

BP4-resumen.pdf



patrivc



Arquitectura de Computadores



2º Grado en Ingeniería Informática



Escuela Técnica Superior de Ingenierías Informática y de
Telecomunicación
Universidad de Granada



Descarga la APP de Wuolah.
Ya disponible para el móvil y la tablet.





Disney Cruella

EN
CINES

O DISFRÚTALA
EN

Disney+
A TRAVÉS DE
ACCESO PREMIUM
Con coste adicional

12

Seminario 4. Optimización de Código en Arquitecturas ILP

Cuestiones generales sobre optimización

Cuestiones generales sobre optimización de Código I

Usualmente la optimización de una aplicación se realiza al final del proceso, si queda tiempo. Esperar al final para optimizar dificulta el proceso de optimización.

Es un error programar la aplicación sin tener en cuenta la arquitectura o arquitecturas en las que se va a ejecutar.

En el caso de que no se satisfagan las restricciones de tiempo, no es correcto optimizar eliminando propiedades y funciones (features) del código.

Cuando se optimiza código es importante analizar donde se encuentran los cuellos de botella. El cuello de botella más estrecho es el que al final determina las prestaciones y es el que debe evitarse en primer lugar.

Se puede optimizar sin tener que acceder al nivel del lenguaje ensamblador (aunque hay situaciones en las que es necesario bajar a nivel de ensamblador).

Cuestiones generales sobre optimización de Código II: Clasificación de optimizaciones

Optimizaciones desde el Lenguaje de Alto Nivel (OHLL)		Optimizaciones desde el Lenguaje Ensamblador (OASM)	
Optimizaciones aplicables a cualquier procesador (OGP)	Optimizaciones específicas para un procesador (OEP)	Optimizaciones aplicables a cualquier procesador (OGP)	Optimizaciones específicas para un procesador (OEP)

Un Compilador puede ejecutarse utilizando diversas opciones de optimización. Por ejemplo, gcc/g++ dispone de las opciones de compilación -O1, -O2, -O3, -Os que proporcionan códigos con distintas opciones de optimización.

Optimización de la Ejecución Unidades de ejecución o funcionales

Es importante tener en cuenta las unidades funcionales de que dispone la microarquitectura para utilizar las instrucciones de la forma más eficaz.

La división es una operación costosa y por lo tanto habría que evitarla (con desplazamientos, multiplicaciones, ...) o reducir su número.

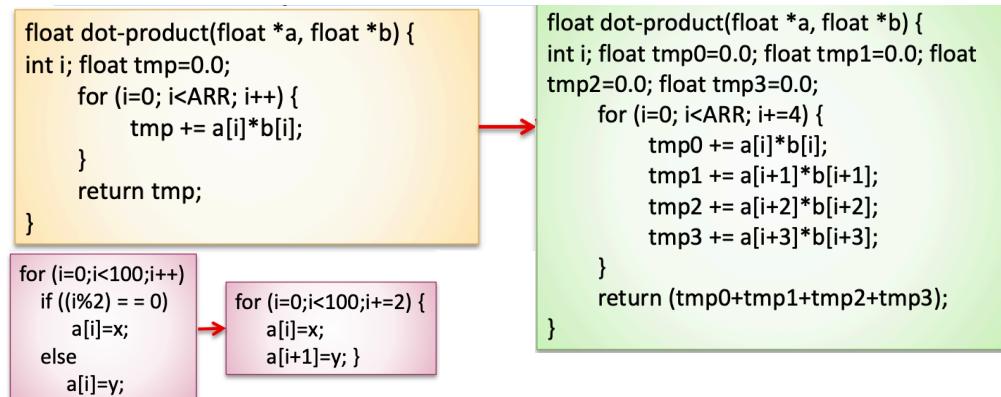
```
for (i=0;i<100;i++) a[i]=a[i]/y;  
temp=1/y;  
for (i=0;i<100;i++) a[i]=a[i]*temp;
```

A veces es más rápido utilizar desplazamientos y sumas para realizar una multiplicación por una constante entera que utilizar la instrucción IMUL.

```
imul eax,10  
lea ecx,[eax+eax]  
lea eax,[ecx+eax*8]
```

Desenrollado de bucles

Utilizar el desenrollado de bucles para romper secuencias de instrucciones dependientes intercalando otras instrucciones.



