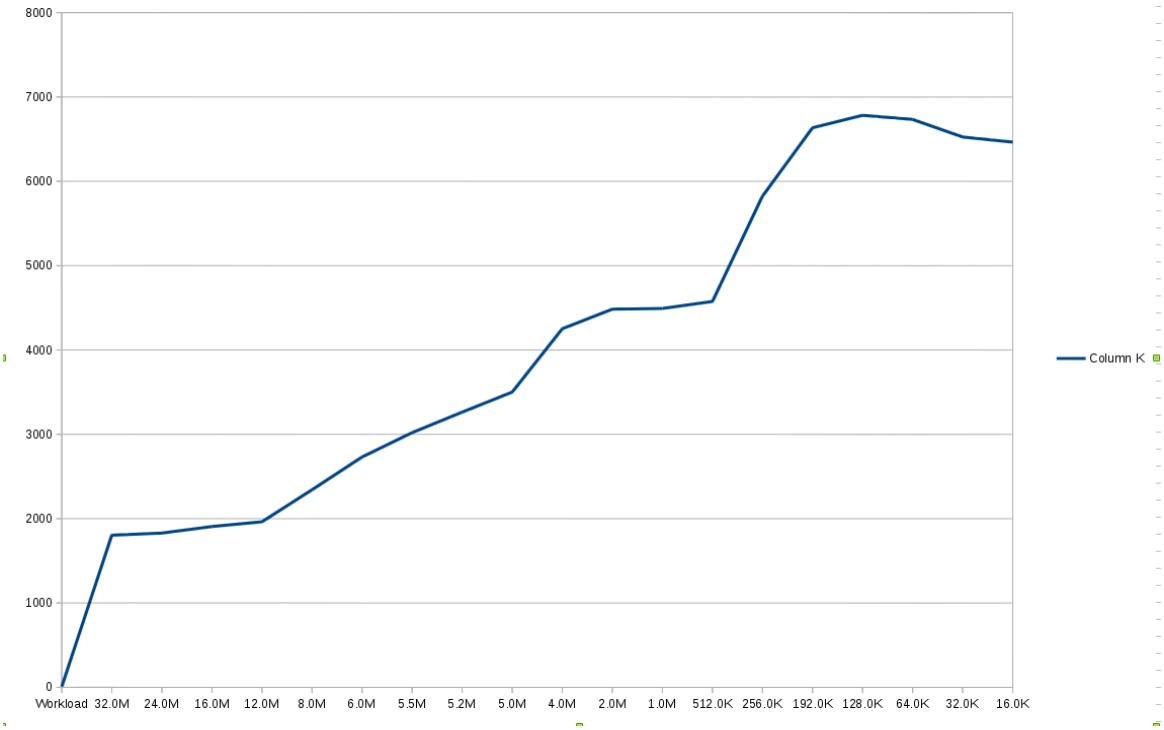
1 Getting knowledge of Our machine

Exercise 1: Think how you would make a program that helps you to figure out the number of caches and their approximate size. Explain it briefly. Haría recorridos con accesos a un array con un determinado stride, reutilizando valores, y iría incrementando el tamaño del array y midiendo el tiempo de acceso hasta ver el punto donde cambia, significando que el array no cabe en nivel actual de caché y algunos accesos tienen que acceder obligatoriamente al siguiente nivel de memoria.

Exercise 2: Think how you would make a program that helps you to figure out the cache line size of a certain level of cache. Explain it briefly. Declararía un array de tamaño potencia de 2, y iría accediendo a él con accesos a una cierta distancia en posiciones (stride). Primero, accedería a arr[0], arr[1], arr[2], luego arr[0], arr[2], arr[4], luego arr[0], arr[4], arr[8], etc. hasta observar una diferencia en el tiempo de acceso..

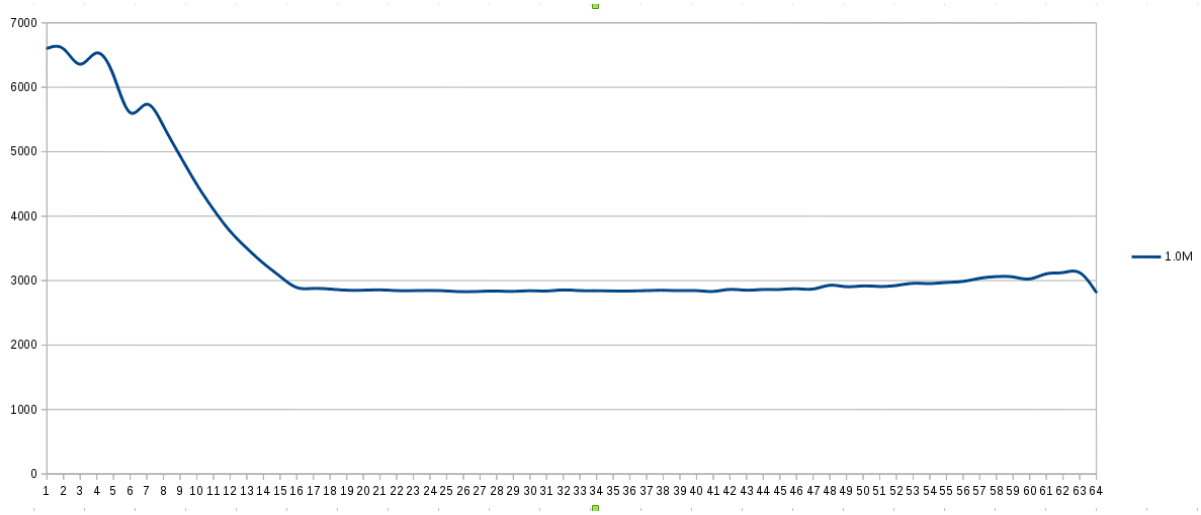
Exercise 3

Cache levels graph



Podemos ver claramente tres niveles de caché, el L1, que tiene un tamaño de **128KB** (según el gráfico, meseta del extremo derecho), el L2, que tiene un tamaño de **2MB**, y el L3, que tiene un tamaño de **5.5MB**.

