Un. VII. Optimización

Las optimizaciones pueden realizarse de diferentes formas. Las optimizaciones se realizan en base al alcance ofrecido por el compilador. La optimización va a depender del lenguaje de programación y es directamente proporcional al tiempo de compilación; es decir, entre más optimización mayor tiempo de compilación.

Como el tiempo de optimización es gran consumidor de tiempo (dado que tiene que recorrer todo el árbol de posibles soluciones para el proceso de optimización) la optimización se deja hasta la fase de prueba final. Algunos editores ofrecen una versión de depuración y otra de entrega o final.

La optimización es un proceso que tiene a minimizar o maximizar alguna variable de rendimiento, generalmente tiempo, espacio, procesador, etc. Desafortunadamente no existen optimizadores que hagan un programa más rápido y que ocupe menor espacio.

La optimización se realiza reestructurando el código de tal forma que el nuevo código generado tenga mayores beneficios. La mayoría de los compiladores tienen una optimización baja, se necesita de compiladores especiales para realmente optimizar el código.

7.1 Tipos de optimización.

Optimización de código
La optimización de código puede realizarse durante la propia generación o como paso adicional, ya sea intercalado entre el análisis semántico y la generación de código (se optimizan las cuádruplas) o situado después de ésta (se optimiza a posteriori el código generado).

Hay teoremas (Aho, 1970) que demuestran que la optimización perfecta es indecidible. Por tanto, las optimizaciones de código en realidad proporcionan mejoras, pero no aseguran el éxito total.

Clasificación de optimizaciones:

1. Dependientes de la máquina.
   - Asignación de registros (ver capítulo anterior).
   - Instrucciones especiales ("idioms").
   - Reordenación del código.

2. Independientes de la máquina.
- Ejecución en tiempo de compilación.
- Eliminación de redundancias.
- Cambio de orden.
- Reducción de frecuencia de ejecución (invariancias).
- Reducción de fuerza.

Optimización y depuración suelen ser incompatibles. Por ejemplo, si se elimina totalmente una instrucción, puede ser imposible poner una parada en ella para depuración.
Ejemplo:

```
x = x;
```

Instrucciones especiales ("idiomas")

Algunas máquinas tienen instrucciones especiales que permiten acelerar ciertos procesos.
Por ejemplo:

• TRT en IBM 390 y XLAT en INTEL permiten realizar una codificación en una sola instrucción máquina.
• MOV en IBM 370 permite copiar bloques de memoria de hasta 255 caracteres.
• REP en INTEL permite copiar, llenar o comparar bloques de memoria utilizando como registros índice SI y DI.
• TEST en INTEL permite realizar fácilmente varias comparaciones booleanas.
Ejemplo:

if (x&4 || x&8) ... se puede representar:

```
TEST X,12
JZ L
...
L:
```

Reordenación del código

En muchas máquinas, la multiplicación en punto fijo de dos operandos de longitud 1 da un operando de longitud 2, mientras la división necesita un operando de longitud 2 y otro de longitud 1 para dar un cociente y un resto de longitud 1. Reordenar las operaciones puede optimizar.

Por ejemplo: sea la expresión a=b/c*d;

```
MOV AX,B
XOR DX,DX
DIV AX,C
MUL AX,D
MOV A,AX
```

Si la reordenamos así: a=b*d/c;, aprovechando que la multiplicación y la división son asociativas, tenemos:

```
MOV AX,B
MUL AX,D
DIV AX,C
MOV A,AX
```

Ahorramos una instrucción. Veamos otro ejemplo:

```
a=b/c;
d=b%c;
```

Los dos códigos siguientes son equivalentes. Puede tratarse como un caso particular del manejo de registros. La realización de la primera división debería guardar constancia de que DX contiene el resultado del resto.

```
MOV AX,B        MOV AX,B
XOR DX,DX       XOR DX,DX
DIV AX,C        DIV AX,C
MOV A,AX        MOV A,AX
MOV AX,B        MOV D,DX
XOR DX,DX
DIV AX,C
MOV D,DX
```

