Problema de práctica 5.4

Cuando usamos gcc para compilar combine3 con la opción de línea de comandos -O2, obtenemos un código con un rendimiento de CPE sustancialmente mejor que con -O1:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  | Integer | | Floating point | |
| Función | Página | Método | + | \* | + | \* |
| Combinación3 | 549 | Compilado-01 | 7.17 | 9.02 | 9.02 | 11.03 |
| Combinación3 | 549 | Compilado-01 | 1.60 | 3.01 | 3.01 | 5.01 |
| Combinacion4 | 551 | Acumulado temporal | 1.27 | 3.01 | 3.01 | 5.01 |

Logramos un rendimiento comparable al de combine4, excepto en el caso de la suma de enteros, pero incluso mejora significativamente. Al examinar el código ensamblador generado por el compilador, encontramos una variante interesante para el ciclo interno:

Bucle interior de combine3. data\_t = doble, OP = \*. Compilado -O2dest en %rbx, datos+i en %rdx, datos+longitud en %rax

Producto acumulado en %xmm0

1 .L22: bucle:

2 vmulsd (%rdx), %xmm0, %xmm0 Multiplicar producto por datos[i]

3 addq $8, %rdx Incrementar datos+i

4 cmpq %rax, %rdx Comparar con datos+longitud

5 vmovsd %xmm0, (%rbx) Almacenar producto en destino

6 jne . L22 Si !=, ir a bucle

Podemos comparar esto con la versión creada con el nivel de optimización 1:

Bucle interior de combine3. data\_t = doble, OP = \*. Compilado -O1 destino en %rbx, datos+i en %rdx, datos+longitud en %rax

1 . L17: bucle:

2 vmovsd (%rbx), %xmm0 Leer producto de destino

3 vmulsd (%rdx), %xmm0, %xmm0 Multiplicar producto por datos[i]

4 vmovsd %xmm0, (%rbx) Almacenar producto en destino

5 addq $8, %rdx Incrementar datos+i

6 cmpq %rax, %rdx Comparar con datos+longitud

7 jne .L17 Si !=, ir a bucle

Vemos que, además de algunos reordenamientos de instrucciones, la única diferencia es que la versión más optimizada no contiene el vmovsd que implementa la lectura desde la ubicación designada por destino (línea 2).

A. ¿Cómo difiere el papel del registro %xmm0 en estos dos bucles?

B. ¿La versión más optimizada implementará fielmente el código C de combine3, incluso cuando haya un alias de memoria entre dest y los datos vectoriales?

C. Explique por qué esta optimización conserva el comportamiento deseado o proporcione un ejemplo en el que produciría resultados diferentes a los del código menos optimizado.

**Respuestas**:

Este código ensamblador demuestra una oportunidad de optimización inteligente detectada por gcc.

A. El registro %xmm0 se utiliza de manera diferente en los dos bucles. En la versión menos optimizada, se utiliza como un valor temporal que se configura con el valor actual de "dest" en la línea 2 y se utiliza para acumular el producto de los elementos vectoriales en la línea 3. En la versión más optimizada, se utiliza como una variable "acc" que acumula el producto de los elementos vectoriales y se actualiza directamente en la ubicación de destino en la línea 5, sin necesidad de leer desde memoria en la línea 2.

Podemos ver que esta versión optimizada funciona de manera muy similar al siguiente Código C:

1 .L22: bucle:

2 vmulsd (%rdx), %xmm0, %xmm0

3 addq $8, %rdx

4 cmpq %rax, %rdx

5 vmovsd %xmm0, (%rbx)

6 jne .L22

En esta versión, se utiliza el registro %xmm0 como una variable "acc" que acumula el producto de los elementos vectoriales y se actualiza directamente en la ubicación de destino en la línea 5, sin necesidad de leer desde memoria en la línea 2. La optimización radica en que, al actualizar la ubicación de destino en cada iteración, se evita la necesidad de almacenar el valor de "acc" en memoria y luego volver a cargarlo en la siguiente iteración. En lugar de eso, se utiliza el registro %xmm0 para mantener el valor de "acc" en todo momento y actualizar directamente la ubicación de destino en cada iteración.

