Memoria Arquitectura de Ordenadores (Práctica 3)

Miguel Arconada Manteca
miguel.arconada@estudiante.uam.es

Alberto González Klein
alberto.gonzalezk@estudiante.uam.es

21 de Noviembre 2018

Introducción

Antes de resolver las cuestiones relacionadas con los ejercicios, vamos a explicar la organización de las carpetas, así como los scripts realizados. Tenemos una carpeta para cada ejercicio y una carpeta **src**, además de los scripts *prepararTodo.sh* y *ejecutarTodo.sh*. El script *prepararTodo.sh*, como su nombre indica, prepara todos los ejercicios para su ejecución, es decir, hace *make clean* para limpiar los viejos ejecutables, *make* para generar nuevos y da permisos de ejecución (con *chmod +x*) a los scripts de todos los ejercicios. Por otro lado, *ejecutarTodo.sh* ejecuta todos los ejercicios en orden (del 1 al 4). Por tanto, si queremos ejecutar la práctica adecuadamente y de forma sencilla, llamaremos primero a *prepararTodo.sh* y posteriormente a *ejecutarTodo.sh*.

La carpeta **src** contiene el makefile, todos los .c y .h (carpeta **c**), los ejecutables de estos .c (carpeta **exes**) y el script de *Gnuplot* que hemos creado, *plotScript.sh* (carpeta **scripts**). Este script funciona con los siguientes argumentos:

- Argumentos obligatorios:
  - $-f$: Nombres de los ficheros de donde se extraen los datos.
  - $-o$: Columna del fichero que es la variable independiente.
  - $-d$: Columnas del fichero que son las variables dependientes.
  - $-p$: Nombre del archivo donde se guarda la gráfica.
- Argumentos opcionales:
  - $-x$: Nombre del eje X.
  - $-y$: Nombre del eje Y.
  - $-t$: Título de la gráfica.
  - $-s$: Estilo de la gráfica. Puesto con *lines* por defecto.
  - $-k$: Posición de la leyenda. Puesto en left top por defecto.
  - $-w$: Anchura de línea. Puesto a 2 por defecto
  - $-l$: Etiquetas de los elementos graficados

En cada una de las carpetas de los ejercicios se encuentra el script que realiza la funcionalidad definida en el enunciado de la práctica, así como dos carpetas: **dat** y **png**. En la carpeta **dat** se encuentran los archivos en los que se han volcado los datos pedidos en el ejercicio correspondiente, mientras que en la carpeta **png** se encuentran las gráficas generadas.

Queda remarcar que en los ejercicios 2, 3 y 4 hemos dividido los valores *Ninicio*, *Npaso* y*Nfinal* entre 4, ya que con los valores indicados en el enunciado de la práctica el tiempo de ejecución era demasiado alto.

Ejercicio 0

A partir de la información expuesta, indique en la memoria asociada a la práctica los datos relativos a las memorias caché presentes en los equipos del laboratorio. Se ha de resaltar en qué niveles de caché hay separación entre las cachés de datos e instrucciones, así como identificarlas con alguno de los tipos de memoria caché vistos en la teoría de la asignatura.

En nuestro caso, la práctica ha sido ejecutada en el ordenador personal de uno de los integrantes de la pareja. Por ello, los datos a continuación son de dicho ordenador:

Primero, al ejecutar el comando *$cat /proc/cpuinfo* obtenemos lo siguiente:

```
$ cat /proc/cpuinfo

processor : 0
[...]
cache size : 4096 KB
[...]
cache_alignment : 64
address sizes : 39 bits physical, 48 bits virtual

processor : 1
[...]
cache size : 4096 KB
[...]
cache_alignment : 64
address sizes : 39 bits physical, 48 bits virtual

processor : 2
[...]
cache size : 4096 KB
[...]
cache_alignment : 64
address sizes : 39 bits physical, 48 bits virtual

processor : 3
[...]
cache size : 4096 KB
[...]
cache_alignment : 64
address sizes : 39 bits physical, 48 bits virtual
[...]
```

De lo anterior podemos deducir que el equipo en cuestión tiene cuatro núcleos de procesador, cada cual con una cache de 4096 kB, con una palabra de 64 bits. Además, podemos ver que la dirección de memoria física es de 39 bits y la virtual de 48 bits. De lo que no podemos saber nada según este comando, es sobre el número de vías de la caché.