Ventajas:

- Reduce el número de saltos
- Aumenta la oportunidad de encontrar instrucciones independientes
- Facilita la posibilidad de insertar instrucciones para ocultar las latencias.
- La contrapartida es que aumenta el tamaño de los códigos.

Desenrollar el bucle genera un impacto en la optimización bastante grande, ya que en menos iteraciones hacemos más operaciones, consiguiendo así una reducción de saltos y tener un repertorio de instrucciones independientes más amplio. La reducción del tiempo es considerable

Código ambiguo I

Si el compilador no puede resolver los punteros (código ambiguo) se inhiben ciertas optimizaciones del compilador:

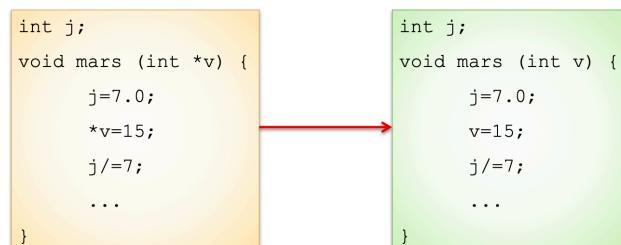
- Asignar variables durante la compilación
- Realizar cargas de memoria mientras que un almacenamiento está en marcha

Si no se utilizan punteros el código es más dependiente de la máquina, y a veces las ventajas de no utilizarlos no compensa.

Cómo evitar código ambiguo o sus efectos:

- Utilizar variables locales en lugar de punteros
- Utilizar variables globales si no se pueden utilizar las locales
- Poner las instrucciones de almacenamiento después o bastante antes de las de carga de memoria.

Código ambiguo II



En el código no optimizado, el compilador no puede asumir que *v no apunta a j.

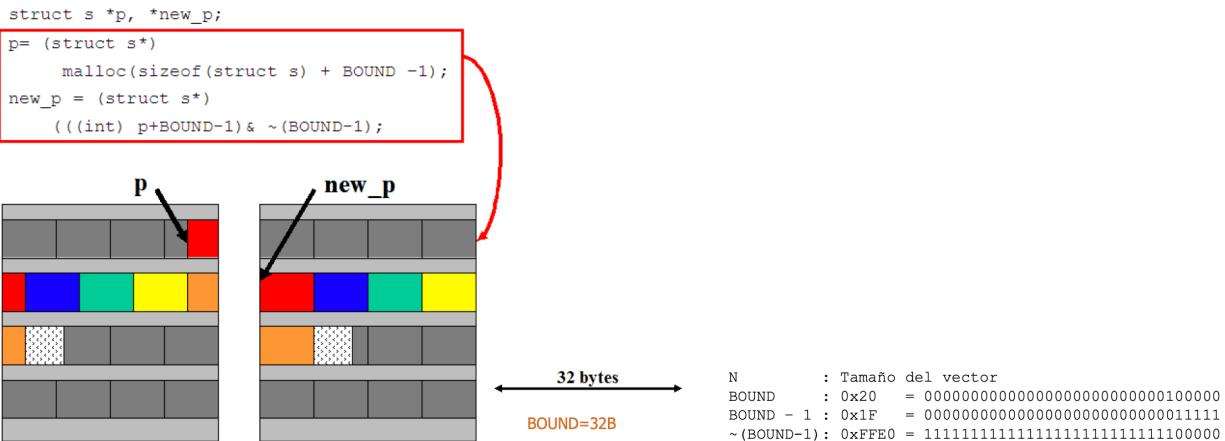
Sin embargo en el código optimizado, el compilador puede hacer directamente j=1.0.

