Memoria Práctica 3

Arquitectura de Ordenadores

Lucía Asencio y Juan Riera, Grupo 1312

I. Ejercicio 0

Hemos usado para ayudarnos los comandos dmidecode y getconf. Al ejecutar el comando “getconf -a | grep -i cache”, hemos obtenido esta salida:

LEVEL1_ICACHE_SIZE	32768
LEVEL1_ICACHE_ASSOC	8
LEVEL1_ICACHE_LINESIZE	64
LEVEL1_DCACHE_SIZE	32768
LEVEL1_DCACHE_ASSOC	8
LEVEL1_DCACHE_LINESIZE	64
LEVEL2_CACHE_SIZE	262144
LEVEL2_CACHE_ASSOC	4
LEVEL2_CACHE_LINESIZE	64
LEVEL3_CACHE_SIZE	8388608
LEVEL3_CACHE_ASSOC	16
LEVEL3_CACHE_LINESIZE	64
LEVEL4_CACHE_SIZE	0
LEVEL4_CACHE_ASSOC	0
LEVEL4_CACHE_LINESIZE	0

De aquí, y de dmidecode, extraemos la siguiente información:

1. La caché de nivel 1 es asociativa de 8 vías, y es de 256 kBytes. Además, separa caché de instrucciones de caché de datos.
2. La caché de nivel 2 es asociativa de 4 vías, y es de 1024 kBytes. No separa caché de datos y de instrucciones.
3. La caché de nivel 3 es asociativa de 16 vías, y su tamaño es 8192 kBytes. Tampoco distingue caché de datos y de instrucciones.

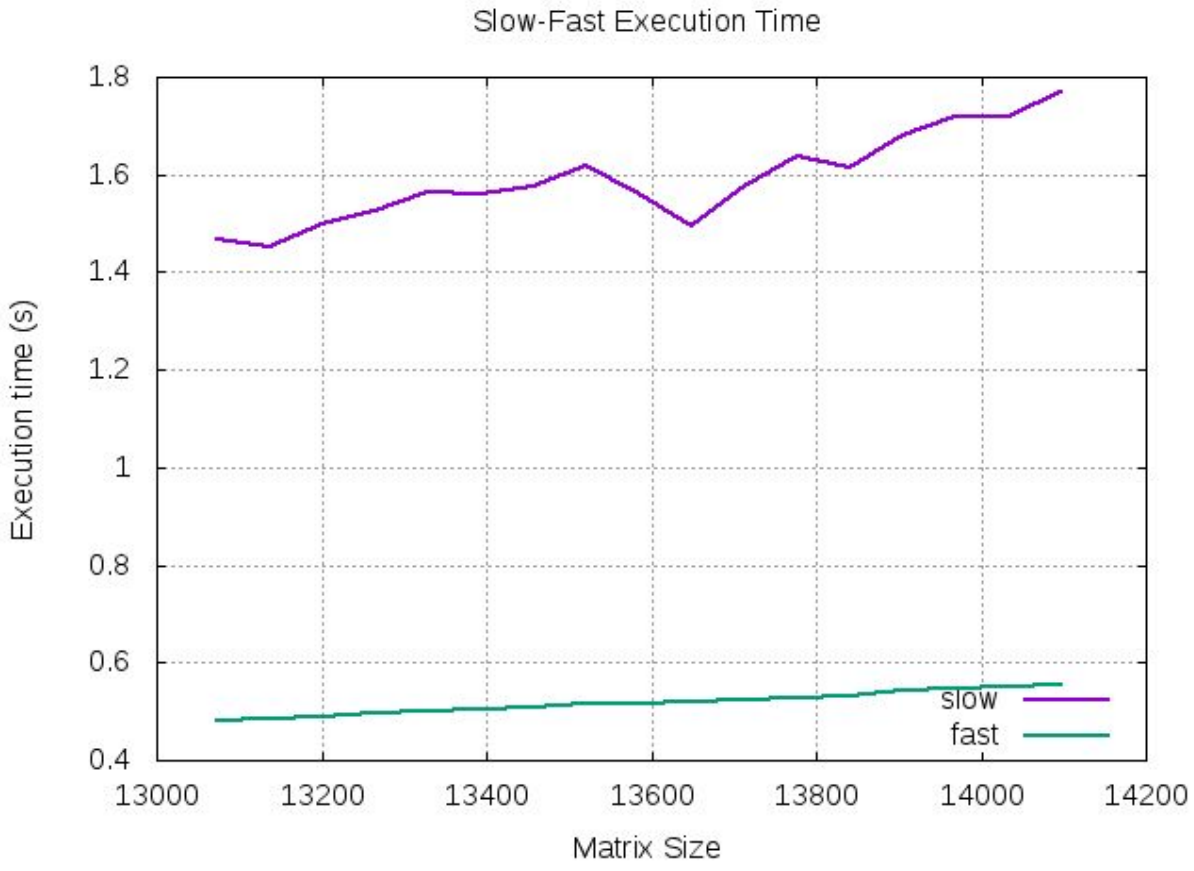
Estos son todos los niveles.