Después, ejecutamos el comando *$ dmidecode* con permisos de administrador, y obtuvimos los siguientes datos sobre la cache:

```
    Handle 0x0005, DMI type 7, 19 bytes
Cache Information
Socket Designation: L1 Cache
Configuration: Enabled, Not Socketed, Level 1
Operational Mode: Write Back
Location: Internal
Installed Size: 128 kB
Maximum Size: 128 kB
Supported SRAM Types:
Synchronous
Installed SRAM Type: Synchronous
Speed: Unknown
Error Correction Type: Parity
System Type: Unified
Associativity: 8-way Set-associative

Handle 0x0006, DMI type 7, 19 bytes
Cache Information
Socket Designation: L2 Cache
Configuration: Enabled, Not Socketed, Level 2
Operational Mode: Write Back
Location: Internal
Installed Size: 512 kB
Maximum Size: 512 kB
Supported SRAM Types:
Synchronous
Installed SRAM Type: Synchronous
Speed: Unknown
Error Correction Type: Single-bit ECC
System Type: Unified
Associativity: 4-way Set-associative

Handle 0x0007, DMI type 7, 19 bytes
Cache Information
Socket Designation: L3 Cache
Configuration: Enabled, Not Socketed, Level 3
```

```
Operational Mode: Write Back
Location: Internal
Installed Size: 4096 kB
Maximum Size: 4096 kB
Supported SRAM Types:
Synchronous
Installed SRAM Type: Synchronous
Speed: Unknown
Error Correction Type: Multi-bit ECC
System Type: Unified
Associativity: 16-way Set-associative
```

De aquí podemos extraer otros datos. Sabemos que hay tres niveles de caché. La primera es una caché de 128 kB, y asociativa de 8 vías. La de segundo nivel ocupa 512 kB y es asociativa de 4 vías. Finalmente, la de nivel tres tiene un tamaño de 4096 kB (4 MB), y es asociativa de 16 vías.

Tras esto, ejecutamos el comando *$ getconf -a — grep -i cache*, que resultó en el siguiente output:

```
    $ getconf -a | grep -i cache
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                  4194304
LEVEL3_CACHE_ASSOC                 16
LEVEL3_CACHE_LINESIZE              64
LEVEL4_CACHE_SIZE                  0
LEVEL4_CACHE_ASSOC                 0
LEVEL4_CACHE_LINESIZE              0
```

De aquí volvemos a ver los tamaños de las cachés de los tres niveles y la asociatividad de las mismas. Obviamente, comprobamos que son los mismos datos que obtuvimos en el apartado anterior, viendo además que la caché de nivel uno en realidad está separada en dos (datos e instrucciones con tamaño de la mitad).

Finalmente, tras ejecutar *$ lstopo*, pudimos ver la siguiente pantalla:

