Optimización de programas

Libro

- La mayor parte de este tema y las imágenes están tomadas de:

  Bryant Randal E. and O'Hallaron David R.

  Computer Systems: A Programmers Perspective.

  Pearson Education Limited, 2016

Introducción

- Objetivo principal de un programa: trabajar correctamente bajo todas las posibles condiciones.
- Un programa que corre rápido pero produce resultados incorrectos no sirve.
- **Optimización de programas**. Es el proceso de modificar un sistema de software para hacer que algún aspecto funcione de manera más eficiente o use menos recursos.
- Ver: https://en.wikipedia.org/wiki/Program_optimization

¿Cuándo se hace la optimización?

- Respuesta: depende del programa.
- En un videojuego el rendimiento (p.e. 60 frames por segundo) puede ser parte de la especificación.
- En otros casos, algún tipo de rendimiento puede estar implícito desde la especificación.
- No es recomendable dejar la optimización para el final.
- La arquitectura del programa puede hacer imposible alcanzar una meta específica de rendimiento.
- Tampoco es recomendable la *optimización prematura*.

¿Cuándo se hace la optimización?

- "*Deberíamos* olvidarnos de las pequeñas eficiencias, digamos alrededor del 97% del tiempo: la optimización prematura es la raíz de todos los males. Sin embargo, no debemos dejar pasar nuestras oportunidades en ese crítico 3%."
- Donald Knuth, **Structured Programming with go to Statements**, Computing Surveys, Vol. 6, No. 4, December 1974, p. 268
- https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.103.6084

¿Qué es optimización prematura?

- La optimización prematura intenta optimizar el rendimiento:

1. Al codificar por primera vez un algoritmo.
2. Antes de que los benchmarks confirmen la necesitad de hacerlo.
3. Antes de que el profiler señale en dónde tiene sentido optimizar.

- https://www.toptal.com/freelance/curse-premature-optimization

Reglas de Pyke

- http://doc.cat-v.org/bell_labs/pikestyle
- **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). Por ejemplo, los árboles binarios son más rápidos que los árboles splay para los problemas cotidianos.

Reglas de Pyke

- **Regla 4**. Los algoritmos elegantes tienen más errores que los simples y son mucho más difíciles de implementar. Use algoritmos simples así como estructuras de datos simples. Las siguientes estructuras de datos son una lista completa para casi todos los programas prácticos: arreglos, listas ligadas, tablas de hash, árboles binarios.
- Por supuesto, también debe estar preparado para combinarlos en estructuras de datos compuestas. Por ejemplo, una tabla de símbolos podría implementarse como una tabla hash que contiene listas ligadas de arreglos de caracteres.

Reglas de Pyke

- **Regla 5**. Los datos dominan. Si ha elegido las estructuras de datos correctas y ha organizado bien las cosas, los algoritmos casi siempre serán evidentes. Las estructuras de datos, no los algoritmos, son fundamentales para la programación. (Consulte The Mythical Man-Month: Essays on Software Engineering de F. P. Brooks, página 102: **la representación es la esencia de la programación**).
- **Regla 6**. No hay Regla 6.

Actividades

- Actividades para escribir un programa eficiente:

1. Seleccionar un conjunto apropiado de algoritmos y estructuras de datos.
2. Escribir código fuente que el compilador pueda optimizar para generar código ejecutable eficiente.
3. Dividir la tarea en porciones que puedan ser ejecutadas en paralelo.

Código optimizado

- Paso 1. Eliminar trabajo no necesario.
- Incluye eliminar llamadas a funciones, pruebas condicionales y referencias a memoria que no son necesarias.
- Paso 2. Explotar la capacidad de los procesadores para proporcionar paralelismo a nivel de instrucción, ejecutando múltiples instrucciones simultáneamente.
- Incluye reducir las dependencias de datos entre diferentes partes de un programa, aumentando el grado de paralelismo con el que se pueden ejecutar.

Compiladores y optimización

- Los compiladores modernos emplean algoritmos sofisticados 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.

Compiladores y optimización

- Los compiladores solo aplican optimizaciones *seguras*.
- El programa optimizado debe tener el mismo comportamiento que el programa original para todos los casos posibles.
- Esto limita algunas posibles optimizaciones.

