Eliminando el Desorden - Llamado a Subrutinas*

José Enrique Alvarez Estrada

Translated By:

José Enrique Alvarez Estrada

Based on *Eliminating Clutter - Subroutine Calls*† by

Charles Severance

Kevin Dowd

Una empresa típica está llena de aterrorizantes ejemplos de sobrecarga. Digamos que un departamento ha preparado una pila de papelería, que debe completar otro departamento. ¿Qué debe usted de hacer para transferir tal trabajo? Primero, debe asegurarse que la parte que le tocó hacer esté completa; no puede pedirles que lo acepten si los materiales que requieren no están listos. Después, necesitará empaquetar los materiales - datos, formas, números de cuenta, etc. Y finalmente viene la transferencia oficial. Una vez que recibió lo que usted les envió, el otro departamento debe desempacarlo, hacer su trabajo, reempaquetarlo y enviarlo de vuelta.

Se malgasta una gran cantidad de tiempo moviendo trabajos entre departamentos. Por supuesto, si la sobrecarga es mínima comparada con la cantidad de trabajo útil que se hace, no hay gran problema. Pero pudiera ser más eficiente llevar a cabo los trabajos pequeños adentro del mismo departamento. Lo mismo puede afirmarse de la invocación de subrutinas y funciones. Si sólo entra y sale de los módulos muy de vez en cuando, la sobrecarga de guardar los valores en los registros y preparar la lista de argumentos no resultará significativa. Sin embargo, si está invocando repetidamente unas pocas subrutinas pequeñas, la sobrecarga puede mantenerlas siempre encabezando la lista del perfil. Puede que fuera mejor si el trabajo permaneciera donde está, en la subrutina emisora de la llamada.

Adicionalmente, los llamados a subrutinas inhiben la flexibilidad del compilador. Si se presenta la oportunidad, usted quiere que su compilador tenga la flexibilidad suficiente para entremezclar instrucciones que no son dependientes las unas de las otras. Y tales instrucciones se encuentran en ambas caras de un llamado a subrutina, en el invocador y en el invocado. Pero la oportunidad se pierde cuando el compilador no puede emparejarlas dentro de subrutinas y funciones. Y así, instrucciones que perfectamente pudieran solaparse, permanecen en sus respectivos lados de la barrera artificial.

*Version 1.2: Oct 19, 2011 8:27 am -0500
†http://cnx.org/content/m33721/1.3/
‡http://creativecommons.org/licenses/by/3.0/

Puede ser de ayuda que ilustremos el reto que representan las fronteras de las subrutinas mediante un ejemplo exagerado. Los siguientes bucles se ejecutan muy bien en una amplia variedad de procesadores:

```
DO I=1,N
  A(I) = A(I) + B(I) * C
ENDDO
```

El código de abajo realiza los mismos cálculos, pero observe qué hemos hecho:

```
DO I=1,N
  CALL MADD (A(I), B(I), C)
ENDDO
SUBROUTINE MADD (A,B,C)
A = A + B * C
RETURN
END
```

Cada iteración invoca a una subrutina para realizar una pequeña cantidad de trabajo, que antes estaba ubicado adentro del ciclo. Este es un ejemplo particularmente doloroso, porque involucra cálculos de punto flotante. La pérdida de paralelismo resultante, junto con la sobrecarga debida al llamado al procedimiento, puede producir código que se ejecuta 100 veces más lento. Recuerde, tales operaciones se colocan en líneas de espera, y toma cierta cantidad de tiempo "de recuperación" antes de que el rendimiento alcance siquiera una operación por ciclo de reloj. Si son pocas las operaciones de punto flotante que se realizan entre llamados a subrutinas, el tiempo gastado en llenar y vaciar las líneas de espera se volverá muy importante.