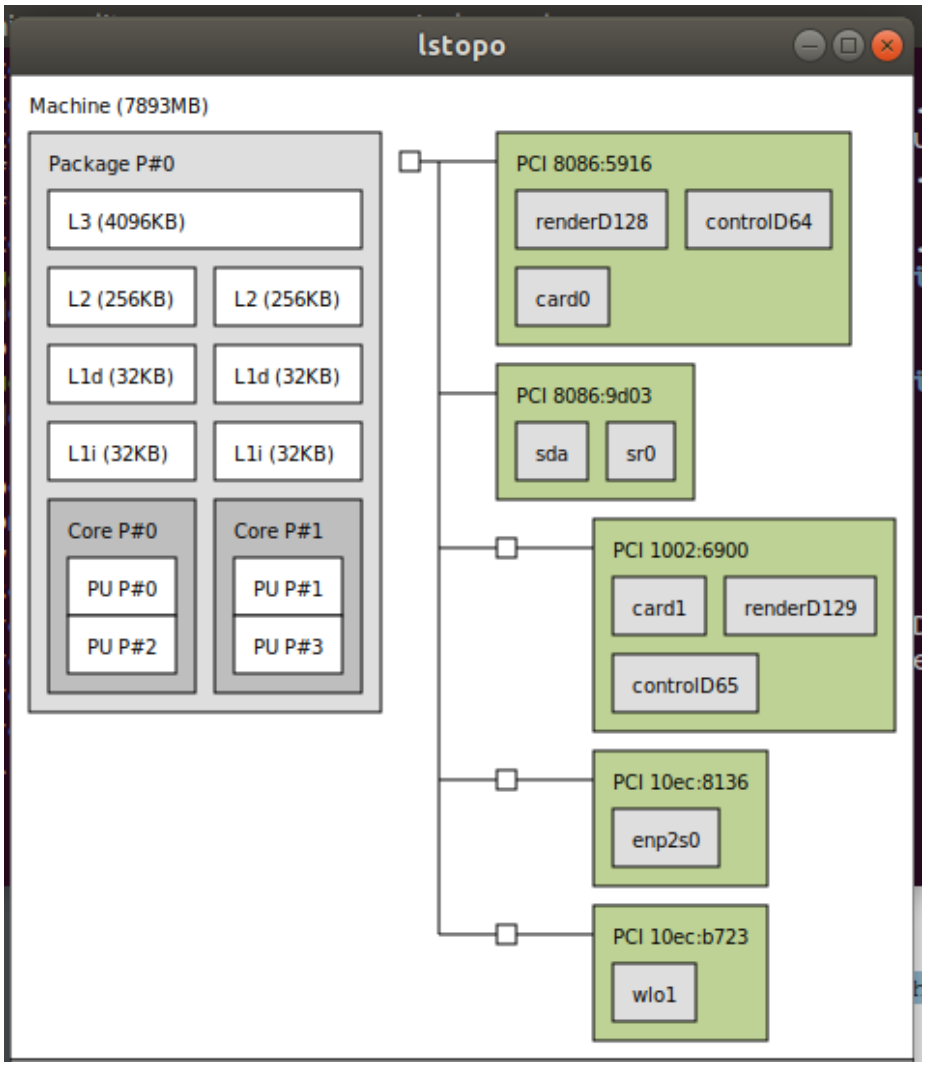

Figura 1: Pantalla de lstopo

De esta imagen nos interesa ver los tamaños de las cachés de los distintos niveles. Una vez más, vemos que son de los tamaños vistos anteriormente, sabiendo además cómose dividen en estas:

- La caché de nivel 3 (4096 kB) es una completa.
- La caché de nivel 2 (512 kB) está formada por dos de 256 kB.
- La caché de nivel 1 (128 kB) está dividida en cuatro (dos para datos y dos para instrucciones) de 32 kB cada una.

Ejercicio 1

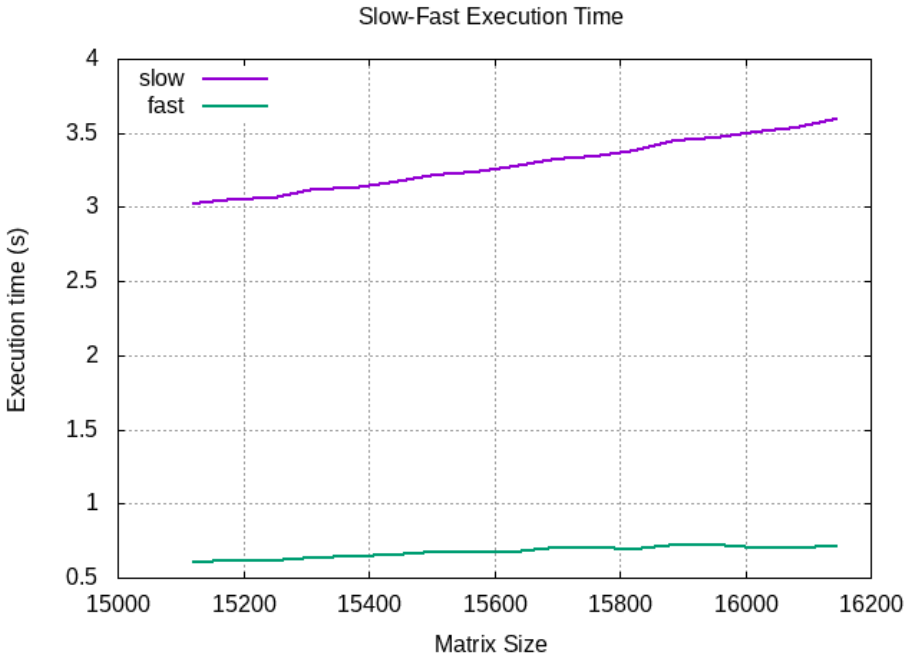

Figura 2: Tiempos de ejecución

Indique en la memoria una explicación razonada del motivo por el que hay que realizar múltiples veces la toma de medidas de rendimiento para cada programa y tamaño de matriz.

Debemos ejecutar múltiples veces para así compensar los posibles picos. Es decir, cuando ejecutamos un programa muchos factores externos al mismo pueden afectar a su ejecución. Esto puede causar que salgan tiempos muy grandes en condiciones muy desfavorables o tiempos muy pequeños en casos mas favorables a la media. Mediante la toma de múltiples valores y el posterior cálculo de su media compensamos estos casos extremos y nos podemos hacer una mejor idea de cuánto es el tiempo de ejecución del programa.

Se pide en la memoria explicar el método seguido para la obtención de los datos, y una justificación del efecto observado.

