gcj: Compile Java into Native Machine Code

Technical Paper

Craig Delger <cdelger@redhat.com>
Mark G. Sobell <sobell@redhat.com>

version 1.00
11 December 2000

Abstract

The GNU Compiler for Java (**gcj**) is an open-source compiler maintained by the Free Software Foundation. It uses the **libgcj** libraries. Version **libgcj-2.96-22** is compatible with Sun’s JDK 1.1 with a few exceptions including AWT and inner classes. Many people are requesting AWT support. It is one of the first priorities for the next release. It also does not have full locale support (internationalization) at this time; it just supports the US locale.

The GNU Compiler for Java (**gcj**) improves the performance of Java code on Linux systems. By compiling into machine code instead of Java byte code your Java programs will not only run faster but leave more CPU cycles available for other tasks.

Despite the fact that compiling Java code to yield native machine code is still in the early stages, the reliability of this process is shown by the number of companies that are already using it to create commercial solutions.

This paper is an introduction to development using **gcj** on an IA32 (Intel Architecture, 32-bit bus, formerly referred to as x86) platform. It helps you set up your development environment and walks you through compiling and debugging simple Java applications using **gcj** and **gdb**.

You can use the same development process with a suitably configured GNU cross-compilation toolchain for an embedded target (**gcj**, **gcc**, **gas**, and **binutils** built to generate code for ARM, PowerPC, MIPS or other supported targets). This approach is ideal for embedded systems as they tend to be resource constrained. A virtual machine may not be readily available for a target system and may consume too much code space, operating memory, and CPU cycles.

Red Hat Linux is a powerful environment for developing and deploying Java programs. Numerous Java development tools and virtual machines are available on Linux. One of the most interesting, **gcj**, is a free, open-source compiler that compiles Java source code or Java byte code into native machine code. A Java program compiled into native machine code runs faster than Java byte code on a Java virtual machine and uses less memory.

This paper assumes that you have a working knowledge of a UNIX-like operating system. If you do not, refer to introductory reference material and explore your shell account. Refer often to **man** and **info** pages: Give the command **man bash** or **info bash** to learn more about **bash** (the Bourne Again Shell). When you see unfamiliar commands use the associated **man** or **info** command to learn more. Refer to the **gcj** home page at **http://sources.redhat.com/java/** and the link to the **gcj** FAQ on that page for helpful hints.

Overview

This tutorial uses the following tools:

- Java compiler: **gcj**
- Java library: **libgcj**
- Java debugger: **gdb**
- An editor of your choice (**vi**, **emacs**, or other)

Each of these tools is included with Red Hat Linux 7. After you understand how to use these tools, explore their many options. With this understanding you can build your preferred development environment. Red Hat Linux and the Linux Applications Library include many additional developer tools that are shipped with the Red Hat boxed CD set and are also available on the Red Hat Web site (see the following section).

The next section starts this tutorial's **gcj** development project.

The Environment

Before you start make sure that you have the following packages (RPMs) installed. Later versions will work, earlier versions will not. Each of the packages is included with Red Hat Linux 7, but may not be installed.

If you have a Red Hat Linux 7 installation CD and one of the packages is not installed, you can install it from the RPMS directory on the CD. Without this disk you can use your Web browser to download the package(s) from **http://www.redhat.com/apps/download/**.[1] Enter the name of the utility (dropping everything from the first hyphen on (enter **binutils** instead of **binutils-2.10.0.18-1.i386.rpm** in the `by keyword` window on **http://www.redhat.com/apps/download/** and you will find the latest version of the package).

Use the following command format to check for the presence of a package on your system and for the version number. Replace ***binutils*** with the name of the package you are looking for.

```
$ rpm -q binutils
binutils-2.10.0.18-1
```

The version numbers in the following list are current as of the date of this paper. You must install **libgcj** before you install **gcc-java**.

- **cpp-2.96-54.i386.rpm** is the C-Compatible Compiler Preprocessor. The C compiler uses this preprocessor to transform your program

1. If you cannot access the Red Hat Public File Server (because it currently has the maximum number of users it can handle) then try one of the mirror sites listed at **http://www.redhat.com/download/mirror.html**.

prior to compiling it. This program is called a macro processor because it allows you to define macros, which are abbreviations for longer constructs.

