6620. Organización de Computadoras Trabajo Práctico 2: Optimización de Software

González, Juan Manuel, *Padrón Nro. 79.979*
juan0511@yahoo.com

Smocovich, Elizabeth, *Padrón Nro. 86.326*
liz.smocovich@gmail.com

Pereira, María Florencia, *Padrón Nro. 88.816*
mflorenciapereira@gmail.com

1er. Cuatrimestre de 2010
66.20 Organización de Computadoras – Prática Jueves
Facultad de Ingeniería, Universidad de Buenos Aires

Resumen

El presente trabajo tiene como objetivo aplicar optimizaciones de software en ambientes reales y familiarizarse con la medición de comportamiento utilizando herramientas de *profiling*.

1. Introducción

Un algoritmo de la multiplicación de matrices en lenguaje C sencillo, pero ineficiente en cuanto a la utilización de la memoria cache es la siguiente:

```
for(i=0;i<N;i++)
    for(j=0;j<M;j++)
        for(k=0;k<L;k++)
            c[i][j]+=a[i][k]*[k][j];
```

Este programa multiplica las matrices $A \in \Re R^{NxL}$ y $B \in \Re^{LxM}$, dando como resultado una matriz $C \in \Re^{NxM}$. Al recorrer una de las matrices por filas y la otra por columnas, para matrices mayores que la memoria cache se producirán ineficiencias a la hora de operar con ella.
En el presente trabajo se hara uso de las técnicas de optimización de uso de la memoria cache vistas en el curso, modificando un programa que resuelve una multiplicación de dos matrices. Se buscará reducir la cantidad de desaciertos (*misses*) en la memoria caché de datos de nivel 1 (L1) y el tiempo de ejecución . Para esto se utilizará el módulo de simulación de memorias caché, llamado Cachegrind, e integrado a Valgrind y la herramienta de profiling gprof. Las optimizaciones a aplicar serán *padding*, *blocking* y *prefetch*.

2. Consideraciones sobre el desarrollo e implementación

A continuación se detallan las consideraciones tenidas en cuenta para el desarrollo del trabajo práctico.

En esta sección se discutirán los métodos de optimización que fueron implementados en la versión final del trabajo práctico. También se expondrán los resultados obtenidos de las mejoras introducidas en el código, basados en las pruebas realizadas según se expone en la sección 4.

2.1. Elección del nivel de memoria a optimizar

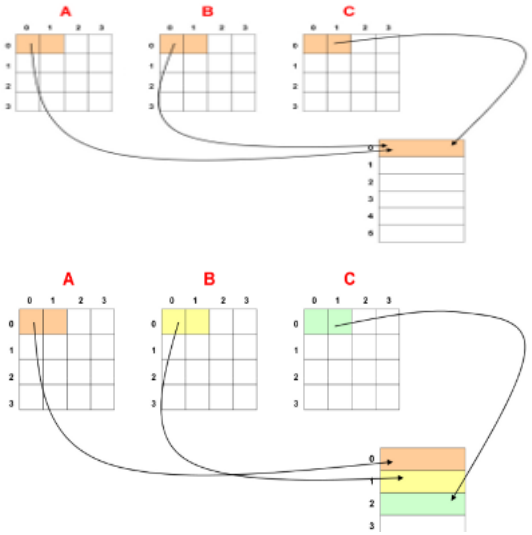
Para reducir la brecha de velocidad entre el procesador y la memoria física, la arquitectura de las computadoras suele incluir varios niveles de memoria. Las memorias caches multi-nivel operan revisando la memoria L1 primero. Si se produce un miss, a continuación se revisa la cache L2 y así sucesivamente con los niveles siguientes hasta la memoria física. Las memorias cache grandes tienen un porcentaje alto de hit rate pero una latencia mayor. El primer nivel de memoria es pequeño para que los ciclos que se consuman sean comparables con los ciclos de reloj del CPU. La memoria cache de nivel 2 es de mayor tamaño para capturar gran parte de los accesos a memoria, disminuyendo así el *miss penalty*. Dado el tamaño de la memoria L1, es probable que su miss rate sea mayor que el de la memoria L2. Por esta razón, se decidió optimizar el código para la memoria cache L1, ya que si se redujera el miss rate de este nivel, el tiempo de ejecución sería menor. Para la memoria cache L2 esto no sería posible ya que el miss rate es bastante menor que el de la L1 y la latencia no es una característica que pueda optimizarse por software.