II. Ejercicio 1

A continuación, mostramos la gráfica obtenida con la ejecución del ejercicio.

Aunque teníamos P = 8, hemos usado P = 3 por la cantidad de tiempo consumido por el programa.

La razón



Para obtener los datos con los que ha sido dibujada la gráfica, hemos usado un script que, para cada tamaño de matriz, ejecutaba los programas slow y fast 20 veces y hacía la media de todos los tiempos conseguidos. La razón de que haya que ejecutar muchas veces el programa con cada tamaño es para evitar “picos” en las gráficas que nos den una información falseada de los tiempos de ejecución.

Cuantas más repeticiones, mayor será la muestra obtenida y el valor medio de la muestra estará más cerca de la media real, que es la que queremos estudiar.

Nuestra manera de intercalar ./slow, ./fast y los tamaños de matrices, ha sido la siguiente. Sean los 16 tamaños que teníamos que probar n1, n2, …, n16, ordenados de menor a mayor. Nosotros ejecutabamos:

slow n1	fast n15
slow n2	fast n16
slow n3	fast n1
slow n4	fast n2
…	
slow n16	fast n14

De esta manera, no sólo intercalábamos los dos programas, sino que además no operábamos dos veces seguidas con la misma matriz. La secuencia de arriba se repetía 20 veces.

Como es de esperar, en la gráfica que observa un aumento del tiempo de ejecución a medida que aumenta el tamaño de la matriz. El incremento de tiempo parece más o menos lineal en ambos programas. Pensamos que (una vez deducido que la matriz se almacena por filas) tiene sentido que la relación entre el tiempo de ejecución de fast y el tamaño de la matriz sea lineal, ya sólo se cargan en memoria n filas. Sin embargo, teniendo en cuenta que slow se ve obligado a cargar n^2 filas, esperábamos encontrar en la gráfica una curva más pronunciada. Esto puede deberse a varios factores: existe la posibilidad de que, efectivamente, la gráfica de slow sea una parábola, pero que esté demasiado “achatada” como para ser apreciado en la imagen. También es posible que (¿a lo mejor por la compilación con la bandera -O3?) el programa dé menos fallos de caché de los que suponemos.

Además, el programa fast se ejecuta en un tiempo considerablemente más bajo que el programa slow, ya que el primero es aproximadamente el doble de rápido que el segundo.

Debido a que el programa fast accede por filas (hace el $\sum_{i=1}^{n} \sum_{j=1}^{n} matrix[i][j]$, y slow por columnas (hace el $\sum_{j=1}^{n} \sum_{i=1}^{n} matrix[i][j]$), y fast se ejecuta considerablemente más rápido que slow, podemos concluir que la matriz se guarda en memoria por filas.

Por otro lado, aunque sólo se aprecia en las centésimas de segundo, es cierto que la diferencia del tiempo de ejecución de slow y fast es mayor cuanto mayor es el tamaño de la matriz. (Para comprobación, ejecútese `awk 'BEGIN{}{print $2 " " $3 " Diferencia de tiempos: " $2-$3}END{}' slow_fast_time.dat`).

