Eliminando el Desorden - Otras Formas de Desorden*

José Enrique Alvarez Estrada

Translated By:

José Enrique Alvarez Estrada

Based on *Eliminating Clutter - Other Clutter*† by

Charles Severance

Kevin Dowd

This work is produced by OpenStax-CNX and licensed under the
Creative Commons Attribution License 3.0‡

El desorden puede asumir muchas formas. Considere que en las secciones previas hemos batallado con grandes piezas de basura como las que puede usted encontrar en el armario de su sala: una tabla de planchar, palos de hockey y palos de billar. Ahora nos dedicaremos a cosas más pequeñas: una goma de borrar, una pelota de tenis y el sombrero que nadie usa. Aquí queremos mencionar unos cuantos. De entrada pedimos disculpas por cambiar los temas tanto, ¡pero así es la naturaleza de limpiar un armario!

1 Conversiones de Tipos de Datos

Las sentencias que incluyen conversiones de tipos a tiempo de ejecución producen algo de penalización del rendimiento cada vez que se ejecutan. Si la sentencia se localiza en una porción muy activa del programa, la penalización total puede resultar significativa.

Las personas tienen sus razones para escribir aplicaciones que mezclan tipos. A menudo es cuestión de ahorrar espacio de memoria, ancho de banda de memoria, o tiempo. En el pasado, por ejemplo, los cálculos de doble precisión tomaban el doble que sus contrapartes de precisión sencilla, así que si alguno de tales cálculos podía arreglarse para hacerse en precisión sencilla, beneficiaba al rendimiento.[1] Pero cualquier tiempo ahorrado realizando parte de los cálculos en precisión sencilla y la otra parte en doble precisión, debe medirse contra la sobrecarga adicional causada por las conversiones de tipos a tiempo de ejecución. En el siguiente código, la suma de `A(I)` con `B(I)` tiene *tipos mezclados*:

*Version 1.2: Oct 19, 2011 9:07 am -0500

†http://cnx.org/content/m33724/1.3/

‡http://creativecommons.org/licenses/by/3.0/

[1] Actualmente, los cálculos en precisión sencilla pueden tardar *más tiempo* que aquellos en doble precisión de registro a registro.

```
INTEGER NUMEL, I
PARAMETER (NUMEL = 1000)
REAL*8 A(NUMEL)
REAL*4 B(NUMEL)
DO I=1,NUMEL
  A(I) = A(I) + B(I)
ENDDO
```

En cada iteración, `B(I)` debe convertirse a doble precisión antes de que pueda ocurrir la suma. Usted no ve la conversión en el código fuente, pero está ahí, y consume tiempo.

Advertencia para los programadores de C: en el libro de C de Kernighan y Ritchie (K&R) C, todos los cálculos de punto flotante que aparecen en los programas son en doble precisión - incluso si todas las variables involucradas están declaradas como *float*. Es posible que usted escriba una aplicación K&R completa en una precisión, si bien sufrirá la penalización derivada de muchas conversiones de tipos.

Otro error relacionado con los tipos de datos, es usar operaciones de caracteres en las pruebas de las sentencias IF. En muchos sistemas, las operaciones sobre caracteres tienen un rendimiento más pobre que sus equivalentes enteras, dado que se realizan mediante llamados a procedimientos. Por lo mismo, puede que los optimizadores no revisen el código que usa variables de tipo caracter como buenos candidatos para optimización. Por ejemplo, el siguiente código:

```
DO I=1,10000
  IF ( CHVAR(I) .EQ. 'Y' ) THEN
    A(I) = A(I) + B(I)*C
  ENDIF
ENDDO
```

puede escribirse mejor usando una variable entera para indicar si debe o no realizarse el cálculo:

```
DO I=1,10000
  IF ( IFLAG(I) .EQ. 1 ) THEN
    A(I) = A(I) + B(I)*C
  ENDIF
ENDDO
```

Otra forma de escribir el código, asumiendo que la variable IFLAG fue 0 o 1, sería la siguiente:

```
DO I=1,10000
  A(I) = A(I) + B(I)*C*IFLAG(I)
```

```
ENDDO
```

El último enfoque puede de hecho provocar mayor lentitud en algunos sistemas de cómputo, respecto al primero, sobre todo si éste usa la variable entera en la sentencia IF.

