Qué Hace un Compilador - Optimizaciones Clásicas*

José Enrique Alvarez Estrada

Translated By:

José Enrique Alvarez Estrada

Based on *What a Compiler Does - Classical Optimizations*† by

Charles Severance

Kevin Dowd

This work is produced by The Connexions Project and licensed under the
Creative Commons Attribution License ‡

Una vez dividido el lenguaje intermedio en bloques básicos, hay diversas optimizaciones que pueden realizarse sobre el código de tales bloques. Algunas son muy simples y afectan a unas pocas tuplas dentro de un bloque básico. Otras mueven código de un bloque básico a otro, sin alterar los resultados del programa. Por ejemplo, a menudo resulta útil mover una cálculo desde el cuerpo de un ciclo al código que precede de forma inmediata al bucle.

En esta sección, vamos a listar por nombre las optimizaciones clásicas, e indicarle a usted para qué son. No estamos sugiriendo que *usted* haga los cambios; la mayoría de lso compiladores desde mitad de la década de 1980 realizan de modo completamente automático tales optimizaciones, excepto en el nivel más bajo de optimización. Como vimos a inicios del capítulo, si comprende usted aquello que pueden (y que no pueden) hacer los compiladores, se convertirá en un mejor programador porque será capaz de jugar con las fortalezas de ellos.

1 Propagación de Copia

Para iniciar, veamos una técnica para desenredar cálculos. Échele un vistazo al siguiente segmento de código: observe los dos cálculos que involucran a `X`.

```
X = Y
```

*Version 1.1: May 23, 2011 8:34 pm -0500

†http://cnx.org/content/m33696/1.3/

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

```
Z = 1.0 + X
```

Tal como está escrito, la segunda sentencia requiere del resultado de la primera antes de que pueda ejecutarse -requiere usted de `X` para calcular `Z`. Las dependencias innecesarias a menudo se traducen en retrasos a tiempo de ejecución. [1] Realizando algunos reacomodos, podemos hacer que la segunda sentencia sea independiente de la primera, al *propagar* una copia de `Y`. El nuevo cálculo para `Z` usa el valor de `Y` directamente:

```
X = Y
Z = 1.0 + Y
```

Observe que dejamos intacta la sentencia `X=Y`. Puede que usted se pregunte, "¿por qué dejarla así?" El problema es que no podemos decir en qué otras partes se requiere el valor de `X`. Eso es algo que deberá decidir otro análisis. Si resulta que ninguna otra sentencia requiere del valor de `X`, la asignación se eliminará posteriormente, por medio de la remoción de código muerto.

2 Plegado de Constantes

Un compilador inteligente puede encontrar constantes a lo largo de su programa. Algunas de ellas son "obvias", como aquellas definidas en las sentencias de parámetros. Otras resultan menos obvias, tales como las variables locales que nunca se redefinen. Cuando las combina en un cálculo, obtiene una *expresión constante*. El pequeño programa siguiente tiene dos constantes, `I` y `K`:

```
PROGRAM MAIN
INTEGER I,K
PARAMETER (I = 100)
K = 200
J = I + K
END
```

Dado que tanto `I` como `K` son constantes individualmente, la combinación de `I+K` es constante, lo cuál significa que `J` también es una constante. El compilador reduce las expresiones constantes como `I+K` a constantes, mediante una técnica llamada *plegado de constantes*.

¿Cómo funciona el plegado de constantes? Puede ver que es posible examinar cada camino en el cuál puede definirse una variable en ruta hacia un bloque básico particular. Si descubre usted que todos los caminos se inician en el mismo valor, éste es constante; puede reemplazar todas las referencias a esa variable con una constante. Este reemplazo tiene un efecto de rizado a lo largo de la ruta. Si el compilador se percata de que encontró una expresión compuesta solamente por constantes, puede evaluarla a tiempo de compilación y reemplazarla por una constante. Tras varias iteraciones, el compilador habrá localizado la mayoría de las expresiones que son candidatas al plegado de constantes.

