Arquitectura de Ordendores, Práctica 3: Memoria Caché y Rendimiento

Leandro García y Fabián Gutiérrez (Grupo 1301_08)

3 de diciembre de 2020

En primer lugar, $P = (8 \mod 7) + 4 = 5$ será el número utilizado a lo largo de la práctica.

Ejercicio 0

Como se observa en la figura 1, el sistema consta de tres niveles de caché:

- **Nivel 1**: caché no unificada, es decir, memorias de instrucciones y de datos separadas. Ambas son de 32 KB, asociativas de 8 vías y con tamaño de línea de 64 B.
- **Nivel 2**: caché unificada. Es de 256 KB, asociativa de 4 vías y con tamaño de línea de 64 B.
- **Nivel 3**: caché unificada. Es de 8 MB, asociativa de 16 vías y con tamaño de línea de 64 B.

```
leandro@ubuntu:~/Documents/ARQO_2020$ 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                  8388608
LEVEL3_CACHE_ASSOC                 16
LEVEL3_CACHE_LINESIZE              64
LEVEL4_CACHE_SIZE                  0
LEVEL4_CACHE_ASSOC                 0
LEVEL4_CACHE_LINESIZE              0
```

Figura 1: Información sobre las memorias caché devuelta por `getconf -a`

Adicionalmente, en la figura 2 se puede observar el diagrama de memoria caché generado por el comando `lstopo`. En particular, el equipo en el que se ejecutan las pruebas consta de cuatro núcleos, cada uno con sus tres niveles de caché descritos anteriormente.

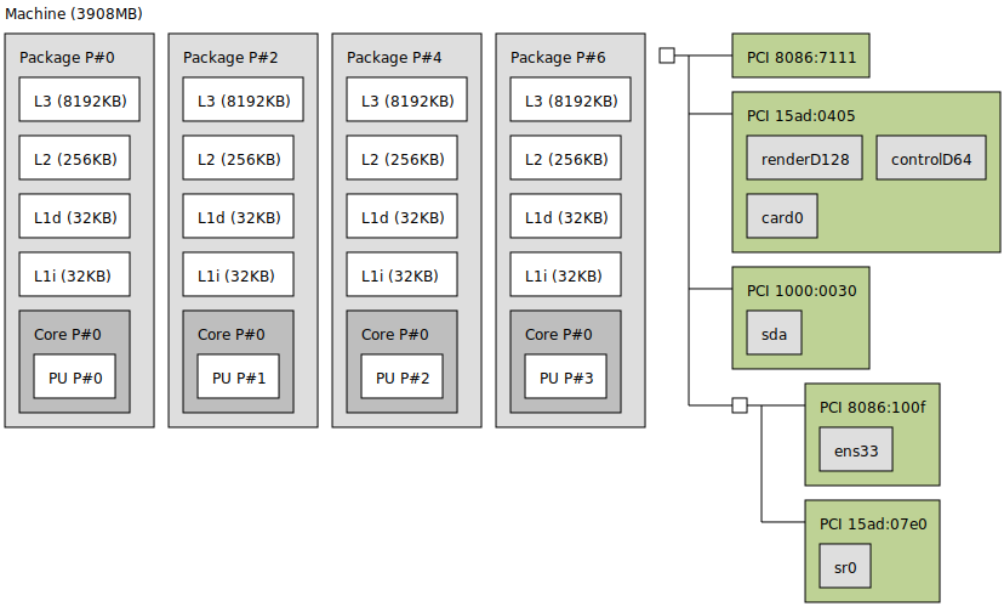

Figura 2: Salida `lstopo`

Ejercicio 1

2) Es necesario realizar múltiples veces la toma de medidas de rendimiento para cada programa y tamaño de matriz para suavizar el efecto de la multiprogramación del sistema donde se ejecutan los programas. El tiempo de procesador que el sistema operativo concede a estos programas depende de los demás procesos que compitan por los recursos en ese momento y de las políticas del sistema, por lo que al realizar el experimento varias veces y tomar la media se obtiene un valor central que depende poco (de manera inversamente proporcional al número de repeticiones) de las decisiones puntuales del sistema operativo, que se pueden considerar aleatorias.

