Estructura de Computadores: Práctica 6

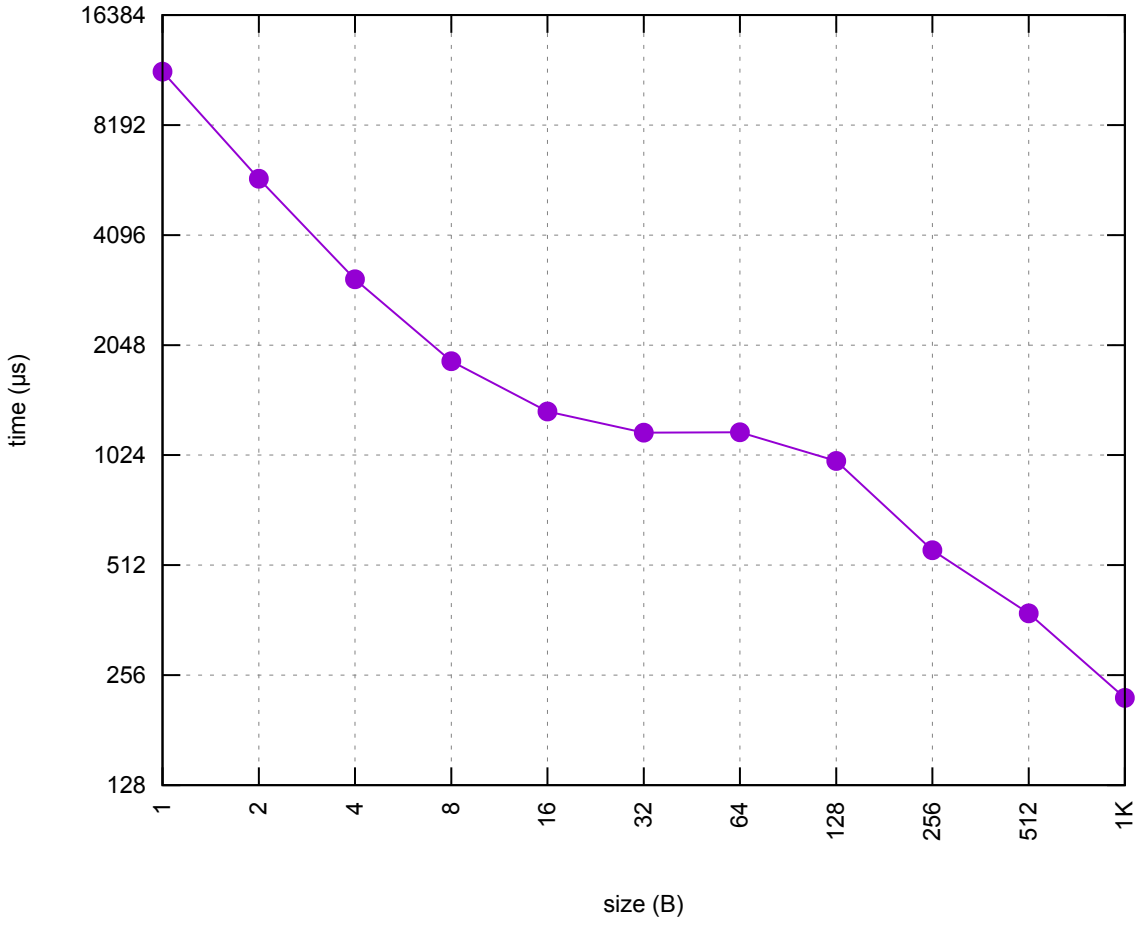
Memoria Caché

Pablo Olivares Martínez

En esta práctica comprobaremos ciertos aspectos sobre la memoria caché. Para ello, vamos a comprobar el tamaño de los bloques de palabras y el de las cachés mediante *line.cc* y *size.cc* respectivamente. Para ello, comenzemos explicando line.cc:

```
                                                ...
        for (unsigned i = 0; i < bytes.size(); i += line)
            bytes[i] ^= 1;
                                                ...
```

Para completar este programa, primero debemos analizar que buscamos: queremos saber el tamaño de las lineas de palabras. Pues para ello, podemos usar una operación que requiera poco tiempo de ejecución para así evitar adulterar los resultados de tiempo con operaciones complejas usando , por ejemplo, la función **or**. También podemos acceder a los bytes a través de saltos según los distintos tamaños de prueba que le asignamos a los bloques, así veremos si el programa se entretiene accediendo al interior de los bloques o salta de bloque en bloque. Por ello, tras la ejecución del **make** obtenemos la siguiente gráfica:

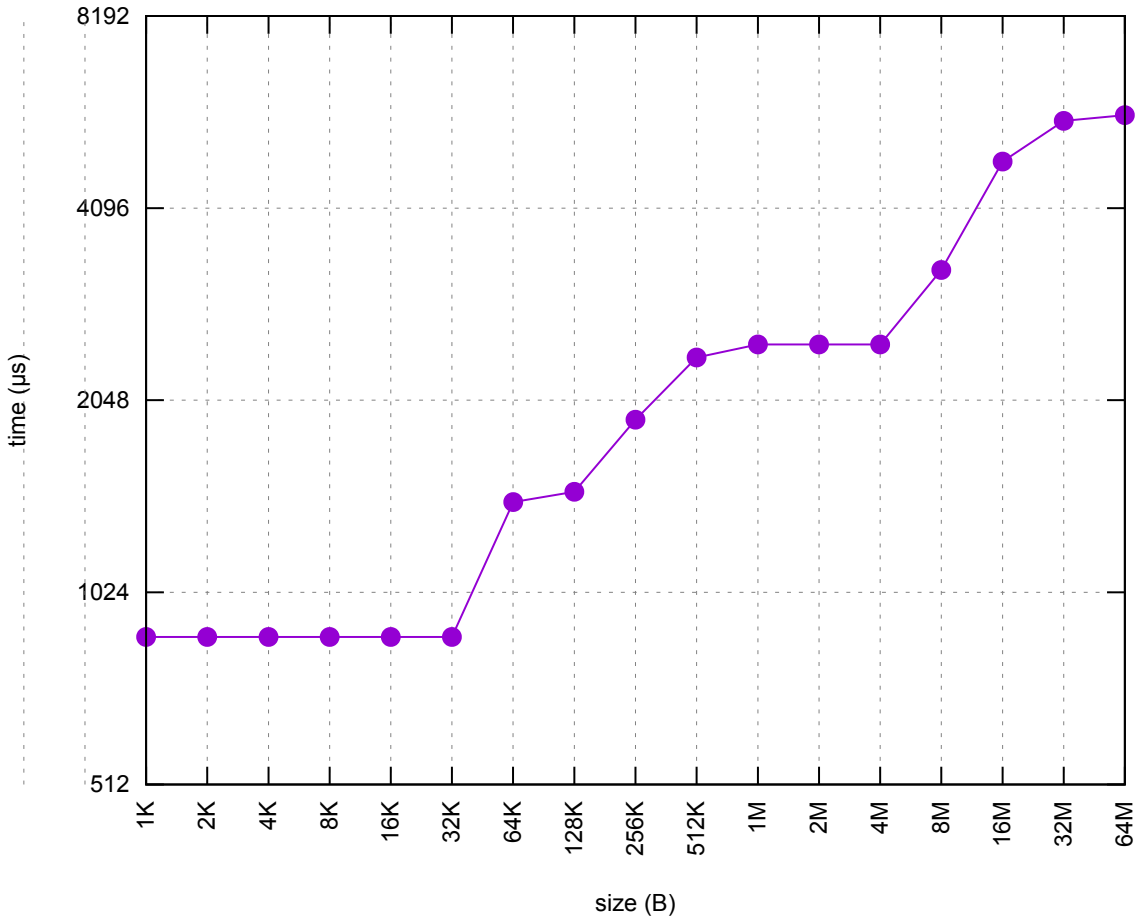


Efectivamente, vemos que a partir de los 64B los tiempos de acceso van disminuyendo drásticamente. Esto es un indicio de que el tamaño de línea es de 64B, ya que no tiene que ir buscando datos internos del bloque y por tanto, el acceso a posiciones múltiplo de ésta hace que disminuya este tiempo.

A continuación, explicaré el fichero `size.cc`. El fichero ha sido finalizado de ésta forma, pero también podría hacerse de otras muchas:

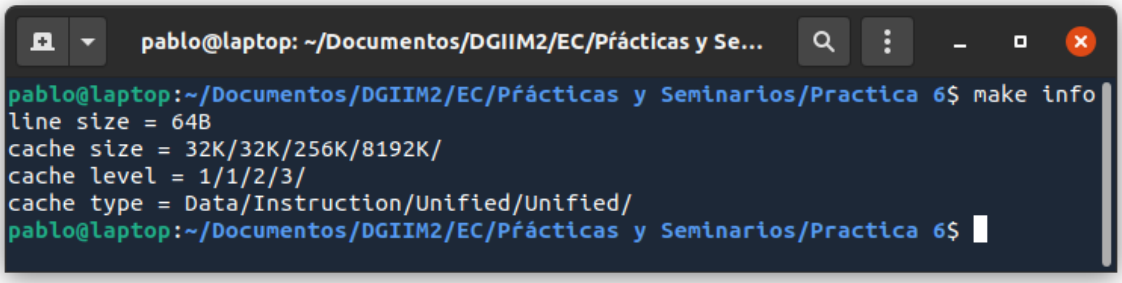
```
                                                    ...
            for (unsigned i = 0; i < STEPS; ++i)
                bytes[(i*64)&(size-1)] ^= 1;
                                                    ...
```