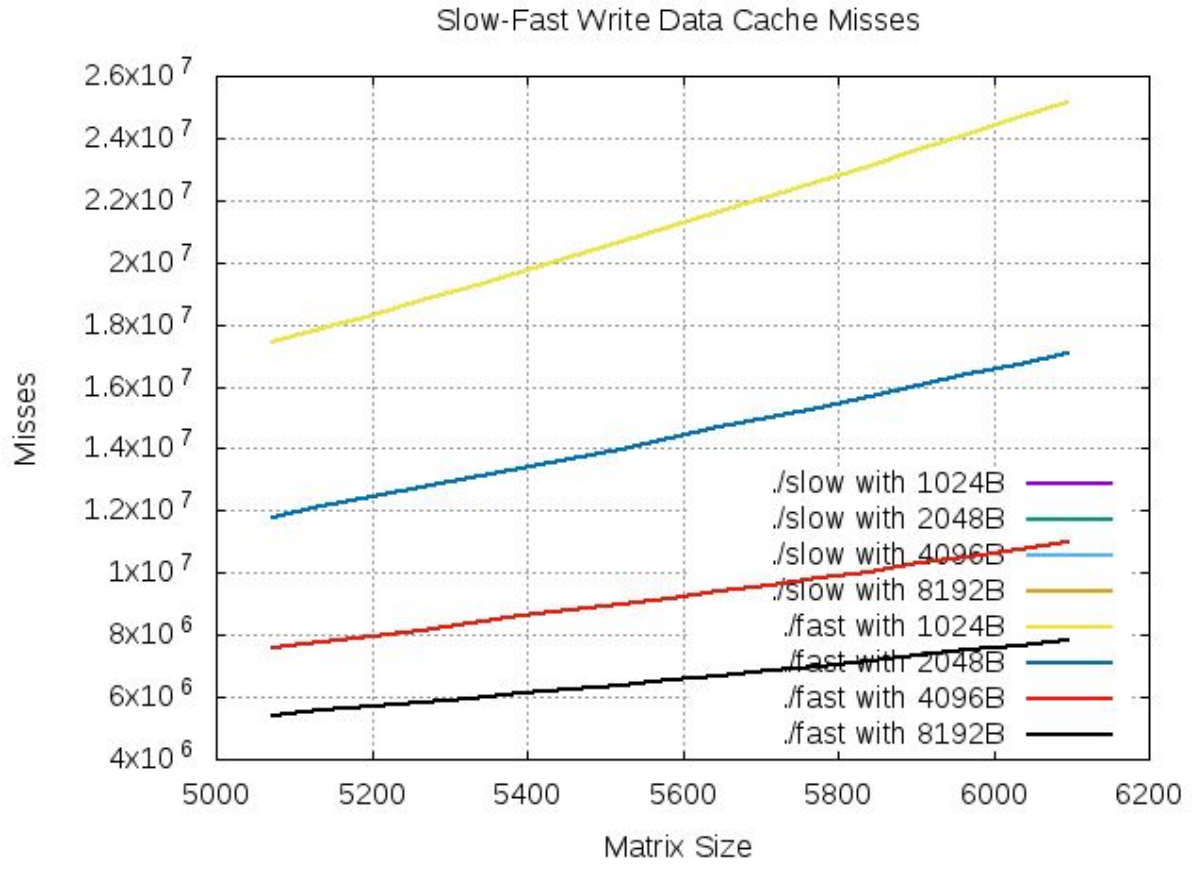
Sabiendo que la matriz se guarda por filas, podemos justificar este hecho de la siguiente manera: el programa fast únicamente cargará n filas en memoria, y por

cada carga añadirá n sumandos a la variable sum, y el programa slow cargará n^2 filas, y por cada carga añadirá sólo un sumando a la variable sum.

Por tanto, la diferencia de tiempo de ejecución entre slow y fast tendrá mucha relación con la función $n^2 - n$, que es creciente. Esta es la razón del aumento del tiempo.