Line sizes graph



En este gráfico podemos distingir una meseta en el bandwith que aproximadamiente empieza con stride=16, lo cual significa que el tamaño de la línea de caché es $16 * 4 =$ **64 Bytes**

Exercise 4

(a) El array `index` se va actualizando primeramente con el número de apariciones de cada uno de los posibles dígitos (`bits` bits altos) de todos los elementos de `S`. Los accesos se producen por lo tanto de forma no necesariamente secuencial, ya que el índice accedido que se incrementa depende del valor de los `bits` bits altos del elemento de S que se esté considerando.

Luego, para cada posición `i` del array se calcula la suma parcial de esa posición, es decir,

$$\sum_{j=0}^{i-1} S[j]$$

El array se accede de forma secuencial, por lo que hay un aprovechamiento de la localidad temporal y espacial.

Finalmente, se usan los valores de cada posición `i` del array `index` para calcular la posición en la que debe ir cada elemento de `S`, incrementando el valor de `index[i]` para el siguiente elemento que tenga el mismo dígito. Nuevamente los accessos son no necesariamente secuenciales tanto para el array `index` como para el array `D`, por la misma razón que en el primer recorrido.

(b) No se pide nada.

(c)
- `make distribution.3`
- `./distribution.3`

(d) Tal y como se puede justificar por los timings y el profiling (los resultados de los cuales están adjuntados en sus respectivos ficheros), el tiempo de ejecución augmenta significativamente cuando el número de bits es relativamente grande y los datos están distribuidos aleatoriamente.

Para justificarlo debemos tener en cuenta cual es la influencia de cada factor que cambiamos en las ejecuciones. Por el profiling tiene sentido afirmar que el augmento del tiempo de ejecución es debido

a los fallos de caché y TLB. Como ya hemos justificado en el apartado anterior, los únicos accesos que nos pueden estar provocando fallos en caché y TLB son los del segundo y último bucles que acceden a los arrays `index` y `D`, por lo que nos concentraremos en su análisis.

Si analizamos el factor de la distribución de los datos (ordenada/aleatoria), nos damos cuenta que la primera distribución favorece a una mayor localidad espacial y temporal en los accesos no necesariamente secuenciales, ya que cuando los datos están distribuidos según un orden el elemento S[i] tiene mayor probabilidad de tener los bits similares a los elementos S[i-1] y S[i+1] que si los datos están distribuidos aleatoriamente, y por lo tanto hay mayor probabilidad de acceder a una posición del array `index/D` cercana a una que hayamos accedido hace poco (localidad espacial), o incluso la misma (localidad temporal).

El hecho que los accesos no tengan un patrón de localidad espacial/temporal cuando los datos están distribuidos aleatoriamente hace que fallemos más en caché, y dado que estaremos accediendo a un número muy grande de direcciones de memoria diferentes en un lapso de tiempo relativamente corto, todas estas posiciones diferentes no cabrán en el TLB y por lo tanto tendremos fallos de TLB.

Notad que en la anterior justificación hemos dicho que estaremos accediendo a un número muy grande de direcciones de memoria, pero esto no es siempre cierto, ya que si el tamaño del array `index` es pequeño, independientemente del patrón de los accesos siempre tendremos localidad espacial y temporal, ya que el rango de direcciones posibles será pequeño y por lo tanto manejable por cualquier caché. Es aquí donde entra el segundo factor, el número de bits, y es que el tamaño del array `index` crece de forma exponencial respecto al número de bits por los que distribuimos los datos. Es por esta razón que cuando el número de bits es pequeño, incluso con los datos distribuidos aleatoriamente el tiempo de ejecución permanece bajo y similar a cuando los datos están distribuidos con un orden.

2 Data alignment

Exercise 5

(a)
- The sources negate an array of data with different types of accesses. Munge8 performs accesses of 1 byte at each iteration, Munge16 performs accesses of 2 bytes, Munge32 of 4 bytes and Munge64 of 8 bytes, and for each of them the accesses are performed with all possible alignments from 0 to 64.
- `make`
- `./munge_vectors8.2`
- `./munge_vectors16.2`
- ...