4) Tras realizarse la ejecución del script `slow_fast_time.sh` con 15 iteraciones se obtiene la gráfica de la figura 3.

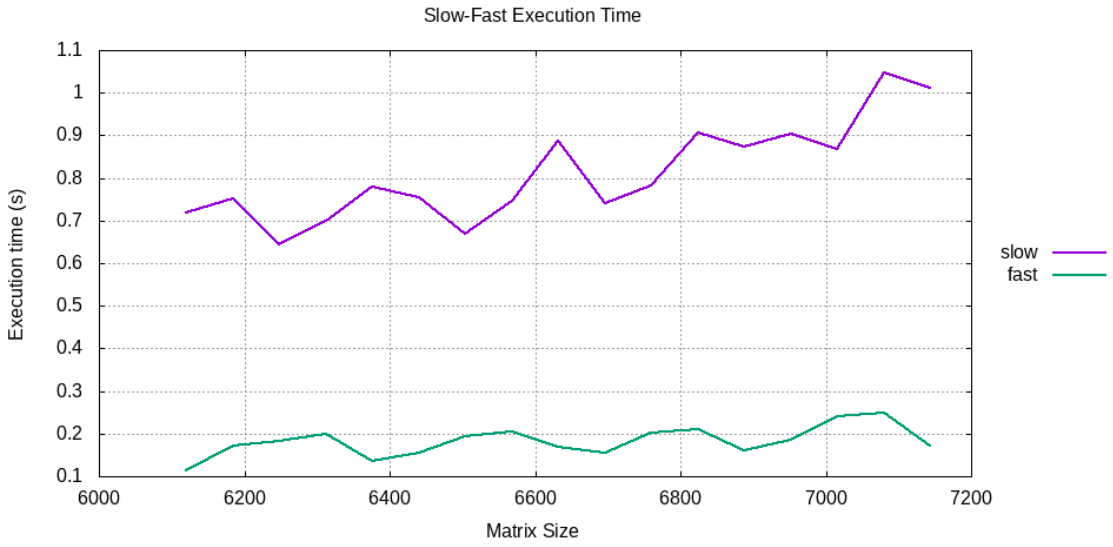

Figura 3: Tiempos de ejecución de `slow` y `fast`

Debido a que se ha utilizado una máquina virtual, la gráfica es aún irregular en ciertos puntos, pero se aprecia la gran diferencia entre ambos algoritmos y la ligera tendencia ascendente de ambos al aumentar el tamaño de las matrices. Esto último sería más evidente tomando un rango mayor en el eje x, pudiéndose apreciar el aumento del coste entre matrices con mayor diferencia de tamaño.

5) Las matrices se almacenan en memoria por filas. En consecuencia, una lectura por filas (versión `fast`) accede a posiciones de memoria contiguas, lo que se traduce en una mayor localidad y un menor número de fallos en caché, pues cada bloque que se carga contiene las posiciones a acceder en el futuro cercano. Por otra parte, la lectura por columnas (versión `slow`) accede a posiciones de memoria que distan entre sí la longitud de una fila. Para matrices de dimensiones reducidas (respecto al tamaño de los bloques de las cachés), este patrón de accesos no tiene grandes penalizaciones comparado con el anterior, pero para matrices de mayor dimensión aumenta la tasa de fallos y las penalizaciones que se derivan de estos, puesto que los datos tienden a estar en bloques distintos. En una palabra, se inhibe la localidad.

   Precisamente, el diseño de las memorias caché se basa en el principio de localidad en los accesos a memoria, por lo que es de esperar que un programa que accede a posiciones muy distantes como `slow` tenga peor rendimiento.

Ejercicio 2

Para generar los datos y las gráficas requeridos se ha empleado el script `cachegrind_slow_-fast.sh`, que automatiza todas las acciones necesarias. Cabe destacar que durante la ejecución de `cachegrind` se muestra el mensaje mostrado en la figura 4, que podría afectar al resultado de la simulación.

```
--6273-- warning: L3 cache found, using its data for the LL simulation.
```

Figura 4: *Warning* mostrado por `cachegrind`

En la primera de las gráficas generadas (figura 5) se analiza el comportamiento de los fallos de lectura en la caché L1 para distintos tamaños de la misma y distintos tamaños de matriz.