- **libgcj-2.96-22.i386.rpm** is the Java runtime library.
- **gcc-java-2.96-54.i386.rpm** adds support for compiling Java programs and bytecode into native code. (For consistency you might expect this package to be named **gcj-*.rpm**.)
- **libgcj-devel-2.96-22.i386.rpm** is the Java compiler development environment
- **gdb-5.0-7.i386.rpm** is the **gdb** full-featured, command-driven debugger. **gdb** allows you to trace the execution of programs and examine their internal state at any time. It works with C and C++ programs that have been compiled with **gcc**, the GNU C compiler.
- **binutils-2.10.0.18-1.i386.rpm** is a collection of binary utilities.

Before trying to install these libraries make sure that the RPM files are in your working directory and that you are working as **root** (Superuser). Give the following commands (omit the **#** which is the **root** prompt)

(You can abbreviate the filenames to a unique prefix followed by an asterisk (*****). For example you can probably abbreviate the argument to **binutil*** in the first command.)

```
# rpm -Uvh binutils-2.10.0.18-1.i386.rpm
# rpm -Uvh gcc-java-2.96-54.i386.rpm
# rpm -Uvh gdb-5.0-7.i386.rpm
# rpm -Uvh libgcj-2.96-22.i386.rpm
# rpm -Uvh libgcj-devel-2.96-22.i386.rpm
# exit
$
```

When you are finished installing these programs exit from the Superuser shell or logout and login again as yourself so that you are once again running as a regular user. To test that **gcj** is properly installed give the following command. You should see the response shown below.

```
$ gcj
gcj: No input file.
```

Compiling

The following example creates, compiles, and runs the classic K&R `Hello world!` program in Java. Although you can create this file in any of your directories it can be helpful to work in an empty directory. Start by creating a directory named **helloworld** (**mkdir helloworld**), change directories so that you are working in the new directory (**cd helloworld**), and use a text editor to create and save a file named **helloworld.java** that contains all but the first of the following lines:

```
$ cat helloworld.java
/**
* The helloworld class implements an application that
* displays "Hello World!" to the standard output.
**/
class helloworld {
   public static void main(String[] args) {
    System.out.println("Hello World!"); //Display the string.
  }
}
```

NOTE
On some systems the **CLASSPATH** variable may be set. In order for the example programs in this paper to work this variable must *not* be set. Give the following command to unset it:

```
$ unset CLASSPATH
```

The following command invokes **gcj** to compile and link the **helloworld.java** source file. The command links the compiled code with the **gcj** runtime, the **libgcj** package, which consists of the core class libraries, a garbage collector library, an abstraction over the system threads, and, optionally, a bytecode interpreter. The addition of the bytecode interpreter means that **gcj** compiled applications can dynamically load and interpret class files, resulting in mixed compiled/interpreted systems.

```
$ gcj --main=helloworld -o helloworld helloworld.java
```

The **––main=helloworld** option generates a stub[2] so that the application starts executing with the main method of the class named. In this example the application should execute the main method of the **helloworld** class. The **–o helloworld** option tells the **gcj** compiler to name the executable file **helloworld**. The **helloworld.java** argument is the name of the source file that **gcj** is compiling. Errors are displayed by **gcj** on the screen.

Test your program by giving the following command. If you do not see the words `Hello World!` then check your program:

```
$ ./helloworld
Hello World!
```

Once you get this program to work you have compiled and run a Java program that executes as native machine code.

The next program demonstrates how to use **gdb** to debug a Java program that has been compiled using **gcj**.

2. *Stub:* A dummy procedure that is named in a program to prevent an undefined label error when the program is linked with a run-time library.

Using *gdb* to Debug a Java Program

The **gdb** utility is the GNU command-line debugger that evaluates binary (object) files that contain compiled-in debugging symbols. The next example covers the basic **gdb** operations of setting breakpoints and stepping through the running code. To follow this example you can either use the working directory or create a new directory named **employee** and use **cd** to make it your working directory. Create the file named **Employee.java** with the following contents:

```
$ cat Employee.java
class Employee{
    String lastname;
    String firstname;
    String title;
    boolean status;

    void showAttributes(){
        System.out.println("Employee name: " + firstname + " " + lastname);
        System.out.println("Employee title: " + title);
        if (status == true)
            System.out.println("Employee status = active");
        else
            System.out.println("Employee status = inactive");
    }

    public static void main (String arguments[]){
        Employee e = new Employee();
        e.lastname = "Smith";
        e.firstname = "Joe";
        e.title = "Finance";
        e.status = true;
        e.showAttributes();
    }
}
```