(b)

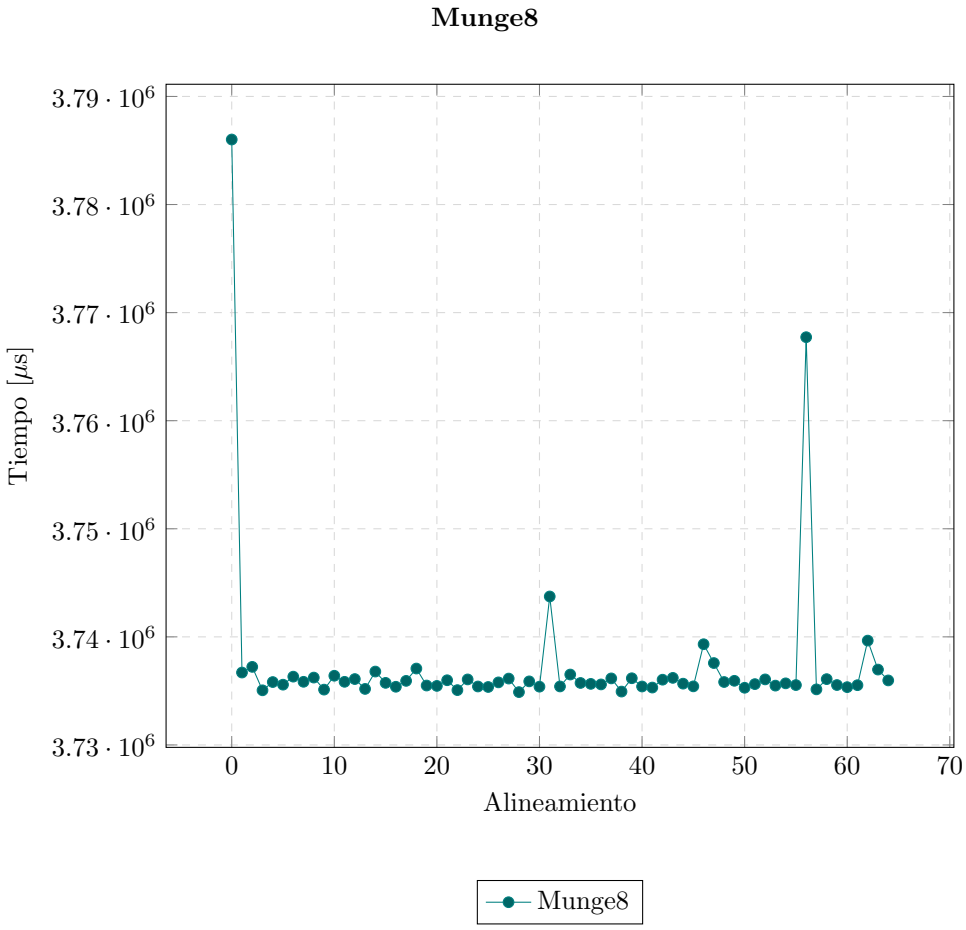


Figure 1: Gráfico del tiempo de ejecución de Munge8 en función del alineamiento

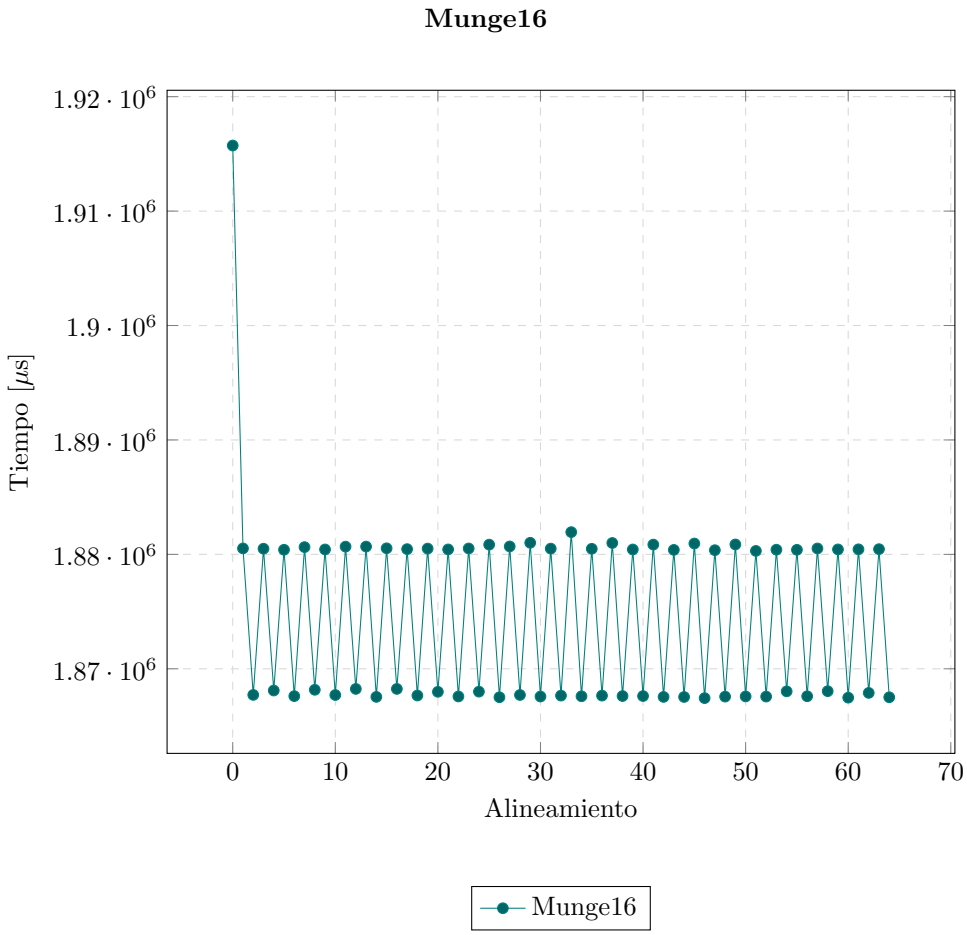


Figure 2: Gráfico del tiempo de ejecución de Munge16 en función del alineamiento

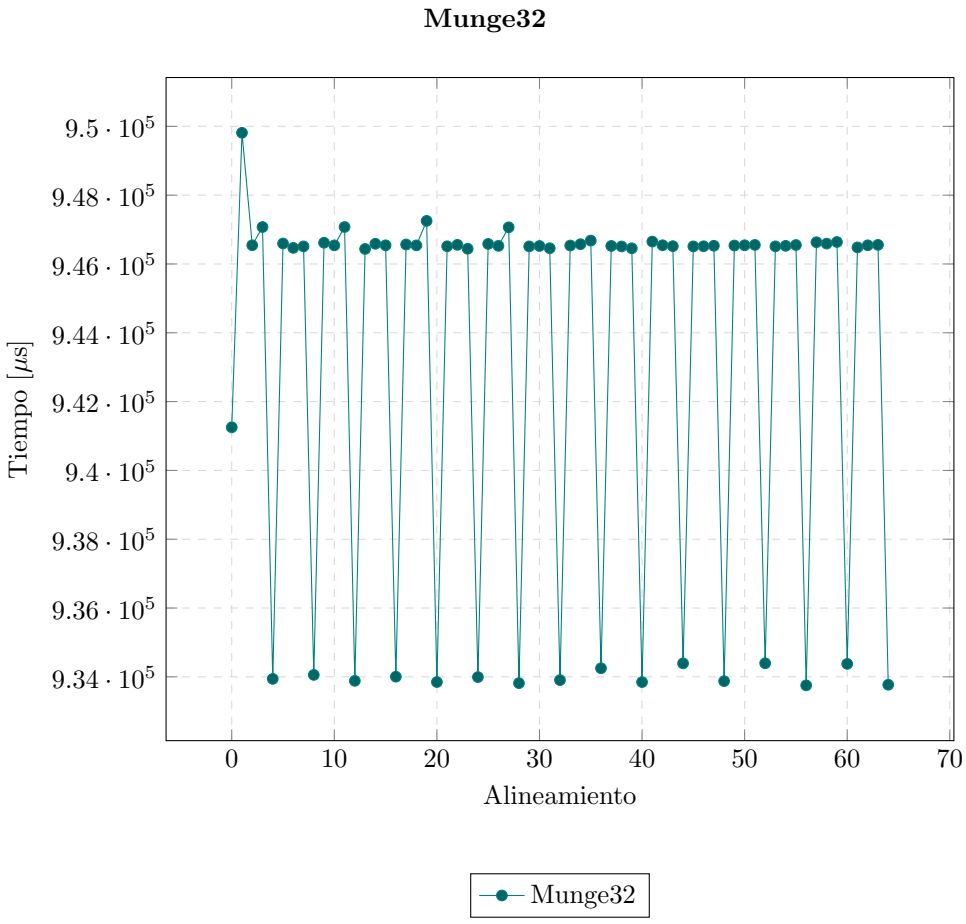