Para obtener los datos, ejecutamos primero *slow* para todos los tamaños indicados y guardamos los datos obtenidos en un array, de forma que el dato de la matriz de tamaño más pequeño está en la primera posición del array y la más grande en la última posición. Posteriormente, ejecutamos *fast* con el mismo

procedimiento guardando los datos en otro array similar.

Para la siguiente iteración, sumamos cada dato obtenido de cada tamaño con el obtenido en la anterior iteración. Al finalizar todas las iteraciones, dividimos el dato final entre el número de iteraciones para obtener la media de cada tamaño. Para realizar esta división y las sumascon números no enteros, hemos creado un programa auxiliar llamado *opsFloat.c*, el cual recibe como argumentos dos números y los opera de la forma indicada como parámetro de entrada (–s para la suma y –d para la división). Además este programa elimina las comas de millares de los números (es decir, convierte 2,043.047 en 2043.047), lo cual nos resultará útil en los próximos ejercicios, puesto que si no el formato es diferente al que podemos usar. Posteriormente, mediante el uso de *awk* guardamos el valor de la operación y volcamos el valor de los arrays a *slow_fast_time.dat* siguiendo el formato indicado.

En la gráfica observamos que el tiempo de *slow* aumenta notablemente más que el tiempo de *fast* con el aumento de la matriz. Esto tiene sentido, puesto que el programa *slow* accede a elementos de la columna, mientras que *fast* accede a elementos de la fila. Si pensamos en cómo funcionan las matrices en c observamos que accedemos a una fila i a partir de un puntero ($matriz[i]$) y si queremos acceder a todos los elementos de esa fila lo podemos hacer a partir de ese mismo puntero ($matriz[i][j]$). Sin embargo, acceder a una columna es más costoso puesto que se tratan de punteros distintos, es decir, acceder a todos los elementos de una columna supondría tener que acceder a todos los punteros de la matriz ($matriz[0][i]$, $matriz[1][i]$, etc.). Esto supone un gran número de accesos a memoria y un gran aumento del coste.

¿Por qué para matrices pequeñas los tiempos de ejecución de ambas versiones son similares, pero se separan según aumenta el tamaño de la matriz?

La separación de tiempos se debe a lo explicado anteriormente. Por ello, a tamaños pequeños hay tan pocos fallos de caché en ambos casos que no se nota apenas una diferencia en cuanto a tiempo. A medida que aumentan los tamaños y, por tanto, la diferencia de fallos entre *slow* y *fast*, aumenta la diferencia de crecimiento de los tiempos de ejecución.

El número de accesos a memoria de ambos problemas es el mismo (se leen los mismos datos). La diferencia radica en en el orden en que se leen estos, y por lo tanto, los fallos de caché que esto produce.

¿Cómo se guarda la matriz en memoria, por filas (los elementos de una filas están consecutivos) o bien por columnas (los elementos de una columna son consecutivos)?

Como ya hemos indicado, se guardan por filas en memoria y por esta razón es menos costoso acceder por filas que por columnas. Por ello, cuando cargamos en caché un dato que no estaba, como se carga un bloque entero, y no solo un

dato, se cargan los siguientes valores, que no darían fallo de caché al ser leídos.

