can be saved for future accesses). This is cumbersome to write and slow at run-time.

To specify a field in JNI, you pass its name as a string to a run-time routine that searches "reflective" data structures. Thus the JNI requires the availability at run-time of complete reflective data (names, types, and positions of all fields, methods, and classes). The reflective data has other uses (there is a standard set of Java classes for accessing the reflective data), but when memory is tight, as in an embedded system, it is a luxury many applications do not need.

As an example, here is a small Java example of a class intended for timing purposes. (This could be written in portable Java, but let us assume for some reason we don't want to do that.)

```
package timing ;
class Timer {
  private long lastTime;
  private String lastNote;
```

```
  /** Return time in milliseconds
   * since last call,
   * and set lastNote. */
  native long sinceLast(String note);
}
```

Figure 1 shows how it could be programmed using the JNI: Note the first `env` parameter, which is a pointer to a thread-specific area, which also includes a pointer to a table of functions. The entire JNI is defined in terms of these functions, which cannot be inlined (since that would make JNI methods no longer binary compatible accross VMs).

```
#include <jni.h>

jdouble Java_Timer_sinceLast (
    JNIEnv *env, /* interface pointer */
    jobject obj, /* "this" pointer */
    jstring note) /* argument #1 */
{
  // Note that the results of the first
  // three statements could be saved for
  // future use (though the results
  // have to be made "global" first).
  jclass cls =
     env->FindClass("timing.Timer");
  jfieldId lTid =
     env->GetFieldID(cls, "lastTime",
                     "J");
  jfieldId lNid =
     env->GetFieldID(cls, "lastNote",
                     "Ljava/lang/String;");

  jlong oldTime =
    env->GetLongField(obj, lTid);
  jlong newTime =
    calculate_new_time();
  env->SetLongField(obj, lTid, newTime);
  env->SetObjectField(obj, lNid, note);
  return newTime - old_Time;
}
```

GCJ supports JNI, but it also offers a more efficient, lower-level, and more natural native API, which we call CNI, for “Compiled Native Interface”. (It can also stand for Cygnus Native Interface, since CNI was designed at Cygnus Solutions before Cygnus was acquired by Red Hat.) The basic idea is to make GNU Java compatible with GNU C++ (G++), and provide a few hooks in G++ so C++ code can access Java objects as naturally as native C++ objects. The rest of this paper goes into details about this integrated Java/C++ model. The key idea is that the calling conventions and data accesses for CNI are the same as for normal nonnative Java methods. Thus there is no extra `JNIEnv` parameter, and the C++ programmer gets direct access to the Java objects. This does require co-ordination between the C++ and Java implementations.

Below is the the earlier example written using CNI.

```
#include "timing/Timer.h"

::timing::Timer::sinceLast(jstring note)
{
  jlong oldTime = this->lastTime;
  jlong newTime = calculate_new_time();
  this->lastTime = newTime;
  this->lastNote = note;
  return newTime - oldTime;
}
```

This uses automatically-generated `timing/Timer.h`

```
#include <cni.h>
class ::timing::Timer
  : public ::java::lang::Object
{
  jlong lastTime;
  jstring lastNote;
public:
  jlong virtual sinceLast(jstring note);
};
```

2 API vs ABI

A fundamental goal of JNI was that it should be independent of the JVM; it should be possible to implement JNI on any reasonable JVM implementation. CNI can also in principle be implemented on any reasonable Java implementation, by putting sufficient knowledge in the C++ compiler. This is possible because the C++ compiler can distinguish C++ and Java types, and thus use different representations for C++ and Java objects. However, CNI works more naturally the closer the C++ and Java data representations are. For GCJ our goal was to make the Java ABI (Application Binary Interface) as close to the C++ ABI as made sense.

Another goal of JNI was to define a portable ABI, rather than just an API (Application Programming Interface). That for any given platform (machine and os), compiled JNI code should not depend on the JVM implementation. However, since JNI is defined in terms of C data types and function calls, it does depend on the C ABI of the given platform. One might say that JNI was designed for applications delivered in compiled form, presumably

on some small number of platforms. It seems a questionable tradeoff to accept the overheads and inconvenience of JNI for this very restricted form of binary portability, especially to those of us who believe source should be available.