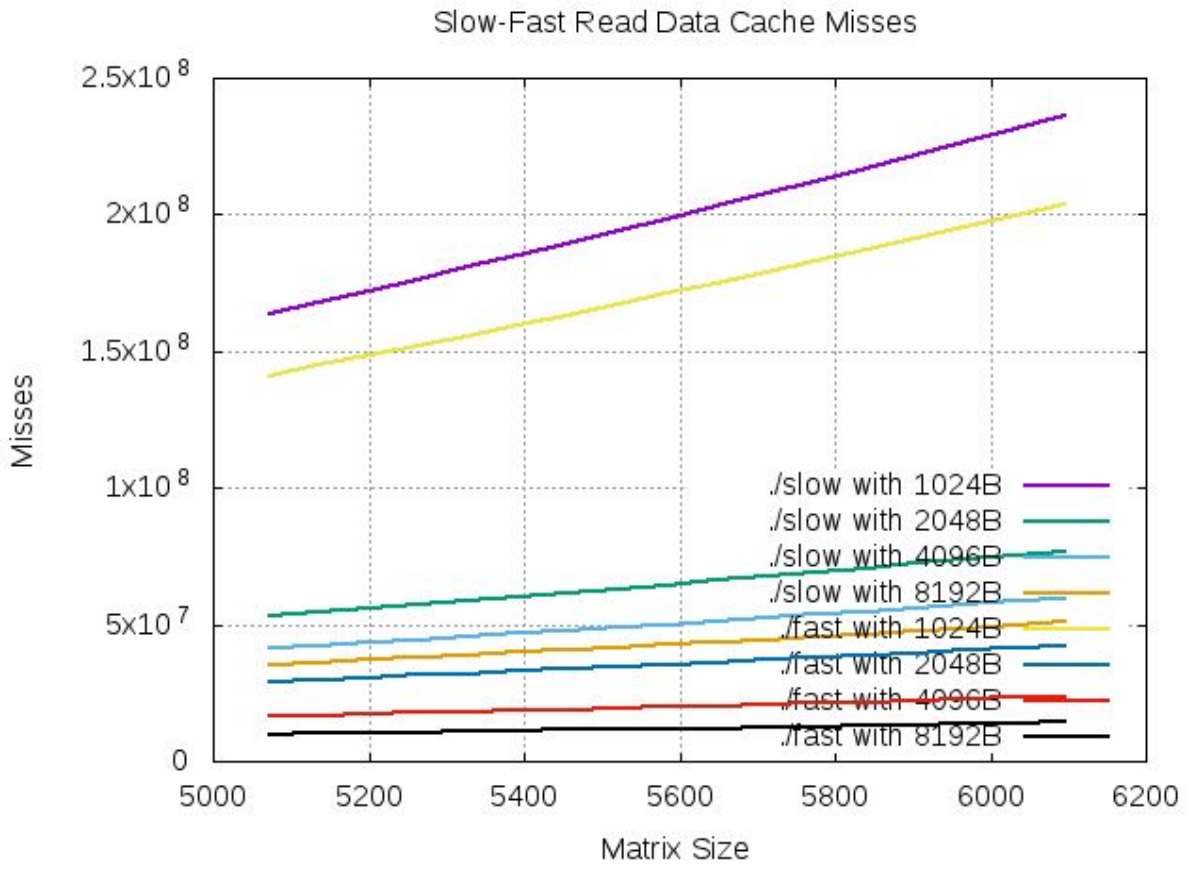
III. Ejercicio 2

Al igual que en el ejercicio anterior, hemos usado P = 3 por la elevada cantidad de tiempo que habría requerido el programa para ejecutarse en caso de haber utilizado P = 8, el correspondiente a nosotros. A continuación tenemos una gráfica con los fallos de caché de lectura para los programas slow y fast, con distintos tamaños de caché tal y como se indica en la leyenda de la gráfica.