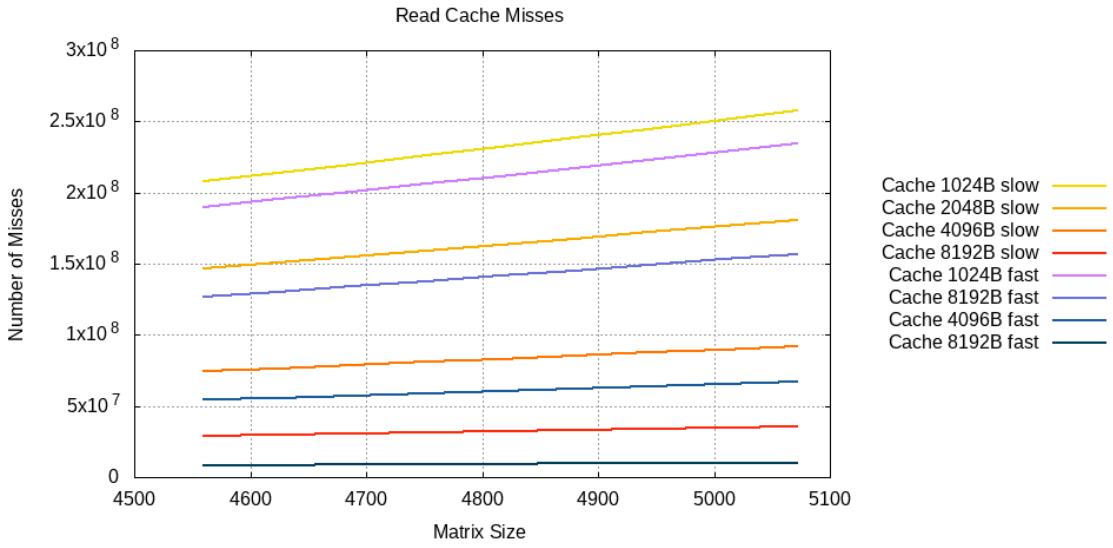

Figura 5: Fallos de lectura en la caché L1 para `slow` y `fast`

Se puede observar que, como cabía esperar, para el mismo tamaño de caché el método `fast` comete menos fallos de lectura. Esto se debe a que al realizar un recorrido por filas de la matriz se aprovecha la localidad de los datos en la caché. El resultado concuerda con las conclusiones del Ejercicio 1, ya que la cantidad de fallos es el motivo del aumento del tiempo de ejecución en el algoritmo `slow`. También se aprecia que, cuanto mayor es el tamaño de la matriz, más errores se cometen.

Finalmente, se puede observar que para un mismo algoritmo, el aumento del tamaño de la caché reduce el numero de fallos de lectura. Esto se produce porque un mayor tamaño de caché permite tener un mayor número de bloques de datos almacenados simultáneamente (sin necesidad de reemplazos), lo que para tamaños de matriz grandes supone una importante diferencia de rendimiento. Además, si bien el número de fallos tiene un crecimiento lineal respecto al tamaño de matriz en todas las configuraciones de caché estudiadas, este crecimiento tiene una menor pendiente cuanto mayor es el tamaño de caché, es decir, cuanto mayor es el tamaño de la caché, un aumento en el tamaño de matriz supone un menor aumento en el número de fallos.

Por otra parte, en la gráfica de la figura 6 se estudia el comportamiento de los fallos de escritura en la caché L1 para distintos tamaños de esta y distintos tamaños de matriz.

