Miscompilation of Gaussian by the Portland Group Fortran Compiler

Roger Sayle[1] and Marcus Daniels[2]

[1] NextMove Software, Santa Fe, New Mexico, USA.
[2] CCS Division, Los Alamos National Labs (LANL), Los Alamos, New Mexico, USA.

2nd September 2010

Introduction

Because Gaussian's Fortran source code can't be compiled by GNU gfortran, the native fortran compiler on most Linux distributions, users of Gaussian on Linux are forced to use one of the two commercial compilers available for Linux. As a result, a significant fraction of Guassian/Linux binary executables are built using the Portland Group's compiler. This whitepaper describes an analysis of a miscompilation of the Gaussian 03 source code by the Portland Group's pgfortran compiler, version 10.4.

Description

As mentioned above, the Gaussian 03 source code can't be compiled by GNU gfortran. The reason for this is that the Fortran code makes use of a number of language idioms that are not defined by the International "J3" Fortran standards technical committee. This is the standards body responsible, in collaboration with ISO/IEC/JTC1/SC22/WG5, for defining Fortran language standards, including Fortran 66, Fortran 77, Fortran 90, Fortran 95, Fortran 2003 and Fortran 2008. This means that g03 is not standards conforming, and should be rejected by a strict compiler, as indeed it is by IBM's xlf compiler and GNU gfortran.

However, many Fortran compilers support so-called "legacy" extensions. Because many Fortran programs were developed before the advent of standards, or using lax compilers, many compilers support language constructs beyond those defined explicitly in the standards, in order to compiler older codes without modification. Indeed, the author is responsible for adding the "-std=legacy" and "-std=gnu" command line options to gfortran to allow handling of discouraged and acceptable language extensions respectively.

In this case, the problematic code looks like:

```
INTEGER*8 I, J, K, MASK
I = LSHIFT(J,32) .OR. (K .AND. MASK)
```

Where the variables I, J, K and MASK are declared as integers. This statement is clearly attempting to perform a bit manipulation operation, equivalent to the C code:

```
I = (J<<32) | (K & MASK)
```

Unfortunately, the Fortran standard(s), explicity states that the operands of the intrinsic operators ".and.", ".or.", ".not" and ".eqv." must be of LOGICAL type. Indeed, the compilation error from GNU fortran, “Error: Operands of logical operator .and. at (1) are INTEGER(8)/INTEGER(8)”, and the equivalent from IBM’s xlf “1516-042 (S) Operand must be type LOGICAL” both diagnose the problem.

To correctly (portably) write this statement in Fortran, it needs to be written as:

```
I = IOR(LSHIFT(J,32),IAND(K,MASK))
```

This form uses the binary intrinsic functions IAND and IOR.

Obviously, because Gaussian can be built by a number of compilers, this code must be accepted by them as an extension. Indeed, the Intel ifort v10.0.023 compiler, the PathScale pathf90 and PGI pgfortran v10.4 compilers all except the code without warning.

The trouble is that by being undefined by a language standard each of these compilers was free to interpret the semantics of the extension differently. Naturally, the author’s of Gaussian assumed that the operators would be treated as the intended bit-wise operations. And fortunately, this is what both Intel and PathScale do. The Portland Group, on the other hand, continues to interpret the operators as logical, so instead perform implicit conversions of the INTEGER arguments to LOGICAL. This compiles the code as equivalent to C’s

```
I = (J << 32) || (K && MASK)
```

or in FORTRAN as

```
I = ((J << 32) .NE. 0) .OR. ((K .NE. 0) .AND. (MASK .NE. 0))
```

All in all wrong.

Bibliography