2º curso / 2º cuatr.

Grado en
Ing. Informática

Arquitectura de Computadores

Seminario 4. Optimización de Código en Arquitecturas ILP

Material elaborado por los profesores responsables de la asignatura:
Julio Ortega – Mancia Anguita

ugr | Universidad de Granada

ETSIIT
Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

ATC
Departamento de Arquitectura y Tecnología de Computadores
UNIVERSIDAD DE GRANADA

Bibliografía

- Manuales de Intel para procesadores x86 (IA-32), especialmente el manual de optimización de código (busque en la web de referencia *Software Optimization Reference Manual*) http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html
- [Gerber 2006] R. Gerber, et al. ”The Software Optimization Cookbook. High Performance Recipes for the IA-32 Platforms”. Intel Press, 2006.
- [Fog 2004] A. Fog. ”How to Optimize for the Pentium family of microprocessors”, http://cr.yp.to/2005-590/fog.pdf, 2004
- [Gerber 2002] R. Gerber. ”The Software Optimization Cookbook. High Performance Recipes for the Intel Architecture”. Intel Press, 2002.

Contenido

- Cuestiones generales sobre optimización
- Optimización de la Ejecución
- Optimización del Acceso a Memoria
- Optimización de Saltos
- Realización de los Ejercicios Prácticos

Contenido

- Cuestiones generales sobre optimización
- Optimización de la Ejecución
- Optimización del Acceso a Memoria
- Optimización de Saltos
- Realización de los Ejercicios Prácticos

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 escribir 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 casos 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 -O1, -O2, -O3, -Os que proporcionan códigos con distintas opciones de optimización.

Contenido

- Cuestiones generales sobre optimización
- Optimización de la Ejecución
- Optimización del Acceso a Memoria
- Optimización de Saltos
- Realización de los Ejercicios Prácticos

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.

```
float dot-product(float *a, float *b) {
int i; float tmp=0.0;
        for (i=0; i<ARR; i++) {
                tmp += a[i]*b[i];
        }
        return tmp;
}
```

```
float dot-product(float *a, float *b) {
int i; float tmp0=0.0; float tmp1=0.0; float
tmp2=0.0; float tmp3=0.0;
        for (i=0; i<ARR; i+=4) {
                tmp0 += a[i]*b[i];
                tmp1 += a[i+1]*b[i+1];
                tmp2 += a[i+2]*b[i+2];
                tmp3 += a[i+3]*b[i+3];
        }
        ret tmp0+tmp1+tmp2+tmp3;
}
```

- 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.

```
for (i=0;i<100;i++)
   if ((i%2) = = 0)
      a[i]=x;
   else
      a[i]=y;
```

```
for (i=0;i<100;i+=2) {
      a[i]=x;
      a[i+1]=y; }
```

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


```
int j;
void mars (int *v) {
        j=7.0;
        *v=15;
        j/=7;
        ...
}
```

```
int j;
void mars (int v) {
        j=7.0;
        v=15;
        j/=7;
        ...
}
```

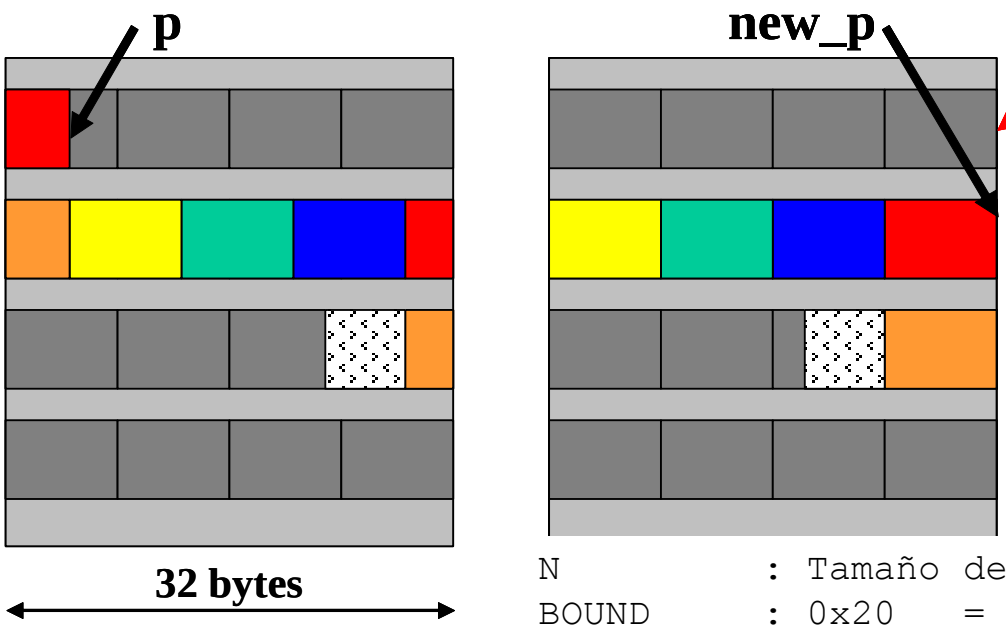
- En el código no optimizado, el compilador no puede asumir que *v no apunta a j.
- Sin embargo en el código optimizado, el compilador puede hacer directamente j=1.0.