Ejecución en tiempo de compilación
Ejemplo:

```
int i;
float f;
i = 2+3;      (+,2,3,t1)        (=,5,,i)
              (=,t1,,i)
i = 4;        (=,4,,i)          (=,4,,i)
f = i+2.5;    (CIF,i,,t2)       (=,6.5,,f)
              (+,t2,2.5,t3)
              (=,t3,,f)
```

La ejecución se aplica principalmente a las operaciones aritméticas (+-*/) y a las conversiones de tipo. La tabla de símbolos puede contener el valor conocido del identificador (ej., i=4), o bien podemos tener una subtabla T con pares (id, valor).

Algoritmo para tratar la ejecución en tiempo de compilación:
• Si la cuádrupla tiene la forma (op, op1, op2, res), donde op1 es un identificador y (op1,v1) está en la tabla T, sustituimos en la cuádrupla op1 por v1.
• Si la cuádrupla tiene la forma (op, op1, op2, res), donde op2 es un identificador y (op2,v2) está en la tabla T, sustituimos en la cuádrupla op2 por v2.
• Si la cuádrupla tiene la forma (op, v1, v2, res), donde v1 y v2 son valores constantes o nulos, eliminamos la cuádrupla, eliminamos de T el par (res, v), si existe, y añadimos a T el par (res, v1 op v2), a menos que v1 op v2 produzca un error, en cuyo caso daremos un aviso y dejaremos la cuádrupla como está.

Ejemplo:
if (false) f = 1/0;

Esta instrucción debe dar un aviso, pero no un error. De hecho, una optimización adicional de código la eliminaría totalmente.

• Si la cuádrupla tiene la forma (=, v1, , res), eliminamos de T el par (res, v), si existe. Si v1 es un valor constante, añadimos a T el par (res, v1).

En el ejemplo:

(+,2,3,t1)	Elim, T = {(t1,5)}
(=,t1,,i)	Sust por (=,5,,i), T = {(t1,5),(i,5)}
(=,4,,i)	T = {(t1,5),(i,4)}
(CIF,i,,t2)	Sust por (CIF,4,,t2), Elim, T = {(t1,5),(i,4),(t2,4.0)}
(+,t2,2.5,t3)	Sust por (+,4.0,2.5,t3) Elim, T = {(t1,5),(i,4),(t2,4.0),(t3,6.5)}
(=,t3,,f)	Sust por (=,6.5,,f)

Y quedan las cuádruplas optimizadas: (=,5,,i), (=,4,,i), (=,6.5,,f).

En cuanto sea posible que los valores de las variables cambien, el compilador debe "olvidar" el valor de las variables (inicializar la tabla T, total o parcialmente).

Esto puede ocurrir si aparece:

- una etiqueta;
- una cuádrupla objetivo de una transferencia;
- una llamada a una subrutina, si se pasan variables por referencia o variables globales;
- una instrucción de lectura externa.

Este proceso no exige la generación de las cuádruplas, puede realizarse directamente durante las rutinas semánticas asociadas al análisis sintáctico, especialmente si es Bottom-up. Problema con la ejecución en tiempo de compilación: si tenemos un "cross-compiler", la precisión puede ser menor en el ordenador que compila que en el que ejecuta.

Eliminación de redundancias

Ejemplo:

```
int a,b,c,d;
a = a+b*c;      (*,b,c,t1)      (*,b,c,t1)
                (+,a,t1,t2)     (+,a,t1,t2)
                (=,t2,,a)       (=,t2,,a)
d = a+b*c;      (*,b,c,t3)
                (+,a,t3,t4)     (+,a,t1,t4)
                (=,t4,,d)       (=,t4,,d)
b = a+b*c;      (*,b,c,t5)
                (+,a,t5,t6)
                (=,t6,,b)       (=,t4,,b)
```

Una solución: el programador podría reescribir su programa así:

```
int a,b,c,d,e;
e = b*c;        (*,b,c,t1)
                (=,t1,,e)
a = a+e;        (+,a,e,t2)
                (=,t2,,a)
d = a+e;        (+,a,e,t3)
                (=,t3,,d)
b = d;          (=,d,,b)
```

Desventaja: esta forma de programar puede ser más larga y menos legible. Además, hay redundancias que el programador no puede eliminar.