2.2. Optimización del algoritmo y resultados

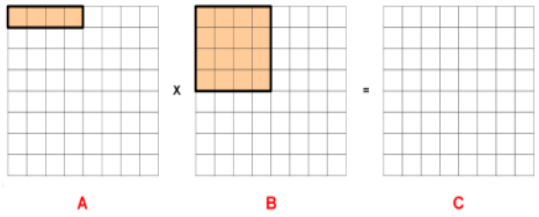
A fin de mejorar el rendimiento, se ensayaron tres soluciones:

Padding

Se conoce como *Padding* al agregado de variables sin usar entre dos *arrays* para reducir la interferencia entre ellos y de esta forma mejorar la tasa de *miss*. Agregar un *pad* modifica el *offset* del segundo *array*, por lo que se puede lograr que estos queden mappeados en distintas partes de la cache, evitando así conflictos. El tamanõs de los *pad* depende del esquema de *mappeo* de la memoria cache, su grado de asociatividad y el patrón de acceso a los datos del código. Valores típicos suelen ser múltiplos de del tamaño de línea. De esta manera, se logra evitar el acceso a datos distintos ubicados en la misma posición de memoria cache (*thrashing*). Esta optimización cambia la forma en que los arreglos son almacenados en memoria principal, reduciendo los accesos conflictivos. Una desventaja es que utiliza mayor cantidad de memoria.
En cuanto a los resultados obtenidos mediante esta técnica, no se observó una mejora significativa en la ejecución del algoritmo. Esto puede deberse a que el tamaño de la matriz sobre la que estamos operando tiene dimensión de N x N donde N=576, que para la arquitectura donde se llevaron a cabo las pruebas, era múltiplo del tamaño de la línea de cache (64 bytes).



Loop blocking o tiling



Loop blocking es una transformación de los ciclos en los cuales se incrementa su profundidad. Si la profundidad original es n, el resultado podrá ser de cualquier profundidad entre n+1 y 2n. Se busca mediante este método resolver mejor la localidad de los datos, para que la mayoría de ellos se encuentren en el cache en el momento y orden que se pretende leerlos de memoria. Esta técnica permite decrementar la cantidad de desaciertos de capacidad cuando se opera con arreglos grandes y mejora la localidad temporal (mantiene en la cache, datos que se usarán en el corto plazo). Asimismo, se reducen los accesos conflictivos, ya que los bloques pequeños pueden ser mantenidos en la cache.
Para esta optimizaciń existe un parámetro llamado blocking factor (B). Este factor deberá ser elegido para que las submatrices B x B entren completamente en la memoria cache. En la práctica, si bien al variar este parámetro se producía una reducción importante en la tasa de miss, esto no sucedía con el tiempo de ejecución. Una explicación para esto es que si bien se reducen la cantidad de miss en el total de accesos, al incorporara más lazos se ejecutarán mayor cantidad de instrucciones, lo cual hace incrementar los ciclos. de ejecución.

Software prefetch

Software prefet es una técnica de optimización que especula con los accesos futuros a datos e instrucciones. El obtjetivo es traer a la memoria cache los datos que se necesitarán en el futuro, al mismo tiempo que se precesarn los datos actuales. En lenguaje C, es posible realizar una lectura adelantada mediante *__builtin_prefetch()*, si la arquitectura lo soporta. La operación de prefetch es de tipo *non-blocking*, razón por la cual para poder implementar software prefetching se deberá contar con una arquitectura que permita al procesador continuar funcionando mientrar se realiza la operación. *Software prefetching* suele utilizarse frecuentemente cuando se tienen lazos de gran dimensiń dado que exhiben un patrón predecible de accesos. Para aplicar esta técnica se dividió el lazo correspondiente a la matriz A (que se recorre por filas) en 3 partes: prólogo, procesamiento principal y epílogo. En el prólogo se traen tantos elementos como entren en una línea de cache. En el procesamiente principal, se traen a memoria los siguientes datos y se procesan los que ya se se encuentran en ella. Finalmente, en el epílogo se termina de procesar los últimos datos. No se aplicó prefetch a la matriz B porque, al ser ésta recorrida por columnas, siempre producirá un miss dada la estructura de la memoria. Un parámetro configurable de esta técnica es la distance de prefetch D, que es igual a la cantidad de bloques leídos en forma adelantada dividida la cantidad de bloques preocesados. Otra definición dice que la distancia $D = l/s$ donde l es la latencia promedio de la memoria y s la cantidad de ciclos estimado de la iteración más