Ejercicio 2

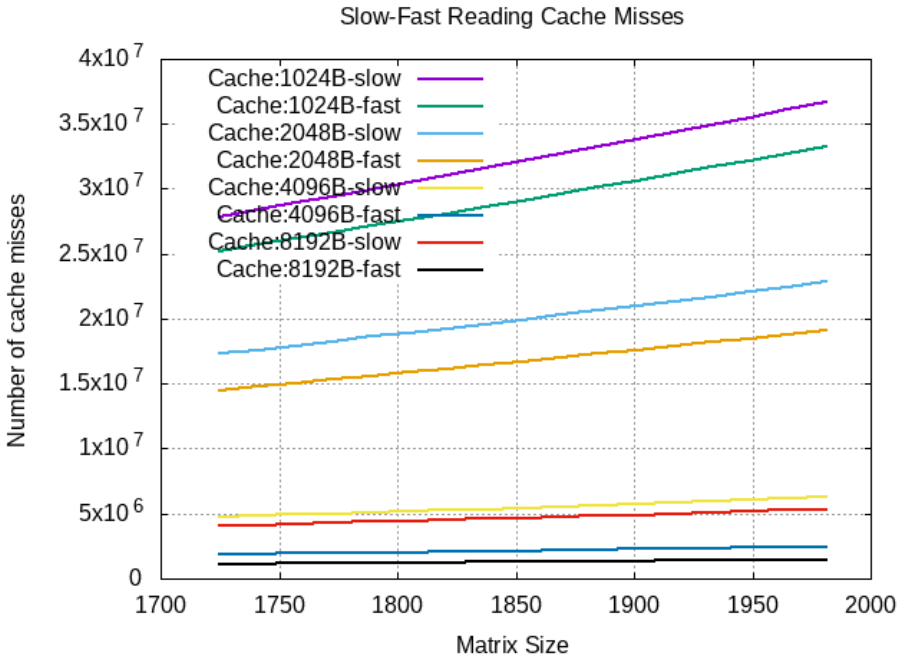

Figura 3: Fallos de lectura de caché

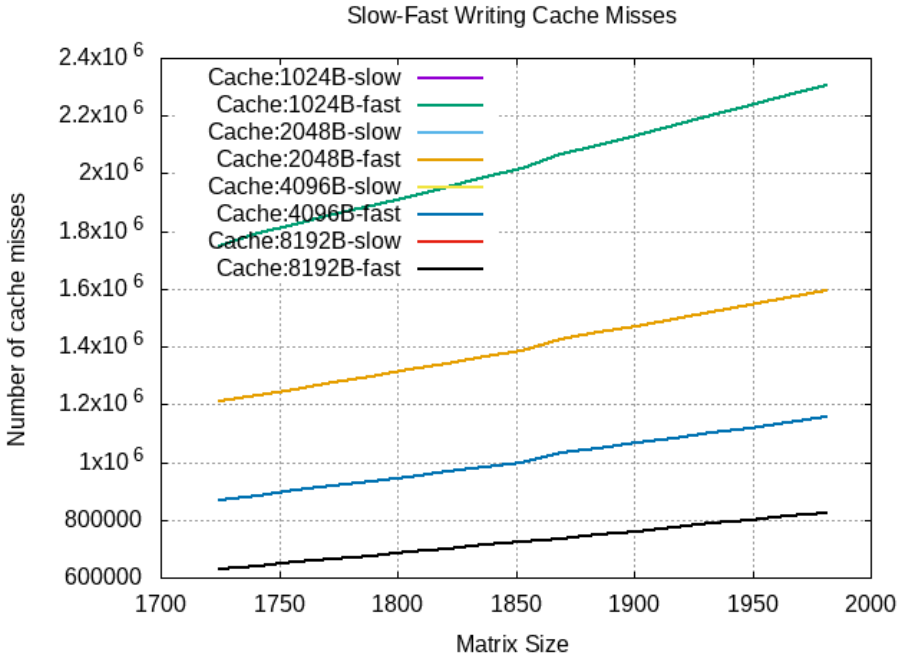

Figura 4: Fallos de escritura de caché

¿Se observan cambios de tendencia al variar los tamaños de caché para el programa slow? ¿Y para el programa fast?

Observamos para ambos programas que el aumento del tamaño de caché supone, a parte de una disminución del número de fallos, una disminución de la pendiente de la recta en la gráfica. Esto quiere decir que aumentar el tamaño de la matriz provoca un menor aumento del número de fallos de caché cuanto más grande sea el tamaño de caché. Esto se debe que al tener un mayor tamaño se puede almacenar un mayor número de datos y por tanto se fallará un menor número de veces, porque es más probable que el dato al que accedemos esté ya cargado en la caché, por lo que no será necesario cargarlo de memoria, que es una operación costosa en cuanto a tiempo.

¿Varía la tendencia cuando se fija en un tamaño de caché concreto y compara el programa slow y fast?

En la gráfica de fallos de escritura (*cache_escritura.png*) solamente observamos 4 líneas cuando deberíamos observar 8. Esto se debe a que los fallos de escritura para *slow* y *fast* con el mismo tamaño de caché son los mismos. Es algo obvio pues la forma de acceder a las matrices en memoria no afecta a cómo se escribe en la caché. Para la escritura, se accede las mismas veces, puesto que

cada vez que actualizamos la suma total, necesitamos ese dato en caché. Por lo tanto, como con cada fila/columna se debe cargar el dato en caché, este se carga las mismas veces independientemente.

Sin embargo, para el caso de lectura sí que hay un cambio de tendencia. Esto se debe a lo que hemos explicado en el ejercicio anterior, ya que para leer una fila de datos se necesitan menos cargas a caché, porque cada una carga algún dato contiguo al cargar un bloque de memoria, que se lee entero o casi. Por el contrario, para leer por columnas, para leer cada dato se carga un bloque entero.