El hecho de que usemos esta expresión es por comodidad. Lo que ahora queremos obtener de `size` es que nos identifique, a través de los tiempos de ejecución, los distintos niveles de memoria. Para ello, vamos a acceder en la memoria por bloques (los bloques son de 64B), ya que los distintos niveles almacenan y transfieren las palabras por bloques. De esta manera, no nos entretenemos en acceso a bloques y vamos a lo que de verdad nos interesa. Además, queremos que a pesar de comparar distintos tamaños desde 1K a 64MB, queremos que cada comprobación lleve a cabo el mismo número de pasos, para así poder comparar tiempos de manera homogénea. Por tanto, podríamos haber hecho un *if else*, pero sin embargo, opto por realizar esta función lógica **and**. Lo que ésto hace es, mientras *i*64* sea menor que *size - 1* (es decir, la última posición del vector, que al ser del tipo 2^n - 1, su expresión binaria está compuesta únicamente con unos y al hacer la función and nos van a salir los valores que corresponden). Cuando *i*64* sea mayor, lo que accederá es a *size-1* hasta completar STEPS, así garantizando una justa comparación. Finalmente, el valor de la tabla queda así:



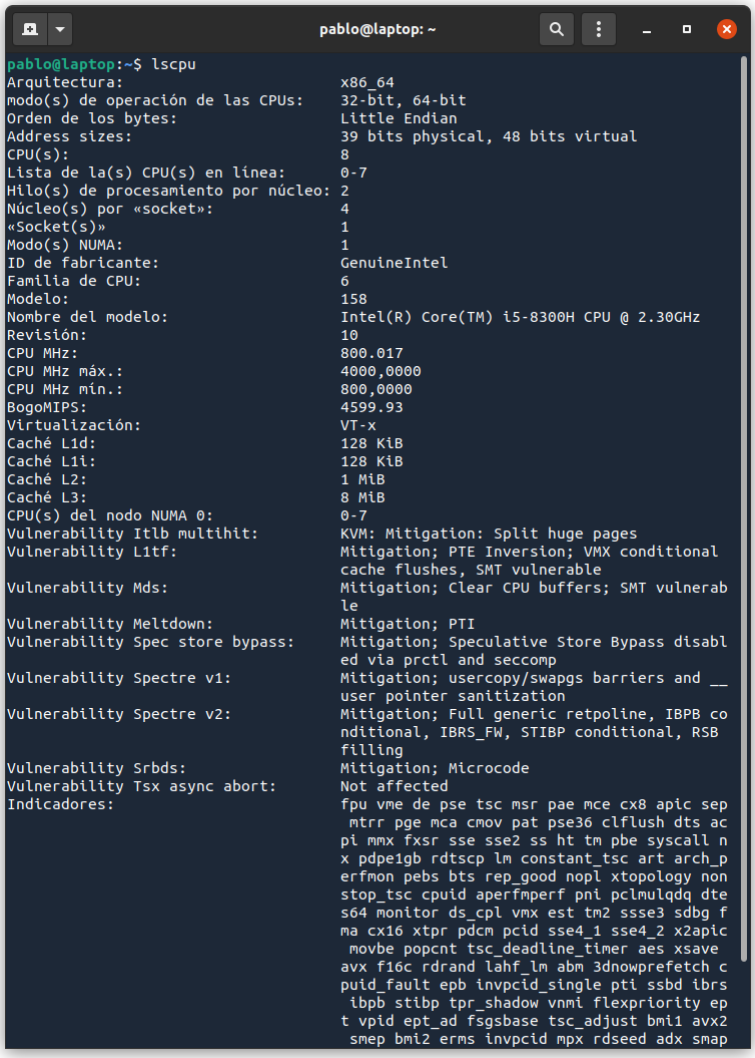
Aquí podemos ver saltos notables en 32K, 256K y 8M. Aquí podemos intuir que estos saltos de tiempo corresponden a los inducidos por los accesos de datos en memoria de siguientes niveles, siendo estas L1, L2, L3 y RAM.

Efectivamente, podemos corroborar los resultados a través de `make info`:



Como dato extra, podemos ver que disponemos de dos niveles 1 de caché. Esto se debe a que hay un L1 para insstrucciones y otro L1 para datos.

Información de mi ordenador:



NOTA: Parte del razonamiento a la hora de elegir hacer la función and en vez de un if else viene influenciado de la siguiente publicación, motivado por la intención de encontrar la función que causase el menor impacto en rendimiento: https://stackoverflow.com/questions/12594208/c-program-to-determine-levels-size-of-cache