That is not to say that we think an ABI is not desirable, far from it. We think that for now an ABI may be premature, and an API such as CNI may make more sense. Note though that much of the C++ community is moving towards a stable defined ABI, and this will become the default for the forthcoming Gcc 3.0. At that point, it may make sense to define a Java ABI partly in terms of a C++ ABI. CNI may be viewed as a pre-cursor to that.

If we view CNI as an ABI for Java, it nails down a number of aspects of the Java implementation (such as field layout and exception handling). CNI leaves other parts of the implementation, such as object allocation and synchronization, unspecified but defines portable hooks. These hooks define an ABI as long as the hooks are function calls, but if the hooks are macros or get inlined to access implementation-specific fields, then binary compatibility is gone.

CNI as currently implemented asumes a conservative garbage collector. For example CNI lets you loop through an array without having to be aware of garbage collection issues. While this seems to prohibit a copying collector, actually it does not. Rather, it means that if a copying collector is used, then the C++ compiler has to be aware the fact, and generate the needed tables so the collector can update all registers and memory locations that point at a moved object.

A related more general disadvantage with GCJ concerns "Binary Compatibility" as discussed in the Java Language Specification [??]. GCJ-compiled classes are much more vulnerable to breaking if a class they depend on is changed and re-compiled than Java `class` files are. Adding a private member to a base class changes the offsets of fields in a class, which means the generated code is changed. This is the same as for C++, and is a cost of GCJ-style compilation you pay in exchange for performance. There are techniques that some people use to reduce these problems in C++ [??]; similar techniques may be applicable to GCJ and hence CNI. One possibility is that the offset of a field in a structure be compiled into a link-time constant, that would be resolved by the (static or dynamic) linker. That would reduce binary compatibility problems quite a bit, though it may produce slightly less optimal code.

3 Utility functions and macros

Both JNI and CNI provide toolkits of utility functions so native code code can request various services of the VM. CNI uses the C++ syntax for operations that have a direct correspondence in C++ (such as accessing an instance field or throwing an exception), For other features, such as creating a Java string from a nul-terminated C string, we need utility functions or macros. Many of these have similar names and functionality as the JNI functions, except that they do not depend on a `JNIEnv` pointer.

For example, the JNI interface to create a Java string from a C string is the following in C:

```
jstring str =
  (*env)->NewStringUTF(env, "Hello");
```

The JNI C++ interface is just a set of inline methods that wrap the C interface, for example:

```
jstring str = env->NewStringUTF("Hello");
```

In the CNI, we do not use a `JNIEnv` pointer, so the usage is:

```
jstring str = JvNewStringUTF("Hello");
```

In general, CNI functions and macros start with the '`Jv`' prefix, for example the function '`JvNewObjectArray`'. This convention is used to avoid conflicts with other libraries. Internal functions in CNI start with the prefix '`_Jv_`'; names with this prefix are reserved to the implementation according the C and C++ standards.

3.1 Strings

To illustrate the available utility functions, CNI provides a number of utility functions for working with Java `String` objects. The names and interfaces are analogous to those of JNI.

▶`jstring JvNewString (const jchar *`*chars*`, jsize` *len*`)`◀ Creates a new Java String object, where *chars* are the contents, and *len* is the number of characters.

▶`jstring JvNewStringLatin1 (const char *`*bytes*`, jsize` *len*`)`◀ Creates a new Java String object, where *bytes* are the Latin-1 encoded characters, and *len* is the length of *bytes*, in bytes.

▶`jstring JvNewStringLatin1 (const char *`*bytes*`)`◀ Like the first `JvNewStringLatin1`, but computes *len* using `strlen`.

▶`jstring JvNewStringUTF (const char *`*bytes*`)`◀ Creates a new Java String object, where *bytes* are the UTF-8 encoded characters of the string, terminated by a null byte.

▶`jchar *JvGetStringChars (jstring `*str*`)`◀ Returns a pointer to the array of characters which make up a string.

▶ `int JvGetStringUTFLength (jstring `*str*`)`◀ Returns number of bytes required to encode contents of *str* as UTF-8.

▶ `jsize JvGetStringUTFRegion (jstring `*str*`, jsize `*start*`, jsize `*len*`, char *`*buf*`)`◀ This puts the UTF-8 encoding of a region of the string *str* into the buffer *buf*. The region of the string to fetch is specified by *start* and *len*. It is assumed that *buf* is big enough to hold the result. Note that *buf* is not nul-terminated.