A veces un programador puede mejorar el rendimiento, al hacer al compilador consciente de los valores constantes en su aplicación. Por ejemplo, en el siguiente segmento de código:

[1] Este código es un ejemplo de dependencia de flujo. Describo las dependencias en detalle en (<http://cnx.org/content/m32775/latest/>).

```
X = X * Y
```

el compilador generará código muy diferente a tiempo de ejecución si sabe que `Y` valía 0, 1, 2, or 175.32. Si no conoce el valor para `Y`, debe generar la secuencia de código más conservadora (no necesariamente la más rápida). Un programador puede comunicarle esos valores a través del uso de la sentencia `PARAMETER` en FORTRAN. Mediante el empleo de dicha sentencia, el compilador conoce los valores para estas constantes a tiempo de ejecución. Otro ejemplo que hemos visto es:

```
DO I = 1,10000
  DO J=1,IDIM
    .....
  ENDDO
ENDDO
```

Tras revisar el código, es claro que `IDIM` era uno de los valores 1, 2, or 3, dependiendo del conjunto de datos en uso. Claramente si el compilador sabía que `IDIM` valía 1, puede generar un código mucho más simple y rápido.

3 Remoción de Código Muerto

A menudo los programas contienen secciones de *código muerto* que no tiene efecto en las respuestas, y que por tanto puede quitarse. Ocasionalmente, quien escribe el código muerto en el programa es el autor, pero es más común que sea el propio compilador quien lo hace; muchas optimizaciones producen código muerto que hay que barrer después.

Hay dos tipos de código muerto:

- Instrucciones inalcanzables
- Instrucciones que producen resultados que jamás se usan

Es fácil que usted escriba código inalcanzable en su programa, al hacer que el flujo de control pase alrededor de él - permanentemente. Si el compilador puede decir que es inalcanzable, lo eliminará. Por ejemplo, es imposible alcanzar la sentencia `I = 4` en este programa:

```
PROGRAM MAIN
I = 2
WRITE (*,*) I
STOP
I = 4
WRITE (*,*) I
END
```

El compilador desechará todo hasta la sentencia `STOP`, y probablemente le envíe un aviso. El código inalcanzable producido por el compilador durante la optimización se retirará silenciosamente.

Los cálculos con variables locales pueden producir resultados que nunca se usan. Al analizar la definición y usos de una variable, el compilador puede ver qué otras partes de la rutina la referencian. Por supuesto el

compilador no puede decir el destino último de las variables que se pasan entre rutinas, externas o comunes, así que tales cálculos siempre se mantienen (siempre y cuando sean alcanzables).[2] En el siguiente programa, los cálculos que involucran a `k` no contribuyen en absoluto a la respuesta final, y son buenos candidatos para la eliminación de código muerto:

```
main ()
{
  int i,k;
  i = k = 1;
  i += 1;
  k += 2;
  printf ("%d\n",i);
}
```

La eliminación de código muerto a menudo produce algunos resultados sorprendentes con bancos de pruebas pobremente escritos. Véase para un ejemplo de este tipo de código.

4 Reducción de Intensidad

Las operaciones o expresiones tienen costos de tiempo asociados. A veces es posible reemplazar un cálculo muy caro por otro más económico. Llamamos a esto *reducción de intensidad*. El siguiente fragmento de código contiene dos operaciones caras:

```
REAL X,Y
Y = X**2
J = K*2
```

En operaciones de exponenciación como la de la primera línea, el compilador generalmente incrusta una llamada a una rutina matemática de biblioteca. En dicha rutina, `X` se convierte a un logaritmo, se multiplica y luego se reconvierte. Sobre todo, elevar `X` a una potencia resulta caro -quizás tome cientos de ciclos de máquina. La clave es observar que `X` se está elevando a una potencia entera y pequeña. Una alternativa mucho más económica consiste en expresarlo como `X*X`, y pagar sólo el costo de una multiplicación. La segunda sentencia muestra la multiplicación entera de una variable `K` por 2. Sumar `K+K` da el mismo resultado, pero toma menos tiempo.

Hay muchas oportunidades para reducción de intensidad generada por el compilador; éstas son sólo dos de ellas. Veremos un caso especialmente importante cuando analicemos la simplificación de variable por inducción. Otro ejemplo de una reducción de intensidad es reemplazar las multiplicaciones de enteros potencia de dos por desplazamientos lógicos.

[2] Si un compilador realiza un análisis interprocedimental suficiente, puede incluso optimizar variables entre fronteras de rutinas. El análisis interprocedimental puede ser la perdición para los códigos de bancos de pruebas si trata de medir el tiempo de un cálculo sin usar los resultados del mismo.

5 Renombramiento de Variables

En here[3], hablamos acerca de renombrar registros. Algunos procesadores pueden tomar decisiones a tiempo de ejecución para reemplazar todas las referencias al registro 1 por referencias al registro 2, por ejemplo, para evitar cuellos de botella. El renombramiento de registros evita que las instrucciones que están reciclando los mismos registros para propósitos diferentes, tengan que esperar hasta que las instrucciones previas hayan dejado de usarlos.