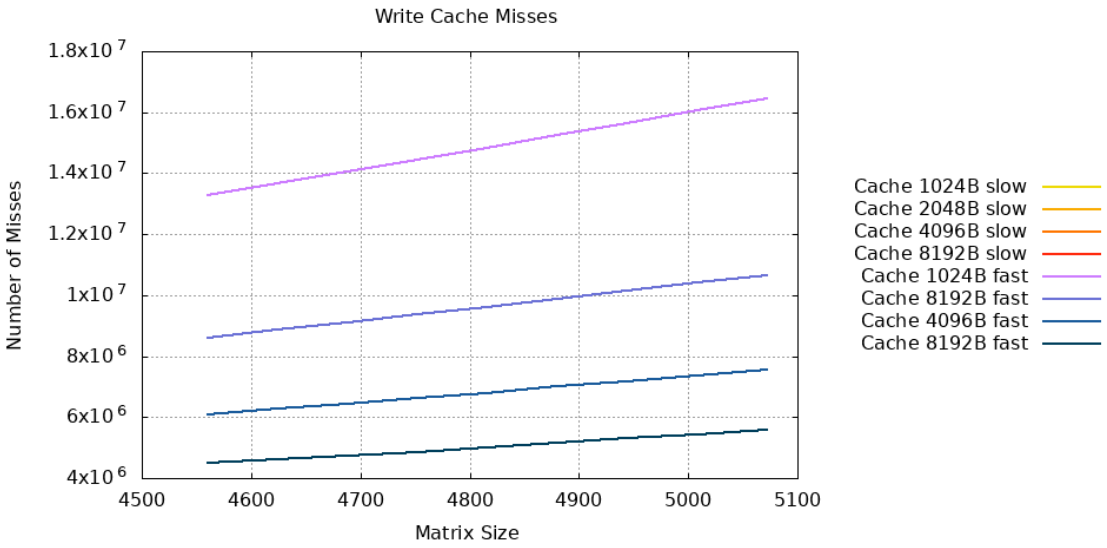

Figura 6: Fallos de escritura en la caché L1 para `slow` y `fast`

A diferencia de los fallos de lectura, los fallos en escritura para ambos algoritmos son idénticos para un mismo tamaño de caché. Esto se debe a que `slow` y `fast` tienen un mismo patrón de accesos a memoria en escritura, pues ambos escriben únicamente la variable que lleva la suma acumulada del algoritmo. Además, se observa que el número de fallos de escritura son mucho menores que los de lectura (aproximadamente en un orden de magnitud) por la misma razón: estos algoritmos leen matrices, es decir, múltiples direcciones de memoria, con lo que en general hay mayor probabilidad de que se produzca un fallo de caché (situación que se agrava si el patrón de acceso inhibe la localidad como el de `slow`); pero escriben en una única variable que almacena la suma, por lo que es más difícil que se produzca un fallo en caché una vez cargado el bloque que la contiene.

Cabe destacar que, al igual que en el caso de la lectura, el número de fallos aumenta de forma lineal con el tamaño de matriz, pues en general un mayor número de direcciones a las que acceder (bien por lectura o por escritura) supone un mayor número de reemplazos, es decir, más fallos de caché. Igualmente se aprecia que para mayores tamaños de caché el crecimiento del número de fallos tiene una menor pendiente. Sin embargo, esta variación de las pendientes para cada configuración de caché es bastante pequeña comparada con lo observado en el caso de la lectura.

Ejercicio 3

Para generar los datos y las gráficas solicitadas se ha utilizado el script `mult_time.sh`. Se realizaron 15 iteraciones en la generación de los tiempos de ejecución por los motivos explicados en el Ejercicio 1.

En la gráfica de la figura 7 se observa que los tiempos de ejecución del producto traspuesto son sensiblemente menores que el algoritmo regular. En ambos casos, el tiempo de ejecución es directamente proporcional al tamaño de las matrices, teniendo el del algoritmo regular un crecimiento más veloz que el del algoritmo traspuesto (es decir, el segundo tiene menor pendiente en general). Además, se puede decir que en ambos casos el crecimiento es lineal, lo que se ve claramente para el algoritmo traspuesto. Se puede argumentar que el algoritmo regular también presenta un crecimiento lineal si se considera que los puntos están suficientemente cerca de estar alineados (omitiendo el primer valor, la recta de regresión se ajusta bien a la curva obtenida).

Por otra parte, se observa que el comportamiento de los tiempos para el algoritmo regular es mucho menos uniforme que el de los tiempos del algoritmo traspuesto, pese a haberse tomado las medidas 15 veces para mitigar los efectos de la multiprogramación y la planificación del sistema operativo. En particular, el punto correspondiente al tamaño de matriz de 1536 es muy superior a los tiempos de tamaños de matriz cercanos [1].