Optimización del Acceso a Memoria

Alineamiento de datos

El acceso a un dato que ocupa dos líneas de cache aumenta tiempo de acceso. Línea de cache

- 32B: Pentium Pro, Pentium II, Pentium III
- 64B: Pentium 4, Core ...



Colisiones en Cache

Típicamente las caches tienen una organización asociativa por conjuntos:

- L1 Intel Core 2 Duo, Intel Core Duo, Intel Core Solo:
 - Líneas cache 64B, asociativa por conjuntos 8 vías, tamaño 32 KB
 - Penalización si se accede a más de 8 direcciones con separación de 4KB (= 32KB/8 vías)
- L2 Intel Core 2 Duo, Intel Core Duo, Intel Core Solo:
 - Penalización si se accede a más de X direcciones con separación de 256KB (X: no de vías de la cache L2)

```
int *tempA, *tempB;
...
pA= (int *) malloc (sizeof(int)*N + 63);
tempA = (int *)(((int)pA+63)&~(63));
tempB = (int *)(((int)pA+63)&~(63))+4096+64;
```

Los punteros tempA y tempB están apuntando a zonas de memoria que empiezan en posiciones que son múltiplos de 64 y que no se asignarán al mismo conjunto de cache L1.

Localidad de los accesos I

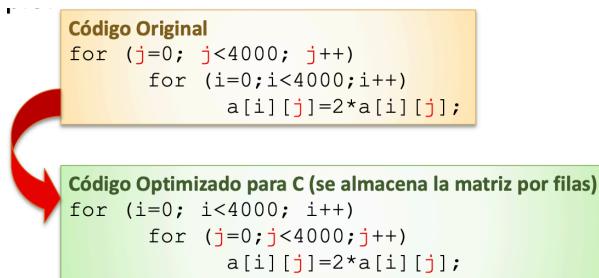
La forma en que se declaran los arrays determina la forma en que se almacenan en memoria. Interesa declararlos según la forma en la que se vaya a realizar el acceso. Ejemplos: Formas óptimas de declaración de variables según el tipo de acceso a los datos.

```
struct {
    int a[500];
    int b[500];
} s;
...
for (i=0; i<500; i++)
    s.a[i]=2*s.a[i];
...
for (i=0; i<500; i++)
    s.b[i]=3*s.b[i];
```

```
struct {
    int a;
    int b;
} s[500];
...
for (i=0; i<500; i++)
{
    s[i].a+=5;
    s[i].b+=3;
}
```

Localidad de los accesos II

Intercambiar los bucles para cambiar la forma de acceder a los datos según los almacena el compilador, y para aprovechar la localidad. Ejemplo:



Acceso a memoria especulativo

Los 'atascos' (stalls) por acceso a la memoria (load adelanta a store, especulativo) se producen cuando:

- Hay una carga (load) 'larga' que sigue a un almacenamiento (store) 'pequeño' alineados en la misma dirección o en rangos de direcciones solapadas.
 - mov word ptr [ebp],0x10
 - mov ecx, dword ptr [ebp]
- Una carga (load) 'pequeña' sigue a un almacenamiento (store) 'largo' en direcciones diferentes aunque solapadas (si están alineadas en la misma dirección no hay problema).
 - mov dword ptr [ebp-1], eax
 - mov ecx,word ptr [ebp]
- Datos del mismo tamaño se almacenan y luego se cargan desde direcciones solapadas que no están alineadas.
 - mov dword ptr [ebp-1], eax
 - mov eax, dword ptr [ebp]

Para evitarlos: Utilizar datos del mismo tamaño y direcciones alineadas y poner los loads tan lejos como sea posible de los stores a la misma área de memoria.