Como podemos apreciar, tal y como cabría esperar, las gráficas ascienden con el tamaño de la matriz de forma lineal, ya que estos fallos solo se producen al escribir las matrices, cosa que solo se hace al crear cada matriz y guardar cada valor, es decir, una vez por cada dato almacenado. El número de éstos asciende linealmente de forma proporcional al tamaño de la matriz. Además dado que slow y fast utilizan la misma matriz para un mismo N, coinciden en cuanto a fallos de caché de escritura, como también se aprecia con total claridad en la gráfica. También

observamos sin sorpresa ninguna como para mayores tamaños de caché el número de fallos se reduce.

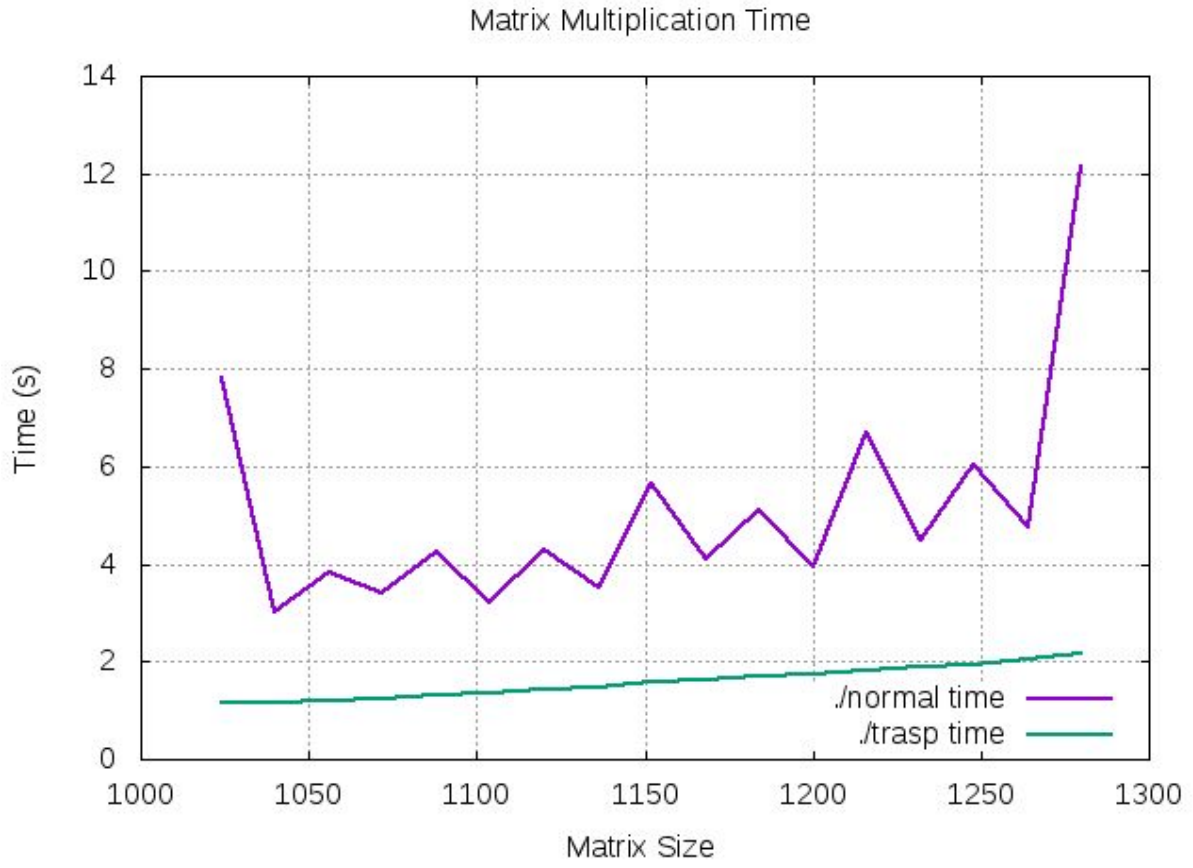


En cuanto a la gráfica de fallos de caché de lectura, tenemos un reparto similar al de la gráfica anterior: gráficas separadas para las ejecuciones de slow y fast para cada tamaño de caché.

