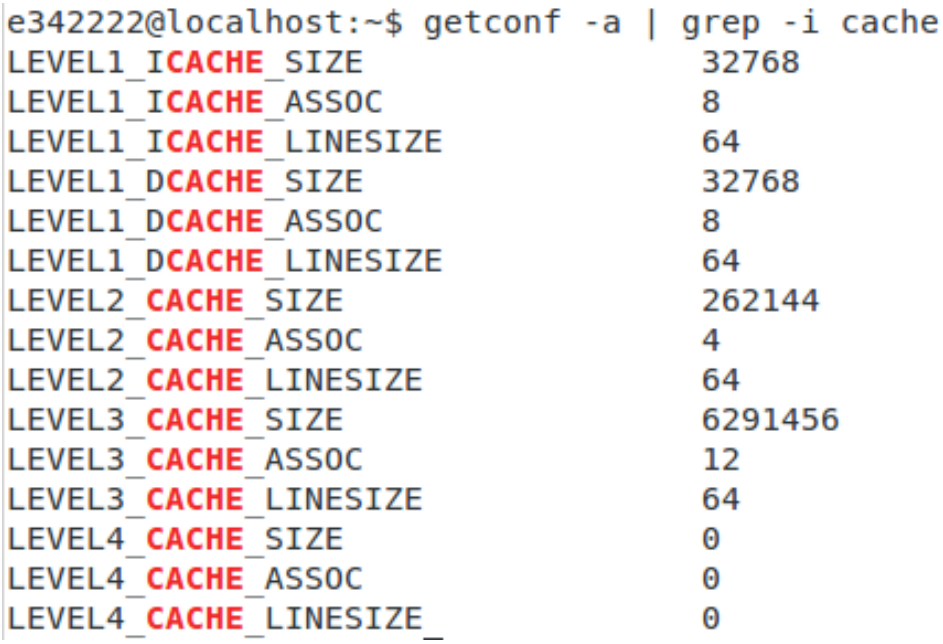
Memoria de la práctica 3 de

Arquitectura de ordenadores

Alejandro Cabana Suárez y Lucía Colmenarejo Pérez. Pareja 38. Grupo 1302

EJERCICIO 0: Información sobre la caché del sistema:

Adjuntamos una imagen del resultado obtenido por terminal al ejecutar el comando mostrado:



Como podemos ver en los ordenadores de los laboratorios existen:

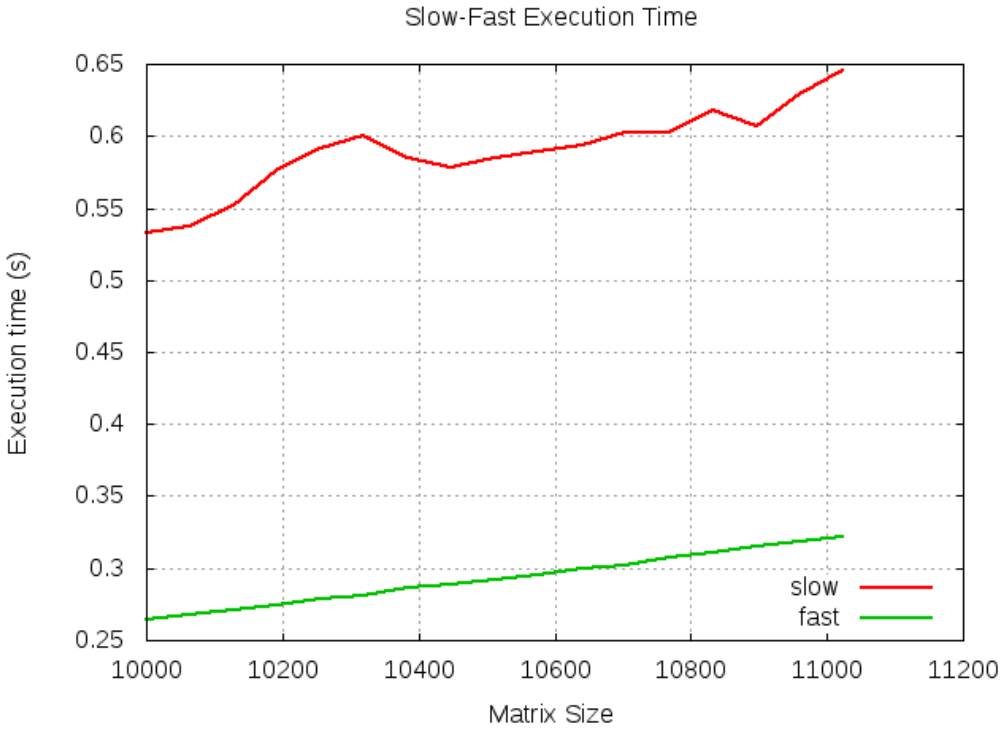
- Una caché de instrucciones de nivel 1 de 32768 bytes (32 KB) asociativa de 8 vías con bloques de 64 bytes.
- Una caché de datos de nivel 1 de 32768 bytes (32 KB) asociativa de 8 vías con bloques de 64 bytes.
- Una caché de nivel 2 de 262144 bytes (256 KB) asociativa de 4 vías con bloques de 64 bytes.
- Una caché de nivel 3 de 6291456 bytes (6 MB) asociativa de 12 vías con bloques de 64 bytes.