Precaptación (Prefetch) I

El procesador, mediante las correspondientes instrucciones de prefetch, carga zonas de memoria en cache antes de que se soliciten (cuando hay ancho de banda disponible). Hay cuatro tipos de instrucciones de prefetch:

Instrucción ensamb.	Segundo parámetro en <code>_mm_prefetch ()</code> (Intel C++ Compiler)	Descripción
<code>prefetchnta</code>	<code>_MM_HINT_NTA</code>	Prefetch en buffer no temporal (dato para una lectura)
<code>prefetcht0</code>	<code>_MM_HINT_T0</code>	Prefetch en todas las caches útiles
<code>prefetcht1</code>	<code>_MM_HINT_T1</code>	Prefetch en L2 y L3 pero no en L1
<code>prefetcht2</code>	<code>_MM_HINT_T2</code>	Prefetch sólo en L3

En gcc se puede usar: `void __builtin_prefetch (const void *addr, ...)`



Cruella

EN
CINES

O DISFRÚTALA
EN

Disney+
A TRAVÉS DE
ACCESO PREMIUM
Con coste adicional

12

Precaptación (Prefetch) II

Una instrucción de prefetch carga una línea entera de cache.

El aspecto crucial al realizar precaptación es la anticipación con la que se pre-captan los datos. En muchos casos es necesario aplicar una estrategia de prueba y error.

Además, la anticipación óptima puede cambiar según las características del computador (menos portabilidad en el código).

Ejemplo:

```
for (i=0; i<1000; i++) {  
    x=function(matriz[i]);  
    _mm_prefetch(matriz[i+16],_MM_HINT_T0);  
}
```

En el ejemplo se precepta el dato necesario para la iteración situada a 16 iteraciones (en el futuro). En un prefetch no se generan faltas de memoria (es seguro precaptar más allá de los límites del array).

Optimización de Saltos

Saltos I

```
if (t1==0 && t2==0 && t3==0)
```

```
if ((t1 | t2 | t3)==0)
```

```
//if ((t1 | t2 | t3)==0) {t4=1;}  
mov  ecx, 1 //t1 -> edi  
or   edi, ebx //t2 -> ebx  
or   edi, ebp //t3 -> ebp  
cmovc eax, ecx //t4 -> eax
```

• Cada una de las condiciones separadas por && se evalúa mediante una instrucción de salto distinta.

Si las variables pueden ser 1 ó 0 con la misma probabilidad, la posibilidad de predecir esas instrucciones de salto no es muy elevada.

• Si se utiliza un único salto, la probabilidad de 1 es de 0.125 y la de 0 de 0.875 y la posibilidad de hacer una buena predicción aumenta.

• Mediante la instrucción de movimiento condicional se pueden evitar los saltos

Saltos II

Se puede reducir el número de saltos de un programa reorganizando las alternativas en las sentencias switch, en el caso de que alguna opción se ejecute mucho más que las otras (más del 50% de las veces, por ejemplo).

Ciertos compiladores que utilizan información de perfiles de ejecución del programa son capaces de realizar esta reorganización (Se recomienda utilizarla si la sentencia switch se implementa como una búsqueda binaria en lugar de una tabla de salto).

Código original

```
switch (i) {  
    case 16:  
        Bloque16  
        break;  
    case 22:  
        Bloque22  
        break;  
    case 33:  
        Bloque33  
        break;  
}
```

Código Optimizado

```
if (i==33)  
    { Bloque33 }  
else  
switch (i) {  
    case 16:  
        Bloque16  
        break;  
    case 22:  
        Bloque22  
        break;  
}
```

Saltos III

CMOVcc hace la transferencia de información si se cumple la condición indicada en cc.

Las instrucciones CMOVcc verifican el estado de uno o más de los indicadores de estado en el registro EFLAGS (CF, OF, PF, SF y ZF) y realizan una operación de movimiento si los indicadores están en un estado (o condición) específico. Se asocia un código de condición (cc) con cada instrucción para indicar la condición que se está probando. Si no se cumple la condición, no se realiza un movimiento y la ejecución continúa con la instrucción que sigue a la instrucción CMOVcc.

```
test ecx,ecx
jne 1h
mov eax,ebx
1h:
```

```
test ecx,ecx
cmovneq eax, ebx
```

FCMOVcc es similar a CMOVcc pero utiliza operandos en coma flotante.

