Práctica 3

Memoria Caché y Rendimiento

Junco de las Heras y Marta Vaquerizo

Índice

Ejercicio 0

La captura de pantalla se encuentra en la carpeta `ejercicio0`.

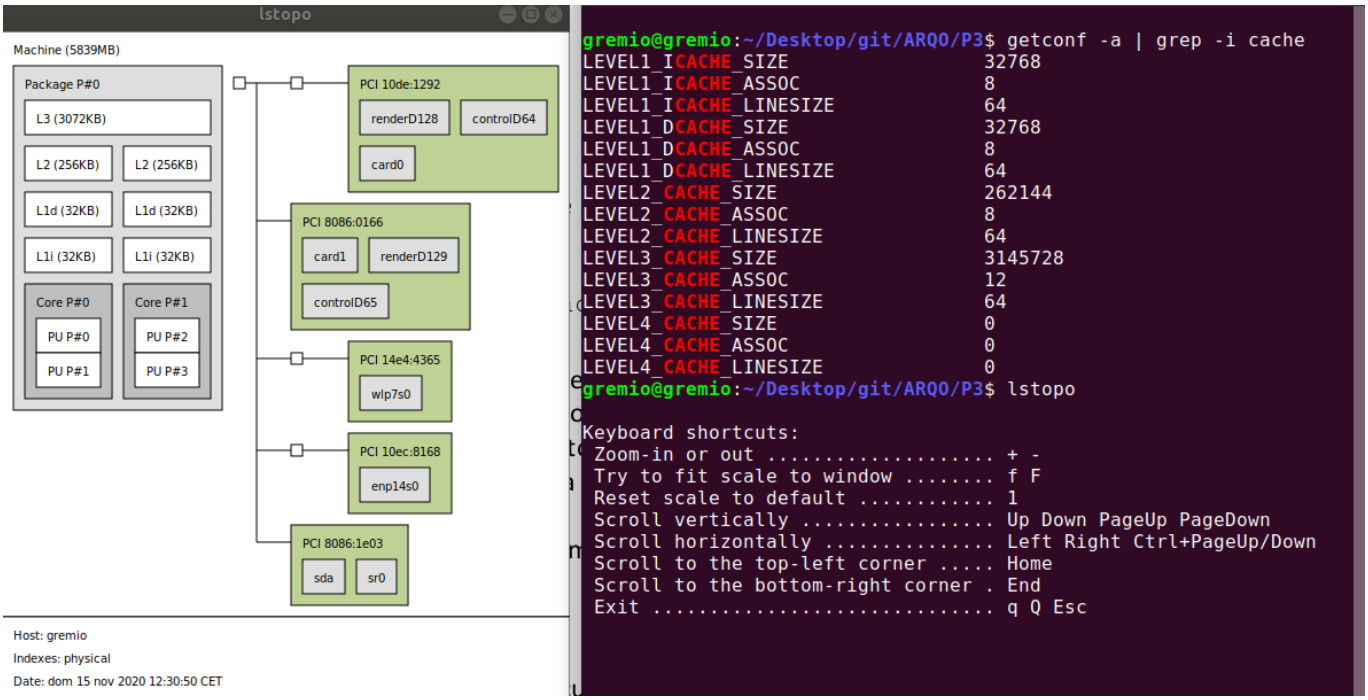

Figura 1: A la izquierda, el gráfico con lstopo. A la derecha, la salida del comando getconf -a filtrando las líneas que contienen la palabra cache.

Se puede observar que en la máquina donde se ha ejecutado hay 3 niveles de caché.
En el nivel 1 hay una parte de instrucciones y otra de datos, ambas con 32768 Bytes en total, de 8 vías con tamaño de línea de 64 Bytes.
El nivel 2 de caché tiene 262144 Bytes en total, de 8 vías con tamaño de línea de 64 Bytes.
El nivel 3 tiene 3145728 Bytes en total, de 12 vías y 64 Bytes por línea.

El caché de nivel 4 muestra que tiene un tamaño de 0, y en el lstopo no lo muestra, eso quiere indicar que no hay caché de nivel 4.

Se puede apreciar como a medida que el nivel de caché aumenta, el tamaño del caché también lo hace, así como del número de vías. Eso es porque a más tamaño de caché, o a más vías, más tiempo le cuesta en conseguir el dato del caché. La longitud de la línea es constante de 64 Bytes siempre.

Ejercicio 1

Los ficeros, scripts, y gráficas generados en este ejercicio se encuentran en la carpeta `ejercicio1`.

Para todos los ejercicios, nuestro número P es 9, ya que somos la pareja 5 y $5\,\%7 + 4 = 9$.

(1) Se toman las medidas con un script bash llamado *slow_fast_time.sh*, ejecutando por cada tamaño de matriz 3 veces el programa slow y el programa fast, intercalándolos para que no se guarde el caché entre una ejecución y otra. El tiempo resultante de cada programa para un tamaño dado es la suma de los tiempos que han tardado para ese tamaño dividido entre el número de ejecuciones con el mismo tamaño.

   En vez de que N variase entre $10000 + 1024 * P$ hasta $10000 + 1024 * (P + 1)$ se ha optado porque valiese entre $1 + 8 * P$ y $1 + 256 * (P + 1)$ para que la gráfica fuese más informativa, ya que a números muy grandes las gráficas salían casi líneas paralelas y separadas.

(2) Hay que realizar varias repeticiones de la ejecución de un mismo programa para un número fijo de tamaño de la matriz porque sino el sistema operativo podría usar el caché de una ejecución para la siguiente, y no mediríamos los tiempos de acceso a caché que nos interesa, sino un tiempo bastante menor.

(3) Se guardan los datos en el fichero *time_slow_fast.dat*.

(4) Se guarda la gráfica en el fichero *time_slow_fast.png*.

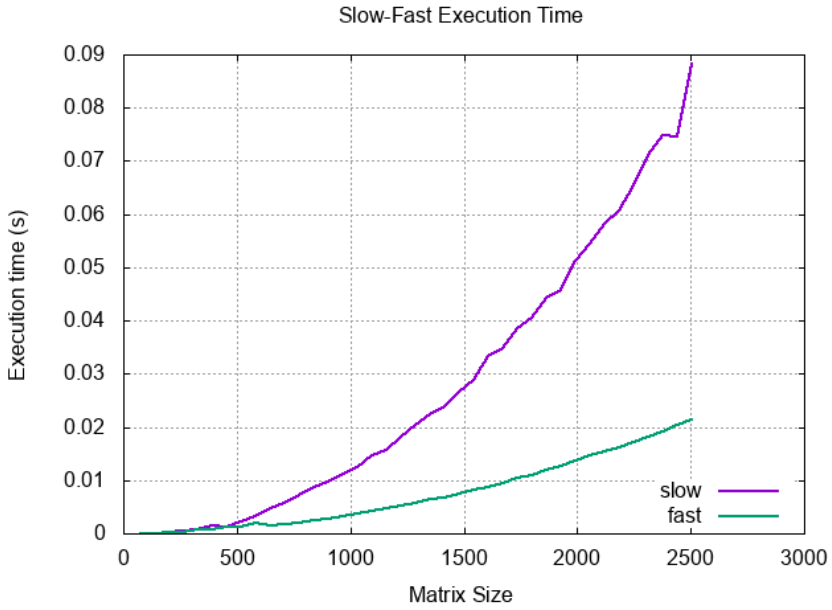

Figura 2: Gráfica de **time_slow_fast.png**

(5) Cuando el tamaño de la matriz es pequeño (Antes de 500 de tamaño), se acceden a menos posiciones de memoria, así que aunque un programa sea más óptimo que el otro, la diferencia no puede ser mucha, pero a medida que aumenta el tamaño, el número de fallos en caché empieza a tomar relevancia.

