3.1 Tipos de Optimización

La optimización es un proceso que tiene a minimizar o maximizar alguna variable de rendimiento, generalmente tiempo, espacio, procesador, etc.

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.

Desafortunamente no existen optimizador 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.

3.1.1 Optimización Local

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 solo se ven reflejados en dichas secciones. La optimización local sirve cuando un bloque de programa o sección es crítico por ejemplo: 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. Como el espacio de soluciones es más pequeño la optimización local es más rápida.

▪ Las optimizaciones locales se realizan sobre el bloque básico

▪ Optimizaciones locales

– Folding
– Propagación de constantes
– Reducción de potencia
– Reducción de sub expresiones comunes

Bloque Básico

▪ Un bloque básico es un fragmento de código que tiene una única entrada y salida, y cuyas instrucciones se ejecutan secuencialmente. Implicaciones:

– Si se ejecuta una instrucción del bloque se ejecutan todas en un orden conocido en tiempo de compilación.

▪ La idea del bloque básico es encontrar partes del programa cuyo análisis necesario para la optimización sea lo más simple posible.

Ensamblamiento (Folding)

▪ El ensamblamiento es remplazar las expresiones por su resultado cuando se pueden evaluar en tiempo de compilación (resultado constante).

– Ejemplo: A=2+3+A+C -> A=5+A+C

▪ Estas optimizaciones permiten que el programador utilice cálculos entre constantes representados explícitamente sin introducir ineficiencias.

Bloque Básico (ejemplos)

- Ejemplos (separación errónea):

```
for (i=1;i<10;++i) {
    b=b+a[i];
    c=b*i;
}
a=3;
b=4;
goto l1;
c=10;
l1: d=3;
e=4;
```

Implementación del Folding

▪ Implementación del floding durante la generación de código realizada conjuntamente con el análisis sintáctico.

– Se añade el atributo de constante temporal a los símbolos no terminales y a las variables de la tabla de símbolos.

– Se añade el procesamiento de las constantes a las reglas de análisis de expresiones.

– Optimiza: 2+3+b -> 5+b

▪ Hay una suma de constantes (2+3)+b

– No optimiza: 2+b+3 -> 2+b+3

▪ No hay una suma de constantes (2+b)+3

▪ Implementación posterior a la generación de código

– Buscar partes del árbol donde se puede aplicar la propiedad conmutativa:

▪ Sumas/restas: como la resta no es conmutativa se transforma en sumas: a+b-c+d -> a+b+(-c)+d

▪ Productos/divisiones: como la división no es conmutativa se transforma en productos: a*b/c*e -> a*b*(1/c)*e

– Buscar las constantes y operarlas

– Reconstruir el árbol.

Propagación de constantes

▪ Desde que se asigna a una variable un valor constante hasta la siguiente asignación, se considera a la variable equivalente a la constante.

▪ Ejemplo: Propagación Ensamblamiento

PI=3.14 -> PI=3.14 -> PI=3.14

G2R=PI/180 -> G2R=3.14/180 -> G2R=0.017

PI y G2R se consideran constantes hasta la próxima asignación.

▪ Estas optimizaciones permiten que el programador utilice variables como constantes sin introducir ineficiencias. Por ejemplo, en C no hay constantes y será lo mismo utilizar

– Int a=10;

– #define a 10

Con la ventaja que la variable puede ser local.

▪ Actualmente en C se puede definir const int a=10;

Implementación de la Propagación de Constantes

▪ Separar el árbol en bloques básicos

– Cada bloque básico será una lista de expresiones y asignaciones

▪ Para cada bloque básico

– Inicializar el conjunto de definiciones a conjunto vacío.

▪ Definición: (variable,constante)

– Procesar secuencialmente la lista de expresiones y asignaciones

– Para expresión y asignación

▪ Sustituir las apariciones de las variables que se encuentran en el conjunto de definiciones por sus constantes asociadas.

– Para asignaciones

▪ Eliminar del conjunto de definiciones la definición de la variable asignada

▪ Añadir la definición de la variable asignada si se le asigna una constante.

Ejecución en tiempo de compilación
Precalcular expresiones constantes (con constantes o variables cuyo valor no cambia)
i = 2 + 3 ! i = 5
j = 4
f = j + 2.5
!
j = 4
f = 6.5
2. Reutilización de expresiones comunes
a = b + c
d = a - d
e = b + c
f = a - d
!
a = b + c
d = a - d
e = a
f = a - d
3. Propagación de copias
Ante instrucciones f = a, sustituir todos los usos de f por a
a = 3 + i
f = a
b = f + c
d = a + m
m = f + d
!
a = 3 + i
b = a + c
d = a + m
m = a + d

Reducción de potencia

Reemplazar una operación por otra equivalente menos costosa

x2 ! x * x

2*x ! x + x (suma); x<<1 (despl. izq.)

4*x, 8*x,... ! x<<2, x<<3,...

x / 2 ! x>>2

3.1.2 Ciclos

Los programas pasan la mayor parte del tiempo o en ciclos, mientras menos instrucciones tengan, más rápido ejecutan cualquier expresión cuyo valor independiente de la cantidad de veces que se ejecute el ciclo es una invariante del ciclo.

Invariante del ciclo

No tiene sentido evaluarla dentro del ciclo. Se mueve el código fuera del ciclo. El destino del código es el bloque de entrada al ciclo, i es una variable de inducción para un ciclo L si existe una constante c tal que cada vez que se asigna un valor a i, su valor aumenta c – note que c puede ser positiva o negativa.

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.

Sea el ejemplo:

```
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.)

3.1.3 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 instrucción es en ensamblador.

3.1.4 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.

Ideas básicas:

Se recorre el código buscando combinaciones de instrucciones que pueden ser reemplazadas por otras equivalentes más eficientes.

Se utiliza una ventana de n instrucciones y un conjunto de patrones de transformación (patrón, secuencias, remplazan).

Las nuevas instrucciones son reconsideradas para las futuras optimizaciones.

Ejemplos:

Eliminación de cargas innecesarias

Reducción de potencia

Eliminación de cadenas de saltos

Por ejemplo: el “break”

```
Switch (expresión que estamos evaluando)
{
Case 1: cout << “Hola” ;
Break;
Case 2: cout << “amigos”;
Break;
}
```

BIBLIOGRAFÍA

Gómez López, A. (2021). Lenguajes y Autómatas 2 3.1 Tipos de optimización. Recuperado 16 de mayo de 2021, de Blogger. website: https://joseandresgomezlopez1997.blogspot.com/2019/10/31-tipos-de-optimizacion.html

López Figueroa, A. B. (2019). Lenguajes y Autómatas 2 3.1 Tipos de optimización. Recuperado 16 de mayo de 2021, de Blogger. website: https://alexisbladimirlopezfigueroa1998.blogspot.com/2019/10/31-tipos-de-optimizacion.html

Extra:

https://dokumen.tips/documents/lenguajes-y-automatas-unidad-3-competencias-optimizacion.html