B. La versión más optimizada implementará fielmente el código C de combine3, incluso con alias de memoria entre "dest" y los datos vectoriales. Esto se debe a que la optimización solo cambia la forma en que se actualiza la ubicación de destino en cada iteración, pero no afecta la forma en que se acumula el producto de los elementos vectoriales. En ambas versiones, el producto de los elementos vectoriales se acumula en el registro %xmm0 y se almacena en la ubicación designada por "dest".

C. Esta optimización conserva el comportamiento deseado del programa porque, con la excepción de la primera iteración, el valor leído desde "dest" al comienzo de cada iteración será el mismo valor escrito en este registro al final de la iteración anterior. Por lo tanto, la instrucción de combinación puede simplemente usar el valor que ya está en %xmm0 al comienzo del ciclo, sin necesidad de leer desde memoria. Esto ahorra tiempo y mejora la eficiencia del código. Sin embargo, si la ubicación de destino cambia en alguna iteración antes de que se escriba en ella, esto podría llevar a resultados diferentes a los del código menos optimizado. Por lo tanto, se debe tener cuidado al usar esta optimización en programas que tienen alias de memoria complejos.

**Problema de práctica 5.5**

Supongamos que deseamos escribir una función para evaluar un polinomio, donde un polinomio de grado n se define para tener un conjunto de coeficientes a0, a1, a2, . . . , un. Para un valor x, evaluamos el polinomio calculando a\_0+a\_1 x+a\_2 x^2 +...+a\_n x^n

Esta evaluación puede ser implementada por la siguiente función, teniendo como argumentos un arreglo de coeficientes a, un valor x, y el grado polinomial grado (el valor n en la Ecuación 5.2). En esta función, calculamos tanto los términos sucesivos de la ecuación como las potencias sucesivas de x dentro de un solo bucle:

1 double poly(double a[], double x, long degree)

2 {

3 long i;

4 double result = a[0];

5 double xpwr = x; /\* Equals x^i at start of loop \*/

6 for (i = 1; i <= degree; i++) {

7 result += a[i] \* xpwr;

8 xpwr = x \* xpwr;

9 }

10 return result;

11 }

A. Para el grado n, ¿cuántas sumas y cuántas multiplicaciones realiza este código?

B. En nuestra máquina de referencia, con operaciones aritméticas que tienen las latencias que se muestran en la Figura 5.12, medimos el CPE para esta función como 5.00. Explique cómo surge este CPE en función de las dependencias de datos formadas entre iteraciones debido a las operaciones que implementan las líneas 7 y 8 de la función.

**Respuestas**

La evaluación de polinomios es una técnica central para resolver muchos problemas. Por ejemplo, las funciones polinómicas se usan comúnmente para aproximar funciones trigonométricas en la biblioteca math.

A. La función realiza n sumas para sumar los términos del polinomio y 2n multiplicaciones para calcular las potencias de x y multiplicarlas por los coeficientes correspondientes.

B. Podemos ver que el cálculo que limita el rendimiento aquí es el cómputo repetido de la expresión xpwr = x \* xpwr. La dependencia de datos entre iteraciones se produce porque el cálculo de xpwr en la línea 8 depende del valor de xpwr de la iteración anterior. Esto significa que el cálculo de una iteración no puede comenzar hasta que se haya completado el de la iteración anterior. Como cada iteración requiere una multiplicación de coma flotante de 5 ciclos de reloj, el tiempo de ejecución de la función se ve limitado por esta dependencia de datos. Por otro lado, la actualización del resultado solo requiere una adición de coma flotante de 3 ciclos de reloj entre iteraciones sucesivas