Figure 3: Gráfico del tiempo de ejecución de Munge32 en función del alineamiento

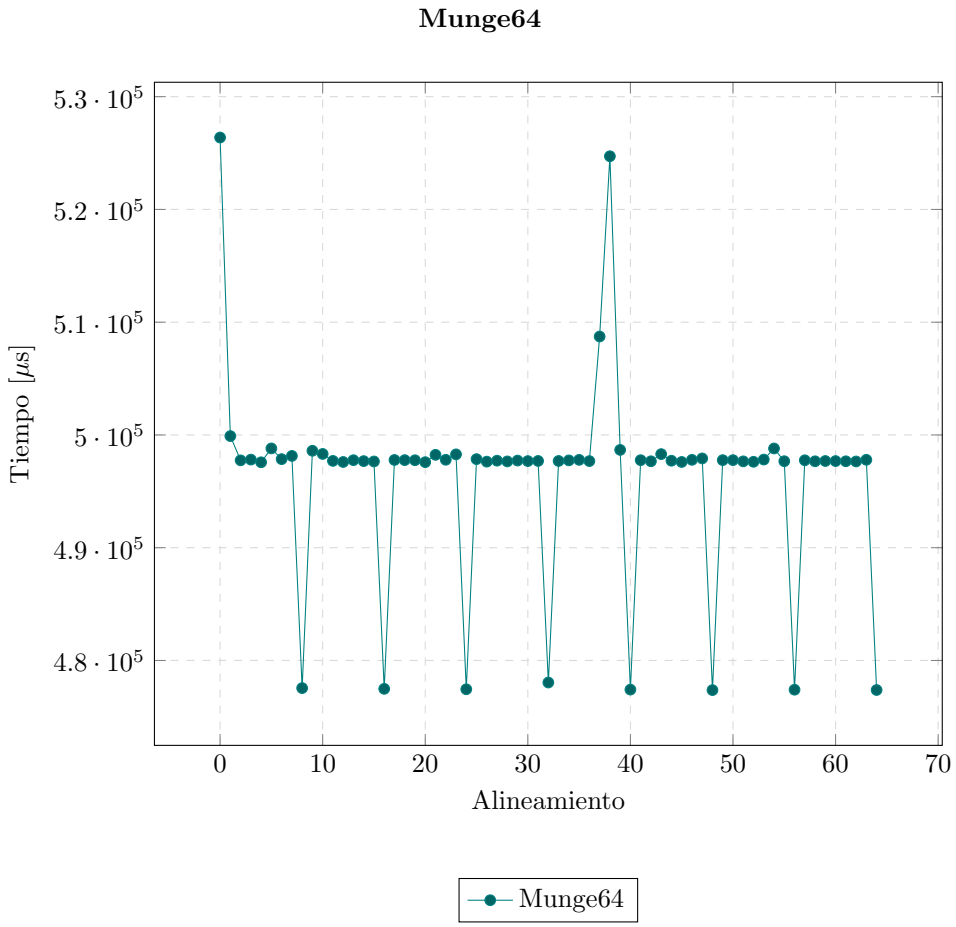


Figure 4: Gráfico del tiempo de ejecución de Munge64 en función del alineamiento

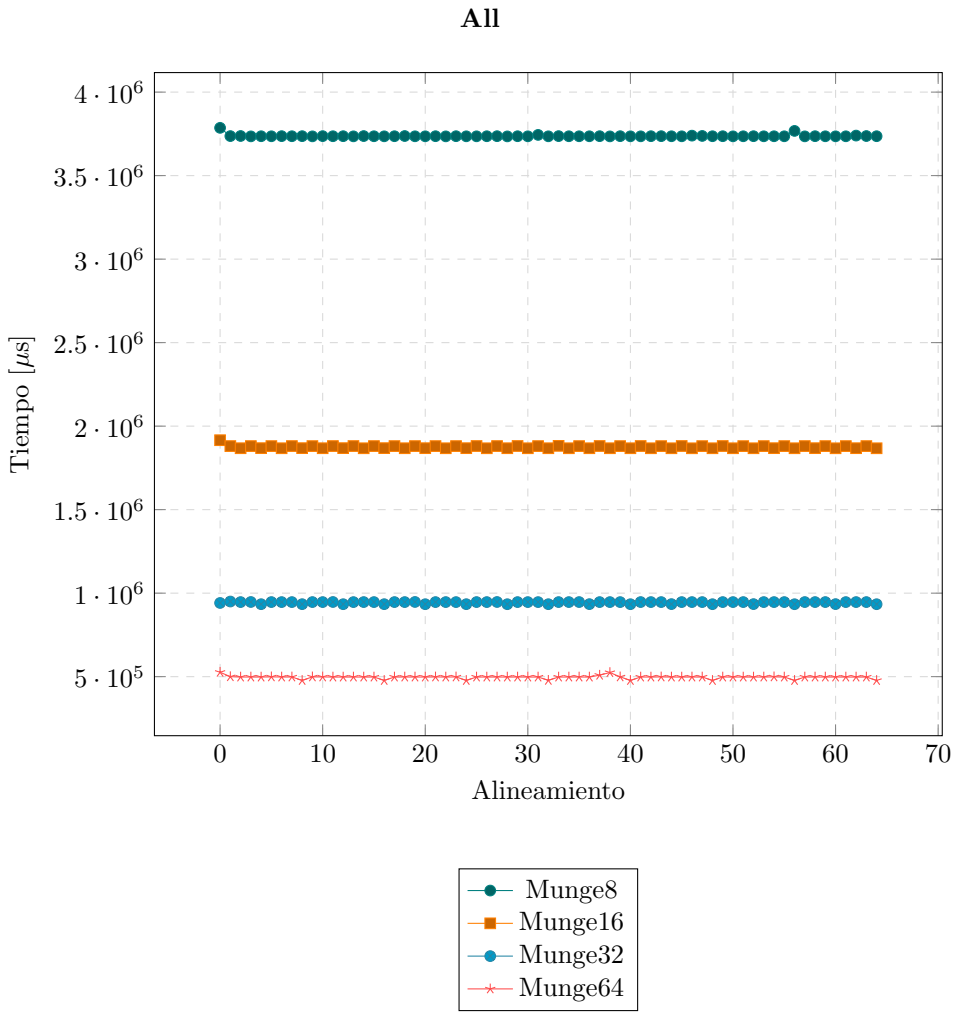


Figure 5: Gráfico del tiempo de ejecución de todas las versiones en función del alineamiento

(c) Nos damos cuenta que cuando el alignment es múltiple del tamaño en bytes de cada acceso, el tiempo de ejecución es menor. Esto es debido a que en la arquitectura que utilizamos los accesos que no son a direcciones múltiples del tamaño de acceso tienen un coste mayor.

3 Memory Bandwith

Exercise 6

(a)