La invocación de subrutinas y funciones complica la habilidad que tiene el compilador de manejar eficientemente variables `COMMON` y `external`, retrasando su almacenamiento en memoria hasta el último momento posible. El compilador usa registros para almacenar los valores "vivos" de muchas variables. Cuando realiza una llamada, el compilador no puede saber cuáles de las variables cambiarán en la subrutina que está declarada como `external` o `COMMON`, y por ello se ve forzado a almacenar en memoria cualquier variable `external` o `COMMON` que haya sido modificada, de modo que la subrutina invocada pueda encontrarla. De igual modo, tras el retorno de la llamada, las mismas variables tienen que cargarse nuevamente en los registros, porque el compilador no puede confiar en las copias antiguas que residen en los registros. La penalización de guardar y recuperar variables puede ser substancial, especialmente si son muchas. También puede resultar riesgoso si las variables que debieran ser locales fueron especificadas como `external` o `COMMON`, como sucede en el siguiente código:

```
COMMON /USELESS/ K
DO K=1,1000
  IF (K .EQ. 1) CALL AUX
ENDDO
```

En este ejemplo, K se ha declarado como una variable `COMMON`. Sólo se usa como contador del ciclo do-loop, así que realmente no hay razón para que sea otra cosa que local. Sin embargo, dado que está en un bloque `COMMON`, la llamada a `AUX` obliga al compilador a almacenar y recargar `K` cada iteración. Ello se debe a que se desconocen los efectos laterales que la llamada puede producir.

Hasta aquí, pareciera como si estuviéramos abonando el camino para ¡programas principales enormes, sin subrutinas o funciones! En absoluto. La modularidad es importante para mantener compacto y comprensible el código fuente. Y francamente, la necesidad de modularidad y facilidad de mantenimiento siempre es más importante que la necesidad de *pequeñas* mejoras de rendimiento. Sin embargo, hay un par de enfoques que permiten racionalizar los llamados a subrutinas, y que no requieren que usted abandone las técnicas de codificación modular: se trata de las macros y los procedimientos en línea.

Recuerde, si la función o subrutina hace una cantidad razonable de trabajo, no importará demasiado la sobrecarga debida a la invocación del procedimiento. Sin embargo, si una rutina pequeña aparece como un nodo hijo en una de las secciones más atareadas del grafo de llamados, puede que deba pensar en insertarla en los lugares apropiados del programa.

1 Macros

Las *Macros* (o macroinstrucciones) son pequeños procedimientos que se substituyen en línea a tiempo de compilación. Al contrario que las subrutinas o funciones, que se incluyen una vez durante el proceso de enlazado, las macros se replican en cada lugar que se usan. Cuando el compilador lleva a cabo su primera pasada a lo largo de su programa, busca patrones que coincidan con las definiciones previas de macros, y las expande en línea. De hecho, en etapas posteriores, el compilador ve una macro expandida como si fuera código fuente creado por usted.

Las macros forman parte tanto de C como de FORTRAN (aunque la noción equivalente a una macro en FORTRAN, la *función sentencia*, ha sido vilipendiada por la comunidad FORTRAN y no sobrevivirá mucho tiempo más).[1] En los programas en lenguaje C, las macros se crean mediante una directiva `#define`, como se demuestra aquí:

```
#define promedio(x,y) ((x+y)/2)
main ()
{
   float q = 100, p = 50;
   float a;
   a = promedio(p,q);
   printf ("%f\n",a);
}
```

El primer paso de compilación de un programa en C consiste en pasar por el preprocesador de C, *cpp*. Ello ocurre automáticamente cuando usted invoca al compilador. *cpp* expande las sentencias `#define` en línea, reemplazando el patrón coincidente por la definición de la macro. En el programa de arriba, la sentencia:

```
a = promedio(p,q);
```

[1] La función sentencia se ha eliminado de FORTRAN 90.

es reemplazada por:

```
a = ((p+q)/2);
```

Debe usted ser cuidadoso al definir la macro, porque literalmente reemplaza al patrón que *cpp* encuentra. Por ejemplo, si la definición de la macro decía:

```
#define multiplicar(a,b) (a*b)
```

y usted lo invocó como:

```
c = multiplicar(x+t,y+v);
```

la expansión resultante será `x+t*y+v` — que probablemente no es lo que usted pretendía.

Si es usted un programador en C, puede que esté usando macros sin siquiera percatarse. Muchos archivos de cabecera en C (*.h*) contienen definiciones de macros. De hecho, algunas funciones "estándar" de biblioteca en C en realidad son macros contenidas en tales archivos. Por ejemplo, la función *getchar* puede enlazarse cuando construye su programa. Si tiene una sentencia:

```
#include <stdio.h>
```

en su archivo, *getchar* se reemplazará con una definición de macro a tiempo de compilación, reemplazando la función de biblioteca de C.