Ejercicio 3

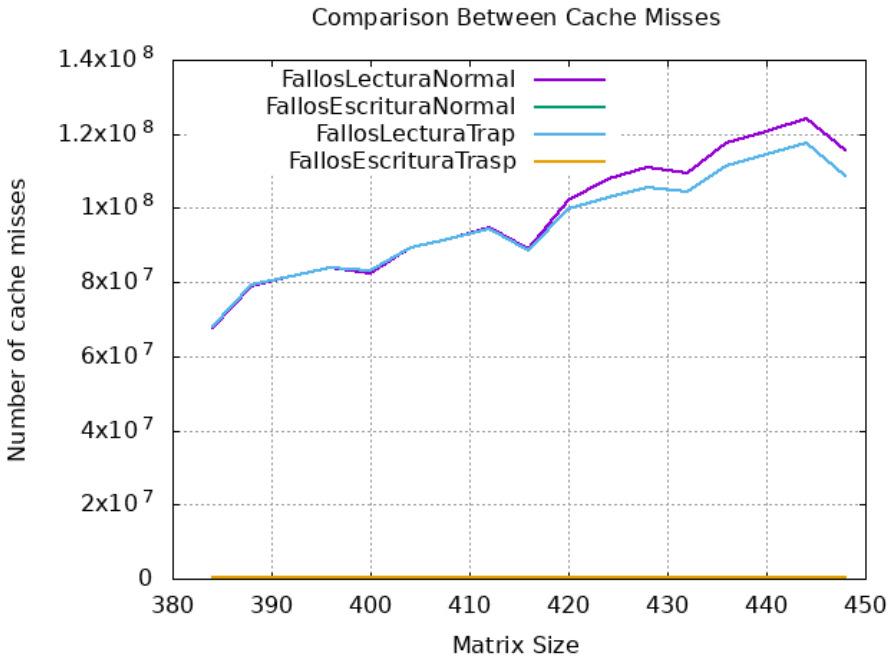

Figura 5: Fallos de caché

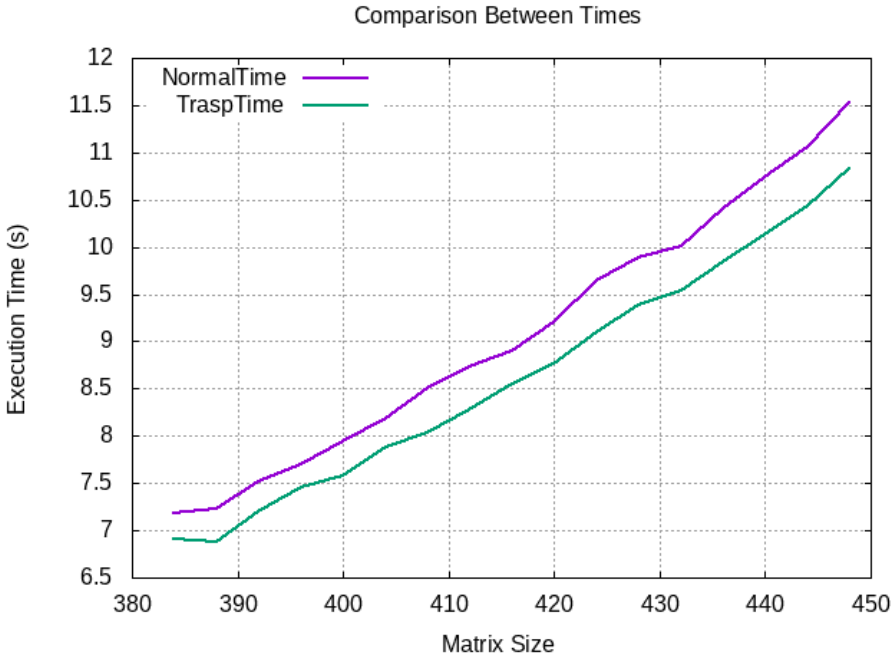

Figura 6: Tiempos de ejecución

¿Se observan cambios de tendencia al variar los tamaños de las matrices? ¿A qué se deben los efectos observados?

Se puede observar que los tiempos de ejecución siguen una tendencia lineal en ambos cálculos, de forma que la diferencia entre ambos crece de forma lineal. Por otra parte, en los fallos de caché, la relación entre el acceso en el cálculo con la traspuesta son tan inferiores, que para la gráfica son demasiado pequeños (del orden de 70 millones frente a 200.000 al inicio).