Figura 7: Tiempos de ejecución para los productos regular y traspuesto de matrices

Las simulaciones con `cachegrind` se realizaron con la configuración por defecto, es decir, con la configuración de las cachés del equipo descrita en el Ejercicio 0. Esto permitirá asociar el número de fallos de las simulaciones con los tiempos de ejecución medidos, lo que conducirá a dar una explicación al comportamiento observado anteriormente. Nótese que en la gráfica de la figura 8 el eje de las ordenadas está en escala logarítmica.

En general, se observa que los fallos de lectura son varios órdenes de magnitud mayores que los de escritura y en ambos casos el número de fallos es directamente proporcional al tamaño de las matrices. Conviene estudiar con más detalle cada tipo de fallo por separado:

[1] En el fichero con las medidas generadas (`mult.dat`), se observa que este fenómeno no ocurre en la simulación de `cachegrind`. Como este último había sido configurado para simular las cachés reales del equipo, se puede afirmar que el tiempo de ejecución para este tamaño puede verse fuertemente afectado por el sistema operativo.

- **Fallos de lectura**: el algoritmo regular produce muchos más fallos de lectura que el traspuesto (en el orden de 10 veces más). Como los fallos en escritura son más similares entre ambos algoritmos, son las lecturas las que provocan el peor rendimiento en tiempo de ejecución mostrado por el algoritmo regular en la figura 7.

  El algoritmo regular recorre para cada fila de A todas las columnas de B, por lo que recorre cada columna N veces, siendo N la dimensión de las matrices. En cambio, el algoritmo traspuesto recorre cada columna una única vez al trasponer y posteriormente recorre tanto A como la traspuesta de B por filas. De este modo, el algoritmo regular recorre las columnas de B, con los fallos de caché que conlleva, N veces y el traspuesto lo hace solo una vez, por lo que el número de fallos será menor. De hecho, la diferencia entre ambos algoritmos se hace mayor cuanto mayor es N, pues el algoritmo regular comete, por este motivo, N veces más fallos asociados al acceso por columnas que el traspuesto.

- **Fallos de escritura**: se aprecia en el gráfico que el algoritmo traspuesto tiene un número de fallos de escritura ligeramente mayor (comparado con la diferencia entre algoritmos en fallos de lectura) que los del algoritmo regular. Esta diferencia es producto de que, si bien ambos algoritmos realizan la multiplicación de matrices y escriben la matriz C, la versión traspuesta además escribe la matriz traspuesta de B, por lo que hace más escrituras y, en consecuencia, produce más fallos de este tipo.

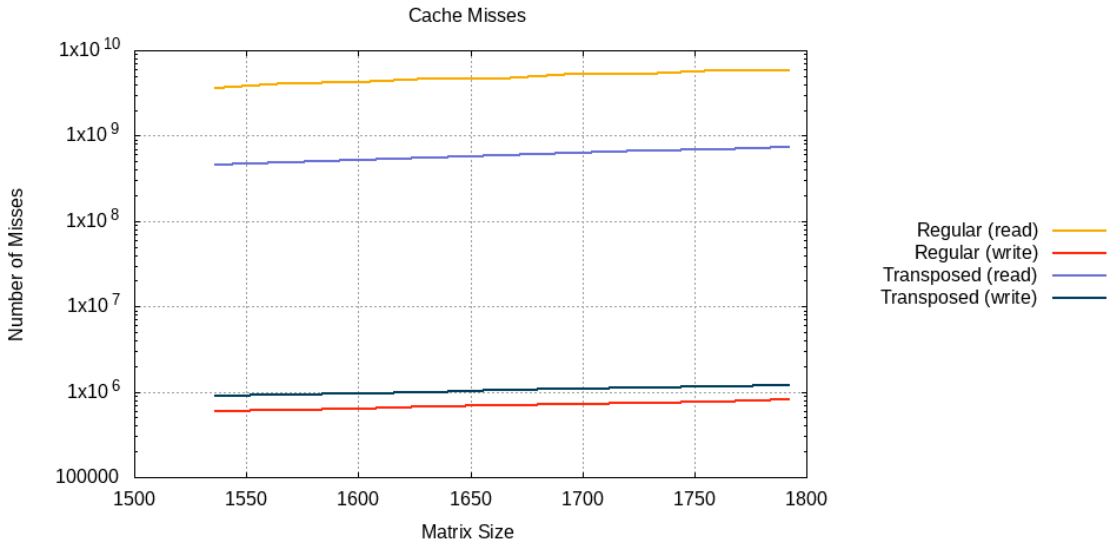

Figura 8: Fallos en la caché L1 para los productos regular y traspuesto de matrices

Ejercicio 4

Dado que se ha analizado en los ejercicios anteriores la influencia del tamaño de la caché y del aprovechamiento de la misma que realice el algoritmo, en este apartado opcional se estudiarán los cambios provocados por otros parámetros de la memoria, como la asociatividad o el tamaño de los bloques.