Prueba los flags de estado en el registro EFLAGS y mueve el operando de origen (segundo operando) al operando de destino (primer operando) si la condición de prueba dada es verdadera. Las instrucciones FCMOVcc son útiles para optimizar pequeñas construcciones IF. También ayudan a eliminar la sobrecarga de ramificación para operaciones IF y la posibilidad de predicciones erróneas de ramificaciones por parte del procesador.

Saltos IV

La instrucción SETcc es otro ejemplo de instrucción con predicado que puede permitir reducir el número de instrucciones de salto.

Establece el operando de destino en 0 o 1 según la configuración de los indicadores de estado (CF, SF, OF, ZF y PF) en el registro EFLAGS. El operando de destino apunta a un registro de bytes o un byte en la memoria. El sufijo del código de condición (cc) indica la condición que se está probando.

Código Original	Explicación:
cmp A,B jge L30 mov ebx,C1 jmp L31 L30: mov ebx,C2 L31:	ebx= (A<B) ? C1 : C2; [Si (A<B) es cierto EBX se carga con C1 y si no con C2]

Código Optimizado	Explicación:
xor ebx,ebx cmp A,B setge bl dec ebx and ebx, (C1-C2) add ebx, C2	//Si A>=B, setge hace BL=1; DEC hace EBX=0; (EBX and C1-C2)=0; //EBX + C2 = C2 //Si A<B, setge deja BL=0; DEC hace EBX=0xFFFFFFFF; //(EBX and C1-C2)= C1-C2; //EBX + C2 = C1

Realización de los Ejercicios Prácticos

Ejemplo de programa de prueba en C

```

/* Ejemplo de Programa de Prueba */

#include <stdio.h>
#include <math.h>
#include <time.h>

int suma_prod(int a, int b, int n);

main()
{
    /* -----
    int i,j,a,b,n,c;
    /* ----- */

    clock_t start,stop;
    start= clock();
    /* ----- */

    n=6000;a=1;b=2;
    for (j=1;j<=10000;j++)
    {
        printf("a=%d b=%d n=%d\n",a,b,n);
        c=suma_prod(a,b,n);
        printf("resultado= %d\n",c);
    }
    /* ----- */

    stop = clock();
    printf("Tiempo= %f",difftime(stop,start));
    return 0;
}

```

Función a optimizar suma_prod()

```

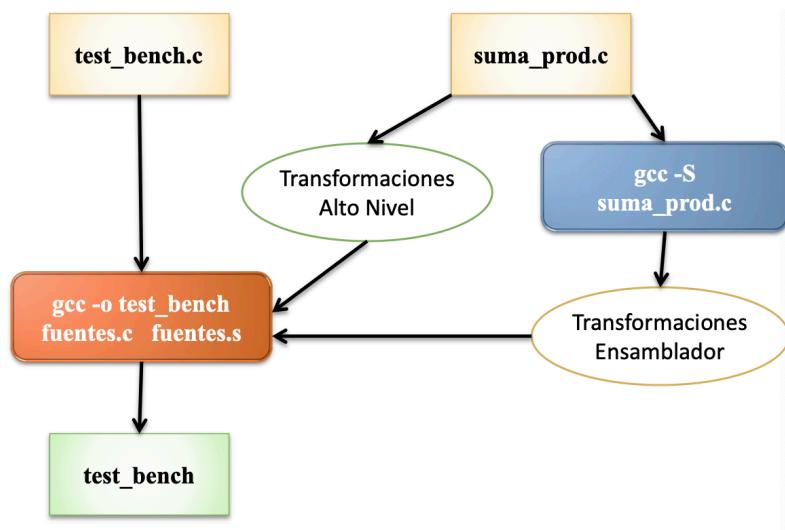
/* Ejemplo de Funcion */
int suma_prod(int a, int b, int n)
{
    return a*b+n;
}