Contenido

- Cuestiones generales sobre optimización
- Optimización de la Ejecución
- Optimización del Acceso a Memoria
- Optimización de Saltos
- Realización de los Ejercicios Prácticos

Alineamiento de datos I


```
struct s *p, *new_p;
p= (struct s*)
     malloc(sizeof(struct s) + BOUND -1);
new_p = (struct s*)
    (((int) p+BOUND-1)& ~(BOUND-1);
```



BOUND=32B

- 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 …

```
N           : Tamaño del vector
BOUND       : 0x20    = 00000000000000000000000000100000
BOUND - 1   : 0x1F    = 00000000000000000000000000011111
~(BOUND-1)  : 0xFFE0  = 11111111111111111111111111100000
```

Alineamiento de datos II

- Cuando el código accede a un array de forma no secuencial es conveniente ajustar y alinear la estructura de forma que ocupe el mínimo número de líneas de cache **completas**. Ejemplo:

Código original

```
struct  s  {
            long int val[7];
};
struct   s   vs;
...
for (i=0;i<7;i++)
   vs.val[i]+=vs.val[i];
```

Código optimizado

```
struct  s  {
            long int val[7];      // 7*64b = 7*8B
            long int pad;         // 64 b = 8B
} ;
struct  s *p, *new_p;
...
p=(struct s*) malloc (sizeof (struct s));
new_p= (struct s *) (((long int)p+63)&(-64) );
```

- Se añaden variables ficticias para llenar 64B (si las líneas de cache son de 64B) y se alinean las estructuras para que empiecen en una línea de cache.

Alineamiento de datos III

- Añade una variable de 64 bits ficticia (`pad`)
- Alinea `new_p` a un múltiplo de 64 B

```
Archivo  Editar  Ver  Terminal  Ayuda
$ gcc -O2 -o alinearp alinearp.c
$ alinearp
Tras el inicio de new_p  (p=0x8b6010    new_p=0x8b6040)
$ alinearp
Tras el inicio de new_p  (p=0x1d49010   new_p=0x1d49040)
$ alinearp
Tras el inicio de new_p  (p=0x16ac010   new_p=0x16ac040)
$ alinearp
Tras el inicio de new_p  (p=0xe82010    new_p=0xe82040)
$ alinearp
Tras el inicio de new_p  (p=0xd56010    new_p=0xd56040)
$ alinearp
Tras el inicio de new_p  (p=0xaff010    new_p=0xaff040)
$ alinearp
Tras el inicio de new_p  (p=0x190e010   new_p=0x190e040)
$ alinearp
Tras el inicio de new_p  (p=0x1a35010   new_p=0x1a35040)
$ alinearp
Tras el inicio de new_p  (p=0x2443010   new_p=0x2443040)
$
```

alinearp.c

```
#include <stdio.h>
#include <stdlib.h>

struct  s  {
            long int val[7];
            long int pad;
} ;
struct  s *p, *new_p;

main ()
{
p=(struct s*) malloc (sizeof (struct s));

new_p = (struct s*) (((long int)p+63) &  (-64)) ;

printf("Tras el inicio de new_p
(p=%p\tnew_p=%p) \n", p, new_p);

}
```

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)

```
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 asignarían al mismo conjunto de cache L1

  - 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)

Localidad de los accesos I

- La forma en que se declaren los arrays determina la forma en que se almacenan en memoria. Interesa declararlos según 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:

Código Original

```
for (j=0; j<4; j++)
      for (i=0;i<4;i++)
            a[i][j]=2*a[i][j];
```

Código Optimizado para C (se almacena la matriz por filas)

```
for (i=0; i<4; i++)
      for (j=0;j<4;j++)
            a[i][j]=2*a[i][j];
```

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 `_mm_prefetch()` (Intel C++ Compiler)**	**Descripción**
`prefetchnta`	_MM_HINT_NTA	Prefetch en buffer no temporal (dato para una lectura)
`prefetcht0`	_MM_HINT_T0	Prefetch en todas las caches útiles
`prefetcht1`	_MM_HINT_T1	Prefetch en L2 y L3 pero no en L1
`prefetcht2`	_MM_HINT_T2	Prefetch sólo en L3

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

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 precapta 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*)

Contenido

- Cuestiones generales sobre optimización
- Optimización de la Ejecución
- Optimización del Acceso a Memoria
- Optimización de Saltos
- Realización de los Ejercicios Prácticos

Saltos I


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

- 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.

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

- 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

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

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

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

```
test ecx,ecx
cmoveq eax, ebx
```

- `FCMOVcc` es similar a `CMOVcc` pero utiliza operandos en coma flotante

Saltos IV

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

Código Original

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

Contenido

- Cuestiones generales sobre optimización
- Optimización de la Ejecución
- Optimización del Acceso a Memoria
- Optimización de Saltos
- 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