2 Haciendo su Propia Eliminación de Subexpresiones Comunes

Hasta ahora le hemos dado a su compilador el beneficio de la duda. La *eliminación de subexpresiones comunes* — la habilidad del compilador de reconocer patrones repetidos en el código, y reemplazar todos excepto uno por una variable temporal- probablemente trabaje en su máquina con expresiones simples. En la siguientes líneas de código, la mayoría de los compiladores reconocerán `a+b` como una subexpresión común:

```
c = a + b + d
e = q + a + b
```

que se transforma en:

```
temp = a + b
c = temp + d
e = q + temp
```

La sustitución de `a+b` elimina algo de la aritmética. Si la expresión se reutiliza muchas veces, los ahorros pueden resultar significativos. Sin embargo, es limitada la habilidad de un compilador para reconocer subexpresiones comunes, especialmente cuando tienen componentes múltiples o éstos aparecen permutados. Un compilador puede no ser capaz de reconocer que `a+b+c` y `c+b+a` son equivalentes.[2] En las partes más importantes del programa, debe usted considerar la eliminación a mano de las subexpresiones comunes en las expresiones más complicadas. Ello garantiza que se realizará. En cierto modo compromete la belleza, pero hay algunas situaciones donde es mejor.

He aquí otro ejemplo en el cuál la función seno se invoca dos veces con el mismo argumento:

```
x = r*sin(a)*cos(b);
y = r*sin(a)*sin(b);
z = r*cos(a);
```

y que se convierte en:

```
temp = r*sin(a);
```

[2] Y por causa del sobreflujo y los errores de redondeo de punto flotante, en algunas situaciones pueden no ser equivalentes.

```
x = temp*cos(b);
y = temp*sin(b);
z = r*cos(a);
```

Hemos reemplazado uno de los llamados por una variable temporal. Estamos de acuerdo en que los ahorros de eliminar la invocación de una función trascendente de entre cinco no nos hará ganar el Premio Nobel, pero nos hace prestar atención a un punto importante: los compiladores típicamente no realizan la eliminación de subexpresiones comunes en subrutinas o llamados a funciones. El compilador no puede estar seguro que el llamado a la rutina no cambie el estado del argumento o de alguna otra variable que no puede ver.

La única ocasión en que un compilador puede eliminar subexpresiones comunes que contengan llamados a funciones, es cuando son intrínsecas, como en FORTRAN. Y puede hacerse porque el compilador puede asumir ciertas cosas acerca de los efectos laterales. Usted, por otra parte, puede ver adentro de las subrutinas, lo cuál significa que está mejor calificado que el compilador para agrupar subexpresiones comunes que involucren subrutinas o funciones.

3 Llevando a Cabo sus Propias Reubicaciones de Código

Donde mejor resulta llevar a cabo estas optimizaciones es adentro de los ciclos, porque es donde se concentra toda la actividad de un programa. Una de las mejores formas de recortar el tiempo de ejecución es mover las instrucciones innecesarias o repetidas (invariantes) fuera del flujo principal del código, a los suburbios. En el caso de los bucles, se le denomina *emerger* instrucciones cuando se desplazan hacia la parte superior y *sumergir* instrucciones cuando se colocan en la parte inferior. He aquí un ejemplo:

```
DO I=1,N
  A(I) = A(I) / SQRT(X*X + Y*Y)
ENDDO
```

que se convierte en:

```
TEMP = 1 / SQRT(X*X + Y*Y)
DO I=1,N
  A(I) = A(I) * TEMP
ENDDO
```

Hicimos emerger una operación cara e invariante hacia afuera del ciclo, y asignamos el resultado a una variable temporal. Observe también que hicimos una simplificación algebraica cuando intercambiamos una división por la multiplicación por un inverso. La multiplicación se ejecutará mucho más rápido. Puede ser que su compilador sea lo suficientemente listo para hacer tales transformaciones por sí mismo, asumiendo que usted le haya instruido de que son transformaciones legales; pero sin rastrear el código ensamblador generado, no puede usted estar seguro. Por supuesto, si usted reacomoda el código a mano y el tiempo de ejecución del ciclo súbitamente disminuye, sabrá que el compilador ha sido un lastre hasta entonces.