La distinción entre instrucciones y datos es a nivel 1.

EJERCICIO 1: Memoria caché y rendimiento:

Hay que realizar múltiples veces la toma de medidas para cada programa y para cada tamaño de matriz para minimizar los posibles efectos que produzca el procesador cuando cede el tiempo de ejecución a otros procesos. Por nuestra manera de medir el tiempo, estaríamos contando el tiempo que consumen los procesos que interrumpan el nuestro. Al tomar varias medidas y calcular la media no eliminamos del todo estos efectos, pero consideramos el tiempo promedio que este efecto nos va a hacer perder. De esta manera, le estaríamos sumando al tiempo real de ejecución una penalización constante que simplemente desplazaría la gráfica hacia arriba.

Aunque a primera vista esta solución parezca efectiva, esto no es así. Con el tamaño de las matrices crece el tiempo de ejecución que necesita nuestro programa y, por tanto, también lo hacen las oportunidades para que el procesador interrumpa su ejecución en favor de otros procesos. Esto quiere decir que al tiempo real que le dedicamos a cada tamaño de matriz le estaremos sumando una penalización cada vez mayor (no constante como pensábamos), de tal manera que contribuimos a que las matrices grandes recojan tiempos mayores que las pequeñas (más allá de que realmente tarden más por tener más elementos).



Alejandro Cabana Suárez y Lucía Colmenarejo Pérez. Pareja 38. Grupo 1302

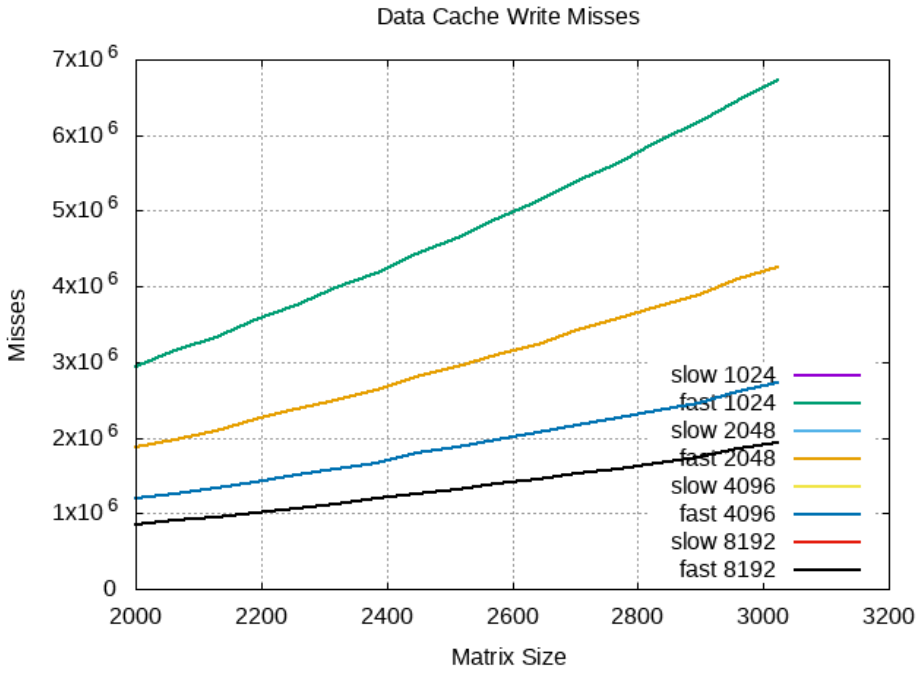
Como esperábamos (y como los nombres sugieren) el programa *slow* consume más tiempo que *fast,* aproximadamente el doble. En nuestra gráfica no se observa el efecto que menciona el enunciado de que para tamaños pequeños los tiempos de los programas estén próximos. Creemos que esto es porque el rango de tamaños que estamos analizando es demasiado pequeño para que se aprecie, pero en cualquier caso, el efecto que mencionábamos antes causado por nuestra manera de medir los tiempos podría tener que ver con esto. Otra causa es que disminuyendo los tamaños de las matrices, los tiempos se reducen a una velocidad muy elevada, por lo que las gráficas de los dos programas parecen acercarse cuando en realidad en proporción están igual de alejadas.