4 Object model

In terms of language features, Java is in essence a subset of C++. Java has a few important extensions, plus a powerful standard class library, but on the whole that does not change the basic similarity. Java is a hybrid object-oriented language, with a few native types, in addition to class types. It is class-based, where a class may have static as well as per-object fields, and static as well as instance methods. Non-static methods may be virtual, and may be overloaded. Overloading in resolved at compile time by matching the actual argument types against the parameter types. Virtual methods are implemented using indirect calls through a dispatch table (virtual function table). Objects are allocated on the heap, and initialized using a constructor method. Classes are organized in a package hierarchy.

All of the listed attributes are also true of C++, though C++ has extra features (for example in C++ objects may also be allocated statically or in a local stack frame in addition to the heap). Because GCJ uses the same compiler technology as g++ (the GNU C++ compiler), it is possible to make the intersection of the two languages use the same ABI (object representation and calling conventions). The key idea in CNI is that Java objects are C++ objects, and all Java classes are C++ classes (but not the other way around). So the most important task in integrating Java and C++ is to remove gratuitous incompatibilities.

4.1 Primitive types

Java provides 8 "primitive" types: `byte`, `short`, `int`, `long`, `float`, `double`, `char`, and `boolean`. These as the same as the following C++ `typedef`s (which are defined in a standard header file): `jbyte`, `jshort`, `jint`, `jlong`, `jfloat`, `jdouble`, `jchar`, and `jboolean`.

Java type	C++ name	Description
byte	jbyte	8-bit signed integer
short	jshort	16-bit signed integer
int	jint	32-bit signed integer
long	jlong	64-bit signed integer
float	jfloat	32-bit IEEE floating-point
double	jdouble	64-bit IEEE floating-point
char	jchar	16-bit Unicode character
boolean	jboolean	logical (Boolean) values
void	void	no value

▶`JvPrimClass (`*primtype*`)`◀ This is a macro whose argument should be the name of a primitive type, e.g. `byte`. The macro expands to a pointer to the `Class` object corresponding to the primitive type. E.g., `JvPrimClass(void)` has the same value as the Java expression `Void.TYPE` (or `void.class`).

4.2 Classes

All Java classes are derived from `java.lang.Object`. C++ does not have a unique "root" class, but we use a C++ `java::lang::Object` as the C++ version of the `java.lang.Object` Java class. All other Java classes are mapped into corresponding C++ classes derived from `java::lang::Object`.

We consider a Java class such as `java.lang.String` and the corresponding C++ class `java::lang::String` to be the same class, just using different syntax.

Interface inheritance (the "`implements`" keyword) is currently not reflected in the C++ mapping.

4.3 Object references

We implement a Java object reference as a pointer to the start of the referenced object. It maps to a C++ pointer. (We cannot use C++ references for Java references, since once a C++ reference has been initialized, you cannot change it to point to another object.) The `null` Java reference maps to the `NULL` C++ pointer.

The original JDK implemented an object reference as a pointer to a two-word "handle". One word of the handle points to the fields of the object, while the other points to a method table. GNU Java, like many newer Java implementations, does not use this extra indirection.

4.4 Casts and Runtime Type Safety

Java casts do runtime type checking when downcasting. GCJ automatically inserts calls to runtime functions to perform these checks as appropriate. When writing CNI code, this checking is not done automatically. C++ code which must check this can call `java::lang::Class::isAssignableFrom`.

4.5 Object fields

Each object contains an object header, followed by the instance fields of the class, in order. The object header consists of a single pointer to a dispatch or virtual function table. (There may be extra fields "in front of" the object, for example for memory management, but this is invisible to the programmer, and the reference to the object points to the word contining the dispatch table pointer.)

The fields are laid out in the same order, alignment, and size as in C++. Specifically, 8-bite and 16-bit native types (`byte`, `short`, `char`, and `boolean`) are not widened to 32 bits, even though the Java VM does extend 8-bit and 16-bit types to 32 bits when on the VM stack or temporary registers.

If you include the `gcjh`-generated header for a class, you can access fields of Java classes in the "natural" way. Given the following Java class:

```
public class Int
{
  public int i;
  public Int (int i) { this.i = i; }
  public static Int zero = new Int(0);
}
```

you can write:

```
#include <gcj/cni.h>
#include <Int.h>
Int*
mult (Int *p, jint k)
{
  if (k == 0)
    // static member access.
    return Int::zero;
  return new Int(p->i * k);
}
```

CNI does not strictly enforce the Java access specifiers, because Java permissions cannot be directly mapped into C++ permission. Private Java fields and methods are mapped to private C++ fields and methods, but other fields and methods are mapped to public fields and methods.