A priori, podríamos pensar que los accesos a memoria, y por lo tanto los tiempos de ejecución, deberían variar con el tamaño de la matriz de forma cuadrática, puesto que ambas dimensiones varían. Sin embargo, esto no se corresponde bien con lo que se puede observar en la gráfica. Esto se puede deber a varias cosas:

- Los tiempos de ejecución no son una medida fija o fiable, puesto que dependen del ordenador y los otros procesos ejecutándose simultáneamente.
- La ejecución con la traspuesta implica también el coste de trasponer la matriz, que se añade al tiempo de cálculo.

- Puede que la variación de tamaño de matriz no sea lo suficientemente grande para apreciar el comportamiento cuaadrático, y lo que apreciamos es una curva muy amplificada que parece recta. Esto se puede deber a que como nos indicó el profesor, reducimos el tamaño de las matrices, y el incremento de los mismos, puesto que el tiempo de ejeución si no se disparaba.

Ejercicio 4

El alumno puede jugar con los diferentes parámetros de configuración de las cachés estudiados a lo largo de la práctica (pudiendo incluir, si así lo desean, la asociatividad y el tamaño de línea de la caché) así como el tamaño de las matrices multiplicadas. Los resultados que reporte pueden contemplar errores de lectura y escritura así como tiempos de ejecución, y cualquier otro dato significativo que se le ocurra obtener.

Todos los experimentos que decida realizar, así como los resultados que obtenga deberán ir acompañados de una explicación que justifique el motivo por el que ha decidido realizar el experimento y las conclusiones que ha extraído del mismo. También se valorará en este ejercicio que el alumno proporcione los scripts de Bash y GNUplot o, en su defecto, los comandos Linux que haya empleado para obtener los resultados.

Para este ejercicio, lo que hicimos fue ejecutar los mismos cálculos del ejercicio anterior, variando el tamaño de caché (1 KB, 2 KB, 4 KB y 8 KB).

Nos hubiera gustado haber podido experimentar con más parámetros (asociatividad o tamaño de palabra), pero el tiempo de ejecución fue muy largo (realizamos varias mediciones para reducir errores, con tamaños de matriz variable y con distintos tamaños de caché, lo cual son muchos bucles):

```
$ time ./ejercicio4.sh
real 546m7,944s
user 545m4,852s
sys 1m31,440s
```

Como se puede ver, este programa tardó casi 9 horas en ejecutarse.

Lo que hemos podido comprobar es que cuanto mayor fuera la caché, más se reducían los fallos de la misma, casi llegándose a reducir a la mitad comparando 1 KB con 8 KB. Esto quiere decir que ampliar la caché cada vez más no es una estrategia óptima, puesto que multiplicando por 8 el tamaño de caché, solo se dividen por 2 los fallos de caché, sin tener en cuenta los problemas

reales que conlleva esto de aumento de precio y tamaño de la caché. Además, si comparamos el tiempo de ejecución, el cambio no es nada relevante, incluso se ve que empíricamente es mejor estrategia usar la estrategia de trasponer, que multiplicar por 8 el tamaño de la caché.

