LA OPTIMIZACIÓN: UNA MEJORA EN LA EJECUCIÓN DE PROGRAMAS

José M. García Carrasco

José M. García Carrasco, Doctor Ingeniero Industrial
Dpto. de Informática. Universidad de Castilla-La Mancha

Resumen

En la actualidad los ordenadores están invadiendo todos los campos del conocimiento. Una gran variedad de programas se desarrollan para diversas aplicaciones de todo tipo. Al usuario final de un programa sólo le interesan dos cosas: que el programa sea fácil de manejar y que su ejecución sea lo más rápida posible. Para mejorar este último aspecto cada día se tienen más en cuenta las optimizaciones en el código del programa. En este artículo se describe cuáles son y en qué consisten cada una de las diversas técnicas de optimización.

1. INTRODUCCIÓN

IDEALMENTE, los compiladores deberían producir código objeto que fuera tan bueno como si estuviera escrito directamente por un buen programador. La realidad es que esto es difícil de conseguir y muy pocas veces se alcanza esa meta. Sin embargo, el código generado por el compilador puede ser mejorado por medio de unas transformaciones que se han denominado tradicionalmente optimizaciones, aunque el término optimización es impropio ya que raramente se consigue que el código generado sea el mejor posible.

El objetivo de las técnicas de optimización es mejorar el programa objeto para que nos dé un rendimiento mayor. La mayoría de estas técnicas vienen a compensar ciertas ineficiencias que aparecen en el lenguaje fuente, ineficiencias que son inherentes al concepto de lenguaje de alto nivel, el cual suprime detalles de la máquina objeto para facilitar la tarea de implementar un algoritmo.

Las distintas técnicas de optimización se pueden clasificar o dividir de diversas formas.

Por una parte podemos hablar de aquellas técnicas que son dependientes de la máquina, y aquellas que son independientes de la máquina (o sea, técnicas que sólo se pueden aplicar a una determinada máquina objeto y técnicas que son aplicables a cualquier máquina objeto).

Por otra parte, las técnicas de optimización se dividen también en locales y globales. Las técnicas de optimización locales analizarán sólo pequeñas porciones de código y en ellas realizarán mejoras, mientras que para la aplicación de las técnicas globales será necesario el análisis de todo el código.

Cada optimización está basada en un *función de coste* y en una *transformación que preserve el significado* del programa. Mediante la función de coste queremos evaluar la mejora que hemos obtenido con esa optimización y si compensa con el esfuerzo que el compilador realiza para poder llevarla a cabo. Los criterios más comunes que se suelen emplear son el ahorro en el tamaño del código, la reducción del tiempo de ejecución y la mejora de las necesidades del espacio para los datos del programa.

En cuanto a preservar el significado del programa, es lógico que no tendría sentido realizar optimizaciones que modificaran el comportamiento del programa. Aunque parezca evidente, puede haber complicadas optimizaciones que fallen en ese aspecto.

Por último, comentar que por muchos optimizaciones que se hayan realizado para mejorar el rendimiento de un programa, siempre se obtendrá un mejor rendimiento si se utiliza un algoritmo mejor. Por todo ello, para obtener un buen programa lo primero es ver qué algoritmo utilizamos y si no es posible desarrollar otro más eficiente. Una vez implementado el mejor algoritmo, ya se puede entonces optimizar el código obtenido a partir de él para mejorar el rendimiento del programa.

2. TIPOS DE OPTIMIZACIÓN

Existen diversas técnicas de optimización que se aplican al código generado para un programa sencillo. Por programa sencillo entendemos aquel que se reduce a un solo procedimiento o subrutina. Las técnicas de optimización a través de varios procedimientos se reducen a aplicar las vistas aquí a cada uno de los procedimientos y después realizar un análisis interprocedural. Este último tipo de análisis no lo vamos a desarrollar en este artículo.