Por ejemplo:

array X[0:4, 0:9];

X[i,j]:=X[i,j]+1;	(*,i,10,t1)	(*,i,10,t1)
	(+,t1,j,t2)	(+,t1,j,t2)
	(+,X[t2],1,t3)	(+,X[t2],1,t3)
	(*,i,10,t4)	(:=,t3,,X[t2])
	(+,t4,j,t5)	
	(:=,t3,,X[t5])	

Algoritmo para eliminar redundancias:

• A cada variable de la tabla de símbolos le asignamos la dependencia -1.
• Numeramos las cuádruplas.
• for (i=0; i<número de cuádruplas; i++) {
• Para cada uno de los dos operandos de la cuádrupla: si la cuádrupla de la que depende es (SAME,j,0,0), se sustituye el operando por el resultado de la cuádrupla (j).
• A la cuádrupla (i) le asignamos como dependencia 1 + el máximo de las dependencias de sus operandos.
• Si la cuádrupla (i) tiene como resultado el identificador id, asignamos a id la dependencia i.
• Si la cuádrupla (i) es idéntica a la cuádrupla (j), j<i, excepto por el resultado generado, y las dependencias de ambas son iguales, sustituimos la cuádrupla i por una nula (SAME,j,0,0), que no genera código. En las asignaciones se exige también que el resultado sea el mismo.

Prueba: si j<k<i, y la cuádrupla k cambiara alguno de los operandos de la cuádrupla i, entonces dep(i)>k. Pero dep(j)<=k, luego dep(i)>dep(j) y no se podría eliminar (i).

Reordenación de operaciones

Tener en cuenta la conmutatividad de algunas operaciones puede mejorar el proceso, pues las cuádruplas (*,a,b,-) y (*,b,a,-) serían equivalentes. Para facilitar el reconocimiento, se puede adoptar un orden canónico para los operandos de las operaciones conmutativas. Por ejemplo: términos que no son variables ni constantes, luego variables indexadas por orden alfabético, luego variables sin indexar por orden alfabético, finalmente constantes. Esto mejora también la ejecución en tiempo de compilación.

Por ejemplo, si tenemos las instrucciones

a=1+c+d+3; a=c+d+1+3; b=d+c+2; b=c+d+2; la reordenación nos permite efectuar en tiempo de compilación la operación 1+3, y reconocer c+d como parte

común de las dos instrucciones. Esto no es completo, sin embargo, ya que a=1+c+d+3; a=c+d+1+3; b=d+c+c+d; b=c+c+d+d; la reordenación no nos permite reconocer que c+d, evaluado en la primera instrucción, puede aplicarse a la segunda.

Otra mejora podría ser la utilización de los operadores monádicos para aumentar el número de cuádruplas equivalentes.
Por ejemplo:

a = c-d;	(-,c,d,t1)	(-,c,d,t1)
	(=,t1,,a)	(=,t1,,a)
b = d-c;	(-,d,c,t2)	(-,t1,,t2)
	(=,t2,,b)	(=,t2,,b)

Que no disminuye el número de cuádruplas, pero sustituye una operación diádica por una monádica, que usualmente son más eficientes.