En la matriz mat[nFilas][nColumnas] tienen nFilas y nColumnas, cada fila tiene sus elementos (columnas) consecutivas. mat[0] es la fila con sus columnas $0 \ldots$ nColumnas -1 consecutivas. Se puede apreciar cuando se reserva la memoria para la matriz en el programa $arqo3.c$ que la matriz se guarda por filas.

Ejercicio 2

Los ficeros, scripts, y gráficas generados en este ejercicio se encuentran en la carpeta `ejercicio2`.

(1) Para este apartado, se ha creado un script "slow_fast_misses.sh" en el que, para cada N variando entre $2000 + 512 * P = 6608$ y $2000 + 512 * (P + 1) = 7120$ y con un incremento en saltos de 64 unidades (salen nueve N distintas), se obtienen los fallos de caché de datos del nivel 1, dependiendo del tamaño de ésta (1024, 2048, 4096, y 8192 Bytes).

(2) En este apartado, los datos generados en el anterior, se guardan en un fichero *cache_<tamaño>.dat* tal y como se indica en el enunciado:
`< N >  < D1mr "slow" >  < D1mw "slow" >  < D1mr "fast" >  < D1mw "fast" >`
para cada tamaño de la caché (1024, 2048, 4096, y 8192 Bytes). Es por eso, que en este apartado se han generado cuatro ficheros .dat.

(3) Para este apartado, se ha creado un script, llamado "slow_fast_misses_graphics.sh", en el que se generan cuatro ficheros .dat, y cuatro gráficas:

Ficheros:

- cache_slow_read.dat
- cache_slow_write.dat
- cache_fast_read.dat
- cache_fast_write.dat

Gráficas:

- cache_lectura_slow.png (número de fallos de lectura al ejecutar `slow.c`)
- cache_escritura_slow.png (número de fallos de escritura al ejecutar `slow.c`)
- cache_lectura_fast.png (número de fallos de lectura al ejecutar `fast.c`)
- cache_escritura_fast.png (número de fallos de escritura al ejecutar `fast.c`)

Se han generado cuatro gráficas, y no dos, para que se pueda visualizar mejor cada gráfica, ya que sino se tendrían 16 "líneas"por cada gráfica.

El objetivo de este script es generar los ficheros con un formato cómodo para generar las gráficas posteriormente en ese mismo script. El formato que tienen los ficheros es el siguiente:

`< TamCache >  < array_datos >`

donde `TamCache` es el tamaño de la caché (varía entre los valores: 1024, 2048, 4096, y 8192 Bytes), y `array_datos` es un array que contiene en cada posición el número de fallos de caché de datos cuando N toma un valor, es decir, que cada posición de `array_datos` corresponde a un N de los nueve que toma. Dependiendo del fichero en el que se guardan estos datos, el número de fallos de caché de datos puede ser de ejecutar el programa `slow` o `fast`, y a su vez pueden ser de lectura (read) o escritura (write). Para obtener el número de fallos, se utilizan los ficheros cache_<TamCache>.dat generados en el apartado anterior, es por ello, que antes de ejecutar el script de este apartado, es necesario ejecutar el script de los apartados anteriores ("slow_fast_misses.sh").

A continuación se van a mostrar las gráficas:

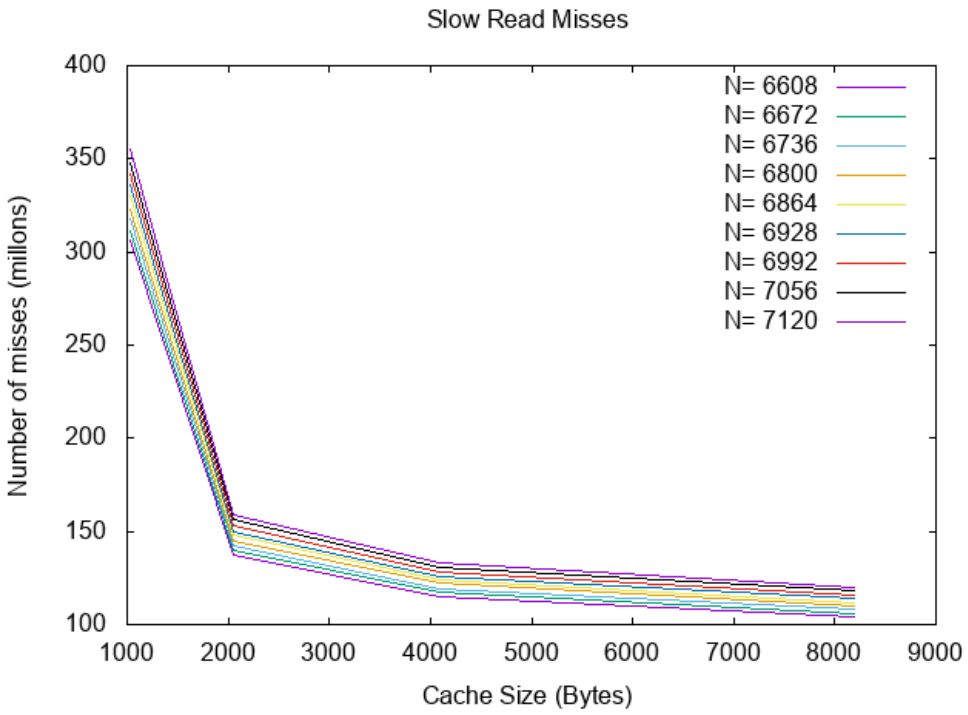

Figura 3: Gráfica de **cache_lectura_slow.png**

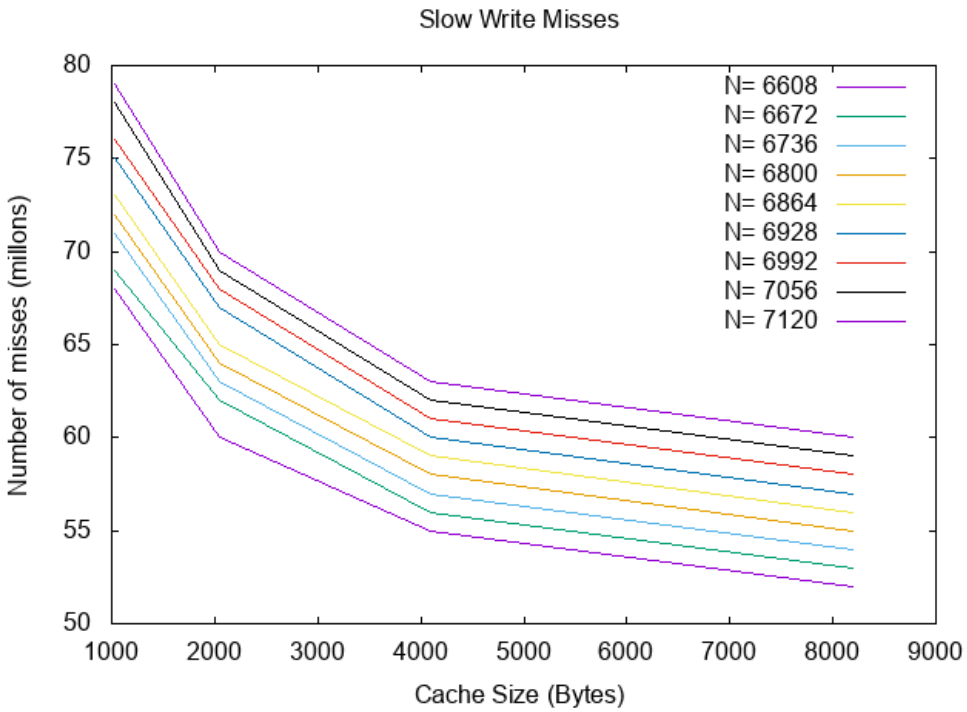

Figura 4: Gráfica de **cache_escritura_slow.png**

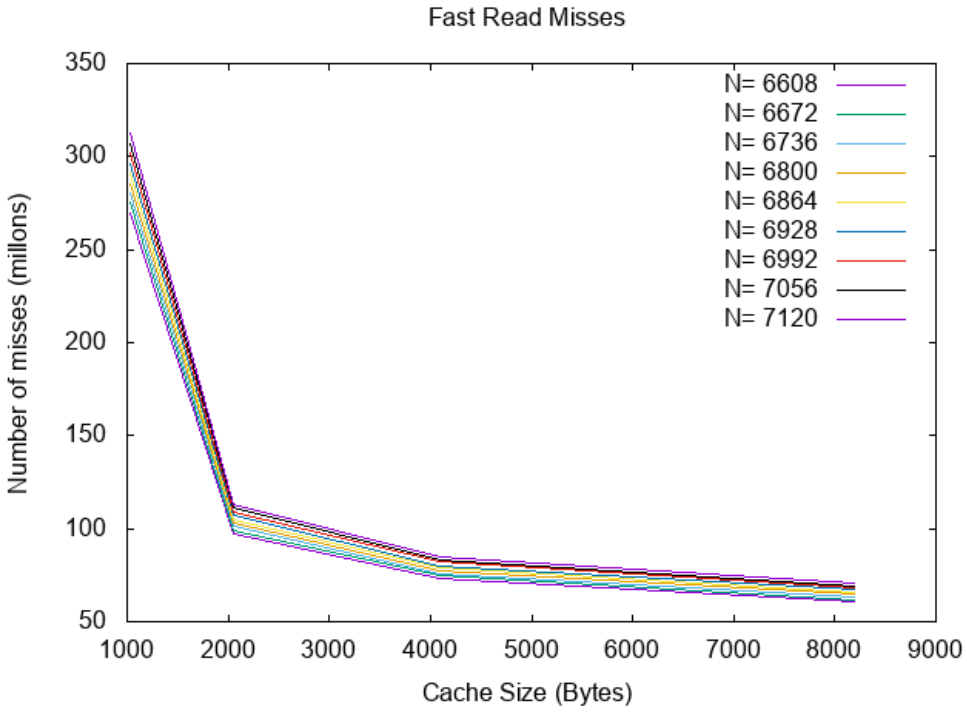

Figura 5: Gráfica de **cache_lectura_fast.png**

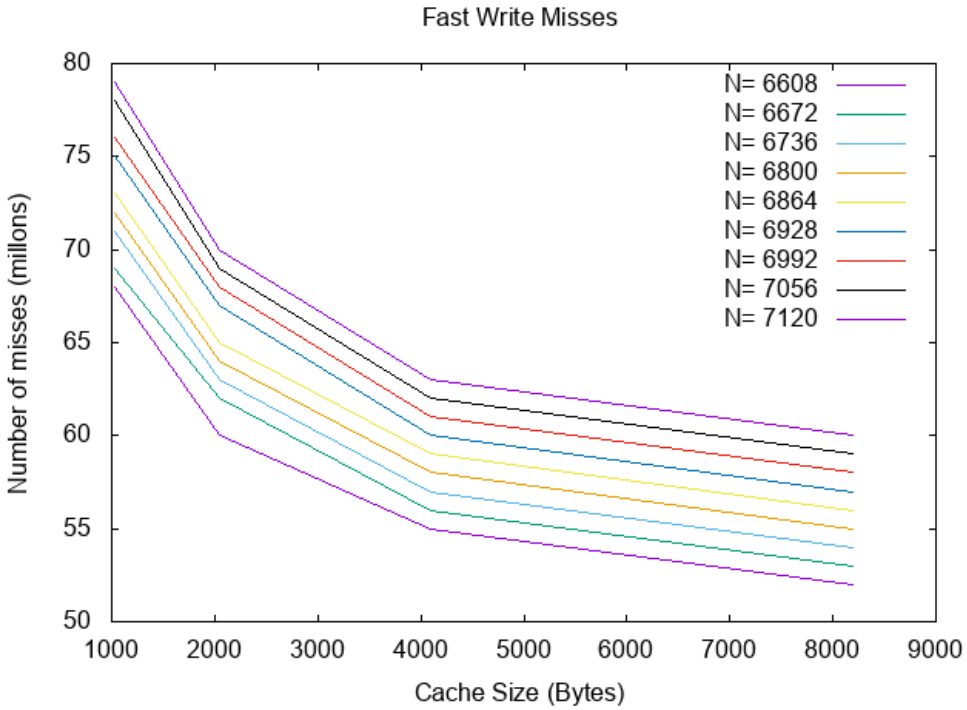

Figura 6: Gráfica de **cache_escritura_fast.png**

Una cosa que se puede observar en las gráficas es que el número de fallos de caché de lectura es muy grande, del orden de 350 millones en el caso del programa `slow`, y de 300 millones, con el programa `fast`, mientras que el de escritura sigue siendo grande, del orden de 80 millones, pero mucho menor que el de lectura. Esto se debe: por una parte, a que se está trabajando de matrices N*N donde N toma valores cercanos al 7000; y por otra parte, a que las cachés son de correspondencia directa, es decir, a la hora de reemplazar un bloque, solo hay una opción.

