Optimización de programas

Optimización de programas

- **Optimización de programas**. Proceso de modificar un sistema de software para hacer que algún aspecto funcione de manera más eficiente o use menos recursos.
- https://en.wikipedia.org/wiki/Program_optimization

Recomendaciones generales

1. Eliminar trabajo no necesario.
2. Aprovechar que los procesadores modernos pueden ejecutar varias instrucciones a la vez (paralelismo a nivel de instrucción).

Eliminar trabajo no necesario

- Reducir:

1. Ineficiencias en las condiciones de los ciclos.

- Ejemplo:

```
// Si el tamaño de s no cambia, en lugar de:
for (int i = 0; i < s.length(); i++) {  … }
// Usar:
int t = s.length();
for (int i = 0; i < t; i++) { … }
```

Eliminar trabajo no necesario

2. Llamadas a funciones dentro de ciclos (inlining de funciones).
3. Dependencias de datos verdaderas dentro de ciclos.
4. Pruebas condicionales (instrucciones `if` / `else`).
5. Referencias a memoria.

Compiladores y optimización

- Los compiladores modernos emplean algoritmos para optimizar un programa.
- Por ejemplo:

1. Simplificar expresiones.
2. Usar un solo cálculo en varios lugares diferentes (subexpresiones comunes).
3. Reducir la cantidad de veces que se debe realizar un cálculo determinado (desenrollado de ciclos).

Reglas de Pyke

- **Regla 1**. No se puede saber dónde va a pasar el tiempo un programa. Los cuellos de botella ocurren en lugares sorprendentes, así que no intente adivinar y hacer un truco de velocidad hasta que haya probado que es ahí donde está el cuello de botella.
- **Regla 2**. Mida. No ajuste la velocidad hasta que haya medido, e incluso entonces no lo haga a menos que una parte del código *abrume* al resto.

Reglas de Pyke

- **Regla 3**. Los algoritmos elegantes son lentos cuando n es pequeño y n suele ser pequeño. Los algoritmos elegantes tienen constantes grandes. Hasta que sepa que n va a ser grande con frecuencia, no sea elegante. (Incluso si n se hace grande, use la Regla 2 primero).
- Las reglas 4, 5 y 6 pueden consultarse en:
- http://doc.cat-v.org/bell_labs/pikestyle

Desenrollado de ciclos

- Objetivo: quitar los ciclos o reducir el número de iteraciones.
- Busca reducir las dependencias de datos verdaderas y los peligros de control.

Desenrollado total

```
// En lugar de:
for (int i = 0; i < 5; i++) {
    System.out.println("Hola mundo");
}
```

Desenrollado total

```
    # Código MIPS
    li $t0, 0                   # i = 0
ciclo:
    bge $t0, 5, Final           # peligro de control
    …
    syscall                     # printString
    addi $t0, $t0, 1            # i++
    j ciclo
Final:
```

Desenrollado total

```
// Usar:
System.out.println("Hola mundo");
System.out.println("Hola mundo");
System.out.println("Hola mundo");
System.out.println("Hola mundo");
System.out.println("Hola mundo");
```

- No hay peligros de control.

Desenrollado parcial

- En muchos casos no es práctico o posible un desenrollado total.
- Una alternativa es un desenrollado parcial.

Ejemplo: suma de un arreglo

```
// Sin optimizar
int sum = 0;
int size = a.length;
for (int i = 0; i < size; i++) {
    sum += a[i];
}
```

Desenrollado 2 x 1

```
int i = 0, sum = 0, size = a.length;
// Combine 2 elements at a time
for (; i < size - 1; i += 2)
    sum += a[i] + a[i + 1];
// Finish any remaining elements
for (; i < size; i++)
    sum += a[i];
```

Explicación

- El ciclo recorre el arreglo sumando dos elementos a la vez.
- El índice del ciclo i se incrementa en 2 en cada iteración y la suma se aplica a los elementos del arreglo i e $i + 1$ en una sola iteración.
- Si el tamaño del arreglo no es un múltiplo de 2 hay que aplicar la suma al último elemento.
- Se llama “desenrollado 2 x 1” porque el ciclo se desenrolla en un factor de 2 pero los valores se acumulan en un solo acumulador

Desenrollado k x 1

- Se puede generalizar esta idea para desenrollar un ciclo por cualquier factor k.
- El índice de ciclo i se incrementa en k en cada iteración.
- Se le llama “desenrollado de ciclo k x 1”, porque el ciclo se desenrolla en un factor k pero los valores se acumulan en un solo acumulador.
- Por lo general el desenrollado k x 1 **no** es mejor que el 2 x 1.
- Razón: dependencias de datos verdaderas.

Reducir dependencias verdaderas

- Este ciclo tiene dependencias entre iteraciones:

```
// Combine 2 elements at a time
for (i = 0; i < size; i += 2) {
    sum += a[i] + a[i + 1];
}
```

- Una solución es utilizar varios acumuladores y combinarlos al final.

Desenrollado 2 x 2

- Hay dos acumuladores: sum0 y sum1.
- sum0 acumula los valores con índices pares.
- sum1 acumula los valores con índices impares.