```

test_bench.c

Figura A2.1 Código C correspondiente a un programa de base (test_bench.c)

Esquema de trabajo en ejercicios



Apuntes extra:

Diferencias en ensamblador:

De los cuatro códigos generados en ensamblador, el que más instrucciones utiliza es la optimización O3, ya que es el nivel de optimización más alto posible. También podemos observar que el código de O2 y O3 son muy parecidos. Esto se debe a que O3 activa todas las opciones de optimización de O2.

Una breve descripción de cada uno puede ser la siguiente:

-O0: Este nivel (que consiste en la letra "O" seguida de un cero) desconecta por completo la optimización y es el predeterminado si no se especifica ningún nivel -O en CFLAGS o CXXFLAGS. El código no se optimizará. Esto, normalmente, no es lo que se desea.

-Os: Optimizará el tamaño del código. Activa todas las opciones de -O2 que no incrementan el tamaño del código generado. Es útil para máquinas con capacidad limitada de disco o con CPUs que tienen poca caché.

-O2: Es el nivel *recomendado* de optimización, a no ser que el sistema tenga necesidades especiales. -O2 activará algunas opciones añadidas a las que se activan con -O1. Con -O2, el compilador intentará aumentar el rendimiento del código sin comprometer el tamaño y sin tomar mucho más tiempo de compilación.

-O3: El nivel más alto de optimización posible. Activa optimizaciones que son caras en términos de tiempo de compilación y uso de memoria. El hecho de compilar con -O3 no garantiza una forma de mejorar el rendimiento y, de hecho, en muchos casos puede ralentizar un sistema debido al uso de binarios de gran tamaño y mucho uso de la memoria. También se sabe que -O3 puede romper algunos paquetes. No se recomienda utilizar -O3.

Preguntas:

1. Cuál de las optimizaciones siguientes reduce el tiempo de ejecución del código que se muestra a continuación? (M y N son múltiplos de dos)

B

```
for (i=0; i<M; i++)  
    for (j=0; j<N; j++){  
        if ((j%2)==0)  
            c[j] += a[j][i]+b[j][i];  
        else  
            c[j] += a[j][i]-b[j][i];  
    }  
     a) for (j=0; j<M; j++)  
        for (i=0; i<N; i++){  
            if ((i%2)==0)  
                c[i] += a[i][i]+b[i][j];  
            else  
                c[i] += a[i][j]-b[i][j];  
        }  
     b) for (i=0; i<M; i++)  
        for (j=0; j<N; j+=2){  
            c[j] += a[j][i]+b[j][i];  
            c[j+1] += a[j+1][i]-b[j+1][i];  
        }  
     c) for (i=0; i<M; i++)  
        for (j=0; j<N; j++){  
            c[j] += a[j][i]+b[j][i];  
            c[j+1] += a[j+1][i]-b[j+1][i];  
        }  
     d) for (i=0; i<M; i+=2)  
        for (j=0; j<N; j+=2){  
            c[j] += a[j][i]+b[j][i];  
            c[j+1] += a[j+1][i+1]-b[j+1][i+1];  
        }
```



Disney Cruella

EN
CINES

O DISFRÚTALA
EN

Disney+
A TRAVÉS DE
ACCESO PREMIUM
Con coste adicional

12

2. ¿Cómo cree que implementará el compilador una función que multiplica un entero por 11 al compilar con optimización máxima (-O3)?