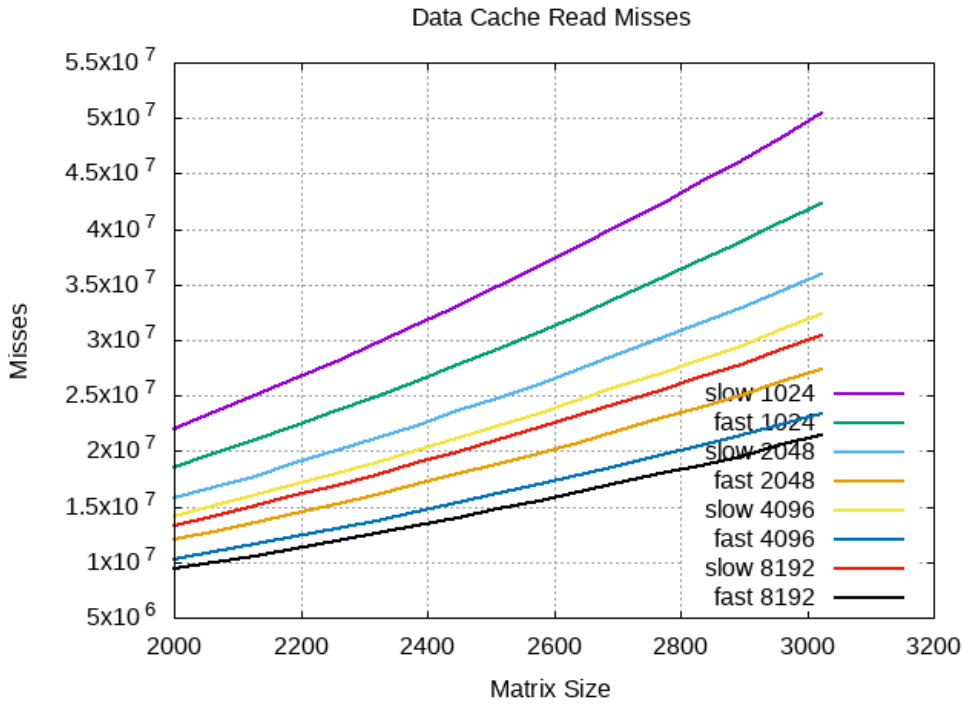
Los datos para esta gráfica los hemos obtenido ejecutando los programas *slow* y *fast* 10 veces con cada tamaño de matriz entre 10000 y 11024, ya que nuestro número P resulta ser 0. Para automatizar este proceso, hemos elaborado el script *slow_fast_time.sh*, que a parte de tomar los datos, genera la imagen de la gráfica en png con el nombre *slow_fast_time.png*.

Si consideramos que *matriz[i][j]* accede al elemento en la i-ésima fila y j-ésima columna de una matriz, entonces el programa que se nos ha entregado guarda los elementos de la misma fila consecutivos en memoria.

EJERCICIO 2: Tamaño de la caché y rendimiento:

Adjuntamos las gráficas que representan los fallos de lectura y los fallos de escritura de datos.





Como podemos observar en las gráficas, cuanto mayor es el tamaño de la caché, menor es el número de fallos (tanto para *slow* como para *fast*), ya que caben más bloques simultaneamente en la caché, y hay que reemplazarlos menos.