Ejemplo

- ¿Cuál es más eficiente? ¿Hacen lo mismo?

```
void twiddle1(long *xp, long *yp) {
    *xp = *xp + *yp;
    *xp = *xp + *yp;
}

void twiddle2(long *xp, long *yp) {
    *xp = *xp + 2 * *yp;
}
```

Análisis

- La función twiddle1 requiere 6 referencias a memoria (2 lecturas de *xp, dos lecturas de *yp, dos escrituras de *xp).
- La función twiddle2 requiere 3 referencias a memoria (1 lectura de *xp, 1 lectura de *yp, 1 escritura de *xp).
- Conclusión: twiddle2 es más eficiente que twiddle1.

Prueba 1

```
int main()
{
    long x = 10;
    long y = 20;

    twiddle1(&x, &y);
    printf("x: %ld, y: %ld\n", x, y);   // imprime x: 50, y: 20

    return 0;
}
```

Prueba 1

```
int main()
{
    long x = 10;
    long y = 20;

    twiddle2(&x, &y);
    printf("x: %ld, y: %ld\n", x, y);   // imprime x: 50, y: 20

    return 0;
}
```

Conclusión

- Parece que twiddle1 y twiddle2 hacen lo mismo.

Prueba 2

```
int main()
{
    long x = 10;
    long y = 20;

    twiddle1(&x, &x);
    printf("x: %ld, y: %ld\n", x, y);   // imprime x: 40, y: 20

    return 0;
}
```

Prueba 2

```
int main()
{
    long x = 10;
    long y = 20;

    twiddle2(&x, &x);
    printf("x: %ld, y: %ld\n", x, y);   // imprime x: 30, y: 20

    return 0;
}
```

Conclusión

- twiddle1 y twiddle2 **no** hacen lo mismo.

Explicación

- Si xp y yp apuntan a la misma dirección, twiddle1 hace lo siguiente:

```
*xp = *xp + *xp    // duplica el valor en xp
*xp = *xp + *xp    // duplica el valor en xp
```

- El resultado es que el valor en xp se incrementa en un factor de 4.
- La función twiddle2 hace lo siguiente:

```
*xp = *xp + 2 * *xp    // triplica el valor en xp
```

- El resultado es que el valor en xp se incrementa en un factor de 3.

Conclusión

- El compilador no sabe cómo twiddle1 será invocada.
- Debe suponer que los argumentos xp y yp pueden tener el mismo valor.
- Por lo tanto, no puede generar el código de twiddle2 como una versión optimizada de twiddle1.

Alias de memoria

- **Alias de memoria (memory aliasing)**. Cuando dos apuntadores apuntan a la misma localidad de memoria.
- El compilador solo debe aplicar optimizaciones seguras.
- Por lo tanto, debe suponer que diferentes apuntadores pueden estar aliased.

Ejemplo

```
x = 1000;
y = 3000;
*q = y;     // 3000
*p = x;     // 1000
t1 = *q;    // 1000 o 3000, dependiendo si p y q apuntan a la misma dirección o no
```

Versión 1

```
int main()
{
   int *p = (int *) malloc(sizeof(int));
   int *q = (int *) malloc(sizeof(int));
   int x = 1000;
   int y = 3000;
   *q = y;          // 3000
   *p = x;          // 1000
   int t1 = *q;
   printf("%d\n", t1);    // imprime 3000
   return 0;
}
```

Versión 2

```
int main()
{
    int *p = (int *) malloc(sizeof(int));
    int *q = p;
    int x = 1000;
    int y = 3000;
    *q = y;          // 3000
    *p = x;          // 1000
    int t1 = *q;
    printf("%d\n", t1);    // imprime 1000
    return 0;
}
```

Bloqueadores de optimización

- **Bloqueador de optimización**. Aspectos de un programa que pueden limitar las oportunidades para que el compilador genere código optimizado.
- El alias de memoria es un bloqueador de optimización.
- Las llamadas a funciones pueden ser bloqueadores de optimización.

Ejemplo

- ¿Cuál es más eficiente? ¿Hacen lo mismo?

```
long f();
long func1() {
    return f() + f() + f() + f();
}

long func2() {
    return 4 * f();
}
```

Ejemplo

- La función `func1` hace 4 llamadas a `f`.
- La función `func2` hace 1 llamada a `f`.
- La función `func2` “parece” una versión optimizada de `func1`.