Asociatividad

En primer lugar, se ejecutan los dos métodos de multiplicación para distintas asociatividades en las cachés (tanto de datos como de intrucciones). Las dimensiones de las matrices varían entre $128 \cdot (P + 1)$ y $128 \cdot (P + 2)$ con un incremento de 16, es decir, las dimensiones utilizadas en el Ejercicio 3 divididas entre 2 (por limitaciones de recursos y tiempo).

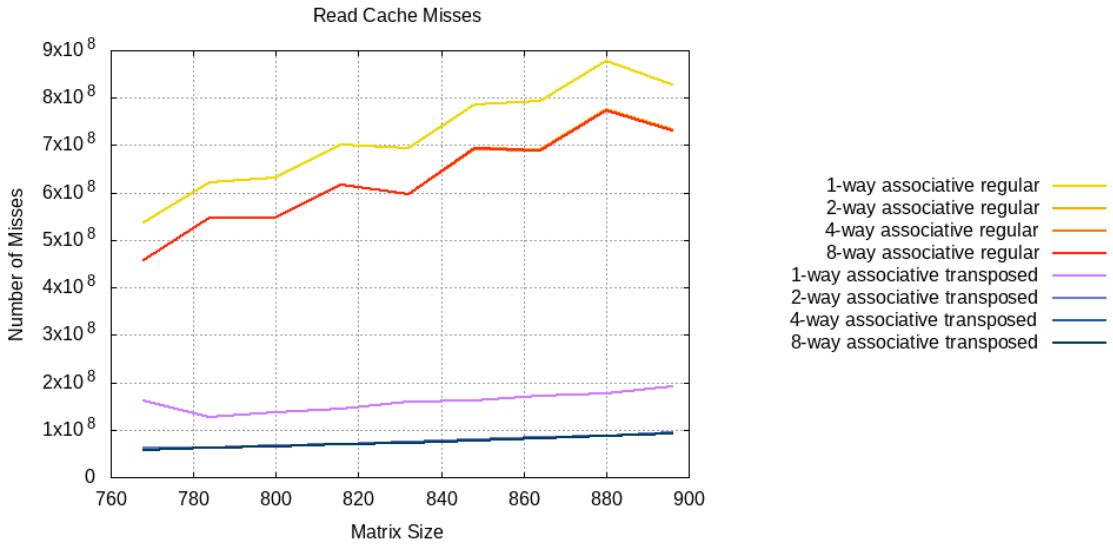

Figura 9: Fallos de lectura en el producto de matrices para distintos niveles de asociatividad

El resultado de la simulación en cuanto a fallos de lectura en el primer nivel concuerda con los resutados teóricos analizados en clase. Se aprecia una clara mejora para la asociatividad de 2 vías frente a la correspondencia directa. Sin embargo, al aumentar este parámetro por encima de dos, la mejora apenas es apreciable. Debido a la escala de los datos no es posible observar claramente la diferencia entre 2, 4 y 8 vías en la gráfica, pero si se estudian los ficheros de datos (`assoc_1.dat`, `assoc_2.dat`, `assoc_4.dat`, `assoc_8.dat`) se puede ver una ligera reducción en los fallos de lectura a medida que se aumenta la asociatividad.

Para los fallos de escritura se ha optado de nuevo por una escala logarítmica en el eje de ordenadas que permita apreciar las diferencias pese a las diferencias de escala. Al igual que en la lectura, se ve que la asociatividad mejora el rendimiento de la caché y que, al igual que en el ejercicio anterior, trasponer la matriz añade fallos de escritura y provoca un mayor número de fallos en total (esto se aprecia en las asociatividades altas, que tienen un comportamiento lineal).