rápida del lazo. Tomamos D=8 dado que los resultados fueron mejores para la arquitectura donde se realizaron las pruebas. Un aspecto importante es que esta optimización permitió reducir el tiempo de ejecución significativamente, solucionando el problema mencionado anteriormente en la descripción de *Loop blocking*.

Otras mejoras posibles

Otras opciones para optimizar aún mas el algoritmo, fuera del alcance de nuestra implementación, incluyen por ejemplo el uso de alineación de datos (*data alignment*), que evita la fragmentación de los datos en la caché.
Otra posible optimización sería loop fusion que aplica a lazos con el mismo espacio de iteración una transformación que fusiona los cuerpos de cada lazo en uno solo.

3. Instalación

A continuación se detallan los pasos para instalar y ejecutar el trabajo práctico

1. Descomprimir el archivo TP2.tar.gz en una carpeta.
2. En un terminal, dentro de la carpeta del paso anterior, escribir ./instalar.
3. En un terminal, dentro de la carpeta del paso anterior, escribir ./tp2 [opciones].

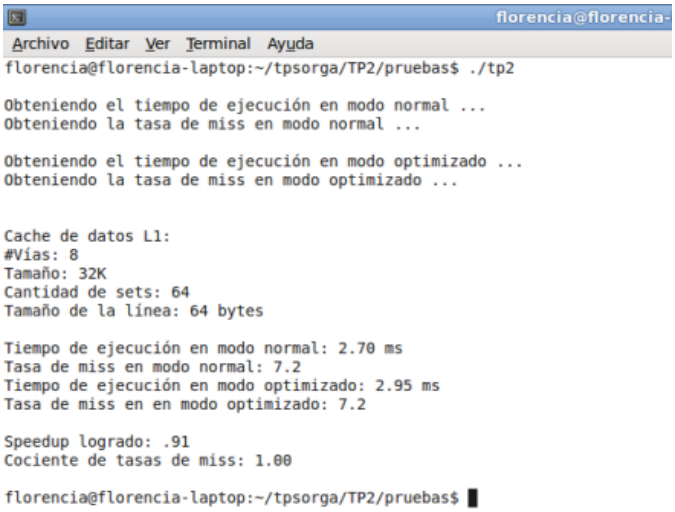
4. Corridas de prueba

En esta sección se presentan, a modo demostrativo, algunas de las corridas efectuada para probar el funcionamiento del trabajo práctico
También se comentan los resultados que se obtuvieron al variar los parámetros de cada optimización Las pruebas se realizaron para una memoria cache de datos, cuyas características son:

- **Cantidad de vías:** 8
- **Tamańo:** 32K
- **Cantidad de sets:** 64

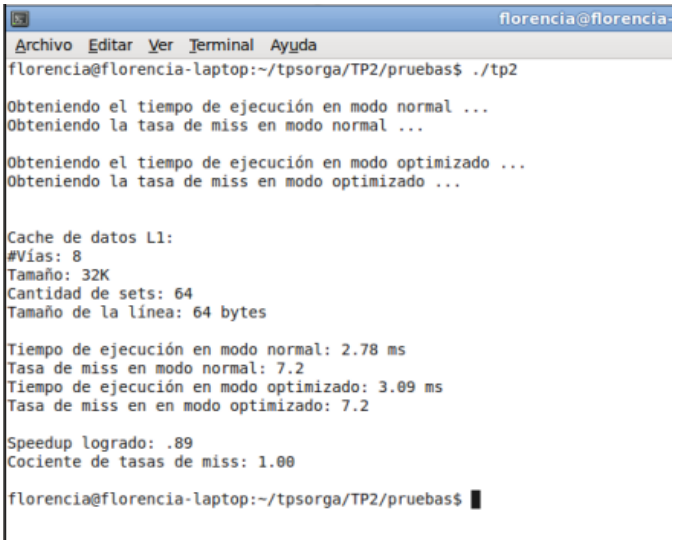
Padding

- pad=2*(tamaño de la línea)



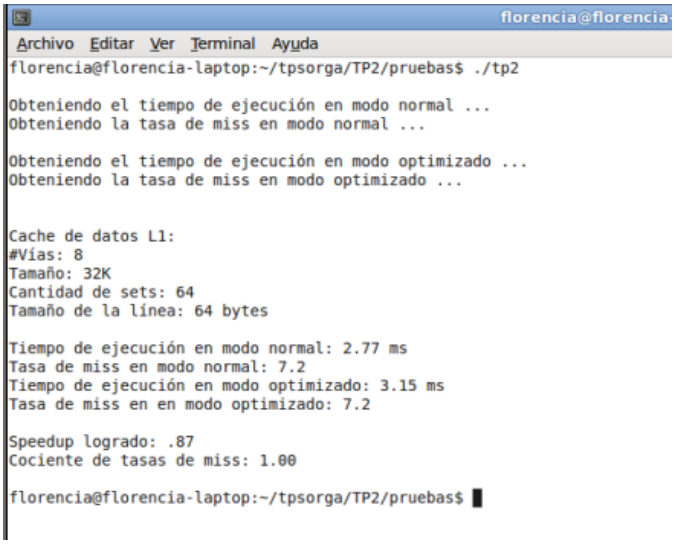
Padding

- pad=4*(tamaño de la línea)



Padding

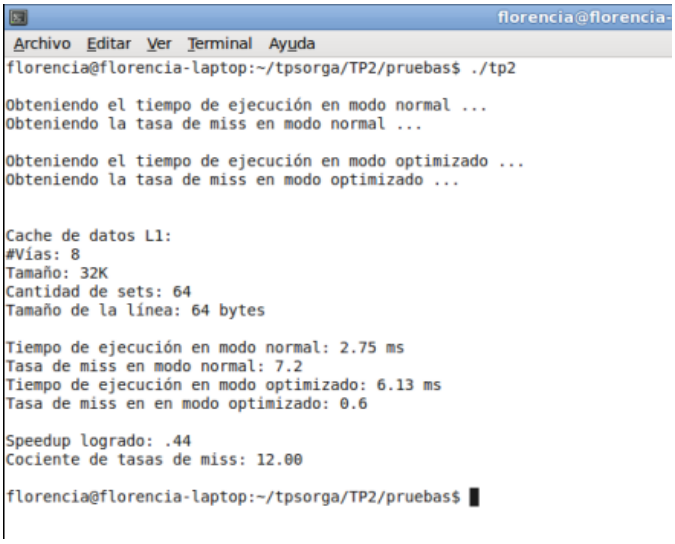
- pad=8*(tamaño de la línea)



Como se mencionó anteriormente, no se observan cambios significativos al agregar padding a las matrices.