Ejemplo

- Suponer que la función f tiene efectos secundarios.
- **Efecto secundario**. La función modifica alguna parte del estado global del programa.

```
long counter = 0;


long f()
{
    return counter++;
}
```

Ejemplo

- La llamada a func1 regresa 0 + 1 + 2 + 3 = 6.
- La llamada a func2 regresa 4 * 0 = 0.
- Muchos compiladores no intentan determinar si una función está libre de efectos secundarios y dejan intactas las llamadas a funciones.

Sustitución inline

- **Sustitución inline**. La llamada a una función se sustituye por el código del cuerpo de la función.
- El compilador puede generar código optimizado a partir de la sustitución.
- No todos los compiladores ofrecen sustitución inline por default.

Ejemplo

```
/* Resultado de hacer inline de f en func1 */
long func1in()
{
    long t = counter++;   /* +0 */
    t += counter++;       /* +1 */
    t += counter++;       /* +2 */
    t += counter++;       /* +3 */
    return t;
}
```

Ejemplo

```
/* Optimización del código inlined */
long func1opt()
{
    long t = 4 * counter + 6;
    counter += 4;
    return t;
}
```

Rendimiento del programa

- Se expresa mediante la métrica CPE (ciclos por elemento).
- Por ejemplo, CPE = 5 indica que si el problema tiene un tamaño n, el programa va a ejecutarse en $5 * n$ ciclos de reloj.

Ejemplo

- Implementar una función para obtener la suma prefijo (suma acumulada) definida como:
- Dado un vector $\vec{a} = \langle a_0, a_1, \ldots, a_{n-1} \rangle$, la suma prefijo es el vector $\vec{p} = \langle p_0, p_1, \ldots, p_{n-1} \rangle$.
- Donde:
- $p_0 = a_0$
- $p_1 = a_0 + a_1$
- $p_2 = a_0 + a_1 + a_2$
- …

Ejemplo

- Es decir:
- $p_0 = a_0$
- $p_i = p_{i-1} + a_i, \quad 1 \leq i < n$
- Se presentan dos soluciones:
- La función $\mathrm{psum1}$ calcula un elemento del vector resultado por iteración.
- La función $\mathrm{psum2}$ desenrolla el ciclo y calcula dos elementos del vector resultado por iteración.

Ejemplo

```
/* Calcula la suma prefijo del vector a */
void psum1(float a[], float p[], long n)
{
    long i;
    p[0] = a[0];
    for (i = 1; i < n; i++)
        p[i] = p[i – 1] + a[i];
}
```

Ejemplo

```
/* Calcula la suma prefijo del vector a */
void psum2(float a[], float p[], long n) {
    long i;
    p[0] = a[0];
    for (i = 1; i < n – 1; i += 2) {
        float mid_val = p[i – 1] + a[i];
        p[i] = mid_val;
        p[i + 1] = mid_val + a[i + 1];
    }
```

Ejemplo

```
    /* Para n par, termina el elemento faltante */
    if (i < n)
        p[i] = p[i – 1] + a[i];
}
```

Ejemplo

- La figura muestra el número de ciclos requieren las dos funciones para un rango de valores de n.