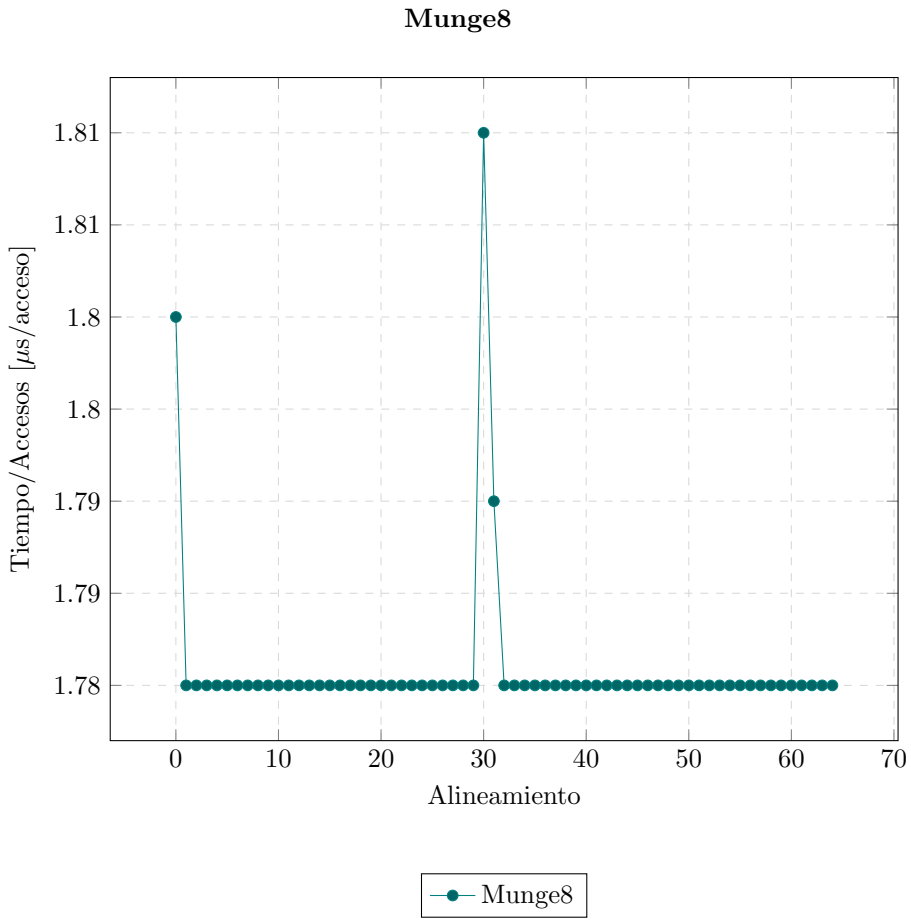


Figure 6: Gráfico del tiempo de ejecución por acceso de Munge8 en función del alineamiento

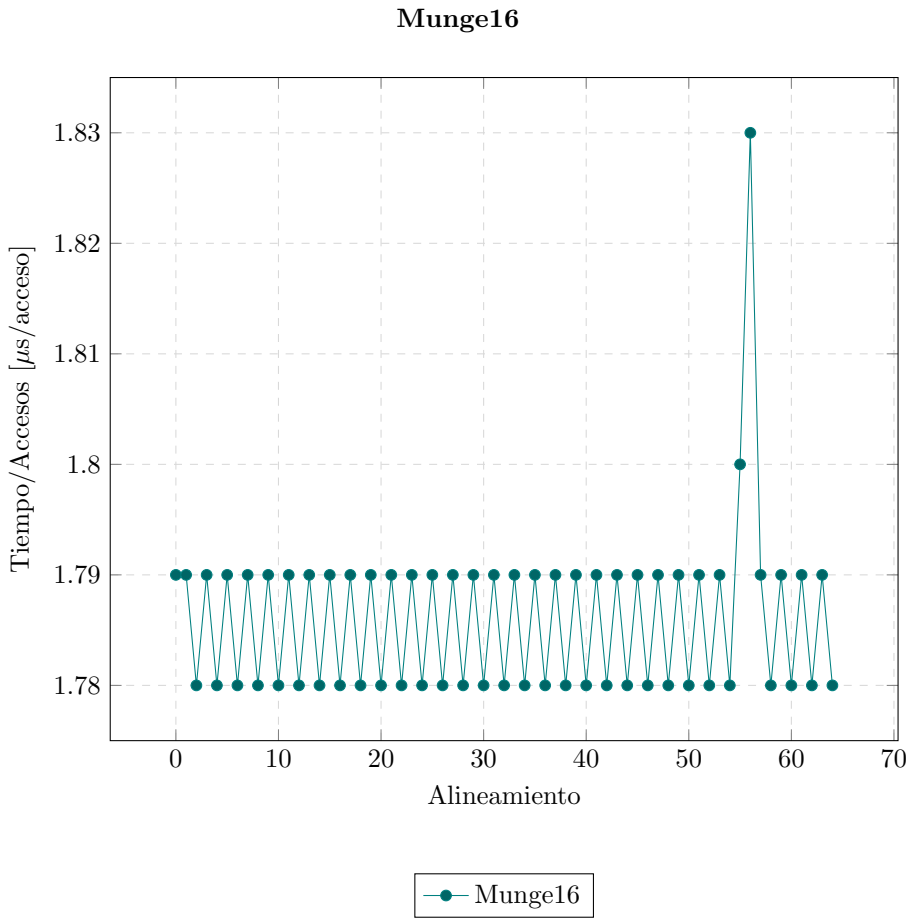


Figure 7: Gráfico del tiempo de ejecución por acceso de Munge16 en función del alineamiento