(4) Por un lado, se puede observar que las gráficas de las Figuras 3 y 5, correspondientes a los fallos de lectura de `slow` y `fast`, son inversamente proporcionales al tamaño de la caché. Se puede destacar en estas dos gráficas una bajada bastante brusca, debido a que cuando la caché tiene tamaño 1024 Bytes, el número de fallos de cachés es muy grande, del orden de 350 millones, y cuando el tamaño del caché se duplica a 2048 Bytes, el número de fallos baja a un valor de alrededor de 150 millones en la gráfica 3 y a 100 millones en la gráfica 5.

Por otro lado, las gráficas de las Figuras 4 y 6, correspondientes a los fallos de escritura de `slow` y `fast`, tienen una tendencia más suave, en el sentido de que entre los valores de los diferentes tamaños de caché no hay tanta diferencia, pero aún así a medida que el tamaño de caché aumenta, el nímero de fallos disminuye. Las gráficas son exáctamente iguales porque el tanto el *slow.c* como el $fast.c$ acceden a la matriz para escribir en el mismo orden, solo cambia el acceso a lectura.

Al fijar el tamaño de la caché, si comparamos ambos programas, nos encontramos con dos casos:

1. Comparando las gráficas del número de fallos de escritura (Figuras 4 y 6). Ambos programas tienen la misma gráfica, se producen el mismo número de fallos porque el tanto el *slow.c* como el $fast.c$ acceden a la matriz para escribir en el mismo orden, solo cambia el acceso a lectura.
2. Comparando las gráficas del número de fallos de lectura (Figuras 3 y 5) se puede observar que el número de fallos en el programa `fast` es 50 millones de fallos menor que el del programa `slow` al principio, y a medida que el tamaño de caché aumenta la diferencia de 50 millones va disminuyendo. Esto ocurre ya que el programa `slow` realiza la suma de los elementos columna por columna, y como las matrices en C se guardan en memoria como arrays contiguos (filas de la matriz), el programa tiene que acceder a la primera posición de cada array para sumarlo por columnas. El problema es que tenemos N arrays de tamaño N (varía entre 6608 y 7120), es decir, cada array ocupa N números de 4 Bytes cada uno (N*4 Bytes), que es mucho mayor que el tamaño de bloque (64 B = 16 enteros). Esto implica que cada array está repartido por varios bloques. Por otro lado, el programa `fast` realiza la suma de los elementos fila por fila, entonces, los fallos de caché que se producen son debido a que el array está repartido en varios bloques bloques. La diferencia del número de fallos tenderá a cero pues el número de fallos tiende a cero a medida que aumenta el tamaño del caché

Ejercicio 3

Los ficeros, scripts, y gráficas generados en este ejercicio se encuentran en la carpeta `ejercicio3`.

En este ejercicio se han implementado dos nuevos programas: `multiplicaction_slow.c` y `multiplication_fast.c`. Por un lado, el primer programa realiza la multiplicación de matrices normal, es decir, fila por columna. Y por otro lado, el segundo programa realiza la multiplicación que llamamos traspuesta. Primero traspone la segunda matriz y de esta manera, el cálculo se realiza fila por fila, lo que permite un cálculo más eficientemente.

Se ha implementado la función void printMatrix(Tipo **, int) en el fichero *arqo3.c*, una función auxiliar, que sirve para imprimir por la salida estándar el contenido de la matriz. Su uso es para debugear, y en el código entregado no se hace ninguna llamada a esta función, pero se ha usado para comprobar que la multiplicación efectivamente funciona.

(1) Para este apartado, se ha creado un script *mult_time.sh*, que calcula y guarda los tiempos de ejecución de cada uno de los programas en el fichero *mult_time.dat*. Este fichero se utiliza en este mismo script para gráficar estos datos. Se ha elegido un intervalo para tamaño de las matrices N, que varía entre $1 + 8 * (P) = 73$ y $1 + 8 * (P + 1) = 270$. Se ha considerado este intervalo de tamaños, ya que el especificado en el enunciado era de tamaños muy grandes y las gráficas quedaban líneas paralelas y separadas, no muy informativas.

(2) En este apartado, se ha creado un script *mult_misses.sh*, que obtiene los fallos de caché de datos de lectura y de escritura al ejecutar ambos programas.

(3) Los datos generados en los apartados (1) y (2) se guardan en el fichero *mult.dat*, que está generado por el script *mult_misses.sh* del apartado (2). Este utiliza el fichero *mult_time.dat* del apartado (1), y los datos que genera él mismo y los guarda tal y como se piden en el enunciado:
`< N >` `< tiempo "normal" >` `< D1mr "normal" >` `< D1mw "normal" >`
`< tiempo "trasp" >` `< D1mr "trasp" >` `< D1mw "trasp" >`

(4) En este apartado se han generado tres gráficas:

- mult_time.png
- mult_cache_lectura.png
- mult_cache_escritura.png

La primera se ha genera con el script *mult_time.sh* del apartado (1) a partir de los datos generados en el mismo (*mult_time.dat*), y las otras dos se generan con el script *mult_misses.sh* del apartado (2), a partir del fichero *mult.dat*.

A continuación se van a mostrar las gráficas:

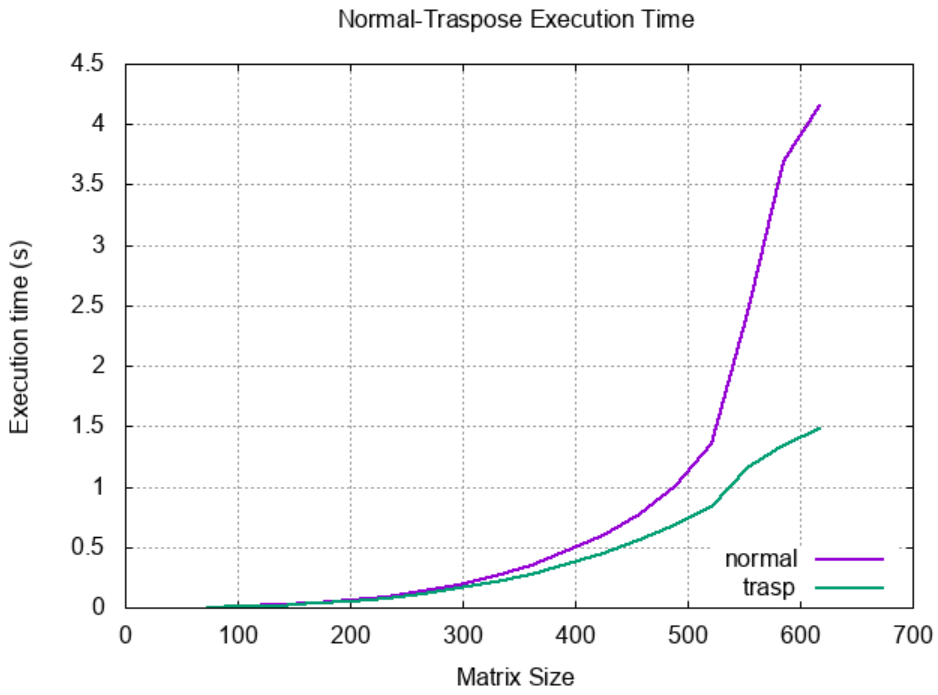

Figura 7: Gráfica de **mult_time.png**

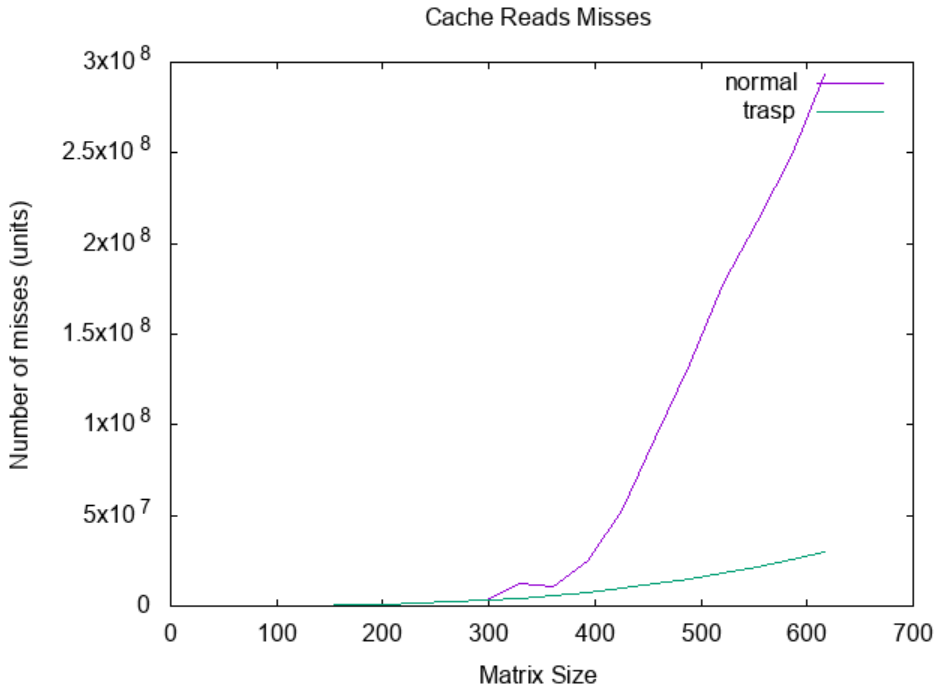

Figura 8: Gráfica de **mult_cache_lectura.png**

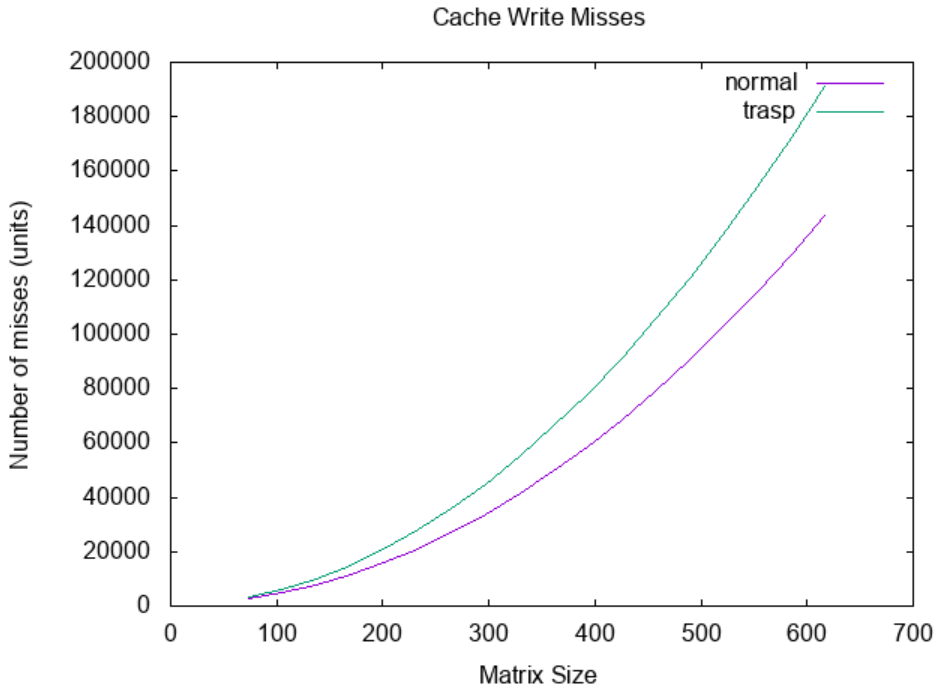

Figura 9: Gráfica de **mult_cache_escritura.png**

(5) En este apartado se va a tener en cuenta lo que se ha observado en el ejercicio 2.
Observación 1: las matrices en C se guardan fila por fila, de manera contigua. Se van a comentar las distintas gráficas del apartado anterior, respectivamente:

1. **Figura 7.**
En ambos programas, se ve claramente que a medida que se aumenta el tamaño de las matrices, aumenta el tiempo de ejecución. Para tamaños medianamente pequeños (menor que 300), ambos programas tardan más o menos lo mismo, pero si se sigue aumentando el tamaño de las matrices(N mayor que 300), se observa que la diferencia de tiempo de ejecución entre ambos programas aumenta cada vez más. Esto se debe a lo que ocurre en la **Observación 1**. Como el programa `multiplication_slow.c` accede a la segunda matriz por columnas, tarda más que el programa `multiplication_fast.c` que accede a ambas matrices por filas. Cabe destacar que en el programa `multiplication_fast.c` la acción de trasponer la segunda matriz consume tiempo, pero es despreciable respecto al consumo de tiempo de la multiplicación como tal ya que transponer tiene una complejidad de $O(N^2)$ frente al $O(N^3)$ de la multiplicación.
2. **Figura 8.**
En esta gráfica se puede apreciar que el programa `multiplication_slow.c` presenta muchos más fallos de lectura que el programa `multiplication_fast.c`. Esto es así como consecuencia de la **Observación 1**, es decir, el programa `multiplication_slow.c` tiene que acceder a muchos más bloques de memoria (para leer las columnas) distintas, y a medida que el tamaño de las matrices aumenta, los fallos de caché de lectura

para este programa crecen potencialmente.

3. **Figura 9.**
   En esta gráfica se puede apreciar que el programa `multiplication_fast.c` presenta más fallos de escritura que el programa `multiplication_slow.c`. Esto es debido a que en la multiplicación se accede para la escritura a las mismas posiciones de la matriz resultado, pero el programa fast además transpone la matriz, obteniendo más fallos de caché.

Ejercicio 4

Los ficeros, scripts, y gráficas generados en este ejercicio se encuentran en la carpeta `ejercicio4`.