Nuevamente, por razones similares, apreciamos con total claridad un incremento lineal de los fallos de caché proporcionalmente al tamaño de la matriz, así como un incremento de los mismos al reducir el tamaño de la caché. Por otro lado, vemos que la diferencia entre los fallos producidos por los programas slow y fast es bastante notable, siendo slow claramente más lento. Esta diferencia se acentúa aún más para tamaños de caché bajos, ya que la diferencia entre slow y fast, tal y como se indica en el enunciado, reside en la eficiencia en términos del uso de la caché. Cuanto más grande es la caché, menos importancia tiene el buen o mal uso que se le dé (aunque se aprecia como siempre tiene algo de importancia) y por eso la mayor diferenciación que mencionábamos.

En cuanto a nuestra implementación, no hemos hecho repeticiones de la ejecución, ya que la herramienta cachegrind es determinista, es decir, para unos mismos datos de entrada, siempre da la misma salida, a diferencia de los tiempos, que se pueden ver influidos por otros factores ajenos a nuestra ejecución. Tampoco hemos alternado tamaños de matrices, ya que cachegrind simula cachés que empiezan estando vacías, para irse llenando más tarde. Cada ejecución de valgrind guardaba los resultados en un par de ficheros slowTmp.dat y fastTmp.dat, dependiendo de si era la ejecución de slow o fast. Más tarde interpretábamos los resultados con la herramienta cg_annotate y parseábamos la salida con sed, cut, head y tail. Finalmente guardábamos los resultados en un fichero cache_NBytes.dat donde NBytes es el número de bytes de la caché en esa ejecución, y ejecutábamos gnuplot para generar las gráficas correspondientes. Todo esto se encuentra en el fichero ej2.sh.