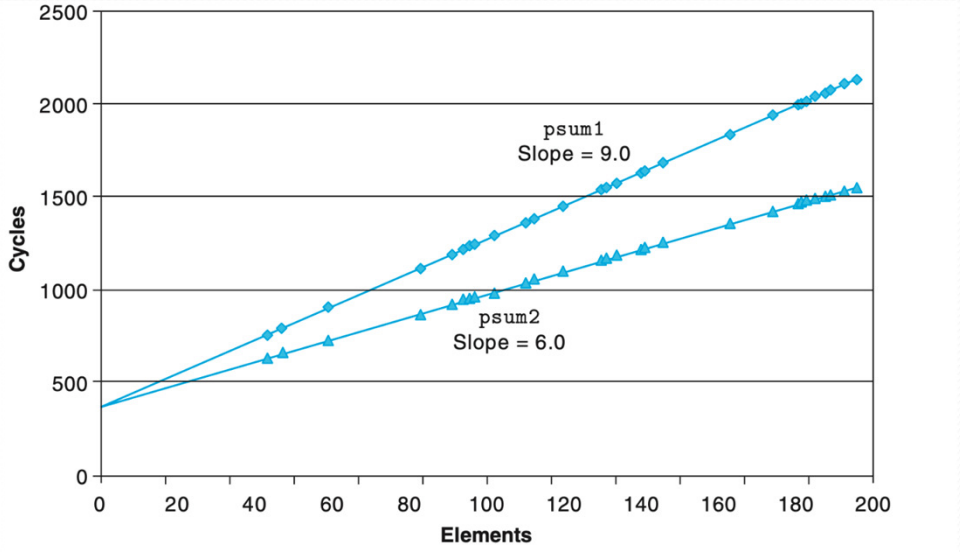

Figure 5.2 **Performance of prefix-sum functions.** The slope of the lines indicates the number of clock cycles per element (CPE).

Ejemplo

- Un ajuste de mínimos cuadrados determina que el número de ciclos se puede aproximar por:
- Para psum1: $C = 368 + 9n$
- Para psum2: $C = 368 + 6n$
- Las ecuaciones indican que hay un overhead de 368 ciclos debido al inicio y finalización de las funciones.
- El resultado es que psum1 tiene un CPE = 9 y psum2 tiene un CPE = 6.

Ejemplo de optimización

- Implementar en C el tipo de datos abstractos Vector.

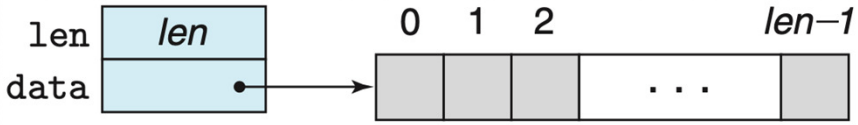

Figure 5.3 **Vector abstract data type.** A vector is represented by header information plus an array of designated length.

ADT Vector

- El ADT Vector tiene 3 funciones:

1. vec_ptr new_vec(long len) crea un vector de la longitud especificada.
2. int get_vec_element(vec_ptr v, long index, data_t *dest) guarda el elemento dado por index en dest. Regresa 0 (index está fuera de rango) o 1 (éxito).
3. long vec_length(vec_ptr v) regresa el número de elementos del vector.

Estructura header

```
/* Create abstract data type for vector */
typedef struct {
    long len;
    data_t *data;
} vec_rec, *vec_ptr;
```

- La declaración data_t designa el tipo de los elementos del vector.
- Por ejemplo, si los elementos son enteros se declara:

```
typedef int data_t;
```

Implementación

```
/* Create vector of specified length */
vec_ptr new_vec(long len)
{
    /* Allocate header structure */
    vec_ptr result = (vec_ptr) malloc(sizeof(vec_rec));
    data_t *data = NULL;
    if (!result)
        return NULL;     /* Couldn't allocate storage */
    result->len = len;
```

Implementación

```
    /* Allocate array */
    if (len > 0) {
        data = (data_t *) calloc(len, sizeof(data_t));
        if (!data) {
            free((void *) result);
            return NULL;     /* Couldn't allocate storage */
        }
    }
    /* Data will either be NULL or allocated array */
    result->data = data;
    return result;
}
```

Implementación

```
/*
* Retrieve vector element and store at dest.
* Return 0 (out of bounds) or 1 (successful)
*/
int get_vec_element(vec_ptr v, long index, data_t *dest)
{
    if (index < 0 || index >= v->len)
        return 0;
    *dest = v->data[index];
    return 1;
}
```

Implementación

```
/* Return length of vector */
long vec_length(vec_ptr v)
{
    return v->len;
}
```

Operación combine

- La operación combine combina los elementos del vector en un resultado de acuerdo a una operación.
- Suma: $s = v_0 + v_1 + \cdots + v_{n-1}$
- Producto: $p = v_0 \times v_1 \times \cdots \times v_{n-1}$

Versión 1 de combine

- Para la suma:

  ```
  #define IDENT 0
  #define OP +
  ```

- Para el producto:

  ```
  #define IDENT 1
  #define OP *
  ```

Versión 1 de combine

```
/* Implementation with maximum use of data abstraction */
void combine1(vec_ptr v, data_t *dest)
{
    long i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

Rendimiento de la versión 1

- Rendimiento (CPE) en una computadora con una CPU Intel Core i7 Haswell:

			Integer		Floating point	
Function	Page	Method	+	*	+	*
`combine1`	543	Abstract unoptimized	22.68	20.02	19.98	20.18
`combine1`	543	Abstract `-O1`	10.12	10.12	10.17	11.14

Versión 2 de combine

- La versión 1 de combine invoca a vec_length en el ciclo for.