La misma situación puede ocurrir en los programas - la misma variable (i.e. posición de memoria) puede reciclarse para dos propósitos no relacionados. Por ejemplo, observe la variable `x` en el siguiente fragmento:

```
x = y * z;
q = r + x + x;
x = a + b;
```

Cuando el compilador reconoce que se está reciclando una variable, esto es que sus usos actual y futuros son independientes, puede sustituirla por una nueva variable para mantener los cálculos separados:

```
x0 = y * z;
q = r + x0 + x0;
x = a + b;
```

El *renombramiento de variables* es una técnica importante, porque deja en claro cuáles cálculos son independientes los unos de los otros, lo cuál mejora el número de cosas que pueden hacerse en paralelo.

6 Eliminación de Subexpresiones Comunes

Las subexpresiones son partes de expresiones más grandes. Por ejemplo, `A+B` es una subexpresión de `C*(A+B)`. Si `A+B` aparece en varios lugares, como sucede a continuación, le llamamos una *subexpresión común*:

```
D = C * (A + B)
E = (A + B)/2.
```

En vez de calcular `A + B` dos veces, el compilador puede generar una variable temporal y usarla dondequiera que se necesite `A + B`:

```
temp = A + B
D = C * temp
E = temp/2.
```

[3] "What is High Performance Computing - Introduction" <http://cnx.org/content/m33671/latest/>

Algunos compiladores se esfuerzan más que otros en encontrar subexpresiones comunes. Se reconocen la mayoría de los pares, tales como `A+B`. Algunos pueden reconocer la reutilización de intrínsecos, tales como `SIN(X)`. Pero no espere que el compilador vaya mucho más allá. Subexpresiones como `A+B+C` no son computacionalmente equivalentes a formas reasociadas como `B+C+A`, aunque sean algebraicamente equivalentes. Con el fin de proporcionar resultados predecibles en los cálculos, FORTRAN debe o bien realizar las operaciones en el orden especificado por el usuario, o bien reordenarlos en una forma que garantice exactamente el mismo resultado. A veces el usuario no se preocupa de la forma en que se asocia `A+B+C`, pero el compilador no puede asumir que al usuario no le preocupa.

El cálculo de direcciones proporciona una oportunidad particularmente rica para la eliminación de subexpresiones comunes. Usted no requiere ver los cálculos en el código fuente, pues los genera el compilador. Por ejemplo, una referencia al elemento de un arreglo `A(I,J)` puede traducirse a una expresión en lenguaje intermedio tal como:

```
address(A) + (I-1)*sizeof_datatype(A)
+ (J-1)*sizeof_datatype(A) * column_dimension(A)
```

Si `A(I,J)` se usa más de una vez, tenemos múltiples copias del mismo cálculo de direcciones. La eliminación de subexpresiones comunes descubrirá y agrupará (con un poco de suerte) los cálculos redundantes.

7 Remoción de Código Invariante en Bucles

Es en los bucles donde los programas de cómputo de alto rendimiento gastan la mayoría de su tiempo. El compilador busca cada oportunidad que tiene de mover cálculos fuera del cuerpo de un bucle hacia el código que lo rodea. Las expresiones que no cambian después de haber ingresado al ciclo (*expresiones invariantes en el bucle*) son la primera opción. El siguiente ciclo tiene dos expresiones invariantes:

```
DO I=1,N
  A(I) = B(I) + C * D
  E = G(K)
ENDDO
```

A continuación, hemos modificado las expresiones para mostrar cómo pueden moverse afuera del ciclo:

```
temp = C * D
DO I=1,N
  A(I) = B(I) + temp
ENDDO
E = G(K)
```

Es posible mover el código antes o después del cuerpo del bucle. Como sucede con la eliminación de subexpresiones comunes, la aritmética de direcciones resulta un blanco particularmente importante para la técnica de movimiento de invariantes en los bucles. Las porciones que cambian lentamente en el cálculo de los índices pueden enviarse a los suburbios, para ejecutarse sólo cuando se requieran.