Para este ejercicio se han creado los siguientes scripts:

- mult_cache_misses.sh
- mult_cache_misses_graphics.sh
- mult_cache_misses_asociative_graphics.sh

El primer script, obtiene y guarda el número de fallos de caché de datos tanto de escritura como de lectura para caché asociativa de hasta n-vías (n tiene que ser múltiplo de 2), que incluye la de acceso directo (1-vía) de distintos tamaños. Por ejemplo, si n = 4 y el tamaño de caché es de 2048 Bytes, se van a generar 3 ficheros: *cache_Directa_*2048*.dat*, *cache_Asociatividad_vias_*2_2048*.dat* y *cache_Asociatividad_vias_*4_2048*.dat*.

El segundo script, cambia el formato del fichero como se hacía en el ejercicio 2, y genera 4 gráficas para la caché de acceso directo. Y el tercer script, es igual que es segundo solo que para la cache de s-vías (siendo s un número entre 2 a n). Para ejecutar estos scripts, es necesario ejecutar el primer script. En todos los scripts, el tamaño de matriz N varía entre 1 + 8*P y 1 + 32*(P+1) de 32 en 32 unidades.

Con estos scripts, hemos decidido variar los tamaños de caché entre: 1024, 2048, 4096 y 8192 Bytes. Además, se van a comparar para estos tamaños, las cachés directa, de 2-vías y de 4-vías para distintos tamaños de matriz. Se va a realizar así el experimento, para comprobar lo estudiado en la teoría, es decir, que para un mismo tamaño de caché, si aumentamos la asociatividad (en nuestro caso, por vías), dismuirán los fallos de caché.

Primeramente, se ha ejecutado el primer script, generando los siguientes ficheros:

Directa	**2-vías**	**4-vías**
cache_Directa_1024.dat	cache_Asociatividad_vias_2_1024.dat	cache_Asociatividad_vias_4_1024.dat
cache_Directa_2048.dat	cache_Asociatividad_vias_2_2048.dat	cache_Asociatividad_vias_4_2048.dat
cache_Directa_4096.dat	cache_Asociatividad_vias_2_4096.dat	cache_Asociatividad_vias_4_4096.dat
cache_Directa_8192.dat	cache_Asociatividad_vias_2_8192.dat	cache_Asociatividad_vias_4_8192.dat

Posteriormente, se han ejecutado el segundo y el tercer script, dando lugar a las siguientes gráficas:

Multiplicación Normal	**Multiplicación Traspuesta**
cache_Directa_escritura_normal.png	cache_Directa_escritura_trasp.png
cache_Asociatividad_vias_2_escritura_normal.png	cache_Asociatividad_vias_2_escritura_trasp.png
cache_Asociatividad_vias_4_escritura_normal.png	cache_Asociatividad_vias_4_escritura_trasp.png
cache_Directa_lectura_normal.png	cache_Directa_lectura_trasp.png
cache_Asociatividad_vias_2_lectura_normal.png	cache_Asociatividad_vias_2_lectura_trasp.png
cache_Asociatividad_vias_4_lectura_normal.png	cache_Asociatividad_vias_4_lectura_trasp.png

A continuación se van a mostrar estas gráficas:

CACHÉ DIRECTA

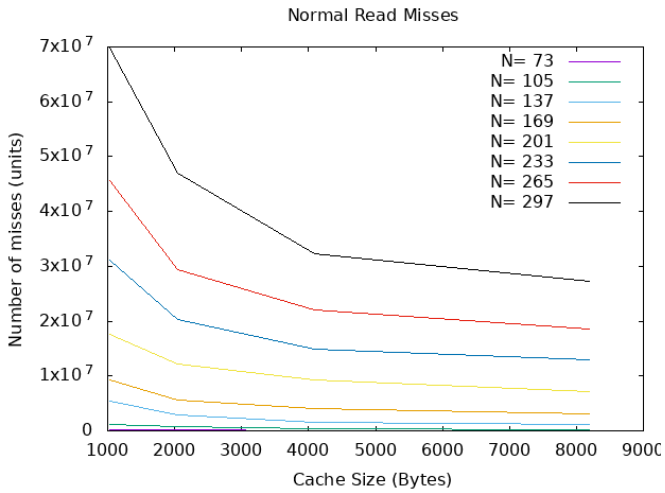

Figura 10: Fallos de lectura (normal)

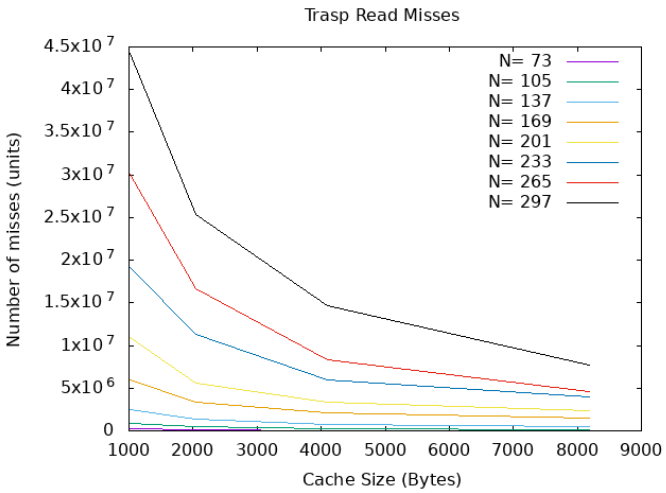

Figura 11: Fallos de lectura (trasp)

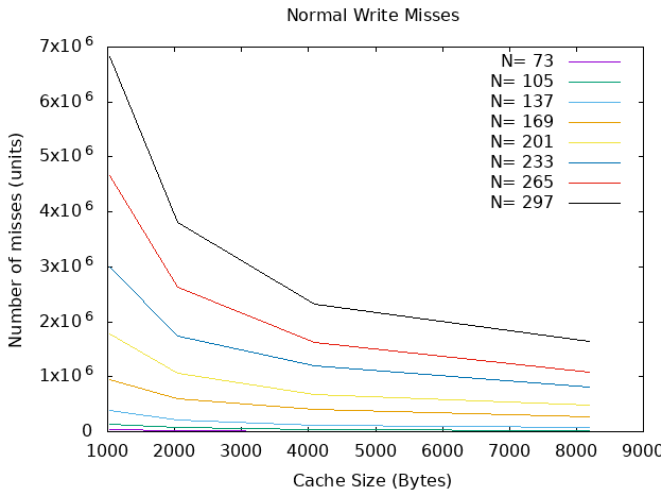

Figura 12: Fallos de escritura (normal)

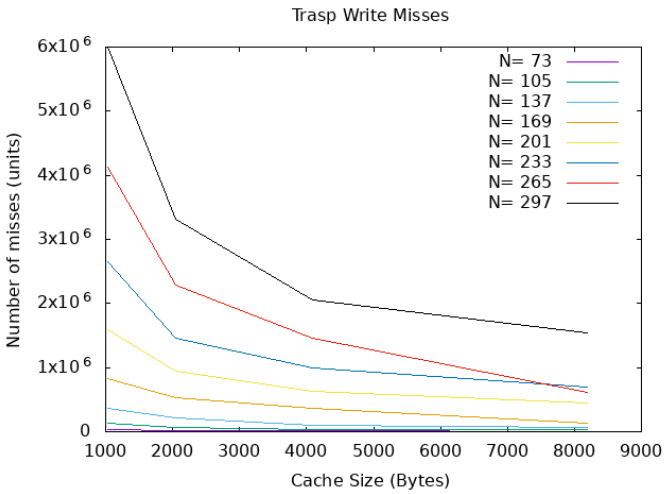

Figura 13: Fallos de escritura (trasp)