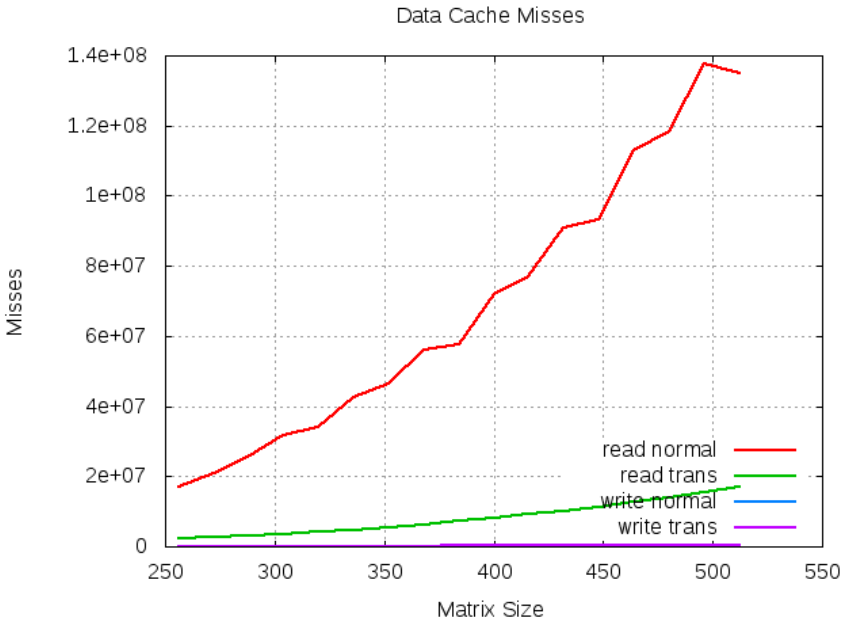
Fijando un tamaño de caché, el número de fallos (tanto de lectura como de escritura, en slow y en fast) aumenta con el tamaño de las matrices. Además, como era de esperar, para el mismo tamaño de caché, el programa *slow* produce más fallos de lectura que *fast*, ya que cuando el programa *fast* (que recorre las matrices por filas) carga un bloque en caché, los próximos datos que pida van a estar en ese mismo bloque y no va a tener que cargar otro hasta que lo haya leído entero. *Slow*, por el contrario, tiene que ir a un bloque distinto por cada dato que lee de la misma columna. Como hemos elegido un tamaño de bloque de 64B, en el mejor de los casos (con caché de 8192B) podremos tener cargados 128 bloques a la vez. Cada elemento de la matriz, como poco ocupa 4B (si es float). Por tanto cada bloque contiene a lo sumo 16 elementos de la matriz. Esto quiere decir que si cuando empieza el programa, se producen unos primeros 128 fallos y se cargan los bloques correspondientes a cada uno (sólo llevaríamos leídos 128 elementos de la matriz), el elemento siguiente que leamos estará en un bloque distinto de esos 128, y tendremos que borrar uno de ellos para cargarlo en la caché.

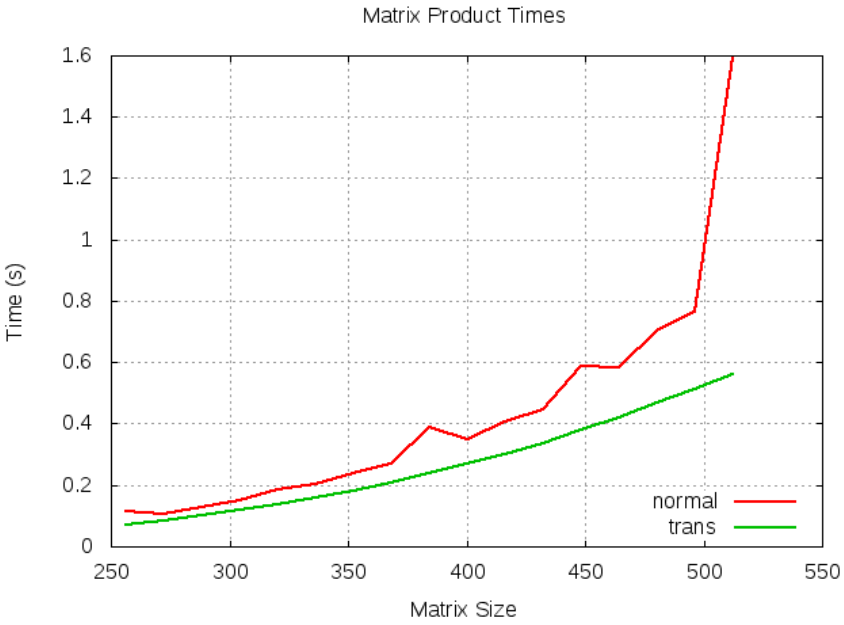
Lo que queremos demostrar con todo esto, es que por cada elemento de una columna que recorra *slow*, siempre se produce un fallo. En cambio, *fast* sólo produce un fallo cada vez que lee todos los elementos que caben en un bloque (en nuestro ejemplo anterior, produciría un fallo por cada 128 elementos leídos).