C

```
int function_f(int x)
{
    return x * 11;
}
 a) 0x401116 <+0>: imul    $0xb,%edi,%eax
 b) 0x401120 <+0>: lea     (%rdi,%rdi,4),%eax
 c) 0x401106 <+0>: push    %rbp
    0x401107 <+1>: mov     %rsp,%rbp
    0x40110a <+4>: mov     %edi,-0x4(%rbp)
    0x40110d <+7>: mov     -0x4(%rbp),%edx
    0x401110 <+10>: mov     %edx,%eax
    0x401112 <+12>: shl     $0x2,%eax
    0x401115 <+15>: add     %edx,%eax
    0x401117 <+17>: add     %eax,%eax
    0x401119 <+19>: add     %edx,%eax
    0x40111b <+21>: pop    %rbp
    0x40111c <+22>: retq
 d) ninguna otra respuesta es correcta
```

3. ¿Qué código cree que calculará de forma correcta y en menor tiempo el producto de dos matrices en un sistema multiprocesador? Suponga matrices cuadradas, c inicializada a cero y N muy grande.

B

```
int a[N][N], b[N][N], c[N][N];
 a) for (int i=0; i<N; ++i)
    #pragma omp parallel for
    for (int j=0; j<N; ++j)
        for (int k=0; k<N; ++k)
            c[i][j] += a[i][k] * b[k][j];
 b) for (int i=0; i<N; ++i)
    #pragma omp parallel for
    for (int j=0; j<N; ++j)
        for (int k=0; k<N; ++k)
            #pragma omp atomic
            c[i][j] += a[i][k] * b[k][j];
 c) for (int i=0; i<N; ++i)
    #pragma omp parallel for
    for (int j=0; j<N; ++j)
        for (int k=0; k<N; ++k)
            #pragma omp critical
            c[i][j] += a[i][k] * b[k][j];
 d) for (int i=0; i<N; ++i)
    for (int j=0; j<N; ++j)
        for (int k=0; k<N; ++k)
            c[i][j] += a[i][k] * b[k][j];
```

4. Sin indicarle un núcleo concreto, ¿cómo ordenaría las instrucciones con enteros en orden creciente de tiempo de ejecución?

- a) **Desplazamiento de bits, división y multiplicación**
 b) Desplazamientos de bits, multiplicación y división
 c) División, desplazamiento de bits y multiplicación
 d) Multiplicación, división y desplazamiento de bits

D

```
0x4005d0 <+0>: cmp     %esi, %edi
0x4005d2 <+2>: mov     %esi, %eax
0x4005d4 <+4>: cmovle %edi, %eax
0x4005d7 <+7>: retq
```

- a) int f(int a, int b, int c, int d) {

 if (a < b)

 return c;

 else

 return d;

}
- b) int f(int a, int b) {

 if (a < b)

 return a;

 else

 return b;

}
- c) ninguna otra respuesta es correcta
- d) int f(int a, int b) {

 if (a > b)

 return a;

 else

 return b;

}

6. ¿Cuál de las siguientes afirmaciones es correcta?

- a) hay optimizaciones que son aplicables a cualquier procesador**
- b) ninguna otra respuesta es correcta
- c) el proceso de optimización se debe realizar siempre al final del desarrollo de la aplicación
- d) la optimización de código siempre debe realizarse en lenguaje ensamblador

7. Dado el siguiente código y suponiendo el vector v inicializado, ¿qué opción es verdadera?

```
for (int i = 0; i < 1000; ++i)
{
    if ((v[i] % 3) == 0)
        foo(v[i]);
    else
        switch((v[i] % 3))
        {
            case 1: foo(v[i] + 2); break;
            case 2: foo(v[i] + 1); break;
        }
}
```

- a) la ejecución finaliza antes si v contiene muchos múltiplos de 3**
- b) la ejecución finaliza antes si v no contiene ningún múltiplo de 3
- c) sólo el desenrollado de bucle puede servir para optimizar el código
- d) los valores contenidos en v no afectan a la velocidad de ejecución

8. ¿Cómo cree que se calcularía más rápido la operación “a=b*c” suponiendo que el valor de c es 5?

- A) a = b+b+b+b+b;
- B) A = b + (b<<2);**
- C) a = b * c;
- D) A = c*b;