CACHÉ ASOCIATIVA 2-VÍAS

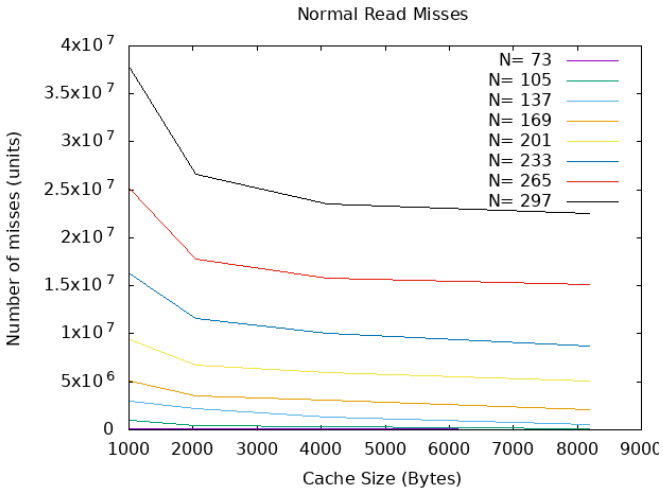

Figura 14: Fallos de lectura (normal)

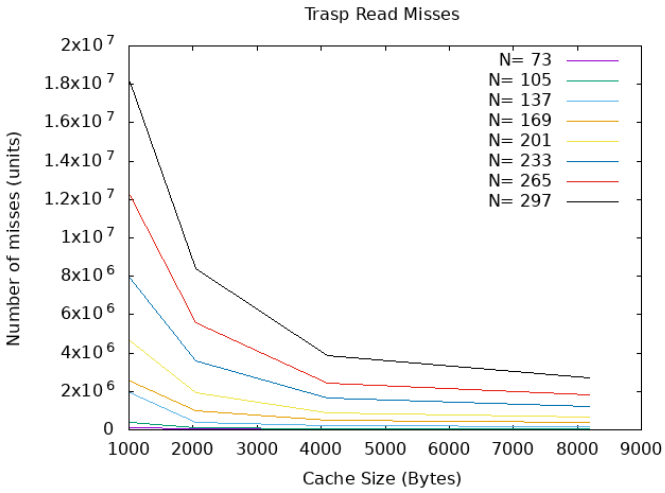

Figura 15: Fallos de lectura (trasp)

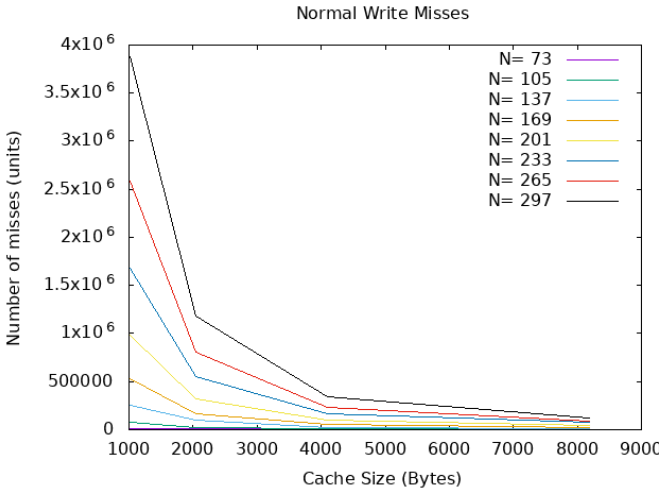

Figura 16: Fallos de escritura (normal)

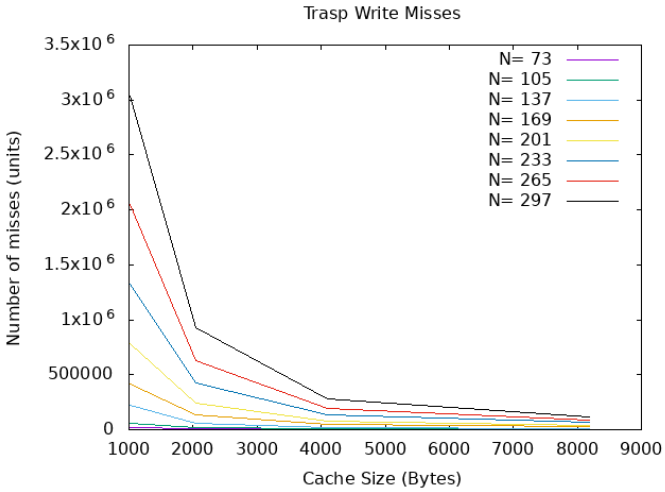

Figura 17: Fallos de escritura (trasp)

CACHÉ ASOCIATIVA 4-VÍAS

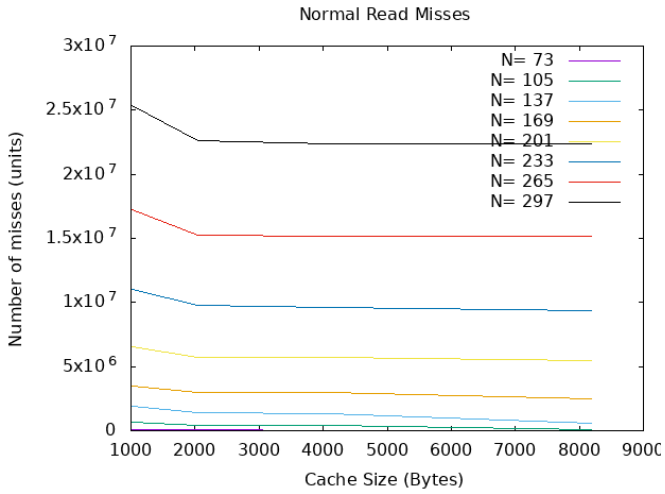

Figura 18: Fallos de lectura (normal)

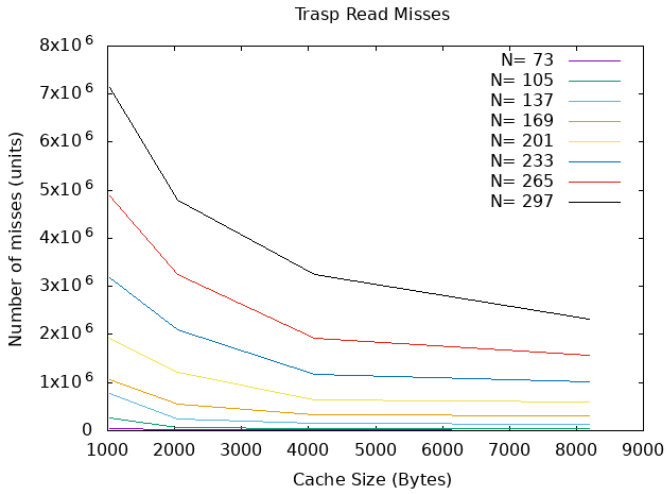

Figura 19: Fallos de lectura (trasp)