4.6 Arrays

While in many ways Java is similar to C and C++, it is quite different in its treatment of arrays. C arrays are based on the idea of pointer arithmetic, which would be incompatible with Java's security requirements. Java arrays are true objects (array types inherit from `java.lang.Object`). An array-valued variable is one that contains a reference (pointer) to an array object.

Referencing a Java array in C++ code is done using the `JArray` template, which is defined as follows:

```
class __JArray : public java::lang::Object
{
public:
  int length;
};

template<class T>
class JArray : public __JArray
{
  T data[0];
public:
  T& operator[](jint i) { re-
turn data[i]; }
};
```

For example, if you have a value which has the Java type `java.lang.String[]`, you can store it a C++ variable of type `JArray<java::lang::String*>*`.

CNI has some convenience typedefs which correspond to typedefs from JNI. Each is the type of an array holding

objects of the appropriate type:

```
typedef __JArray *jarray;
typedef JArray<jobject> *jobjectArray;
typedef JArray<jboolean> *jbooleanArray;
typedef JArray<jbyte> *jbyteArray;
typedef JArray<jchar> *jcharArray;
typedef JArray<jshort> *jshortArray;
typedef JArray<jint> *jintArray;
typedef JArray<jlong> *jlongArray;
typedef JArray<jfloat> *jfloatArray;
typedef JArray<jdouble> *jdoubleArray;
```

▶`template<class T> T *elements (JArray<T> &`*array*`)`◀ This template function can be used to get a pointer to the elements of the *array*. For instance, you can fetch a pointer to the integers that make up an `int[]` like so:

```
extern jintArray foo;
jint *intp = elements (foo);
```

The name of this function may change in the future.

You can create an array of objects using this function: ▶`jobjectArray JvNewObjectArray (jint` *length*`, jclass` *klass*`, jobject` *init*`)`◀ Here *klass* is the type of elements of the array; *init* is the initial value to be put into every slot in the array.

For each primitive type there is a function which can be used to create a new array holding that type. The name of the function is of the form `JvNew`*Type*`Array`, where *Type* is the name of the primitive type, with its initial letter in upper-case. For instance, `JvNewBooleanArray` can be used to create a new array of booleans. Each such function follows this example: ▶`jbooleanArray JvNewBooleanArray (jint` *length*`)`◀

▶`jsize JvGetArrayLength (jarray` *array*`)`◀ Returns the length of *array*.

Unlike Java, array bounds checking for C++ code is not automatic but instead must be done by hand.

5 Methods

Java methods are mapped directly into C++ methods. The header files generated by `gcjh` include the appropriate method definitions. Basically, the generated methods have the same names and "corresponding" types as the Java methods, and are called in the natural manner.

5.1 Overloading

Both Java and C++ provide method overloading, where multiple methods in a class have the same name, and the correct one is chosen (at compile time) depending on the argument types. The rules for choosing the correct method are (as expected) more complicated in C++ than in Java, but the fundamental idea is the same. Given a set of overloaded methods generated by `gcjh` the C++ compiler will choose the expected one, as long as each primitive Java type maps to a distict C++ type.

Common assemblers and linkers are not aware of C++ overloading, so the standard implementation strategy is to encode the parameter types of a method into its assembly-level name. This encoding is called *mangling*, and the encoded name is the *mangled name*. The same mechanism is used to implement Java overloading. The name mangling used by CNI must be the same as that generated by GCJ.

5.2 Instance methods

Virtual method dispatch is handled essentially the same in C++ and Java – i.e. by doing an indirect call through a function pointer stored in a per-class virtual function table. C++ is more complicated because it has to support multiple inheritance, but this does not affect Java classes. G++ historically used a different calling convention that was not compatible with the one used by GCJ.

The first two elements of the virtual function table are used for special purposes in both GNU Java and C++; in Java, the first points to the class that owns the virtual function table, and the second is used for an object descriptor that is used by the GC mark procedure.

