Optimización código: Condicionales, Bucles y funciones

Optimizando funciones:

Las llamadas a funciones inhiben la optimización. Por lo tanto, el rendimiento en torno a funciones pequeñas que se invocan con frecuencia mejora drásticamente si la función llamada está insertada.

Source File

```
callee(int *c, int d)
{
    *c = d;
}

caller (int *a, int *b)
{
    callee(a, 5);
    callee(b, 7);
}
```

Optimizer Comments After Compiling with –o2

```
_caller:
;** 8   ----------------------    callee(a, 5);
;** 9   ----------------------    callee(b, 7);
;** 9   ----------------------    return;
```

Optimizer Comments After Compiling with –o3

```
_caller:
;** 3   ----------------------    *a = 5;  // [0]
;** 3   ----------------------    *b = 7;  // [0]
;** 3   ----------------------    return;  // [0]
. . .
;; Inlined function references:
;; [  0] callee
```

Optimizando condicionales: Conversión “else” en “if”

El compilador C6000 convertirá pequeñas declaraciones "if" (declaraciones "if" con bloques "if" y "else" que son cortos o vacíos).

Ejemplo, transformará:

```
if (p) x=5; else x=7;
```

en:

```
[ p] x = 5
[!p] x = 7
```

Optimizando condicionales: Reducción del condicional

Suponga que se utilizan <i1> hasta <im> para calcular el valor de y [i] sin otros efectos secundarios. Entonces solo se debe proteger la asignación ay. El tamaño de la instrucción "if" se puede reducir radicalmente sacando <i1> a <im>.

Esta optimización es rentable si "x" es densa. Puede que no sea rentable si "x" es escasa, porque <i1> a <im> se ejecutarían con mucha más frecuencia que en el ciclo original, lo que podría superar los beneficios del pipelining.

Optimizando condicionales: Reducción del condicional

Original Loop

```
for (i=0; i<n; i++)
{
    if (x[i])
    {
        <i1>
        <i2>
        ...
        <in>
        y[i] += ...
    }
}
```

Hand-Tuned Loop After "If" Statement Reduction

```
for (i=0; i<n; i++)
{
    <i1>
    <i2>
    ...
    <in>
    if (x[i])
    {
        y[i] += ...
    }
}
```

Optimizando condicionales: Eliminación de condicional

Una alternativa para reducir las declaraciones de condicionales es eliminarlas por completo.

Ocasionalmente, el compilador lo hace automáticamente. De lo contrario, la transformación se puede aplicar manualmente.

Optimizando condicionales: Eliminación de condicional

Original Loop

```
for (i=0; i<n; i++)
{
    if (x[i])
    {
        <i1>
        <i2>
        ...
        <im>
        y[i] += ...
    }
}
```

Hand-Tuned Loop After "If" Statement Elimination

```
for (i=0; i<n; i++)
{
    <i1>
    <i2>
    ...
    <im>
    p = (x[i] != 0);
    y[i] += p * (...);
}
```

Optimizando condicionales: Usando intrínsecos

Pasar de:

```
if (a>b) max = a; else max = b;
```

a:

```
max = _max2(a, b);
```

Optimizando condicionales: Reducción vía una consolidación común

Con frecuencia, el bloque "if" y el bloque "else" tienen un código común:

Original Loop

```
for (i=0; i<n; i++)
{
    if (x[i])
    {
        int t = z[i];
        t    += ...
        y[i]  = t;
        x[i]= ...

    }
    else
    {
        int t = z[i];
        y[i]  = t;
    }
}
```

Hand-Tuned Loop After Common Code Consolidation

```
for (i=0; i<n; i++)
{
    int t = z[i];
    if (x[i])
        t += ...
    y[i] = t;
    if (x[i])
        x[i] = ...
}
```

Optimizando bucles: MUST_ITERATE

Los argumentos min y max son recuentos mínimos y máximos de viajes garantizados por el programador. El recuento de viajes es el número de veces que se repite un bucle. El recuento de viajes del bucle debe ser uniformemente divisible por múltiplos. Todos los argumentos son opcionales

#pragma MUST_ITERATE(límite_inferior, límite_superior, factor)

Optimizando bucles: MUST_ITERATE

El factor es si es múltiplo de un número. Ejemplo:

pragma MUST_ITERATE(8, 48, 8);

for(i = 0; i < trip_count; i++) { ...

Este ejemplo le dice al compilador que el ciclo se ejecuta entre 8 y 48 veces y que la variable “trip_count” es un múltiplo de 8 (8, 16, 24, 32, 40, 48). El argumento múltiple permite al compilador desenrollar el bucle.

Optimizando bucles: Dividir bucles

Los bucles con una gran cantidad de instrucciones también tardan mucho en compilarse. En muchos casos, se puede lograr un mejor rendimiento dividiendo bucles excesivamente grandes. La sobrecarga del bucle adicional podría compensarse con creces por el programa más eficiente que el compilador podrá generar para los dos bucles más pequeños en relación con el bucle grande.

Optimizando bucles: Dividir bucles

Original Loop

```
#MUST_ITERATE(1,20)
for (i=0; i<n; i++)
{
    int v = 0;
    if (x[i])
    {
        <largeblock1>
        v =...
    }
    if (v)
    {
        <largeblock2>
    }
}
```

Hand-Tuned Loops After Scalar Expansion/Loop Splitting

```
int tmp[20];
#MUST_ITERATE(1,20)
for (i=0; i<n; i++)
{
    int v = 0;
    if (x[i])
    {
        <largeblock1>
        v = ...
    }
    tmp[i] = v;
}

#MUST_ITERATE(1,20,)
for (i=0; i<n; i++)
{
    int v = tmp[i];
    if (v)
    {
        <largeblock2>
    }
}
```

Optimizando bucles: Desenrollar bucle

Bucle normal	Bucle desenroscado

`for (int indice = 0; indice < 100; ++indice)`
`{`
`    borrar(indice);`
`}`
	`for (int indice = 0; indice < 100; indice += 5)`
	`{`
	`    borrar(indice);`
	`    borrar(indice + 1);`
	`    borrar(indice + 2);`
	`    borrar(indice + 3);`
	`    borrar(indice + 4);`
	`}`

Optimizando bucles: Desenrollar bucle con Pragma Unroll

El pragma UNROLL especifica al compilador cuántas veces se debe desenrollar un bucle.

```
#pragma UNROLL( n );
```

Bibliografía

1. Hand-Tuning Loops and Control Code on the TMS320C6000
2. TMS320C6000 Programmer's Guide
3. TMS320C6000 Optimizing Compiler v7.4