Las variables intermedias para resultados parciales pueden reutilizarse para minimizar la memoria (aunque eso puede ir en contra de las optimizaciones anteriores). Por ejemplo: sea la expresión (a*b)+(c+d). Sus cuádruplas equivalentes serían:

(*,a,b,t1)
(+,c,d,t2)
(+,t1,t2,t1)

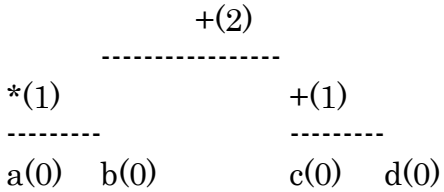
En este caso, utilizamos dos variables auxiliares (t1, t2). Pero si aprovechamos la asociatividad de la suma para reordenar de esta manera:

(*,a,b,t1)
(+,t1,c,t1)
(+,t1,d,t1)

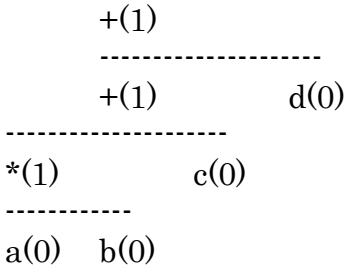
Necesitaremos sólo una variable auxiliar. El número mínimo de variables auxiliares se puede calcular construyendo un grafo de la expresión y aplicando las siguientes reglas:

1. Marcar las hojas con 0.
2. Si (j,k) son las marcas de los hijos del nodo i, si j=k, asociar (k+1) al nodo i, en caso contrario asociarle max(j,k).

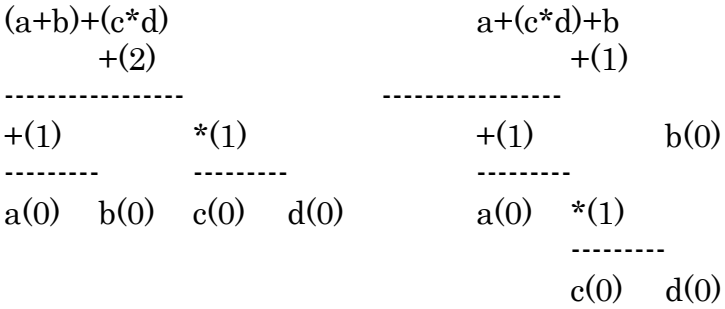
Por ejemplo, el grafo de (a*b)+(c+d) es:



Pero el grafo de ((a*b)+c)+d es:



También se puede aprovechar la conmutatividad, como en el ejemplo:



7.1.1 Locales

• La optimización local se realiza sobre módulos del programa. En la mayoría de las ocasiones a través de funciones, métodos, procedimientos, clases, etc.
• La característica de las optimizaciones locales es que sólo se ven reflejados en dichas secciones.

Optimización Local

• La optimización local sirve cuando un bloque de programa o sección es crítico por ejemplo: la E/S, la concurrencia, la rapidez y confiabilidad de un conjunto de instrucciones.
• Como el espacio de soluciones es más pequeño la optimización local es más rápida

7.1.2 Bucles

• Los ciclos son una de las partes más esenciales en el rendimiento de un programa dado que realizan acciones repetitivas, y si dichas acciones están mal realizadas, el problema se hace N veces más grandes.
• La mayoría de las optimizaciones sobre ciclos tratan de encontrar elementos que no deben repetirse en un ciclo.

Ciclos

• while(a == b)
• {
• int c = a;
• c = 5; …;
• }

• En este caso es mejor pasar el int c =a; fuera del ciclo de ser posible.
• El problema de la optimización en ciclos y en general radica es que muy difícil saber el uso exacto de algunas instrucciones. Así que no todo código de proceso puede ser optimizado.
• Otros uso de la optimización pueden ser el mejoramiento de consultas en SQL o en aplicaciones remotas (sockets, E/S, etc.)

Optimización de bucles

Una operación es invariante respecto a un bucle, si ninguno de los operandos de los que depende cambia de valor durante la ejecución del bucle. La optimización consiste en sacar la operación fuera del bucle.

Otra optimización es la reducción de la fuerza de una operación (sustituir una operación fuerte por otra más débil, como la multiplicación por la suma o la diferencia por el cambio de signo, como en el apartado anterior).