Calling a Java instance method in CNI is done using the standard C++ syntax. For example:

```
  java::lang::Number *x;
  if (x->doubleValue() > 0.0) ...
```

Defining a Java native instance method is also done the natural way:

```
#include <java/lang/Integer.h>
jdouble
java::lang:Integer::doubleValue()
{
  return (jdouble) value;
}
```

5.3 Interface method calls

A Java class can *implement* zero or more *interfaces*, in addition to inheriting from a single base class. An interface is a collection of constants and method specifications. An interface provides a subset of the functionality of C++ abstract virtual base classes, but they are currently implemented differently. Since interfaces are infrequently used by Java native methods, we have not modified G++ to allow for method calls via interface pointers. In the future we might add an explicit mechanism to CNI to allow this.

5.4 Static methods

Static Java methods are invoked in CNI using the standard C++ syntax, using the '`::`' operator rather than the '`.`' operator. For example:

```
jint i =
  java::lang::Math::round((jfloat) 2.3);
```

Defining a static native method uses standard C++ method definition syntax. For example:

```
#include <java/lang/Integer.h>
java::lang::Integer*
java::lang::Integer::getInteger(jstring s)
{
  ...
}
```

5.5 Object allocation

New Java objects are allocated using a *class-instance-creation-expression*:

`new` *Type* `(` *arguments* `)`

The same syntax is used in C++. In both languages, the `new`-expression actually does two separate operations: Allocating an instance, and then running the instance initializer (constructor).

Using CNI, you can allocate a new object using standard C++ syntax. The C++ compiler is smart enough to realize the class is a Java class, and in that case generates a call to a run-time routine that allocates a garbage-collected object. If you have overloaded constructors, the compiler will choose the correct one using standard C++ overload resolution rules. For example:

```
java::util::Hashtable *ht
   = new java::util::Hashtable(120);
```

In G++, methods get passed an extra magic argument, which is not passed for Java constructors. G++ also has the constructors set up the vtable pointers. In Java, the object allocator sets up the vtable pointer, and the constructor does not change the vtable pointer. Hence, the G++ compiler needs to know about these differences.

Allocating an array is a special case, since the space needed depends on the run-time length given.

6 Sharing code for JNI and CNI

It would be nice to combne the advantages of CNI with the portability of JNI. That is people should be able to write native code that can be compiled for either CNI or JNI. This can be done using conditional compilation, with separate code for JNI and CNI, but of course that makes writing native methods even worse than plain JNI. It should be possible to use some pre-processing tricks to reduce the duplication, though.

It would be appealing to be able to write CNI code and automatically translate it to JNI code. However, recognizing where JNI calls are needed would require something with the sophistication of a C++ compiler. The logical thing would be to use a C++ compiler with a suitable option, say `--emit-jni`. Then instead of actually generating JNI C source, this compiler would generate machine code that calls the appropriate JNI routines. I.e. rather than actually getting JNI C code, you would get machine code equivalent to that generated from JNI C code. This is not quite as portable as pure JNI source, but it would be portable to all JVMs that run on platforms to which this compiler has been ported. For G++, that is almost all platforms in use.

Compiling CNI to JNI-using binaries might involve some combination of G++ changes, an extension to gcjh, and run-time code. Some ideas have been suggested, but there are no actual plans for such a project. Note that 100% automatic translation might be difficult, so you might have to put in `#ifdef JNI` conditionals occasionally, but the goal would be to minimize JNI-specific code.

7 Packages

The only global names in Java are class names, and packages. A *package* can contains zero or more classes, and also zero or more sub-packages. Every class belongs to either an unnamed package or a package that has a hierarchical and globally unique name.

A Java package is mapped to a C++ *namespace*. The Java class `java.lang.String` is in the package `java.lang`, which is a sub-package of `java`. The C++ equivalent is the class `java::lang::String`, which is in the namespace `java::lang`, which is in the namespace `java`.

The suggested way to do that is:

```
// Declare the class(es).
// (Possibly in a header file.)
namespace java {
  namespace lang {
    class Object;
    class String;
  }
}

class java::lang::String
  : public java::lang::Object
{
  ...
};
```

7.1 Leaving out package names

Having to always type the fully-qualified class name is verbose. It also makes it more difficult to change the package containing a class. The Java `package` declaration specifies that the following class declarations are in the named package, without having to explicitly name the full package qualifiers. The `package` declaration can be followed by zero or more `import` declarations, which allows either a single class or all the classes in a package to be named by a simple identifier. C++ provides something similar with the `using` declaration and directive.