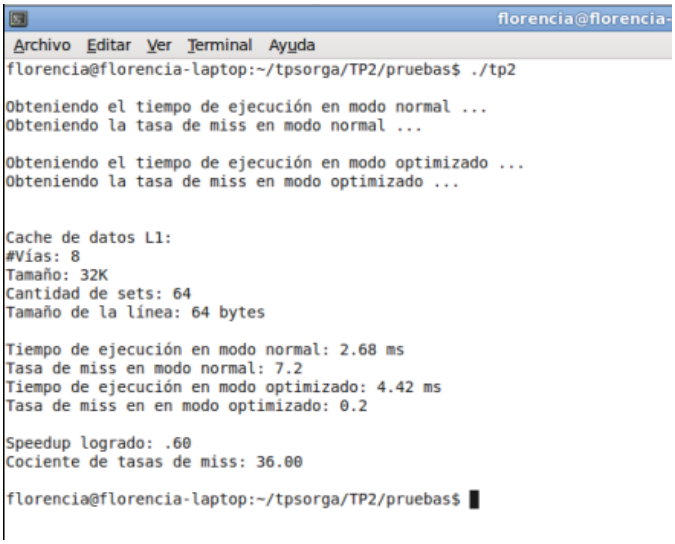
Padding y Blocking

- pad=2*tamaño de la línea
- B=2



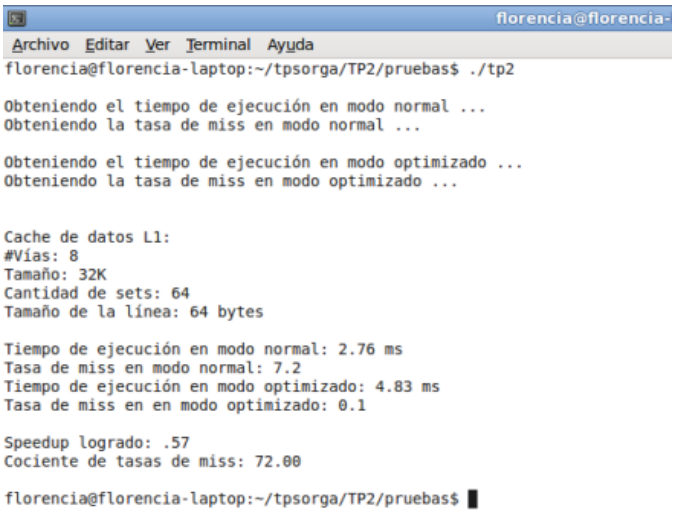
Padding y Blocking

- Con pad=2*tamaño de la línea
- B=4



Padding y Blocking

- pad=2*tamaño de la línea
- B=(cache_size)/8

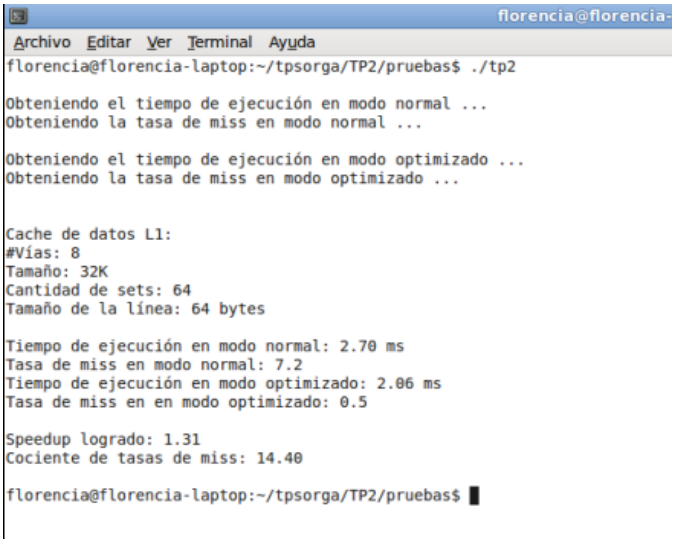


Podemos observar que el mientras más grande sea B (siempre siendo menor al tamaño de la memoria cache), disminuye más la tasa de miss. Sin embargo, el tiempo de ejecución es mayor al original. Como se mencionó antes, esto puede deberse a que ahora se ejecutan más instrucciones por el agregado de más lazos. Para poder implementar prefetch, se tomó B=(cache_size)/8*D