This program does not do much, but provides an example for learning how to use **gdb**. Use the **–g** argument to compile debugging symbols into the object code and an uppercase `E` as shown (because the name of the class begins with an `E`):

```
$ gcj -g --main=Employee -o Employee Employee.java
```

Now, call **gdb** with the name of the program, **Employee**, as an argument:

```
$ gdb Employee
.
.
.
(gdb)
```

After several lines of information you will see the prompt: `(gdb)`. The **gdb** utility can tell when a program it is debugging receives a signal. You can instruct **gdb** in advance how to respond to each type of signal.

Some versions of **glibc** use the SIGPWR and SIGXCPU signals in the implementation of POSIX threads for Linux. If you do not handle these signals, the debugger may stop when, from your point of view, nothing has happened. You can give a **continue** (**c**) command when this happens or you can use the following commands to handle the **SIGPWR** and **SIGXCPU** signals:

```
(gdb) handle SIGPWR nostop noprint
Signal        Stop      Print   Pass to program Description
SIGPWR        No        No      Yes             Power fail/restart
(gdb) handle SIGXCPU nostop noprint
Signal        Stop      Print   Pass to program Description
SIGXCPU       No        No      Yes             CPU time limit exceeded
```

To set a break point on line 20, type **break** (or just **b**), followed by the name of the source file, a colon, and the line number from the source file:

```
(gdb) break employee.java:20
Breakpoint 1 at 0x804bcfd: file Employee.java, line 20.
```

When you run the program it stops at the breakpoint:

```
(gdb) run
Starting program: /home/cdelger/employee/Employee
[New Thread 1024 (LWP 2080)]
[New Thread 2049 (LWP 2081)]
[New Thread 1026 (LWP 2082)]
[Switching to Thread 1026 (LWP 2082)]

Breakpoint 1, Employee.main (arguments=@8083ff0) at Employee.java:20
20              e.title = "Finance";
Current language:  auto; currently java
```

Now step through the program by giving the command **next (n)** to execute the next line of source code.

```
(gdb) next
21              e.status = true;
```

To view the current value of a variable, give the command **print** (**p**) followed by the variable name. The following command displays the value of the **lastname** attribute of the **Employee** object **e**:

```
(gdb) print e.lastname
$1 = java.lang.String "Smith"
```

Use the **backtrace** command to display a history of the execution of your program. This command lists the entire stack:[3] one line per frame[4] for all frames in the stack. The commands **where** and **info stack** are aliases for **backtrace**:

3. *Stack:* A data structure that stores items that are accessed in last-in first-out (LIFO) order. Data items are said to be “pushed onto” and “popped off of” a stack.

4. *Stack frame* or *activation record:* A data structure containing the variables belonging to one particular scope incarnation of a function.

```
(gdb) backtrace
#0  Employee.main (arguments=@8083ff0) at Employee.java:21
#1  0x401cf23d in gnu::gcj::runtime::FirstThread::run (this=@8065ea0)
    at ../../../libjava/gnu/gcj/runtime/natFirstThread.cc:146
#2  0x401d8d0a in java::lang::Thread::run_ (obj=@8065ea0)
    at ../../../libjava/java/lang/natThread.cc:263
#3  0x401e930d in really_start (x=@808dff8)
    at ../../../libjava/posix-threads.cc:344
#4  0x403064d1 in GC_start_routine () from /usr/lib/libgcjgc.so.1
#5  0x4031fc87 in pthread_start_thread_event (arg=@bf7ffc00)
    at manager.c:274
```

The stack trace includes both Java code and the native methods in the **libgcj** runtime library.

The **libgcj** library is implemented as a mixture of Java and C++ code because it is not always possible to implement all of Java using Java [for example low level system routines must be accessed via native (C++) code]. The Java core classes provide the user with Java methods that are wrappers to the native code, minimizing the need for the average Java programmer to write in native code. The stack trace includes both Java code and C++ code (the *native methods*) from the **libgcj** runtime library.