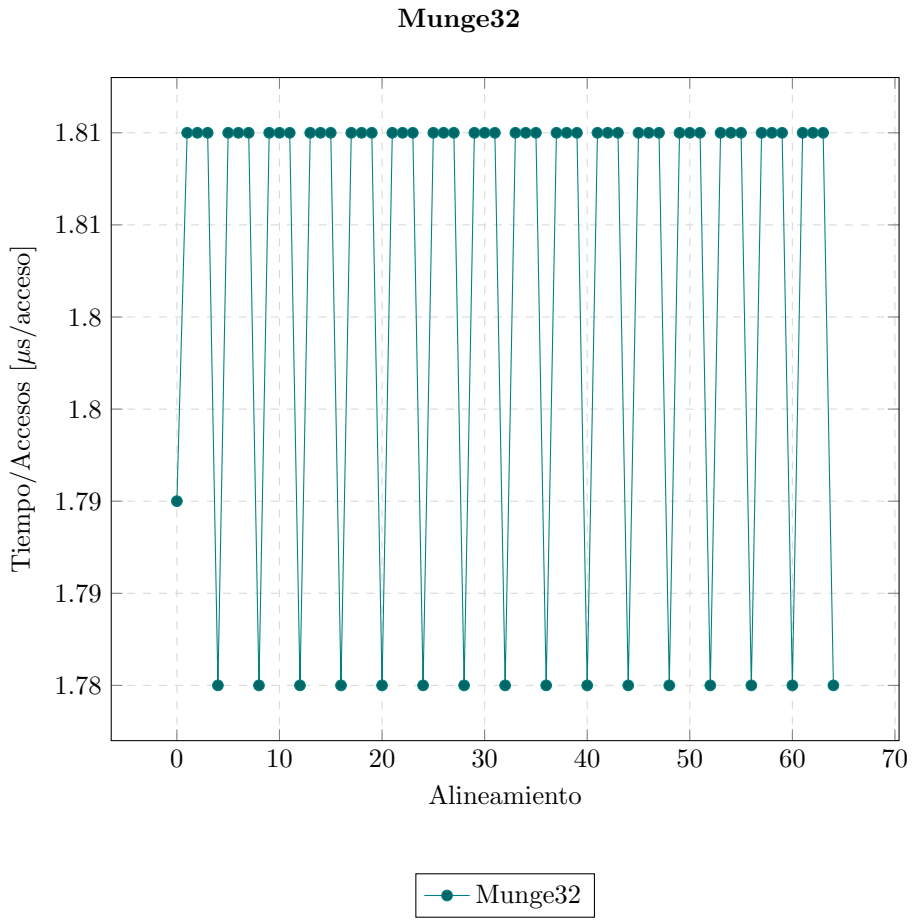


Figure 8: Gráfico del tiempo de ejecución por acceso de Munge32 en función del alineaminento

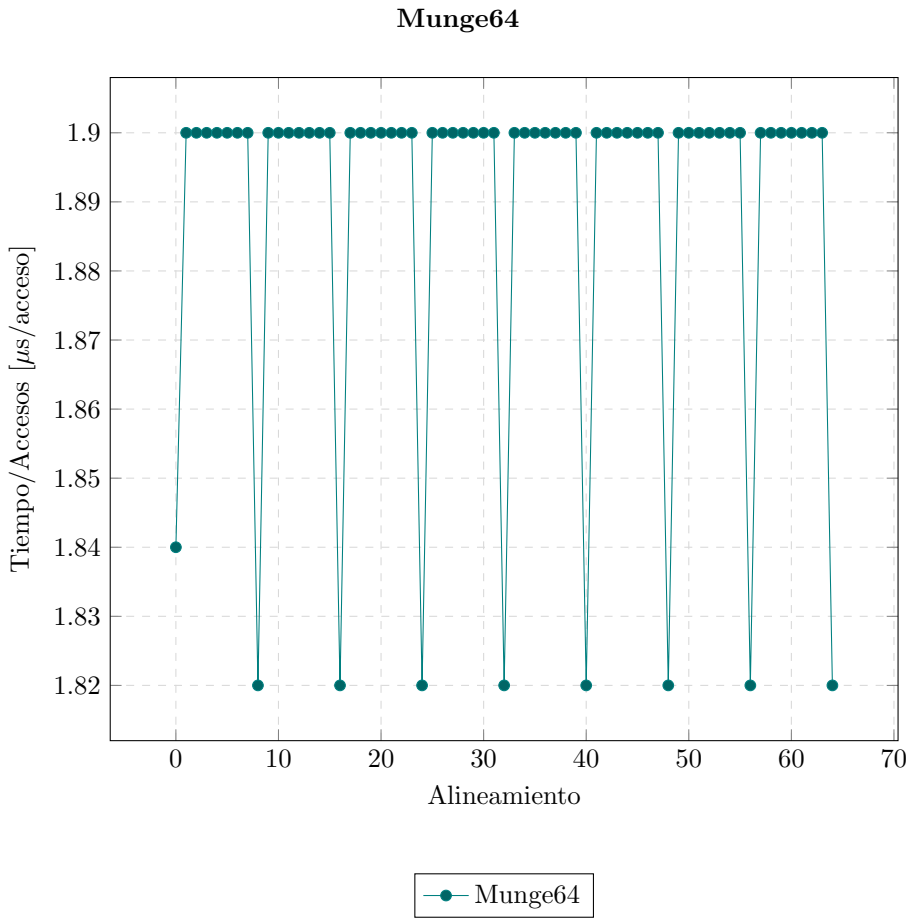


Figure 9: Gráfico del tiempo de ejecución por acceso de Munge64 en función del alineamiento