```
for (i = 0; i < vec_length(v); i++) { … }
```

- Sin embargo, el tamaño del vector es constante.
- La versión 2 de combine mueve la llamada a vec_length fuera del ciclo for.

Versión 2 de combine

```
/* Move call to vec_length out of loop */
void combine2(vec_ptr v, data_t *dest)
{
    long i;
    long length = vec_length(v);
    *dest = IDENT;
    for (i = 0; i < length; i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

Rendimiento de la versión 2

- CPE:

			Integer		Floating point	
Function	Page	Method	+	*	+	*
`combine1`	543	Abstract `-O1`	10.12	10.12	10.17	11.14
`combine2`	545	Move `vec_length`	7.02	9.03	9.02	11.03

Versión 3 de combine

- La versión 2 invoca a get_vec_element dentro del ciclo for:

```
for (i = 0; i < length; i++) {
    …
    get_vec_element(v, i, &val);
    …
}
```

- La versión 3 obtiene un apuntador al arreglo de datos y obtiene el elemento a partir de ese arreglo.

Apuntador del arreglo

- A la implementación del TDA Vector se le agrega la siguiente función:

```
data_t *get_vec_start(vec_ptr v)
{
    return v->data;
}
```

Versión 3 de combine

```
/* Direct access to vector data */
void combine3(vec_ptr v, data_t *dest)
{
    long i;
    long length = vec_length(v);
    data_t *data = get_vec_start(v);
    *dest = IDENT;
    for (i = 0; i < length; i++) {
        *dest = *dest OP data[i];
    }
}
```

Rendimiento de la versión 3

- CPE:

			Integer		Floating point	
Function	Page	Method	+	*	+	*
`combine2`	545	Move `vec_length`	7.02	9.03	9.02	11.03
`combine3`	549	Direct data access	7.17	9.02	9.02	11.03

Versión 4 de combine

- La versión 3 de combine acumula el valor calculado en la localidad apuntada por dest:

```
void combine3(vec_ptr v, data_t *dest)
    …
    for (i = 0; i < length; i++) {
        *dest = *dest OP data[i];
    }
```

- Esto se puede ver examinando el código ensamblador de combine3.

Código ensamblador de combine3

```
; Inner loop of combine3. data_t = double, OP = *
; dest in %rbx, data+i in %rdx, data+length in %rax
.L17:
    vmovsd (%rbx), %xmm0          ; read product from dest
    vmulsd (%rdx), %xmm0, %xmm0   ; multiply product by data[i]
    vmovsd %xmm0, (%rbx)          ; store product at dest
    addq $8, %rdx                 ; increment data + i
    cmpq %rax, %rdx               ; compare to data + length
    jne .L17                      ; if != goto loop
```

Explicación

- El valor acumulado se lee y se escribe en la memoria en cada iteración.
- Esta lectura y escritura es un desperdicio, ya que el valor leído de dest al comienzo de cada iteración es el valor escrito al final de la iteración anterior.
- La versión 4 utiliza una variable local en lugar del apuntador.

Versión 4 de combine

```
/* Accumulate result in local variable */
void combine4(vec_ptr v, data_t *dest)
{
    long i;
    long length = vec_length(v);
    data_t *data = get_vec_start(v);
    data_t acc = IDENT;
    for (i = 0; i < length; i++) {
       acc = acc OP data[i];
    }
    *dest = acc;
}
```

Código ensamblador de combine4

```
; Inner loop of combine4.  data_t = double, OP = *
; acc in %xmm0, data+i in %rdx, data+length in %rax
.L25:
    vmulsd (%rdx), %xmm0, %xmm0      ; multiply acc by data[i]
    addq $8, %rdx.                   ; increment data + i
    cmpq %rax, %rdx                  ; compare to data + length
    jne                              ; if != goto loop
```

Rendimiento de combine4

- CPE:

			Integer		Floating point	
Function	Page	Method	+	*	+	*
`combine3`	549	Direct data access	7.17	9.02	9.02	11.03
`combine4`	551	Accumulate in temporary	1.27	3.01	3.01	5.01

Conclusión

- Algunas recomendaciones para mejorar el rendimiento de un programa:

1. Eliminar ineficiencias en las condiciones de los ciclos.
2. Reducir las llamadas a funciones dentro de los ciclos.
3. Eliminar referencias innecesarias a la memoria.