Si miramos la gráfica de escritura, vemos que los fallos disminuyen con el tamaño de la caché, pero para un mismo tamaño *slow* y *fast* dan los mismos fallos. Esto tiene sentido, ya que los dos escriben el mismo número de veces (una por cada elemento de la matriz).

EJERCICIO 3: Caché y multiplicación de matrices:

Adjuntamos las gráficas que representan los fallos de caché y los tiempos de ejecución.





Mirando a la gráfica de fallos de caché, podemos ver que se dan los mismos efectos que en el apartado anterior pero a una escala bastante superior, ya que multiplicar matrices implica muchos más accesos a memoria que sumar sus elementos (que solo requiere uno por cada elemento en la matriz). De nuevo, los fallos de escritura son los mismos para las dos versiones porque se escribe el mismo número de veces en las dos: una por cada elemento de la matriz resultado. La multiplicación traspuesta parece ser un éxito: produce menos fallos para el tamaño de matriz más grande de los que produce la multiplicación normal para las matrices más pequeñas. Los tiempos reflejan lo mismo: tarda mucho menos la multiplicación traspuesta que la normal, y se empieza a notar de manera más drástica la diferencia a partir del tamaño 500.

La razón por la que la multiplicación traspuesta es tan rápida es está relacionada con lo que comentábamos antes sobre *slow* y *fast*. La multiplicación de matrices normal recorre una matriz por filas y otra por columnas, y va sumando los productos de los elementos. Como hemos visto, recorrer por filas es eficiente porque estamos guardando las filas de manera consecutiva en memoria; pero recorrer por columnas no lo es. Al hacer la traspuesta de la segunda matriz, estamos eliminando la necesidad de recorrerla por columnas. Así, al recorrer ambas matrices por filas, se reduce drásticamente el tiempo necesario y los fallos de caché que se producen.

SUGERENCIAS PARA MEJORAR EL ESTUDIO:

- El generador de números aleatorios que utiliza ***arqo3.c*** siempre usa la misma semilla (0). Esto quiere decir que cada vez que se ejecuta el programa, se generan los mismos números en el mismo orden. Si cambiamos esta semilla en cada ejecución del programa, cada vez se generarían matrices distintas, y ya no sería tan necesario alternar las ejecuciones de *slow* y *fast* e ir cambiando los tamaños, ya que la caché siempre almacenaría datos distintos. El único aspecto positivo que puede tener usar siempre la misma semilla es que para un mismo tamaño, *slow* y *fast* se ejecutan sobre la misma matriz, y la comparación está más justificada. Aun así, hay otras maneras de conseguir este efecto.

- Para medir el tiempo de ejecución tomamos una medida del tiempo antes de empezar las operaciones y otra después, y la diferencia es lo que utilizamos para nuestro estudio. El problema de esta manera de medir es que contamos el tiempo real desde que comienza la ejecución hasta que termina, sin tener en cuenta que no todo ese tiempo de ejecución ha estado dedicado exclusivamente a este programa. Cada vez que el planificador haya cedido cierto tiempo de ejecución a otro proceso (interrumpiendo el nuestro), habremos añadido ese tiempo a nuestra medida. Utilizando el tipo *clock_t* de la biblioteca *time.h*, podríamos medir los ciclos de reloj dedicados exclusivamente a nuestro proceso, y esto nos ahorraría tener que repetir pruebas para obtener datos fiables.