IV. Ejercicio 3

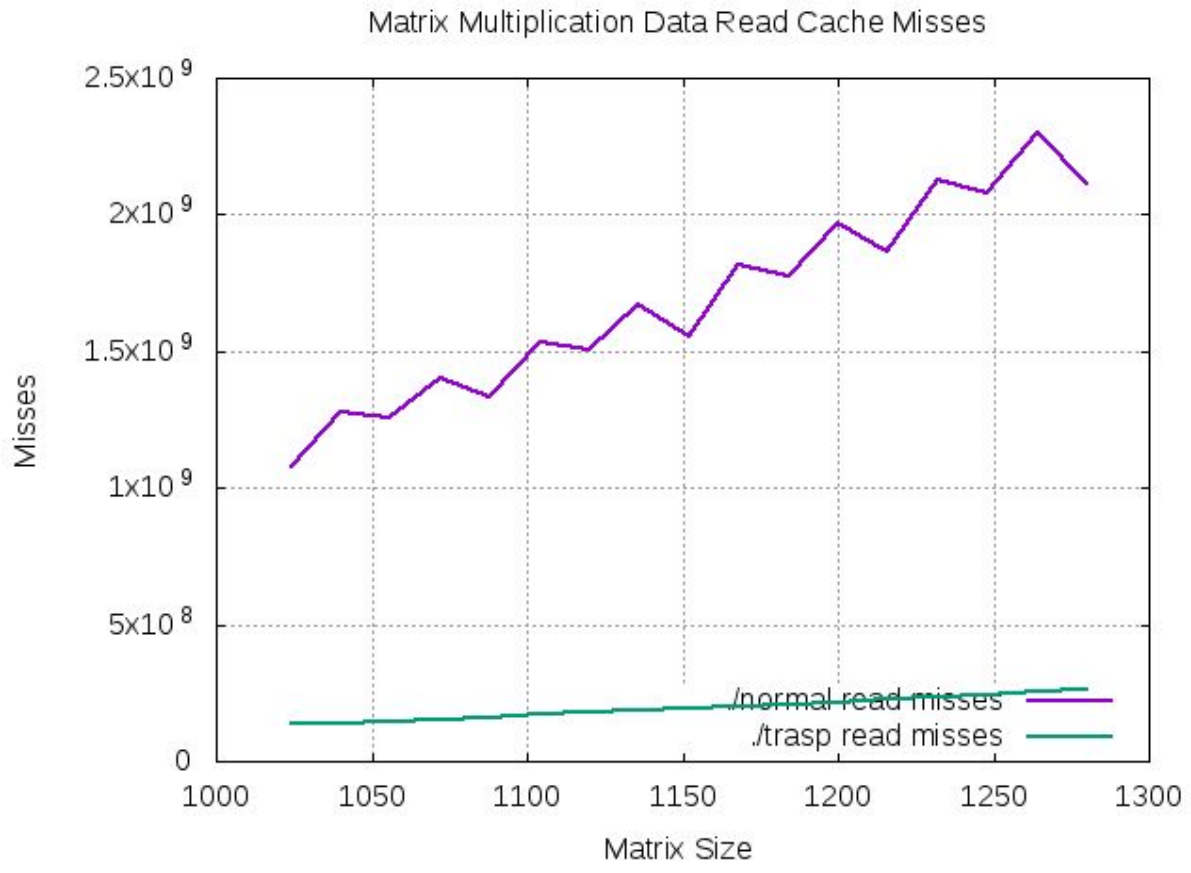


Tenemos en lila la gráfica de tiempos en segundos de la multiplicación de matrices, y en verde los tiempos en segundos de la multiplicación de matrices con el algoritmo que utiliza la matriz traspuesta. Como vemos, tal y como cabría esperar,

ambas ascienden en media de forma lineal a medida que aumenta el tamaño de la matriz, sin embargo, apreciamos que la ejecución del algoritmo de multiplicación “normal” está contaminada por una sucesión de subidas y bajadas periódicas que se deben probablemente al peor uso que se hace de la caché en este algoritmo.

A la hora de comparar la ejecución de multiplicación normal y traspuesta, queda claro que el algoritmo que multiplica vectores fila es mucho mejor que el que multiplica vectores filas por columna. De hecho, el algoritmo traspuesta es más o menos, según se observa en la gráfica, entre 2 y 3 veces mayor que el algoritmo normal. De nuevo, esto es por la manera que tiene la matriz de almacenarse en memoria y, por tanto, la manera en que se carga la misma en caché.
Como la matriz se almacena por filas, en el caso de la multiplicación normal multiplicar fila por columna supone extraer elementos de n+1 filas, mientras que multiplicar filas por filas sólo supone extraer elementos de 2 filas.

A continuación, presentamos la gráfica obtenida representando el número de fallos de lectura en caché en función del tamaño de la matriz.