- Estas optimizaciones son independientes de la CPU.
- Para mejorar el rendimiento hay que considerar optimizaciones que tomen ventaja de la arquitectura de la CPU.

Intel Core i7 Haswell

- Tiene ocho unidades funcionales, numeradas del 0 al 7:
- 0. Aritmética de enteros, multiplicación de punto flotante, división de enteros y de punto flotante, brincos
- 1. Aritmética de enteros, suma de punto flotante, multiplicación de enteros, multiplicación de punto flotante
- 2. Carga, cálculo de direcciones
- 3. Carga, cálculo de direcciones
- 4. Store
- 5. Aritmética entera

Intel Core i7 Haswell

- 6. Aritmética entera, brincos
- 7. Cálculo de la dirección de store

Intel Core i7 Haswell

	Integer			Floating point		
Operation	Latency	Issue	Capacity	Latency	Issue	Capacity
Addition	1	1	4	3	1	1
Multiplication	3	1	1	5	1	2
Division	3–30	3–30	1	3–15	3–15	1

Figure 5.12 Latency, issue time, and capacity characteristics of reference machine operations. Latency indicates the total number of clock cycles required to perform the actual operations, while issue time indicates the minimum number of cycles between two independent operations. The capacity indicates how many of these operations can be issued simultaneously. The times for division depend on the data values.

Latencia y throughput

	Integer		Floating point	
Bound	+	*	+	*
Latency	1.00	3.00	3.00	5.00
Throughput	0.50	1.00	1.00	0.50

Versión 4 de combine

Function	Page	Method	Integer +	Integer *	Floating point +	Floating point *
`combine4`	551	Accumulate in temporary	1.27	3.01	3.01	5.01
Latency bound			1.00	3.00	3.00	5.00
Throughput bound			0.50	1.00	1.00	0.50

- Las medidas coinciden con el límite de latencia del procesador, excepto en el caso de la suma de números enteros.

Desenrollado de ciclos

- Objetivo: quitar los ciclos o reducir el número de iteraciones.
- Busca eliminar o reducir los peligros de control.

Desenrollado 2 x 1 (versión 5)

```
/* 2 x 1 loop unrolling */
void combine5(vec_ptr v, data_t *dest)
{
    long i;
    long length = vec_length(v);
    long limit = length – 1;
    data_t *data = get_vec_start(v);
    data_t acc = IDENT;
```

Desenrollado 2 x 1 (versión 5)

```
    /* Combine 2 elements at a time */
    for(i = 0; i < limit; i += 2) {
        acc = (acc OP data[i]) OP data[i + 1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        acc = acc OP data[i];
    }
    *dest = acc;
}
```

Explicación

- El ciclo recorre el arreglo dos elementos a la vez.
- Es decir, el índice del ciclo i se incrementa en 2 en cada iteración y la operación de combinación 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 operación al último elemento.

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 “desenrollamiento de ciclo k x 1”, porque el ciclo se desenrolla por un factor k pero se acumulan los valores en una sola variable acumulador.

Comparación

- Factores de desenrollamiento $k = 2$ vs $k = 3$:

			Integer		Floating point	
Function	Page	Method	+	*	+	*
`combine4`	551	No unrolling	1.27	3.01	3.01	5.01
`combine5`	568	2×1 unrolling	1.01	3.01	3.01	5.01
		3×1 unrolling	1.01	3.01	3.01	5.01
Latency bound			1.00	3.00	3.00	5.00
Throughput bound			0.50	1.00	1.00	0.50

- Se alcanzó el límite de la latencia.

Desenrollado de ciclos

- Muchos compiladores hacen desenrollado de ciclos como parte de su colección de optimizaciones.
- gcc puede realizar algunas formas de desenrollar ciclos cuando se invoca con el nivel de optimización 3 o superior (gcc –O3 …) .