A Java simple-type-import declaration:

`import` *PackageName*`.` *TypeName*`;`

allows using *TypeName* as a shorthand for *PackageName*`.`*TypeName*. The C++ (more-or-less) equivalent is a `using`-declaration:

`using` *PackageName*`::` *TypeName*`;`

A Java import-on-demand declaration:

`import` *PackageName*`.*;`

allows using *TypeName* as a shorthand for *PackageName*`.`*TypeName* The C++ (more-or-less) equivalent is a `using`-directive:

`using namespace` *PackageName*`;`

8 Exception Handling

It is a goal of the Gcc exception handling mechanism that it as far as possible be language independent. The Java features are again a subset of the G++ features, in that C++ allows near-arbitrary values to be thrown, while Java only allows throwing of references to objects that inherit from `java.lang.Throwable`. While G++ and GCJ share a common exception handling framework, things are not yet perfectly integrated. The main issue is that the "run-time type information" facilities of the two languages are not integrated.

Still, things work fairly well. You can throw a Java exception from C++ using the ordinary `throw` construct, and this exception can be caught by Java code. Similarly, you can catch an exception thrown from Java using the C++ `catch` construct.

Note that currently you cannot mix C++ catches and Java catches in a single C++ translation unit. This is caused by a limitation in GCC's internal processing of exceptions, and we do intend to fix this eventually.

C++ code that needs to throw a Java exception would just use the C++ throw statement. For example:

```
if (i >= count) {
  jstring msg =
    JvNewStringUTF("I/O Error!");
  throw new java::io::IOException(msg);
}
```

There is also no difference between catching a Java exception, and catching a C++ exception. The following Java fragment:

```
try {
  do_stuff();
```

```
} catch (java.IOException  ex) {
  System.out.println("caught I/O Error");
} finally {
  cleanup();
}
```

could be expressed this way in G++:

```
try {
  try {
    do_stuff();
  } catch (java::io::IOException* ex) {
     printf("caught I/O Error\n;");
  }
catch (...) {
  cleanup();
  throw;  // re-throws exception
}
```

Note that in C++ we need to use two nested `try` statements.

9 Exception Generation

Java code is extensively checked at runtime. For instance, if a Java program recurses too deeply, a `StackOverFlowException` is generated. Likewise, if a null pointer is dereferenced, a `NullPointerException` is generated. This is typically done by the Java runtime.

GCJ is currently weak on these checks. Explicit null pointer checks are generated in the specific case of calling a `final` function. In other cases we rely on the runtime to trap segmentation violations and turn them into `NullPointerException`. However, this approach only works on platforms with MMU support. In the future we plan to give gcj the ability to automatically generate explicit checks for null pointers and then generate the appropriate exception. When this happens we will most likely not modify the C++ compiler to do this, but will instead rely on the CNI programmer to add explicit checks by hand.

The same considerations apply to stack overflows.

10 Synchronization

Each Java object has an implicit monitor. The Java VM uses the instruction `monitorenter` to acquire and lock a monitor, and `monitorexit` to release it. The JNI has corresponding methods `MonitorEnter` and `MonitorExit`. The corresponding CNI macros are `JvMonitorEnter` and `JvMonitorExit`.

The Java source language does not provide direct access to these primitives. Instead, there is a `synchronized` statement that does an implicit `monitorenter` before entry to the block, and does a `monitorexit` on exit from the block. Note that the lock has to be released even the block is abnormally terminated by an exception, which means there is an implicit `try-finally`.

From C++, it makes sense to use a destructor to release a lock. CNI defines the following utility class.

```
class JvSynchronize() {
  jobject obj;

  JvSynchronize(jobject o)
  { obj = o; JvMonitorEnter(o); }

  ~JvSynchronize()
  { JvMonitorExit(obj); }
};
```

The equivalent of Java's:

```
synchronized (OBJ) { CODE; }
```

can be simply expressed:

```
{ JvSynchronize dummy(OBJ); CODE; }
```

Java also has methods with the `synchronized` attribute. This is equivalent to wrapping the entire method body in a `synchronized` statement. (Alternatively, an implementation could require the caller to do the synchronization. This is not practical for a compiler, because each virtual method call would have to test at runtime if synchronization is needed.) Since in `GCJ` the `synchronized` attribute is handled by the method implementation, it is up to the programmer of a synchronized native method to handle the synchronization (in the C++ implementation of the method). In otherwords, you need to manually add `JvSynchronize` in a `native synchronized` method.