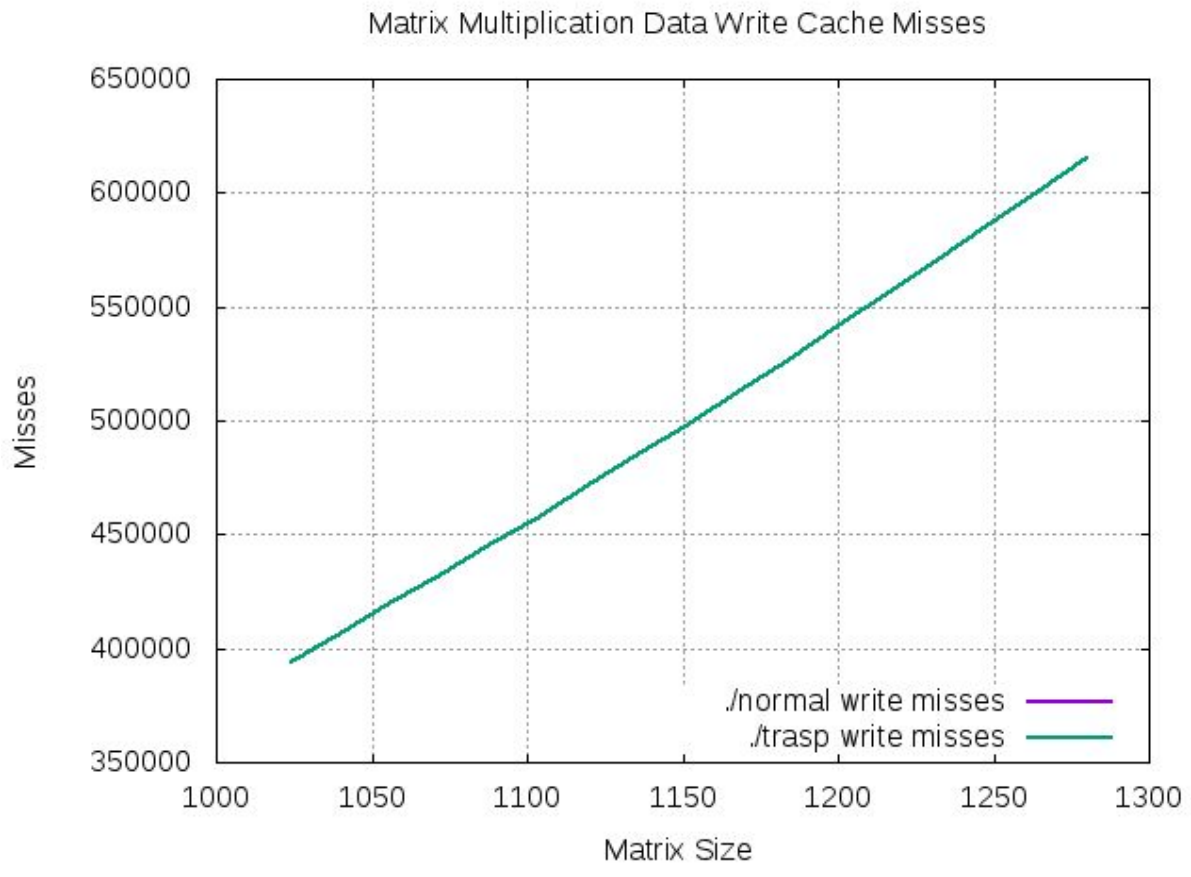


Apreciamos una subida de ambas gráficas proporcional al tamaño de la matriz y lineal en media. Vemos con total claridad la mayor eficiencia de la

multiplicación de la matriz con el algoritmo de la traspuesta. También vemos picos periódicos en la ejecución del algoritmo normal debidos al peor uso de la caché.

Comparando con el gráfico de los fallos de lectura, hemos observado que los mínimos de una función corresponden con los máximos de la otra, y viceversa. Esto nos desconcertó por ser lo contrario a lo que podíamos esperar, ya que creíamos que, a mayor número de fallos de caché, mayor sería el tiempo de ejecución. Una posible causa de esta discordancia podría ser la prioridad que se da a determinado proceso. Pongamos que hay un proceso que se ejecuta en el ordenador cada x tiempo, con una prioridad mayor que la de nuestro programa. Pongamos que en ese momento estamos multiplicando una matriz de tamaño n_1, y se estén dandose f_1 fallos de página. Aunque f_1 sea bajo, el tiempo de ejecución podría ser alto si nuestro programa se ve interrumpido por la ejecución del que tiene mayor prioridad. Después, podría multiplicarse la matriz de tamaño n_2 (con f_2 fallos de lectura en caché) y que, a pesar de ser f_2 mayor que f_1, el tiempo de ejecución fuera menor por no estar ya ejecutándose el proceso de mayor prioridad.

A continuación, la gráfica de los fallos de caché de escritura en función del tamaño de la matriz.



Apreciamos la proporcionalidad lineal perfecta entre el tamaño de la matriz y la cantidad de fallos de caché. Esto de debe a que las únicas escrituras que se realizan a lo largo de la ejecución son las correspondientes a la creación de las matrices, y la escritura de los resultados, que no dan lugar a grandes irregularidades en los fallos de caché de escritura al realizarse por filas. Además vemos que para ambos algoritmos las gráficas coinciden, lo cual tiene sentido, ya que para un mismo tamaño de matriz, las matrices a multiplicar y la matriz resultado son las mismas y del mismo tamaño.