En las simulaciónes de memorias de baja asociatividad se pueden apreciar grandes irregularidades. Esto, sumado al hecho de haber reducido a 16 la diferencia entre tamaños observados, induce a pensar que el tamaño de la matriz tiene un gran impacto sobre la ubicación en memoria de los bloques, lo que puede provocar reemplazamientos poco eficientes y, consecuentemente, un alto número

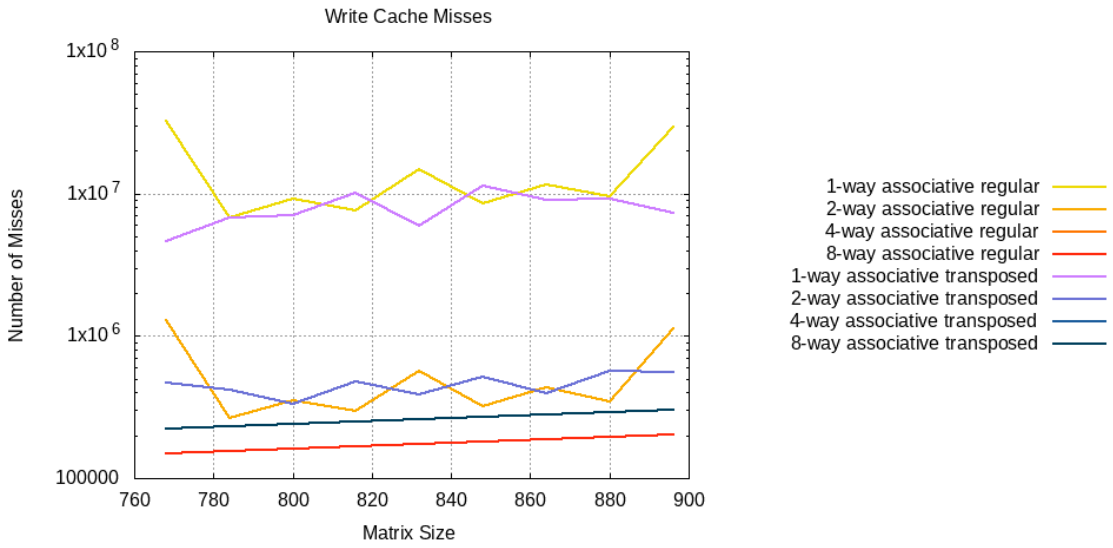

Figura 10: Fallos de escritura en el producto de matrices para distintos niveles de asociatividad

de fallos por conflicto.

Tamaño de bloque

Para concluir con el estudio del rendimiento de las memorias caché, se han simulado con `cachegrind` multiplicaciones siguiendo los dos algoritmos de los ejercicios previos empleando cachés de tamaño fijo y longitud de bloque variable (tomando valores entre 32kB y 256kB).