8 Simplificación de Variables de Inducción

Los ciclos pueden contener lo que se conoce como *variables de inducción*. Su valor cambia como una función lineal del contador de iteraciones del ciclo. Por ejemplo, `K` es una variable de inducción en el siguiente bucle. Su valor está ligado al índice del ciclo:

```
DO I=1,N
  K = I*4 + M
   ...
ENDDO
```

La *simplificación de variables de inducción* reemplaza los cálculos que involucran variables como `K` por otras más simples. Dado un punto de inicio y la primera derivada de la expresión, puede usted llegar al valor de `K` para la *n*-ésima iteración, recorriendo paso a paso las *n-1* iteraciones que intervienen:

```
K = M
DO I=1,N
  K = K + 4
   ...
ENDDO
```

Ambas formas del bucle no son equivalentes; la segunda no le proporciona a usted el valor de `K`, dado cualquier valor de `I`. Dado que no podemos saltar en la mitad del bucle a la *n*-ésima iteración, `K` siempre toma los mismos valores que hubiera tenido si hubiéramos dejado la expresión original.

Las simplificación de variables de inducción probablemente no sea una optimización muy importante, excepto que el cálculo de direcciones de los arreglos se parece mucho al cálculo de `K` en el ejemplo anterior. Por ejemplo, el cálculo de direcciones para `A(I)` dentro de un bucle iterando sobre la variable `I` se ve más o menos así:

```
address = base_address(A) + (I-1) * sizeof_datatype(A)
```

Realizar toda esta matemática es innecesario. El compilador puede crear una nueva variable de inducción para las referencias a A y simplificar el cálculo de direcciones:

```
afuera del ciclo...
address = base_address(A) - (1 * sizeof_datatype(A))
dentro del ciclo...
address = address + sizeof_datatype(A)
```

La simplificación de variables de inducción resulta especialmente útil en aquellos procesadores que pueden incrementar automáticamente un registro cada vez que se utiliza un apuntador para una referencia a memoria. Mientras se recorre el ciclo paso a paso, tanto la referencia a memoria como la aritmética de direcciones pueden exprimirse en una sola instrucción -un gran ahorro.

9 Generación de Código Objeto

La precompilación, los análisis lexicográfico y sintáctico y muchas técnicas de optimización son a menudo transportables, pero la generación de código es muy específica del procesador destino. En cierta forma es en esta fase donde los compiladores desquitan su precio, en los sistemas RISC de un solo procesador.

Cualquier cosa que no se haga en hardware debe realizarse en software. Ello significa que si el procesador no puede resolver conflictos de recursos, tales como la sobreutilización de un registro o fila de procesamiento, el compilador deberá hacerse cargo del asunto. Permitir que el compilador se ocupe de ello no es necesariamente malo, más bien es una decisión de diseño. Un compilador complicado y un hardware simple y rápido pueden tener una alta relación costo/beneficio para ciertas aplicaciones. Dos procesadores que se encuentran en los extremos de este espectro son el MIPS R2000 y el HP PA-8000. El primero depende fuertemente del compilador para planificar las instrucciones y distribuir los recursos equitativamente. El segundo administra ambas cosas a tiempo de ejecución, aunque ambos dependan del compilador para proporcionar una mezcla de instrucciones equitativa.

En todas las computadoras, la selección de registros es un reto porque, dependiendo de su número, los registros son recursos preciosos. Usted quiere asegurarse que las variables más activas residan permanentemente en los registros, a expensas de otras. En aquellas máquinas sin renombramiento de registros (véase here[4]), usted debe asegurarse que el compilador no trate de reciclar los registros demasiado rápido, o de otra forma el procesador tendrá que retrasar los cálculos, en espera de que alguno de ellos se libere.

Algunas instrucciones del repertorio también hacen que su compilador no tenga que ejecutar otras. Ejemplo de esto son el autoincremento de los registros que se usan como índices de arreglos, o el uso de asignaciones condicionales en vez de bifurcaciones. Ambas ahorran al procesador realizar cálculos extras, produciendo un flujo de instrucciones más compacto.

Finalmente, están las oportunidades para incrementar el paralelismo. Los programadores generalmente piensan serialmente, especificando pasos en sucesión lógica. Desafortunadamente, el código fuente serial produce código objeto serial. Un compilador que aspire a usar eficientemente el paralelismo del procesador, deberá ser capaz de mover instrucciones y encontrar operaciones que puedan realizarse simultáneamente. Este es uno de los mayores retos de los creadores de compiladores actualmente. Conforme los diseños superescalares y de *tamaño de instrucción muy grande* (VLIW, Very Long Instruction Word) se vuelven capaces de ejecutar más instrucciones por ciclo de reloj, el compilador tendrá que excavar más profundamente en busca de operaciones que puedan ejecutarse a la vez.

[4] "What is High Performance Computing - Introduction" <http://cnx.org/content/m33671/latest/>