Mejorar el paralelismo

- Los procesadores actuales son multicore y pueden correr código en paralelo.
- Las dependencias de datos verdaderas son bloqueadores de optimización.
- Este ciclo tiene dependencias entre iteraciones:

```
/* Combine 2 elements at a time */
for(i = 0; i < limit; i += 2) {
    acc = (acc OP data[i]) OP data[i + 1];
}
```

Mejorar el paralelismo

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

Desenrollamiento 2 x 2

- Hay dos acumuladores: acc0 y acc1.
- El acumulador acc0 acumular los valores con índices pares.
- El acumulador acc1 acumular los valores con índices impares.

Versión 6 de combine

```
/* 2 x 2 loop unrolling */
void combine6(vec_ptr v, data_t *dest)
{
    long i;
    long length = vec_length(v);
    long limit = length – 1 ;
    data_t *data = get_vec_start(v);
    data_t acc0 = IDENT;
    data_t acc1 = IDENT;
```

Versión 6 de combine

```
/* Combine 2 elements at a time */
for (i = 0; i < limit; i += 2) {
    acc0 = acc0 OP data[i];
    acc1 = acc1 OP data[i + 1];
}
```

Versión 6 de combine

```
    /* Finish any remaining elements */
    for (; i < length; i++) {
        acc0 = acc0 OP data[i];
    }
    *dest = acc0 OP acc1;
}
```

Comparación

Function	Page	Method	Integer +	Integer *	Floating point +	Floating point *
`combine4`	551	Accumulate in temporary	1.27	3.01	3.01	5.01
`combine5`	568	2 × 1 unrolling	1.01	3.01	3.01	5.01
`combine6`	573	2 × 2 unrolling	0.81	1.51	1.51	2.51
Latency bound			1.00	3.00	3.00	5.00
Throughput bound			0.50	1.00	1.00	0.50

- Se rompió la barrera impuesta por el límite de latencia.

Desenrollamiento de ciclo $k \times k$

- Se puede generalizar la transformación de múltiples acumuladores para desenrollar el ciclo por un factor de k y acumular k valores en paralelo, lo que produce un desenrollamiento de ciclo $k \times k$.