En ocasiones querrá usted sumergir una operación tras el ciclo. Usualmente, se trata de algún cálculo realizado cada iteración, pero cuyo resultado sólo se requiere al final. Para ilustrarlo, he aquí una clase de ciclo distinto de los que hemos visto hasta ahora, uno que busca el caracter final en una cadena:

```
while (*p != ' ')
  c = *p++;
```

se convierte en:

```
while (*p++ != ' ');
c = *(p-1);
```

La nueva versión del ciclo mueve la asignación de `c` más allá de la última iteración. Debemos admitir que esta transformación puede ser un logro para un compilador y que el ahorro no sea muy grande. Pero ilustra la idea de sumergir una operación muy bien.

Nuevamente, emerger y sumergir instrucciones para colocarlas afuera de ciclos es algo que su compilador puede que sea capaz de hacer. Pero a menudo usted puede reestructurar ligeramente los cálculos por sí mismo sólo con moverlos, y obtener un beneficio todavía mayor.

4 Manipulando los Elementos de un Arreglo en Bucles

Hay otra área donde debe usted confiar en que el compilador hará lo correcto. Cuando emplee repetidamente un elemento de un arreglo adentro de un bucle, querrá que se cargue una sola vez de la memoria. Tome el siguiente bucle como un ejemplo. Reutiliza `X(I)` dos veces:

```
DO I=1,N
  XOLD(I) = X(I)
  X(I)= X(I) + XINC(I)
ENDDO
```

En realidad, los pasos encargados de recuperar `X(I)` sólo son subexpresiones comunes adicionales: un cálculo de direcciones (posiblemente) y una operación de carga de memoria. Puede usted ver que la operación se repite al reescribir el ciclo ligeramente:

```
DO I=1,N
  TEMP= X(I)
  XOLD(I) = TEMP
  X(I)= TEMP + XINC(I)
ENDDO
```

Los compiladores de FORTRAN *deben* reconocer que se está usando dos veces el mismo elemento `X(I)` y que por tanto sólo se requiere cargarlo una vez, pero no siempre un compilador es tan inteligente. A veces

tiene usted que crear una variable escalar temporal para almacenar el valor de un elemento del arreglo en el cuerpo de un bucle. Esto es particularmente cierto cuando hay llamados a subrutinas o funciones en el ciclo, o cuando alguna de las variables está declarada como `external` o `COMMON`. Asegúrese de emparejar los tipos entre las variables temporales y las otras variables, pues no quiere incurrir la sobrecarga derivada de la conversión de tipos sólo por estar "ayudando" al compilador. En el caso de los compiladores de C, el mismo tipo de expresiones indexadas son un reto incluso mayor. Considere este código:

```
doinc(int xold[],int x[],int xinc[],int n)
{
  for (i=0; i<n; i++) {
    xold[i] = x[i];
    x[i]= x[i] + xinc[i];
  }
}
```

A menos que el compilador pueda ver las definiciones de `x`, `xinc` y `xold`, debe asumir que son apuntadores señalando a la misma celda de almacenamiento, y repetir las operaciones de carga y almacenamiento. En este caso, introducir variables temporales para almacenar los valores de `x`, `xinc`, y `xold` es una optimización que el compilador no es libre de hacer.

Es interesante señalar que si bien usar variables escalares temporales en el ciclo es útil para las máquinas RISC y superescalares, no ayuda al código que se ejecuta sobre hardware paralelo. Un compilador paralelo busca oportunidades de eliminar los escalares o, cuando menos, de reemplazarlos con vectores temporales. Si ejecuta su código sobre una máquina paralela de vez en cuando, debe ser cuidadoso antes de introducir variables escalares temporales en un ciclo. Una dudosa ganancia de rendimiento en una instancia puede convertirse en una pérdida de rendimiento real en otra.