Por ejemplo:

```
for (i=a; i<c; i+=b) {... d=i*k; ...}
```

donde b,k son invariantes respecto al bucle. (b podría ser una expresión, en cuyo caso todos sus operandos deben ser invariantes). Además, i no se modifica dentro del bucle, excepto en la instrucción de cierre, i+=b, y d no se usa ni modifica antes de la instrucción indicada y no se modifica después. En este caso, podemos sustituir el código generado por su equivalente:

```
d=a*k;
t1=b*k;
for (i=a; i<c; i+=b, d+=t1) {...}
```

Con lo que hemos reducido la fuerza de una multiplicación a una suma (dentro del bucle).
Esto no se debe hacer si i o k son reales, pues podría perderse precisión al sumar i veces en vez de multiplicar una. Pero sí se puede hacer si i,k son enteros.
Otro ejemplo:

```
for (i=0; i<10; i++) {... a=(b+c*i)*d; ...}
        INIT: (=,0,,i)
        LOOP:       ...
                (*,c,i,t1)
                (+,b,t1,t2)
                (*,t2,d,t3)
                (=,t3,,a)
                ...
        INCR: (+,i,1,i)
```

donde b,c,d son invariantes respecto al bucle, e i es la variable del bucle. Supongamos que se cumplen todas las condiciones. Podemos aplicar reducción de fuerza a la primera cuádrupla del bucle así:

```
        INIT: (=,0,,i)
                (*,c,0,t1)
                (*,c,1,t4)
        LOOP:       ...
                (+,b,t1,t2)
                (*,t2,d,t3)
                (=,t3,,a)
                ...
        INCR:(+,i,1,i)
                (+,t1,t4,t1)
```

Ahora t1 desempeña el mismo papel que i. Se le asigna un valor inicial y en cada paso del bucle se le incrementa en t4. Por tanto, podemos aplicar reducción de fuerza a la cuádrupla siguiente:

```
        INIT: (=,0,,i)
                (*,c,0,t1)
                (*,c,1,t4)
                (+,b,t1,t2)
        LOOP:       ...
                (*,t2,d,t3)
                (=,t3,,a)
                ...
        INCR:(+,i,1,i)
```

```
        (+,t1,t4,t1)
        (+,t2,t4,t2)
```

Ahora pasa lo mismo con t2, luego podemos aplicar reducción de fuerza a la siguiente cuádrupla:

```
INIT: (=,0,,i)
        (*,c,0,t1)
        (*,c,1,t4)
        (+,b,t1,t2)
        (*,t2,d,t3)
        (*,t4,d,t5)
LOOP:       ...
        (=,t3,,a)
        ...
INCR:(+,i,1,i)
        (+,t1,t4,t1)
        (+,t2,t4,t2)
        (+,t3,t5,t3)
```

Todavía podemos optimizar más notando que ahora t1 y t2 no se emplean dentro del bucle, luego no es necesario incrementarlas:

```
INIT: (=,0,,i)
        (*,c,0,t1)
        (*,c,1,t4)
        (+,b,t1,t2)
        (*,t2,d,t3)
        (*,t4,d,t5)
LOOP:       ...
        (=,t3,,a)
        ...
INCR:(+,i,1,i)
        (+,t3,t5,t3)
```

Si sacamos operaciones fuera de un bucle, pueden quedar dentro de otro bucle más externo. El proceso podría repetirse. Si hay alguna llamada de subrutina dentro del bucle, es difícil saber si se cambia alguna de las variables (podrían ser globales o pasarse como argumento por referencia). En tal caso, sólo pueden aplicarse las optimizaciones si el compilador sabe qué variables se cambian. Esto suele ocurrir sólo para ciertas funciones y subrutinas predefinidas.