Partiendo de un programa sencillo, obtenemos código intermedio de tres direcciones [AHO77], pues es una representación adecuada del programa sobre la que emplear las diversas técnicas

de optimización. A partir de aquí dividimos el programa en bloques básicos [AHO86], o secuencia de sentencias en las cuales el flujo de control empieza al principio y acaba al final, sin posibilidad de parar o de tener saltos. Las sentencias de un bloque básico constituyen una unidad sobre la cual se aplican las optimizaciones locales.

Estas optimizaciones se pueden dividir en:

a) **Optimizaciones que no modifican la estructura**. Son:
   1. Eliminación de sub-expresiones comunes.
   2. Eliminación de código muerto.
   3. Renombrar variables temporales.
   4. Intercambio de sentencias independientes adyacentes.

b) **Transformaciones algebraicas**. Son aquellas transformaciones que simplifican expresiones y/o reemplazan operaciones costosas de la máquina por otras menos costosas.

Además de este tipo de optimizaciones locales a un bloque básico, existe otro tipo de optimizaciones aún más *locales*, pues su ámbito se reduce a una breve secuencia de instrucciones. A este tipo de optimización local se le llama optimización *peephole*, e intenta mejorar el rendimiento del programa por medio de reemplazar esa breve secuencia de instrucciones objeto por otra secuencia más corta y/o más rápida. Hay varios tipos de optimización *peephole*, siendo los más usuales los siguientes:

1. Eliminación de instrucciones redundantes.
2. Optimizaciones en el flujo de control.
3. Simplificaciones algebraicas.
4. Uso de instrucciones máquina específicas.

Debido a la naturaleza de este tipo de optimización, su salida es susceptible de ser optimizada de nuevo, con lo que serán necesarias varias pasadas para lograr la máxima optimización. Una aplicación de este tipo de optimización se puede encontrar en el kit ACK desarrollado por Tanembaum [Tane82], donde se describen 123 combinaciones distintas de sentencias de código intermedio con sus correspondientes equivalencias.

Las técnicas de optimización global se basan todos ellos en el *análisis global de flujo de datos*. Este análisis se realiza para el código de todo el programa, es decir, a lo largo de los distintos bloques básicos que forman el código del programa. Suponiendo que tenemos la información que nos proporciona este análisis, que lo veremos en el siguiente apartado, hay dos tipos de optimizaciones importantes que se realizan: la localización y la asignación global de registros para las variables, y las optimizaciones que se realizan en los bucles.

Localización y asignación de registros

Para una máquina con registros –lo común en los procesadores actuales– las instrucciones cuyos operandos están en los registros de la máquina son más cortas y más rápidas que aquellas que tratan con operandos que están en la memoria. Es por tanto importante decidir qué variables se deben almacenar en los registros (localización) y en qué registro se debe almacenar cada variable (asignación).

Existen diversas estrategias para la localización y asignación de los registros. Es frecuente asignar algún número fijo de registros que contengan las variables más usadas en un bucle interno, sirviendo los registros restantes para las variables locales a cada bloque básico.

Un método sencillo para determinar la ganancia que obtenemos por almacenar la variable "x" durante el bucle L es el siguiente:

1. Contaremos un ahorro de una unidad por cada referencia a x en el bucle si x está en un registro y no está precedida por una asignación a x en el mismo bloque básico (debido a la naturaleza del algoritmo que empleamos para general código –ver [AHO86]–).
2. Contaremos un ahorro de dos unidades si x es una variable viva a la salida de un bloque básico en el cual se le ha asignado un valor a x, debido a que evitamos almacenar el valor de x a la salida de ese bloque.
   Entonces, una fórmula aproximada para calcular el beneficio obtenido por colocar la variable x en un registro en el bucle L es:

$$\sum_{\text{bloques } B \text{ en } L} (\text{use}\ (x, B) + 2 * \text{live}\ (x, B))$$

   donde use (x, B) es el número de veces que x es usada en B antes de que sea definida, y live (x, B) es 1 si está *viva* a la salida de B y se le asigna un valor en B, y 0 en cualquier otro caso. El cálculo de use (x, B) y live (x, B) se realiza gracias al análisis global del flujo de datos.