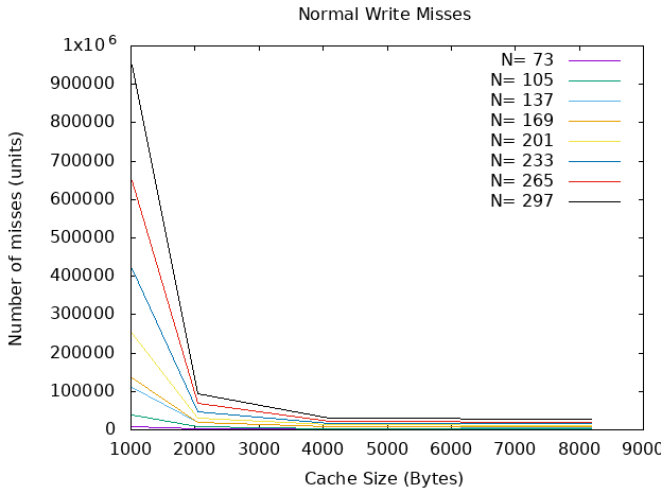

Figura 20: Fallos de escritura (normal)

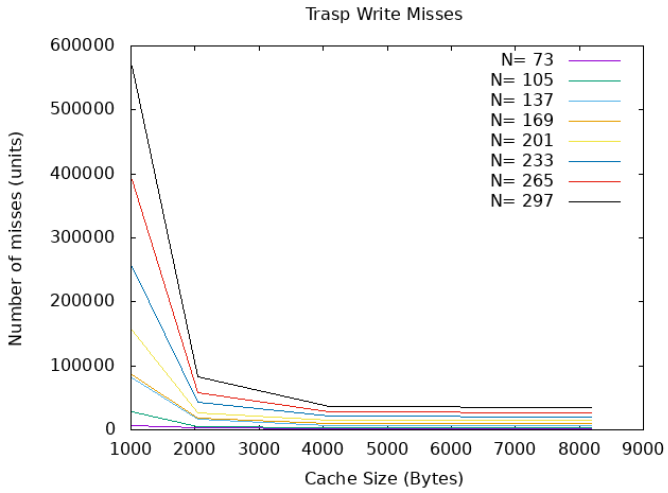

Figura 21: Fallos de escritura (trasp)

Se van a comentar los resultados para un tamaño de caché fijado (por ejemplo 1024 Bytes). Si, por ejemplo, comparamos las figuras 10, 14 y 18, se ve claramente que el número de fallos para un tamaño de matriz N, en la caché directa (Figura 10) es mayor que en las otras dos asociativas; y que en la asociativa de 2-vías es mayor de en la de 4-vías. Este efecto en el que el número de fallos de caché de datos cumple lo siguiente:

`directa` > `asociativa 2-vías` > `asociativa 4-vías`

se cumple en todas las gráficas, tanto para fallos de lectura como de escritura. Esto es lo que se quería probar con este experimento, ya que es lo que se ha visto en teoría.

Ejercicio extra, ejecución en cluster

Los ficeros, scripts, y gráficas generados en este ejercicio se encuentran en la carpeta `ejercicio1_en_cluster`.

En este ejercicio se muestran los resultados del ejercicio 1 ejecutado en el clúster. Como la ejecución en el clúster es optativa, solo hemos ejecutado el ejercicio 1 a modo de ejemplo. Para realizar las gráficas de los ejercicios 2, 3 y 4 sería seguir el mismo procedimiento, dado que ya tenemos los .sh creados.

```
[arqo06@labomat36 codigo]$ qsub cluster_script.sh slow_fast_time.sh
Your job 86680 ("cluster_script.sh") has been submitted
[arqo06@labomat36 codigo]$ qstat
job-ID  prior   name       user         state submit/start at     queue
                      slots ja-task-ID
-----------------------------------------------------------------------------
-----------------------------------
  86680 0.50500 cluster_sc arqo06       r     11/28/2020 23:36:29 amd.q@comput
e-0-5.local            1
[arqo06@labomat36 codigo]$ ls
arqo3.c             Makefile              slow
arqo3.h             multiplication_fast   slow.c
cluster_script.sh   multiplication_fast.c slow_fast_time.sh
cluster_script.sh~  multiplication_slow   time_slow_fast.dat
fast                multiplication_slow.c time_slow_fast.png
fast.c              salida.out
[arqo06@labomat36 codigo]$
```

Figura 22: Prueba de la ejecución del ejercicio 1 en el clúster.

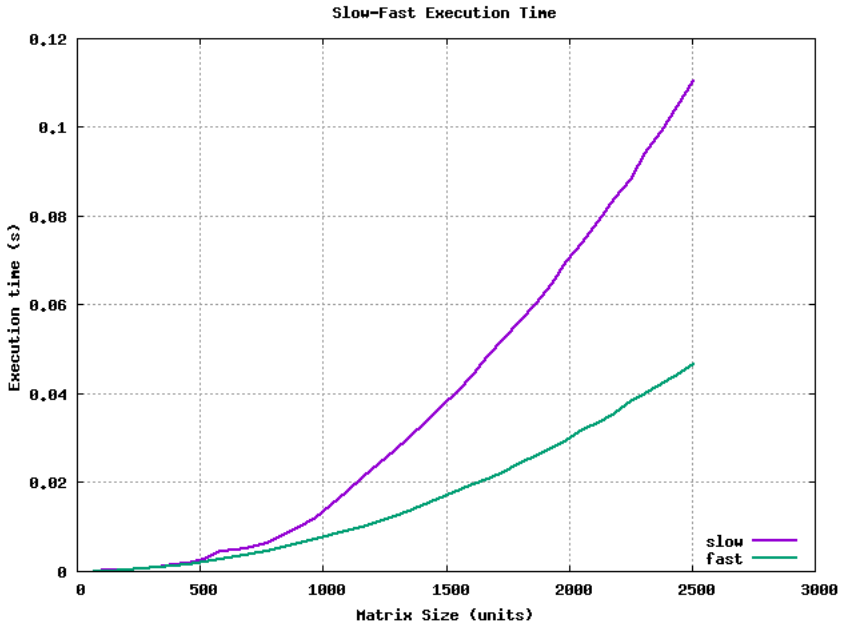

Figura 23: Gráfica de la ejecución del ejercicio 1 en el clúster.

Comparando esta gráfica con la presentada en el ejercicio 1 se puede ver que los tiempos son ligeramente menores en nuestra máquina que en la del clúster. La gráfica del slow y del fast muestran los mismos crecimientos y diferencias, entre ambas gráficas, que los descritos en el ejercicio 1.