Prefetch, Padding y Blocking

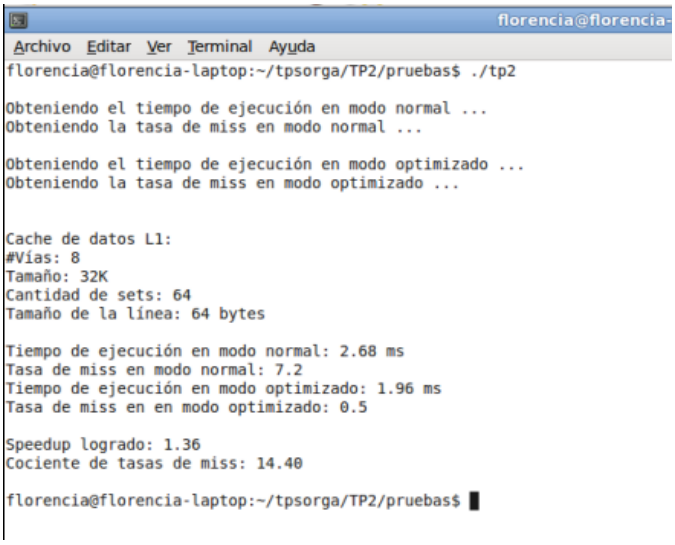
- pad=2*tamaño de la línea

- D=2
- B=D*(cache_size)/8



Prefetch, Padding y Blocking

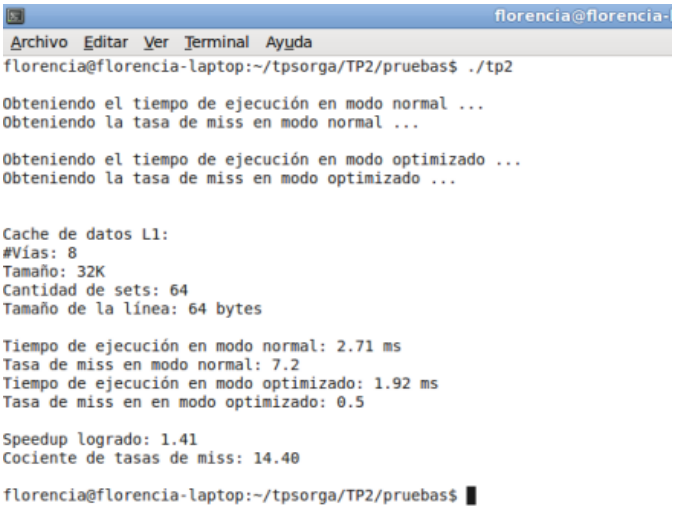
- pad=2*tamaño de la línea
- D=4
- B=D*(cache_size)/8



Prefetch, Padding y Blocking

- pad=2*tamaño de la línea

- D=8
- B=D*(cache_size)/8



Observamos que con el aumento de la distancia de prefetch, mejora el tiempo de ejecución y la tasa de miss se mantiene constante.

Como conclusión decidimos presentar esta última opción dado que presenta una mejora importante tanto en el tiempo de ejecución como en la tasa de miss.

5. Código fuente original

```
int main(void)
{

double a[N][N], b[N][N], c[N][N];
  int i,j,k;
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      for (k=0; k<N; k++)
          c[i][j] += a[i][k] * b[k][j];

  return 0;
}
```