Desenrollamiento de ciclo $k \times k$

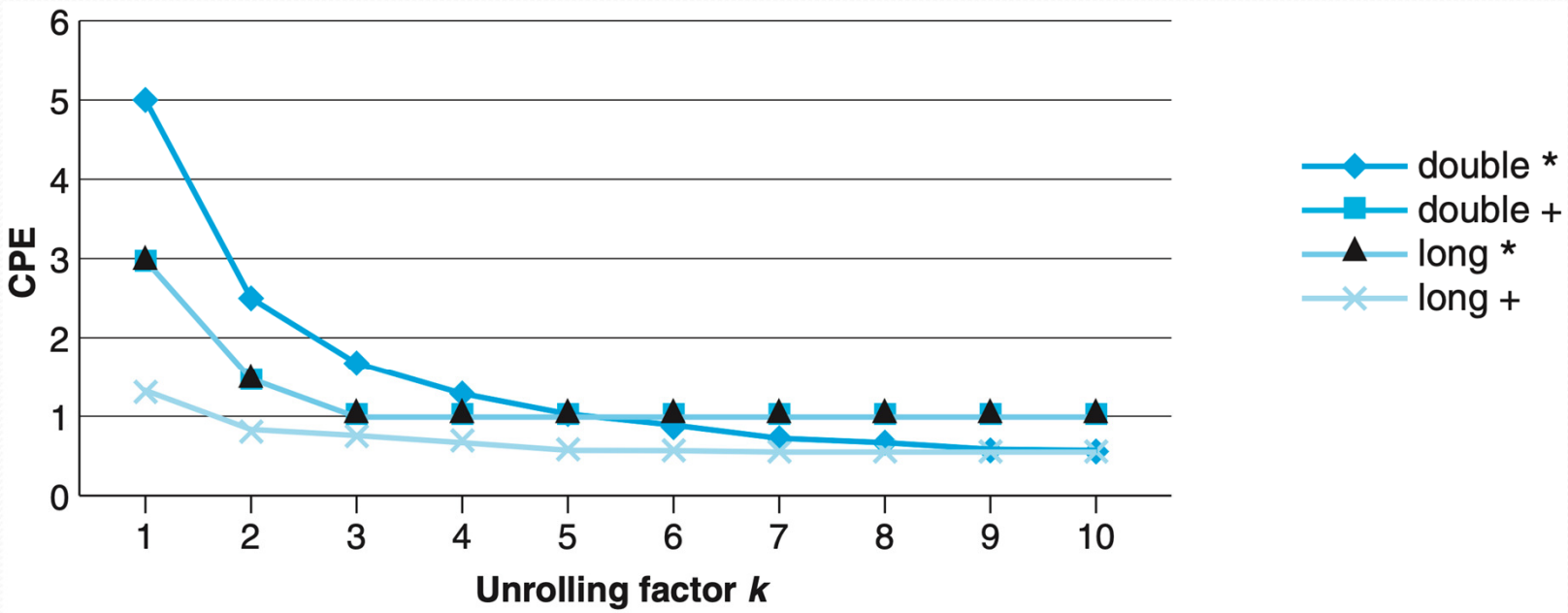

Figure 5.25 CPE performance of $k \times k$ loop unrolling. All of the CPEs improve with this transformation, achieving near or at their throughput bounds.

Consideraciones

- Al realizar la transformación de desenrollado $k \times k$, se debe considerar si se conserva la funcionalidad de la función original.
- La aritmética en complemento a dos es conmutativa y asociativa.
- La aritmética de punto flotante **no** es asociativa.
- Ejemplo: $(1 + 1e100) - 1e100 = 0$ y $1 + (1e100 - 1e100) = 1$.
- La mayoría de los compiladores **no** intentan transformaciones con código de punto flotante.

Consideraciones

- En general, desenrollar un ciclo y acumular múltiples valores en paralelo es la forma más confiable de mejorar el rendimiento de un programa.

Resumen

Function	Page	Method	Integer +	Integer *	Floating point +	Floating point *
`combine1`	543	Abstract `-O1`	10.12	10.12	10.17	11.14
`combine6`	573	2 × 2 unrolling	0.81	1.51	1.51	2.51
		10 × 10 unrolling	0.55	1.00	1.01	0.52
Latency bound			1.00	3.00	3.00	5.00
Throughput bound			0.50	1.00	1.00	0.50

Limitaciones

- Si el desenrollado de ciclos 10 x 10 es mejor que el 2 x 2, entonces ¿el desenrollado 20 x 20 será aun mejor?
- Respuesta: tal vez no.
- Motivo: derrame de registros (*register spilling*).
- 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.

Efecto del derrame de registros

			Integer		Floating point	
Function	Page	Method	+	*	+	*
`combine6`	573					
		10 × 10 unrolling	0.55	1.00	1.01	0.52
		20 × 20 unrolling	0.83	1.03	1.02	0.68
Throughput bound			0.50	1.00	1.00	0.50

Peligros de control

- Cuando se encuentra un brinco, el procesador debe especular en qué dirección continua la ejecución.
- 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 obtención de instrucciones en la ubicación correcta.

Peligros de control

- Al hacer esto, se incurre en un castigo por error de predicción, porque el pipeline de instrucciones debe llenarse antes de que se generen resultados útiles.
- Los procesadores modernos tienden a tener pipelines largos, por lo que el castigo por una predicción errónea es de entre 10 y 20 ciclos de reloj.
- ¿Qué puede hacer un programador para que el castigo no afecte la eficiencia de un programa?
- No hay una respuesta simple a esta pregunta, pero se aplican los siguientes principios generales:

Predicción de brincos

1. No preocuparse demasiado por las predicciones de brincos.
2. Escribir código buscando que el compilador genere instrucciones de movimientos condicionales (lenguaje ensamblador x86).

No preocuparse demasiado

- 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) y realizan una operación de movimiento si las banderas están en un estado (o condición) específico.
- 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
```

Ejemplo – forma tradicional

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

Ejemplo – movimiento condicional

```
// 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;
}
```

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

- 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

- La versión imperativa (minmax1) tiene un CPE de 13.5 para números aleatorios y de 2.5 a 3.5 para datos predecibles.
- La versión funcional (minmax2) tiene un CPE de 4 para números aleatorios y para datos predecibles.

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

Resumen

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