11 Class Initialization

Java requires that each class be automatically initialized at the time of the first active use. Initializing a class involves initializing the static fields, running code in class initializer methods, and initializing base classes. There

may also be some implementation specific actions, such as allocating `String` objects corresponding to string literals in the code.

The GCJ compiler inserts calls to `JvInitClass` (actually `_Jv_InitClass`) at appropriate places to ensure that a class is initialized when required. The C++ compiler does not insert these calls automatically - it is the programmer's responsibility to make sure classes are initialized. However, this is fairly painless because of the conventions assumed by the GCJ system.

First, `libgcj` will make sure a class is initialized before an instance of that object is created. This is one of the responsibilities of the `new` operation. This is taken care of both in Java code, and in C++ code. (When the G++ compiler sees a `new` of a Java class, it will call a routine in `libgcj` to allocate the object, and that routine will take care of initializing the class.) It follows that you can access an instance field, or call an instance (non-static) method and be safe in the knowledge that the class and all of its base classes have been initialized.

Invoking a static method is also safe. This is because the Java compiler adds code to the start of a static method to make sure the class is initialized. However, the C++ compiler does not add this extra code. Hence, if you write a native static method using CNI, you are responsible for calling `JvInitClass` before doing anything else in the method (unless you are sure it is safe to leave it out).

Accessing a static field also requires the class of the field to be initialized. The Java compiler will generate code to call `_Jv_InitClass` before getting or setting the field. However, the C++ compiler will not generate this extra code, so it is your responsibility to make sure the class is initialized before you access a static field.

12 Changes to G++

Here is a summary of changes made to G++, the GNU C++ compiler, to make it aware of Java types, and thus provide the C++/Java interoperability we have discussed:

- For each Java primitive type (such as `long`), G++ defines a C++ primitive type, with a name like `__java_long`. (A name with two initial underscore is reserved to the implementation, so it cannot clash with a valid user identifier.) These types are distinct from all other types. The CNI header files will then do:

  ```
  typedef __java_long jlong;
  ```

  This mechanism makes it easy for the compiler to distinguish Java types from standard C++ types, giving them the correct size in bits. Each of these types has a special "Java type" flag bit set. When mangling the name of a method into an assembler label, Java types are recognized. For example `__java_long` is mangled the same as the C++ 64-bit integer type `long long`.

- The compiler recognizes `extern "Java"` in addition to the standard `extern "C"` and `extern "C++"`. If a class is defined inside the scope of `extern "Java"`, then the compiler set the "Java type" bit on the class and the corresponding pointer type. The "Java type" bit is also set for a class if its base class has the "Java type" bit set. The standard libgcj/CNI header files define `java::lang::Object` inside `extern "Java"`; thus all classes that inherit from `java::lang::Object` have the "Java type" bit set.

- There is a compiler internal function that checks if a type is a valid Java type. This is used to catch errors such as when a programmer accidentally types `long` instead of `jlong` in the function header of a Java class.

- The compiler has an internal function that takes a Java type, and generates a declartion that refers to the corresponding (run-time) `java.lang.Class` instance. This is used for exception handling and object allocation.

- If when compiling a `new`-expression the allocated type is a Java type, then the compiler generates a call to the run-time routine `_Jv_AllocObject`. The compiler also suppresses generating code to cause the object to be de-allocated if the constructor throws an exception. (Such de-allocation is mandated by the C++ standard, but is not correct for Java, which assumes a garbage collector.)

- The interface to constructors needs to be changed so magic vtable pointer initialization and the extra constructor argument do not happen when constructing a Java object.

- C++ has the problem that the compiler cannot tell which compilation unit needs to emit a class's virtual function table. Various rules and heuristics are

used, but sometimes the same vtable has to be emitted by more than one compilation unit. This is not an issue for Java types: G++ never emits the vtable, since that is done when GCJ compiles the Java class.

- G++ handles `catch` and `throw` by generating appropriate libgcj calls.