Desenrollado 2 x 2

```
int i = 0, sum0 = 0, sum1 = 0, size = a.length;
// Combine 2 elements at a time
for (; i < size - 1; i += 2) {
    sum0 += a[i];
    sum1 += a[i + 1];
}
// Finish any remaining elements
for (; i < size; i++)
    sum0 += a[i];
int sum = sum0 + sum1;
```

Desenrollado 2 x 2

- El desenrollado 2 x 2 puede mejorar el rendimiento en un 100% con respecto al desenrollado 2 x 1.
- El desenrollado 3 x 3 puede mejorar el rendimiento en un 50% con respecto al desenrollado 2 x 2.

Limitaciones

- Si el desenrollado de ciclos 3 x 3 es mejor que el 2 x 2, entonces ¿el desenrollado 20 x 20 será mejor?
- Respuesta: tal vez no.
- Motivo: derrame de registros.
- Los procesadores tienen un número finito de registros.
- Una versión con ciclos desenrollados utiliza más registros que la versión original.
- Si al compilador se le terminan los registros ocurre un derrame y comenzará a utilizar la memoria para variables temporales.

Peligros de control

- Cuando se encuentra un brinco, el procesador debe especular cuál es la siguiente instrucción.
- Si la predicción es correcta, el procesador continua ejecutando instrucciones.
- Si la predicción es incorrecta, el procesador debe descartar las instrucciones ejecutadas especulativamente y reiniciar el proceso de obtener instrucciones de la ubicación correcta.
- El castigo por una predicción errónea puede ser de hasta 20 ciclos de reloj.

Peligros de control

- ¿Qué puede hacer un programador para que el castigo no afecte la eficiencia de un programa?
- No hay una respuesta simple.
- Pero se aplican los siguientes principios generales:

1. No hacer nada.
2. Escribir código buscando que el compilador genere instrucciones de movimientos condicionales (lenguaje ensamblador x86).

No hacer nada

- La predicción de brincos en los procesadores modernos es muy buena (90% de aciertos).

Movimiento condicional (x86)

- Las instrucciones CMOVcc verifican el estado de una o más de las banderas de estado en el registro EFLAGS (CF, OF, PF, SF y ZF).
- Si la condición se cumple, se realiza el movimiento.
- Si la condición no se cumple, no se realiza un movimiento y la ejecución continúa con la instrucción que sigue a la instrucción CMOVcc.

Movimiento condicional (x86)

```
cmp r0, r1      ; compara r0 con r1
cmove r2, r3    ; si ZF == 1 (r0 y r1 son iguales), r2 ← r3
```

Ventajas

- Se elimina la especulación de brincos.
- No hay castigo por una posible mala predicción.

Problemas

- La generación de instrucciones CMOVcc no puede ser controlado directamente por el programador.
- gcc puede generar movimientos condicionales para código escrito en un estilo más “funcional” a diferencia de un estilo más “imperativo”.
- En un estilo imperativo se usan operaciones condicionales (if).
- En un estilo funcional se usan instrucciones condicionales (operador ternario).

Ejemplo – estilo imperativo

```
// Calcula |x – y|
long absdiff(long x, long y)
{
    long result;
    if (x < y)
        result = y – x;
    else
        result = x – y;
    return result;
}
```

Ejemplo – estilo funcional

```
// Calcula |x – y|
long cmovdiff(long x, long y) {
    long rval = y – x;
    long eval = x – y;
    long ntest = x >= y;
    // Line below requires single instruction
    if (ntest) rval = eval;
    return rval;
}
```

Ejemplo

- Programar una función que reciba dos arreglos de enteros a y b y que, al final, para cada posición i, $a[i]$ tenga el mínimo de $a[i]$ y $b[i]$, y $b[i]$ el máximo.

Ejemplo – estilo imperativo

```
// Rearrange two vectors so that for each i, b[i] >= a[i]
void minmax1(long a[], long b[], long n) {
    long i;
    for (i = 0; i < n; i++) {
        if (a[i] > b[i]) {
            long t = a[i];
            a[i] = b[i];
            b[i] = t;
        }
    }
}
```

Ejemplo – estilo funcional

```
// Rearrange two vectors so that for each i, b[i] >= a[i]
void minmax2(long a[], long b[], long n) {
    long i;
    for (i = 0; i < n; i++) {
        long min = a[i] < b[i] ? a[i] : b[i];
        long max = a[i] < b[i] ? b[i] : a[i];
        a[i] = min;
        b[i] = max;
    }
}
```

Comparación

- El libro CS:APP (p. 580) reporta que la versión funcional (minmax2) es 3.4 veces más rápida que la versión imperativa (minmax1) para números aleatorios.

Conclusión

- Se requiere cierta cantidad de experimentación, escribiendo diferentes versiones de la función y luego examinando el código ensamblador generado y midiendo el rendimiento.

Resumen

- Estrategias básicas para mejorar el rendimiento:

1. Diseño de alto nivel:
   - Algoritmos
   - Estructuras de datos
2. Principios básicos de codificación:
   - Eliminar el exceso de llamadas a funciones
   - Eliminar las referencias de memoria innecesarias
   - Utilizar librerías SIMD (p.e. NumPy para Python)
   - Utilizar funciones paralelas (p.e `parallelSort` vs `sort` en Java)

Resumen

3. Optimizaciones de bajo nivel:
   - Desenrollar ciclos
   - Aumentar el paralelismo
   - Reescribir las operaciones condicionales en un estilo funcional