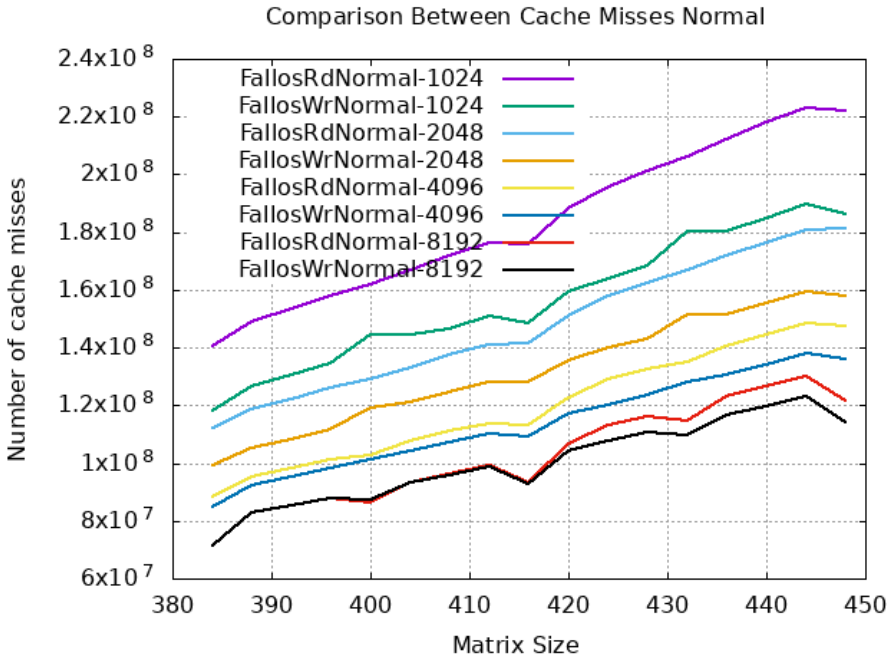

Figura 7: Fallos de caché multiplicación normal

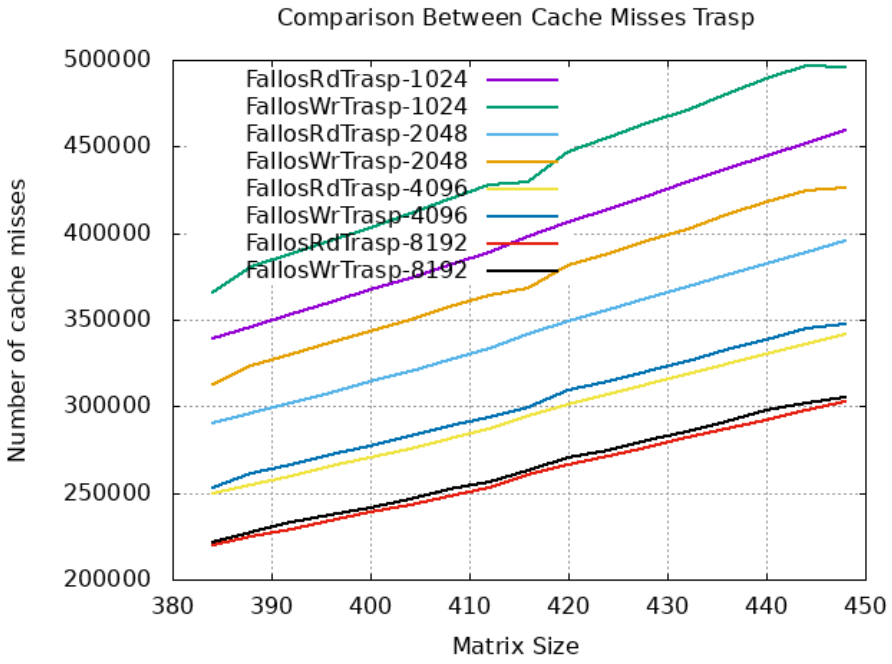

Figura 8: Fallos de caché multiplicación traspuesta

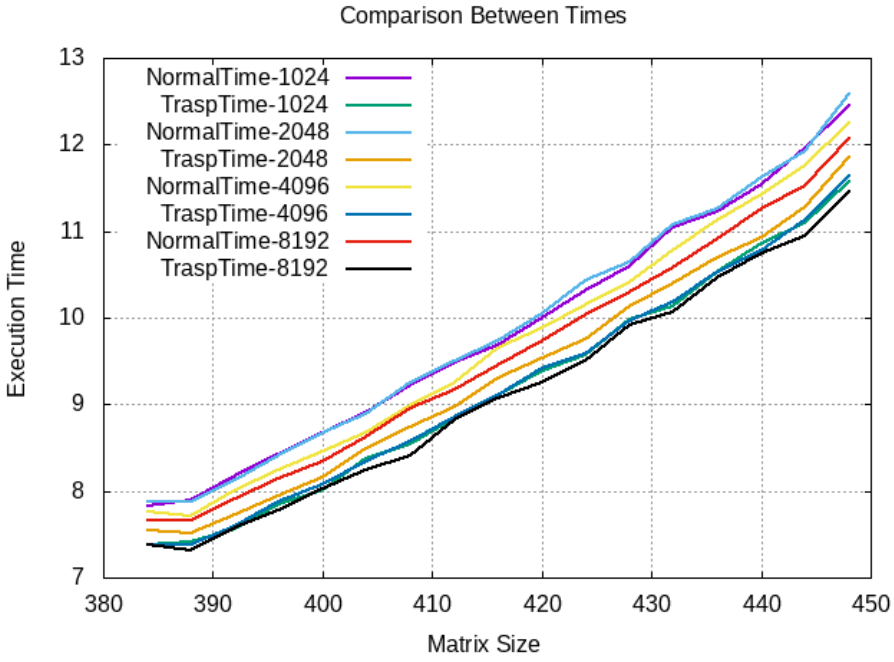

Figura 9: Tiempos de ejecución

Desde luego, aunque para esta práctica no hayamos podido comparar todos los aspectos posibles, sin duda en nuestro tiempo libre intentaremos realizar una comparación de esto para ver cómo influyen en un problema real estos cambios, puesto que sabemos que este es un problema real, con el que quizás nos encontremos en nuestra vida laboral.