Optimizaciones en bucles

Habitualmente, un programa pasa la mayor parte del tiempo de la ejecución en un trozo de código pequeño. A este fenómeno se le conoce como la regla 90-10, queriendo decir que el 90% del

tiempo es pasado en el 10% del código. Este 10% del código suele estar constituido por bucles, y de ahí la importancia de una correcta optimización del código que forma parte de los bucles.

Las principales optimizaciones que se realizan en los bucles son las siguiente:

1. Movimiento de código.
2. Eliminación de variables inducidas.
3. Sustitución de variables costosas por otras menos costosas.

Y también se suelen aplicar (aunque con menor importancia):

4. Expansión de código (loop unrolling).
5. Unión de bucles (loop jamming).

3. ANÁLISIS GLOBAL DE FLUJO DE DATOS

Para que el compilador pueda realizar la mayoría de las optimizaciones vistas hasta ahora, es necesario que posea información de todo el programa, para poder determinar si una variable está *viva*, o si dos subexpresiones son comunes, o si una variable se puede sustituir por un valor constante, etc.

Esta información es recogida por el compilador mediante la resolución de las dos ecuaciones siguientes:

$$X = (Y - S_1) \cup S_2 \tag{1}$$

$$Y = \text{op } X \tag{2}$$

donde X, Y, S_1 y S_2 son diversos conjuntos específicos de cada problema y que definiremos más adelante.

En función de que el análisis se realice hacia adelante (de la primera instrucción de código a la última) o hacia atrás, y en función del tipo de operador que aparece en la segunda ecuación (operador unión u operador intersección), se presentan los siguientes cuatro tipos de problemas en el análisis del flujo de datos:

		operador	
		∪	∩
Dirección	Hacia delante	Enlace ud	Expresiones disponibles Propagación de copia
	Hacia atrás	Variables vivas Enlace du	Expresiones muy usadas

Mediante la resolución de los cuatro tipos de problemas anteriores se obtiene toda la información necesaria para poder realizar las optimizaciones anteriores.

Para concretar, y a modo de ejemplo, vamos a desarrollar el problema denominado **enlace ud**.

4. ENLACE UD

El enlace ud, o enlace uso-definición, trata de determinar qué definiciones alcanzan un punto dado de un programa.

Por definición de A entendemos una asignación a A o bien la lectura de un valor para A.

Por uso de una variable A entendemos cualquier aparición de A en que actúa como operando.

Por un punto en el programa queremos decir la posición antes o después de cualquier sentencia de código.

Una definición de una variable A alcanza un punto p si hay un camino en el flujo del programa de esa definición a p, tal que ninguna otra definición de A aparece en ese camino.

Un camino de p_1 a p_n es una secuencia de puntos $p_1, p_2, \ldots, p_n$ tal que para cada punto i entre el 1 y el $n-1$ se cumple:

a) p_i es el punto que precede inmediatamente a una sentencia y p_{i+1} es el punto que sigue inmediatamente a esa sentencia en el mismo bloque.
b) O bien p_i es el final de algún bloque y p_{i+1} es el comienzo de algún bloque sucesor.

Para calcular qué definiciones alcanzan un punto dado, hemos de calcular para cada bloque básico:

a) El conjunto de definiciones generadas por ese bloque, es decir, aquellas definiciones que alcanzan el final del bloque. A este conjunto lo denominaremos GEN[B].
b) El conjunto de definiciones exteriores a ese bloque que son redefinidas dentro del bloque. A este conjunto lo denominaremos KILL[B].
c) El conjunto de todas las definiciones que alcanzan el inicio del bloque, denominado IN[B], y el conjunto de todas las definiciones que alcanzan el final del bloque, denominado OUT[B].