Java classes can interact with non-Java code in 2 ways: using the standard **JNI**[5] methods or more seamlessly via **CNI**. The **gcj** compiler uses **CNI** because it lets you treat compiled Java as though it were C++ because object layouts are the same and the ABI (application binary interface) is the same. For example **CNI** code can make method calls on Java objects as though they were C++ objects.

A program may have more than one thread[6] of execution. To examine threads use the **info threads** and the **threads** commands. The **info threads** command lists the existing threads and their ID numbers:

```
(gdb) info threads
* 3 Thread 1026 (LWP 2082)  Employee.main (arguments=@8083ff0)
    at Employee.java:21
  2 Thread 2049 (LWP 2081)  0x40410670 in __poll (fds=0x80a2e6c, nfds=1,
    timeout=2000) at ../sysdeps/unix/sysv/linux/poll.c:63
  1 Thread 1024 (LWP 2080)  0x4036c722 in __sigsuspend (set=0xbffff760)
    at ../sysdeps/unix/sysv/linux/sigsuspend.c:45
```

An asterisk (*****) to the left of the **gdb** thread number marks the current thread. To change the current thread, give the **thread** command followed by the id of the thread you want to switch to. The next example changes to thread ID number 1:

5. JNI stands for Java Native Interface and CNI stands for C/C++ Native Interface. JNI is Sun's spec and has a fair bit of overhead to use (from the programmer's standpoint). CNI allows for a more seamless integration of Java code with native code (that is, the native code can refer to Java objects directly rather than having to call JNI methods).
6. A portion of a program that can run independently of and concurrently with other portions of the same and other programs.

```
(gdb) thread 1
[Switching to thread 1 (Thread 1024 (LWP 2080))]
#0  0x4036c722 in __sigsuspend (set=0xbffff760)
    at ../sysdeps/unix/sysv/linux/sigsuspend.c:45
45      ../sysdeps/unix/sysv/linux/sigsuspend.c: No such file or directory.
Current language:  auto; currently c
```

To verify that you have switched threads, give the **info threads** command again:

```
(gdb) info threads
  3 Thread 1026 (LWP 2082)  Employee.main (arguments=@8083ff0)
    at Employee.java:21
  2 Thread 2049 (LWP 2081)  0x40410670 in __poll (fds=0x80a2e6c, nfds=1,
    timeout=2000) at ../sysdeps/unix/sysv/linux/poll.c:63
* 1 Thread 1024 (LWP 2080)  0x4036c722 in __sigsuspend (set=0xbffff760)
    at ../sysdeps/unix/sysv/linux/sigsuspend.c:45
```

The asterisk to the left of the thread number 1 indicates that it is the current thread.

If you want to run through the rest of the program without interruption remove the existing break point at line 20 with the **clear** command:

```
(gdb) clear 20
No breakpoint at 20.
```

Now run through the rest of the program using the **continue** (**c**) command:

```
(gdb) continue
Continuing.
Employee name: Joe Smith
Employee title: Finance
Employee status = active

Program exited normally.
(gdb) quit
$
```

There are many ways to run **gdb**. To learn more, read the **man** page for **gdb** (the **man** page refers to additional resources). Typing help in **gdb** displays a concise list of commands.

Graphical front ends to **gdb**, such as Insight and DDD, can make **gdb** easier to use. The **emacs** utility has integrated **gdb** functionality that allows you to run **gdb** in one window while displaying the source code, including an indication of the line being executed in a second window.

Conclusion

This paper covered the use of **gcj** and **libgcj** to compile Java source code into native machine code under Linux. It also provided examples of running and debugging (using **gdb**) the resulting object code.

There are many resources with information about the tools covered in this paper. With the information you now have you should be able to use these resources to expand your knowledge. Some good places to start looking for information are:

- **http://www.redhat.com/devnet** The Red Hat Developer Network (RHDN) home page.
- **http://sources.redhat.com** News about free software.
- **http://www.redhat.com/devnet** Ask questions at The Java Developer Forum.
- **http://sources.redhat.com/java/** The **gcj** home page. The FAQ listed on this page is very helpful.
- **http://www.gnu.org/manual/gdb/html_mono/gdb.html** The GNU Source-Level Debugger manual.
- **http://www.redhat.com/embedded** Discusses Red Hat commercial tools and custom engineering for embedded systems.