Para realizar las optimizaciones pueden hacer falta dos pasos: uno primero, en el que se analizan los bucles y se obtiene información sobre las

variables que cambian, y otro segundo, en el que se realiza la optimización propiamente dicha. Pero también se puede fusionar el proceso con el analizador semántico y el generador de código y hacerlo todo en un solo pasó. Para esto, a veces hay que retrasar o cambiar el orden de algunas de las operaciones del bucle. Por ejemplo, podríamos generar un código como el siguiente:

```
        GOTO INIT
LOOP:   ...
INCR:...
        GOTO TEST
INIT: ...
TEST:   IF (no fin de bucle) GOTO LOOP
```

con lo que INIT y INCR (que son los que cambian con la optimización) quedan al final. Hay que tener cuidado con estas optimizaciones. Si el bucle se ejecuta normalmente 0 (o 1) veces, y es muy raro que se entre en él, las optimizaciones anteriores degradarán (o dejarán invariante) la eficiencia.

7.1.3 Globales

• La optimización global se da con respecto a todo el código.
• Este tipo de optimización es más lenta pero mejora el desempeño general de todo programa.
• Las optimizaciones globales pueden depender de la arquitectura de la máquina.

Optimización global

• En algunos casos es mejor mantener variables globales para agilizar los procesos (el proceso de declarar variables y eliminarlas toma su tiempo) pero consume más memoria.
• Algunas optimizaciones incluyen utilizar como variables registros del CPU, utilizar instrucciones en ensamblador.

7.1.4 De mirilla

• La optimización de mirilla trata de estructurar de manera eficiente el flujo del programa, sobre todo en instrucciones de bifurcación como son las decisiones, ciclos y saltos de rutinas.
• La idea es tener los saltos lo más cerca de las llamadas, siendo el salto lo más pequeño posible.

7.2 Costos

Los costos son el factor más importante a tomar en cuenta a la hora de optimizar ya que en ocasiones la mejora obtenida puede verse no reflejada en el programa final pero si ser perjudicial para el equipo de desarrollo.

La optimización de una pequeña mejora tal vez tenga una pequeña ganancia en tiempo o en espacio pero sale muy costosa en tiempo en generarla.

Costos

• Pero en cambio si esa optimización se hace por ejemplo en un ciclo, la mejora obtenida puede ser N veces mayor por lo cual el costo se minimiza y es benéfico la mejora.
• Por ejemplo: for(int i=0; i < 10000; i++); si la ganancia es de 30 ms 300s

7.2.1 Costo de ejecución.

• Los costos de ejecución son aquellos que vienen implícitos al ejecutar el programa.
• En algunos programas se tiene un mínimo para ejecutar el programa, por lo que el espacio y la velocidad de los microprocesadores son elementos que se deben optimizar para tener un mercado potencial más amplio.

Costos de ejecución

• Las aplicaciones multimedia como los videojuegos tienen un costo de ejecución alto por lo cual la optimización de su desempeño es crítico, la gran mayoría de las veces requieren de procesadores rápidos (e.g. tarjetas de video) o de mucha memoria.
• Otro tipo de aplicaciones que deben optimizarse son las aplicaciones para dispositivos móviles.
• Los dispositivos móviles tiene recursos más limitados que un dispositivo de cómputo convencional razón por la cual, el mejor uso de memoria y otros recursos de hardware tiene mayor rendimiento.
• En algunos casos es preferible tener la lógica del negocio más fuerte en otros dispositivos y hacer uso de arquitecturas descentralizadas como cliente/servidor o P2P.

7.2.2 Criterios para mejorar el código

• La mejor manera de optimizar el código es hacer ver a los programadores que optimicen su código desde el inicio, el problema radica en que el costo podría ser muy grande ya que tendría que codificar más y/o hacer su código más legible.
• Los criterios de optimización siempre están definidos por el compilador
• Muchos de estos criterios pueden modificarse con directivas del compilador desde el código o de manera externa.
• Este proceso lo realizan algunas herramientas del sistema como los ofuscadores para código móvil y código para dispositivos móviles.

7.2.3 Herramientas para el análisis del flujo de datos

• Existen algunas herramientas que permiten el análisis de los flujos de datos, entre ellas tenemos los depuradores y desambladores.
• La optimización al igual que la programación es un arte y no se ha podido sistematizar del todo.