6. Código fuente modificado

A continuación se incluye el código fuente del programa, con las modificaciones incorporadas.

```
#define N 576
#include <string.h>
#include <stdio.h>

#define D 8
#define CE 8 /*NOTA: esto es solo un ejemplo para un tamano de linea de 64 bytes. Esta con
cada arquitectura */


/*obtiene el blocking factor B */
int calcularB(int distancia){


return (CE*distancia);

}

int min(int a,int b)
{
if (a<b)
return(a);
else
```

```
return(b);
}


/* ejecucion del programa*/

void ejecutar(int pad,int B, int distancia){

double a[N][N],pad1[pad],b[N][N],pad2[pad],c[N][N];
int i0,j0,k0,i,j,k;
   for (i0=0;i0<N;i0+=B)
    for (j0=0;j0<N;j0+=B)
     for (k0=0;k0<N;k0+=B){
      /* PROLOGO*/
      /* proceso la primera linea*/

      for (k=k0; k<=min(k0+B-1,(N-1)); k+=CE){
        __builtin_prefetch(&a[i0][k]);
        /* proceso toda la matriz a menos la ultima fila */

        for (i=i0;i<=min(i0+B-1,(N-1))-1;i++)
          for (k=k0;k<=min(k0+B-1,(N-1));k+=CE){
            __builtin_prefetch(&a[i+1][k]);
            for (j=j0;j<=min(j0+B-1,(N-1));j++){

/* NOTA: esto es un ejemplo de codigo para una computadora donde entran 8 doubles en una l
El desarrollo del lazo varia de acuerdo a la computadora donde se instala el programa, par
 contamplar la cantidad de doubles que entran en una linea*/

                    c[i][j]+=a[i][k+0]*b[k+0][j];
                    c[i][j]+=a[i][k+1]*b[k+1][j];
                    c[i][j]+=a[i][k+2]*b[k+2][j];
                    c[i][j]+=a[i][k+3]*b[k+3][j];
                    c[i][j]+=a[i][k+4]*b[k+4][j];
                    c[i][j]+=a[i][k+5]*b[k+5][j];
                    c[i][j]+=a[i][k+6]*b[k+6][j];
                    c[i][j]+=a[i][k+7]*b[k+7][j];
                    /*EXPANSION*/
                    }
                   }


             /* EPILOGO*/
            /*proceso la ultima liInea*/
             for (i=min(i0+B-1,(N-1))-1;i<=min(i0+B-1,(N-1));i++)
                 for (k=k0;k<=min(k0+B-1,(N-1));k++)
                    for (j=j0;j<=min(j0+B-1,(N-1));j++)
                         c[i][j]+=a[i][k]*b[k][j];
```

```
                        }



}


int main(int argc, char** argv)
{

int m,pad,B;
/*obtengo el tamano de la linea*/
m=atoi(argv[1]);

/* calculo el pad, multiplo del tama\~no de la linea*/
pad=m*2;

/*obtengo el blocking factor*/
B=calcularB(D);

ejecutar(pad,B,D);



return(0);
```

7. Código en shell script de instalar y tp2

A continuación se incluyen los archivos ejecutables que realizan la instalación y medición de tiempos de ejecución y miss rates.

8. Enunciado

A continuación se incluye el enunciado del trábajo práctico.

9. Conclusiones

El estudio de la problemática de la multiplicación entre matrices cuadradas generó diversos algoritmos para su optimización. Las condiciones que se presentan en la misma (saltos en la memoria principal, *thrashing*, gran cantidad de ciclos) la vuelven ideal para tomarla como caso de estudio. Utilizando *padding*, no logramos mejorar significativamente el rendimiento de la memoria cache al ejecutar la aplicación. Por otro lado, encarando la resolución del problema con la técnica de *blocking* se logró una importante mejora en el rendimiento, aunque no así en el tiempo de ejecución. Finalemente, incluir *software prefetch* produce una mejora en este último aspecto.
Al juntar todas las optimizaciones en un solo código fuente, pudimos observar que se obtiene una mejora global. Es importante seleccionar los parámetros de cada optimización de forma tal que se obtenga un mayor beneficio. En definitiva, la multiplicación de matrices es una situación ideal para ejercitar los distintos métodos de optimización investigados.

A modo de conclusión, podemos comentar que una vez concluidos los casos de prueba y luego de sendos testeos, notamos que se obtiene una mejora en el procesamiento y la aplicación hace mejor uso de los recursos del sistema.

Referencias

[1] Herramientas de profiling Valgrind. http://valgrind.org/

[2] The Cache Performace and Optimizations of Blocked Algorithms. Monica S. Lam an Edward E. Rothberg and Michael E. Wolf. Computer Systems Laboratory, Stanford University. http://suif.standford.edu/papers/lam91.ps

[3] GCC: Online Documentation. http://gcc.gnu.org/onlinedocs/

[4] Oetiker, Tobias, "The Not So Short Introduction To LaTeX2", http://www.physics.udel.edu/∼dubois/lshort2e/

[5] Kernighan, Brian W. / Ritchie, Dennis M. El Lenguaje De Prorgamación C. Segunda Edición, PRENTICE-HALL HISPANOAMERICANA SA, 1991.

[6] Patterson / Hennesy. Computer Organization and Design; The Hardware-Software Interface. Second Edition. Morgan Kaufman, 1997.