**Problema de práctica 7.1**

Este problema afecta a los módulos m.o y swap.o de la figura 7.5. Para cada símbolo definido o referenciado en swap.o, indique si tendrá o no una entrada en la tabla de símbolos en la sección .symtab del módulo swap.o. En caso afirmativo, indique el módulo que define el símbolo (swap.o o m.o), el tipo de símbolo (local, global o externo) y la sección (.text, .data, .bss o COMMON) a la que está asignado en el módulo.

|  |  |
| --- | --- |
| (a)m.c | (b)swap.c |
| 1 void swap(); | 1 extern int buf[]; |
| 2 | 2 |
| 3 nt buf[2] = {1, 2}; | 3 int \*bufp0 = &buf[0]; |
| 4 | 4 int \*bufp1; |
| 5 int main() | 5 |
| 6 { | 6 void swap() |
| 7 swap(); | 7 { |
| 8 return 0; | 8 int temp; |
| 9 } | 9 |
|  | 10 bufp1 = &buf[1];bufp1 = |
|  | 11 temp = \*bufp0; |
|  | 12 \*bufp0 = \*bufp1; |
|  | 13 \*bufp1 = temp; |
|  | 14 } |

**Respuesta**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Símbolo | Entrada .symtab | Tipo de símbolo | Módulo where definido | Sección |
| buf | Si | Externo | m.o | .data |
| bufp0 | SI | Global | swap.o | .data |
| bufp1 | Si | Global | swap.o | COMMON |
| swap | SI | Global | swap.o | .text |
| temp | No |  |  |  |

En el programa presentado, hay dos módulos: m.o y swap.o. El módulo m.o define un símbolo externo llamado "buf", que se utiliza en el módulo swap.o. Además, el módulo swap.o define una función llamada "swap" y dos punteros a int llamados "bufp0" y "bufp1", así como una variable local llamada "temp".

buf: Este símbolo se utiliza en swap.o pero se define en m.o. Por lo tanto, habrá una entrada en la tabla de símbolos de swap.o que indica que es un símbolo externo definido en m.o y que pertenece a la sección .data.

bufp0: Este símbolo se define en swap.o y se utiliza en la función swap. Como se define en el mismo módulo donde se utiliza, será un símbolo global en la sección .data de la tabla de símbolos de swap.o.

bufp1: Este símbolo también se define en swap.o y se utiliza en la función swap. Dado que se define en el mismo módulo donde se utiliza, será un símbolo global en la sección COMMON de la tabla de símbolos de swap.o.

swap: Esta función se define en swap.o y se utiliza en el módulo main. Será un símbolo global en la sección .text de la tabla de símbolos de swap.o.

temp: Esta variable se define dentro de la función swap y no se utiliza fuera de ella, por lo que no tendrá una entrada en la tabla de símbolos de swap.o.

**Problema de práctica 7.2**

En este problema, REF(x.i) → DEF(x.k) denota que el enlazador asociará una referencia arbitraria al símbolo x en el módulo i a la definición de x en el módulo k. Para cada ejemplo que sigue, use esta notación para indicar cómo el enlazador resolvería las referencias al símbolo definido de forma múltiple en cada módulo. Si hay un error de tiempo de enlace (regla 1), escriba "ERROR". Si el enlazador elige arbitrariamente una de las definiciones (regla 3), escriba "UNKNOWN".

**Respuesta**

|  |  |
| --- | --- |
| **A**. /\* Module 1 \*/ | /\* Module 2 \*/ |
| int main() | int main; |
| { | int p2() |
| } | { |
|  | } |

A: El enlazador elige el símbolo fuerte definido en el módulo 1 sobre el símbolo débil definido en el módulo 2 (regla 2):

(a) REF(main.1) → DEF(main.1)

(b) REF(main.2) → DEF(main.1)

|  |  |
| --- | --- |
| **B**. /\* Module 1 \*/ | /\* Module 2 \*/ |
| void main() | int main = 1; |
| { | int p2() |
| } | { |
|  | } |

B. Esto es un error, porque cada módulo define un símbolo fuerte principal (regla 1):

(a) REF(main.1) → DEF( . )