También puede usted hacer que las macros de *cpp* trabajen para los programas en FORTRAN.[2] Por ejemplo, una versión FORTRAN del programa en C anterior pudiera verse así:

```
#define PROMEDIO(X,Y) ((X+Y)/2)
C
      PROGRAM MAIN
      REAL A,P,Q
      DATA P,Q /50.,100./
      A = PROMEDIO(P,Q)
      WRITE (*,*) A
      END
```

[2] Algunos programadores prefieren usar para FORTRAN el preprocesador estándar de UNIX, *m4* .

Sin algo de preparación, la sentencia `#define` será rechazada por el compilador de FORTRAN. El programa debe preprocesarse previamente mediante *cpp* para reemplazar el uso de `PROMEDIO` con su definición de macro. Ello convierte a la compilación en un procedimiento de dos pasos, pero que no debe ser una carga mayor, especialmente si está construyendo sus programas bajo el control del programa de utilidad *make*. También le sugerimos almacenar los programas en FORTRAN que contienen directivas para *cpp* mediante la nomenclatura *filename.F* para distinguirlos de los programas FORTRAN sin adornos. Sólo asegúrese de hacer los cambios sólo a los archivos *.F*, y no a la salida de *cpp*. Así es como debe preprocesar los archivos *.F* de FORTRAN a mano:

```
% /lib/cpp -P < promedio.F > promedio.f
% f77 promedio.f -c
```

El compilador de FORTRAN nunca ve el código original. En vez de ello, la definición de la macro se sustituye en línea, tal y como si la hubiera tecleado usted mismo:

```
C
      PROGRAM MAIN
      REAL A,P,Q
      DATA P,Q /50.,100./ A = ((P+Q)/2)
      WRITE (*,*) A
      END
```

Por cierto, algunos compiladores de FORTRAN también reconocen la extensión *.F*, haciendo innecesario el proceso en dos pasos. Si el compilador ve la extensión *.F* invoca automáticamente a *cpp*, y desecha el archivo *.f* intermedio. Trate de compilar un archivo *.F* en su computadora para ver si funciona.

También, tome conciencia de que la expansión de macros puede provocar que las líneas de código fuente excedan la columna 72, lo cuál hará que su compilador de FORTRAN reclame (o peor, que le pase desapercibido). Algunos compiladores soportan líneas de entrada más largas de 72 caracteres. En los compiladores de Sun la opción **–e** permite extender las líneas de entrada hasta 132 caracteres de longitud.

2 Procedimientos En Línea

Las definiciones de macros tienden a ser muy cortas, usualmente de sólo una sentencia de longitud. A veces tiene usted porciones de código ligeramente más largas (pero no demasiado largas), que también pueden obtener beneficio si se copian en línea, en vez de invocarse como subrutina o función. Nuevamente, la razón de hacerlo es eliminar la sobrecarga debida a la invocación de procedimientos, y exponer el paralelismo. Si su compilador es capaz de definir subrutinas y funciones *inline* (en línea) al interior de los módulos que las invocan, entonces dispone de una forma muy natural y portátil de escribir código modular sin sufrir del costo debido a la invocación de la subrutina.

Dependiendo del vendedor, puede preguntarle al compilador si soporta la declaración de procedimientos en línea mediante:

- Especificar desde la línea de comandos aquellas rutinas que deben insertarse en línea
- Poner directivas para la declaración en línea dentro del código fuente
- Permitir que el compilador decida automáticamente qué poner en línea

Las directivas y las opciones de la línea de comandos del compilador no están estandarizadas, así que deberá revisar la documentación de su compilador. Desafortunadamente, puede que nuevamente descubra que no hay tal característica (“nuevamente,” siempre nuevamente), o que es una característica extra que por la que deberá pagar. La tercera forma de declaración en línea de la lista, la automática, sólo la tienen disponible unos pocos vendedores. La declaración en línea automática depende de que el compilador sea sofisticado y pueda ver las definiciones de varios módulos a la vez.

Unas últimas palabras de precaución respecto a la colocación de procedimientos en línea. Es fácil exagerar en su uso. Si todo termina enquistado en el cuerpo o en sus padres, el ejecutable resultante puede ser tan grande que repetidamente desborde la cache de instrucciones y se convierta en una pérdida de rendimiento neto. Nuestro consejo es que use la información de invocador/invocado que le proporcionan los perfiladores para tomar decisiones inteligentes acerca de la ubicación en línea, en vez de tratar de aplicarlo a cada subrutina que genere. De nuevo, los mejores candidatos generalmente son aquellas rutinas pequeñas que se invocan muy a menudo.