Los conjuntos GEN[B] y KILL[B] se determinan por simple inspección en cada bloque básico. Para obtener IN[B] y OUT[B], se resuelven las ecuaciones (1) y (2) dadas en el apartado anterior, que en este caso adoptan la siguiente forma:

$$OUT[B] = (IN[B] - KILL[B]) \cup GEN[B]$$

$$IN[B] = \bigcup_{\substack{P \text{ es un} \\ \text{predecesor de B}}} OUT[P]$$

Para la resolución de estas ecuaciones, hay que tener en cuenta lo siguiente:

1. Los conjuntos IN, OUT, GEN, KILL se representan como vectores de bits, donde el bit i-ésimo tendrá el valor 1 si la definición numerada con "i" está en el conjunto. Ello permite que el espacio ocupado por estos conjuntos sea pequeño y que las operaciones anteriores se puedan implementar con el *and* y el *or* lógicos.
2. La resolución de las ecuaciones se debe realizar de forma iterativa. Aunque se pueden resolver con otras suposiciones, es normal y muy eficiente utilizar el siguiente algoritmo para el cálculo de las ecuaciones:

```
{inicializaciones}
for cada bloque B do begin
    IN[B] : = 0;
    OUT[B] : = GEN[B]
    end;
{cuerpo del algoritmo}
cambio : = true;
while cambio do begin
    cambio : = false;
    for cada bloque B do begin
        IN[B] : = ∪ OUT[P];
                  P es un
              predecesor de B
        oldout : = OUT[B];
        OUT[B] : = (IN[B] – KILL[B]) U GEN[B];
        if OUT[B] <> oldout then cambio : = true
        end
    end
```

NOTAS:

- El algoritmo finaliza cuando los conjuntos IN y OUT permanecen constantes para todos los bloques. Se puede probar que un límite superior de iteraciones es el número de nodos –bloques básicos– que tiene el diagrama de flujo [KIL73].
- El número medio de iteraciones para programas reales es menor que cinco.

– En el cálculo de los predecesores, asumiremos que cualquier flecha en el diagrama de flujo puede ser recorrida. Aunque en algún caso esto no sea verdad, esta es una postura conservativa que lo único que hará es que perdamos alguna posible optimización, pero nunca que modifiquemos el comportamiento del programa, cosa que con la suposición contraria podría ocurrir.

Una vez resueltas las anteriores ecuaciones, se realiza el cálculo del enlace ud, es decir, asociar a cada uso de la variable *A* las definiciones que alcanzan ese uso.

Hay una variedad de aplicaciones al enlace ud. Entre otras, se puede determinar la sustitución de una variable por un valor constante, y también determinar si el uso de una variable *A* está indefinida en algún punto del programa.

Por último señalar que este tipo de análisis de flujo de datos no sólo se aplica para optimizar código en compilación, sino que se utiliza también para otras aplicaciones. Una aplicación interesante se puede encontrar en [DEJ89].

5. CONCLUSIONES

En este trabajo se ha presentado la técnica de la **optimización** de código, como un medio de mejora del código objeto producido por un compilador. Dicha mejora va a ser evaluada en una reducción del tamaño de código objeto generado y, sobre todo, en una mayor velocidad de ejecución del programa objeto.

Una vez presentados los diversos tipos de optimización que se pueden presentar en un compilador, se ha dedicado una especial atención al análisis global de las ecuaciones de flujo de datos. Dentro de este análisis global, se ha desarrollado con atención el denominado análisis **ud** (uso-definición), mostrando las ecuaciones y los posibles tipos de optimización a que da lugar este análisis de un programa.

BIBLIOGRAFÍA

[AHO77] AHO, A. & ULLMAN, J. (1977): *Principles of compiler design.* London, Addison-Wesley.

[AHO86] AHO, A.; SETHI, R.; ULLMAN, J. (1986): *Compilers. Principles, techniques and tools.* London, Addison-Wesley.

[DEJ89] DEJEAN, D. & ZOBRIST, G. (1989): "A Definition optimization technique used in a code translation algorithm". *Communications of the ACM.* 32 (1) pp. 94-105.

[KIL73] KILDALL, G. A. (1973): *A unified approach to global program optimization.* New York, ACM Press.
[WAI84] WAITE, W. & GOOS, G. (1984): *Compiler construction.* Berlin, Springer-Verlag.
[ZIM91] ZIMA, H. (1991): *Supercompilers for parallel and vector computers.* New York, ACM Press.