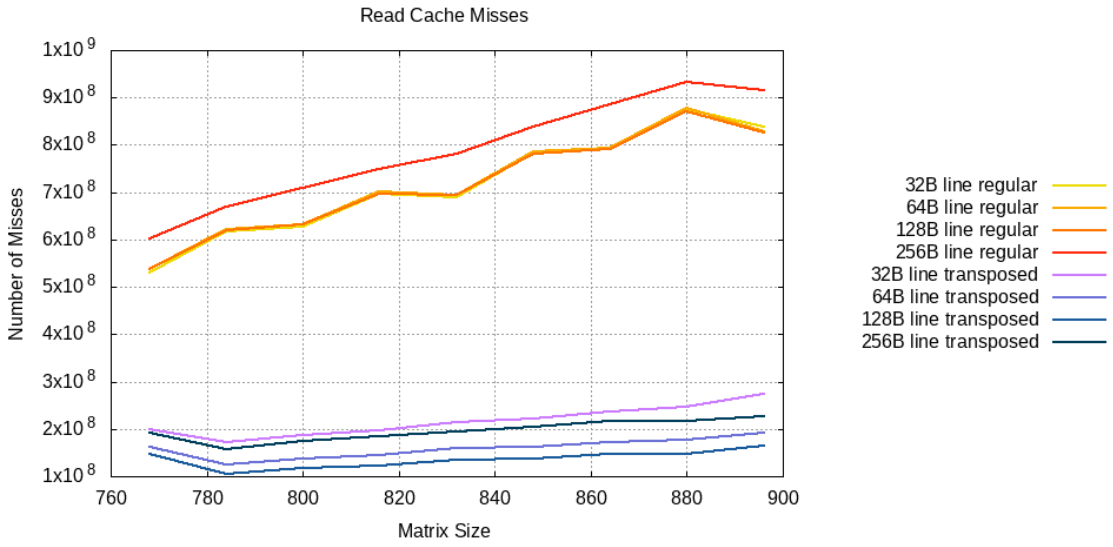

Figura 11: Fallos de lectura en el producto de matrices para distintos tamaños de bloque

Los fallos de lectura para estas simulaciones muestran un resultado bastante curioso: el método regular es mejor con bloques pequeños, mientras que el algoritmo de trasposición se optimiza al aumentar el tamaño de linea. En este último caso, puesto que se leen ambas matrices por filas

(aprovechando así la localidad de los valores), es lógico que, cuanto mayor sea la longitud de un bloque, más números se podrán leer consecutivamente sin producirse un fallo. El hecho de que la multiplicación habitual se comporte de forma opuesta puede deberse a la polución. Cuando se lee por columnas una matriz de gran tamaño, la longitud del bloque es indiferente, ya que solo interesaría leer uno de sus valores (en el mejor de los casos, se leen varios, pero pocos comparativamente). Sin embargo, como se ha fijado el tamaño de la memoria, un mayor tamaño de bloques implica una menor cantidad de los mismos presentes simultáneamente, es decir, menos datos útiles para el acceso por columnas, lo que deriva en más reemplazamientos.

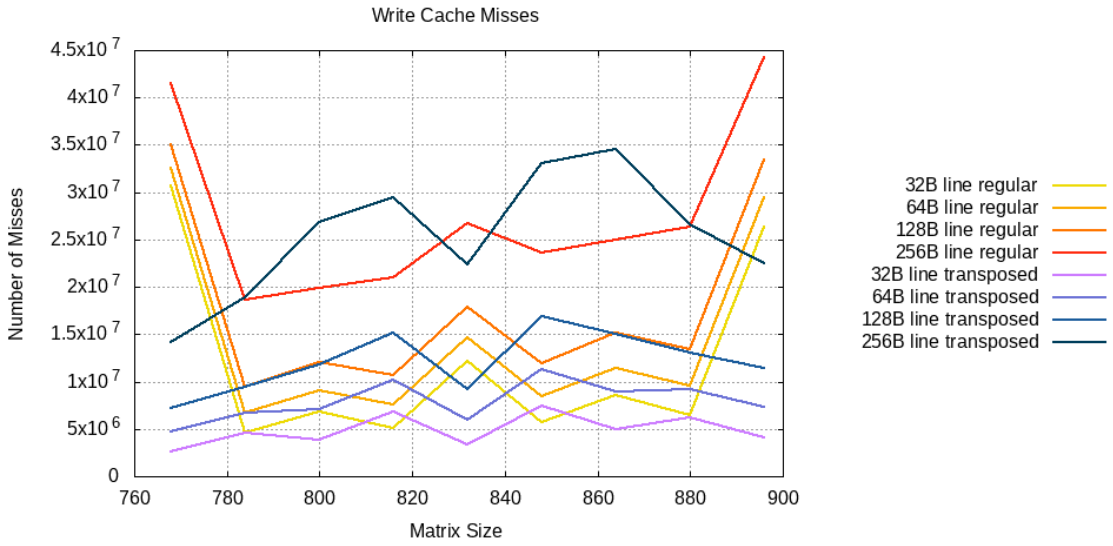

Figura 12: Fallos de escritura en el producto de matrices para distintos tamaños de bloque

En cuanto a los fallos de escritura, se observa en la figura 12 que para mayor tamaño de bloque se producen más fallos. Por idénticos motivos que en la lectura por columnas, un mayor tamaño de bloque supone cargar un gran número de bytes para escribir pocos en comparación. Por tanto, la caché almacena muchos datos no utilizados y se ve forzada a realizar más reemplazamientos.