13 Performance numbers

Figure 1 shows some benchmark numbers comparing JNI and CNI. (The benchmarks are based on code by Matt Welsh.) (ADD MORE FOR FINAL PAPER.) They are a number of micro-benchmarks, all run on a 600 MHz Athlon running RedHat Linux 7 with a pre-2.4 kernel. Each column measures:

1. Running JDK 1.3 from Sun, calling a test method written in pure Java.
2. Again running JDK 1.3 from Sun, with the test method written using in JNI. For those rows that have two numbers, the second number is when using cashing of `jfieldID`s.
3. Using GCJ (version shipped with RedHat 7), calling a test method written in pure Java.
4. Again using GCJ (version shipped with RedHat 7), calling a test method written in C++ using CNI.

14 Usage and features of gcjh

The gcjh is used to generate C++ header files from Java class files. By default, gcjh generates a relatively straightforward C++ header file. However, there are a few caveats to its use, and a few options which can be used to change how it operates. We don't list the options here, as they aren't too relevant to this discussion.

gcjh will generate all the required namespace declarations and `#include`'s for the header file.

Gcjh also has the ability to decompile simple Java methods. Currently it does this via ad hoc pattern matching – it recognizes empty methods, accessor methods, and methods which return `this` or `null`. These choices were made primarily because they were simple to implement, but also because they seemed likely to provide good inlining opportunities. This feature in effect implements cross-language inlining, and we anticipate more work here in the future.

gcjh has to handle mismatches between the C++ and Java programming languages. There are a few such problems:

- gcjh must be careful to generate fields and methods in the same order as the compiler itself so that the G++ view of object layout is compatible with the GCJ view. This means that gcjh has to be updated whenever either compiler changes its object layout.
- Some valid Java identifiers, like `register`, are keywords in C++. We handle this case in gcjh with some collusion by gcj. We add a `$` to the end of any name that conflicts with a C++ keyword. For instance, a method `typename` appears in the header, and in the generated object file, as `register$`. This encoding is robust – if the Java code has a method named `struct$`, it is renamed to `struct$$`. Thus, conflicts are never possible.
- In C++ it isn't possible to have a field and a method with the same name, while in Java this is possible. If the conflicting field is static, gcjh simply issues an error. Otherwise, the field will be renamed by appending '`_`' in the generated header; this is safe because object code that refers to instance fields will not use the field's name. In the future we plan to eliminate this problem. One approach would be to modify gcj and gcjh to mangle conflicting field names as is done for identifiers matching C++ keywords.
- In Java it is often convenient to refer to a Class object, for instance via the `Foo.class` syntax. In C++ there is no way to do this, so gcjh makes the class object appear to be a static field of the class itself. This field is named `class$`. Typical code uses a pointer to the Class object, e.g. `&java::lang::Class::class$`.
- gcjh assumes that all the methods and fields of a class have ASCII names. The C++ compiler cannot correctly handle non-ASCII identifiers. gcjh does not currently diagnose this problem. In the future we hope to generate headers which use C++ UCNs (the C++ equivalent of Java `\u` escapes) to avoid this problem.

gcjh can automatically generate C++ stubs for a given class. For each native method in the class it will generate a C++ stub whose default implementation throws

Test	JDK pure Java	JDK with JNI	GCJ pure Java	cj with CNI
Void no-op method	.027μs	.077μs	.026μs	.028μs
Instance field increment	.06μs	2.88μs / .35μs	.02μs	.02μs
Static field increment	.033μs	4.39μs / .395μs	.027μs	.029μs

Figure 1: Benchmark measurements

```
// DO NOT EDIT THIS FILE - it is machine generated -*- c++ -*-

#ifndef __timing_Timer__
#define __timing_Timer__

#pragma interface

#include <java/lang/Object.h>

extern "Java"
{
  namespace timing
  {
    class Timer;
  }
};

class ::timing::Timer : public ::java::lang::Object
{
public: // actually package-private
  virtual jlong sinceLast (::java::lang::String *);
  Timer ();
private:
  jlong lastTime;
  ::java::lang::String *lastNote;
public:

  static ::java::lang::Class class$;
};

#endif /* __timing_Timer__ */
```

Figure 2: `Timer.h` as gnerated by gjch

an exception. This feature is convenient when writing a class for the first time.

An alternative to a separte gcjh program would be modifying G++ so it could read `.class` files directly. However, using gcjh is almost as convenient, and G++ is already very complex, so adding significant Java-specific changes seems ill-advised from the perspective of long-term compiler maintenance..

Figure 2 is the output of gcjh on the Timer class from the first section.

References

[1] Per Bothner. *A Gcc-based Java Implementation*. IEEE Compcon '97, 1997.

[2] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, .