(b) REF(main.2) → DEF( . )

|  |  |
| --- | --- |
| **C**. /\* Module 1 \*/ | /\* Module 2 \*/ |
| Int x; | double x = 1.0; |
| void main() | int p2() |
| { | { |
| } | } |

El enlazador elige el símbolo fuerte definido en el módulo 2 sobre el símbolo débil definido en el módulo 1 1 (regla 2):

(a) REF(x.1) → DEF(x.2)

(b) REF(x.2) → DEF(x.2)

**Problema de práctica 7.3**

Deje que a y b denoten módulos de objetos o bibliotecas estáticas en el directorio actual, y deje que a→b denote que a depende de b, en el sentido de que b define un símbolo al que a hace referencia. Para cada uno de los siguientes escenarios, muestre la línea de comando mínima (es decir, una con la menor cantidad de argumentos de archivo de objeto y biblioteca) que permitirá que el vinculador estático resuelva todas las referencias de símbolos.

**A. p.o → libx.a**

En este escenario, se tiene un archivo de objeto "p.o" que hace referencia a símbolos definidos en la biblioteca estática "libx.a". Para resolver estas referencias, se necesita vincular el archivo de objeto con la biblioteca estática. La línea de comando mínima para hacer esto sería:

$ gcc -o prog p.o -L. -lx

Donde el comando "gcc" se utiliza para compilar y vincular archivos de objeto y bibliotecas estáticas en un programa ejecutable. La opción "-o prog" indica el nombre del archivo ejecutable que se creará; "-L." se utiliza para especificar el directorio actual como ubicación de la biblioteca estática y "-lx" se utiliza para especificar el nombre de la biblioteca estática a vincular. En este caso, "libx.a".

**B. p.o → libx.a → liby.a**

Dado un archivo de objeto "p.o" que hace referencia a símbolos definidos en las bibliotecas estáticas "libx.a" y "liby.a". Para resolver estas referencias, primero se necesita vincular el archivo de objeto con la biblioteca "libx.a", y luego vincular el resultado con la biblioteca "liby.a". La línea de comando mínima para hacer esto sería:

$ gcc -o prog p.o -L. -lx -ly

En este código similar al anterior podemos observar una nueva opción "-ly" la cual se utiliza para especificar el nombre de la segunda biblioteca estática a vincular, en este caso, "liby.a".

**C. p.o → libx.a → liby.a and liby.a → libx.a → p.o**

El archivo de objeto "p.o" que hace referencia a símbolos definidos en las bibliotecas estáticas "libx.a" y "liby.a". Además, "liby.a" también hace referencia a símbolos definidos en "libx.a", y "p.o" hace referencia a símbolos definidos en "libx.a". Para resolver estas referencias, se necesita vincular todas las bibliotecas y archivos de objetos en el orden correcto.

$ gcc -o prog p.o -L. -ly -lx

La opción "-ly" se utiliza para especificar el nombre de la primera biblioteca estática a vincular, en este caso, "liby.a".

La opción "-lx" se utiliza para especificar el nombre de la segunda biblioteca estática a vincular, en este caso, "libx.a". Es importante que "libx.a" se incluya después de "liby.a" para asegurar que las referencias a símbolos definidos en "liby.a" se resuelvan primero.

El orden en que se especifican las opciones "-ly" y "-lx" puede ser importante para resolver correctamente todas las referencias a símbolos definidos en las bibliotecas estáticas cuando hay dependencias circulares entre ellas.

En el escenario anterior, hay una dependencia circular entre las bibliotecas "libx.a" y "liby.a". Esto significa que "liby.a" depende de símbolos definidos en "libx.a" y "libx.a" depende de símbolos definidos en "liby.a". Por lo tanto, si se vincula primero "libx.a" y luego "liby.a", es posible que algunas referencias a símbolos definidos en "liby.a" no se resuelvan correctamente. Por tanto se debe vincular "liby.a" primero garantiza que se definan todos los símbolos necesarios antes de vincular "libx.a".