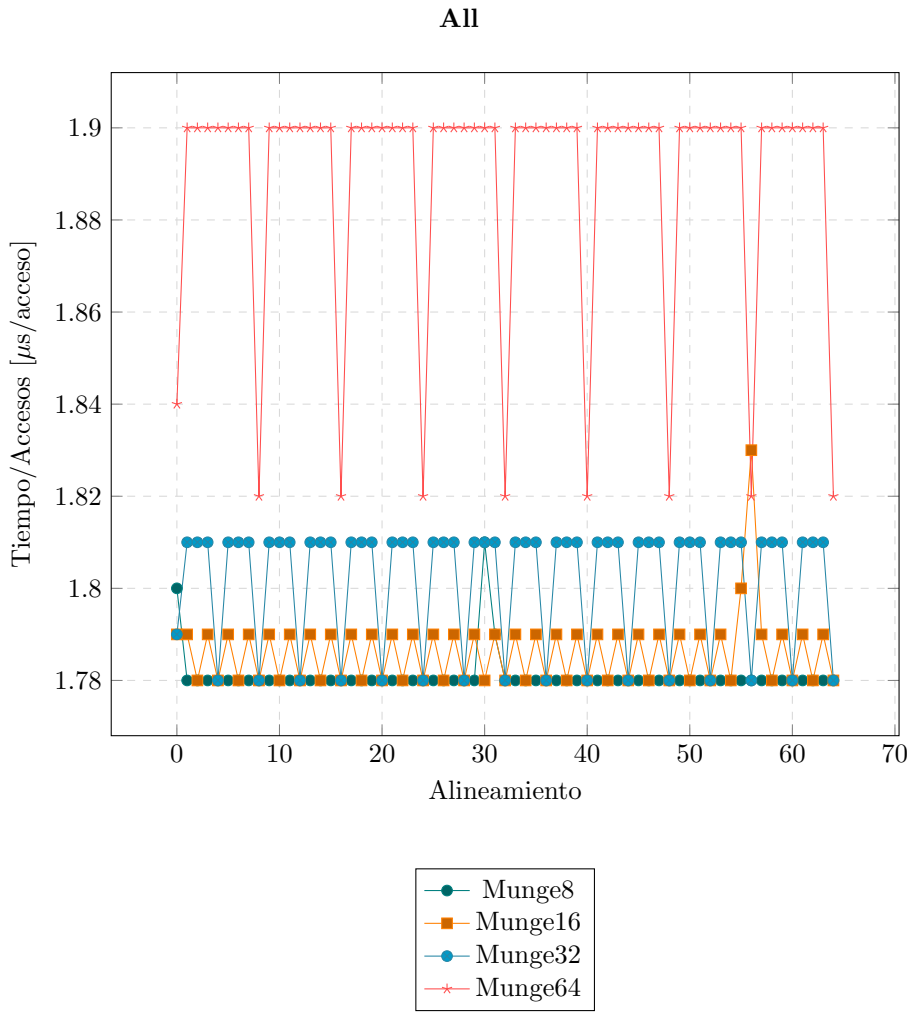


Figure 10: Gráfico de el tiempo de ejecución por acceso de todas las versiones en función del alineamiento

(b) Tal y como podemos ver en las gráficas, los accesos que aprovechan mejor el bandwith de memoria son los que tienen un tamaño de 64 bits, ya que el número de datos que movemos es el doble que en 32 bits y el tiempo por acceso es menor que el doble.

Mirando el código ensamblador, podemos ver como se ejecutan dos instruciones de tipo `mov` para cargar el valor y luego escribirlo:

```
400754:    0f b6 10          movzbl (%rax),%edx
400757:    f7 da             neg    %edx
400759:    48 39 d8          cmp    %rbx,%rax
40075c:    88 50 ff          mov    %dl,-0x1(%rax)
```

De esto y de los datos que hemos observado (tardamos 1.78 μs por cada par load/store para 64 bits) podemos deducir que el bandwith de los registros de propósito general es de $(8bytes)/(1.78*10-6s) = 4.5$ millones de accesos por segundo, aunque debemos tener en cuenta que el coste es aproximado ya que estamos contando también el coste del salto del bucle.

```
int main(int argc, char *argv[])
{
    uint64_t buffer[BUFFER_SIZE/8];
    int n1, i;

    while ((n1=read(0, &buffer, BUFFER_SIZE)) > 0)
    {
            //printf("%d   %d\n", strlen(buffer), n1);
        for (i = 0; i < n1/8; i++)
        {
                buffer[i] =   ((buffer[i] >> 8) & 0x00FF00FF00FF00FF)
                            | ((buffer[i] << 8) & 0xFF00FF00FF00FF00);
        }

        char* bff = (char*) buffer;
        char aux;

        for (i = n1 - n1%8; i < n1; i += 2)
        {
            aux = bff[i];
            bff[i] = bff[i+1];
            bff[i+1] = aux;
        }

        if (write(1, &buffer, n1) < 0) panic("writebuffer");
    }
}
```

La optimización consiste en leer y escribir el buffer por chunks de 8 bytes, teniendo en cuenta que puede que el número de bytes leídos (n1) no sea múltiple de 8.

4 Spatial Locality

Exercise 8

(a) No se pide nada

(b)

(c)
```
for ( i=0 ; i<ROWS; i++)
{
    for ( j=0 ; j<COLUMNS; j++)
    {
        buffer[i][j].r = 0;
        buffer[i][j].g = 1;
        buffer[i][j].b = 0;
    }
}
```

La optimización consiste en acceder a la matriz por filas en vez de por columnas, lo que augmenta la localidad espacial, dado que en C las matrices se guardan en memoria por filas.

(d)
```
for ( i=0 ; i<ROWS; i++)
{
  for ( j=0 ; j<COLUMNS; j++)
  {
    short* buffs = &buffer[i][j];
     *buffs = 0x0100;
     buffer[i][j].b = 0;
  }
}
```

La optimización consiste en escribir los dos bytes del red y el green en un solo acceso.

(e) Porque si miramos el código ensamblador, el compilador ya está haciendo la optimización con instrucciones SIMD.

5 empleat.c Optimizations

I've performed the two following optimizations:

- Buffering: He creado un buffer en el que voy escribiendo trozos del array de empleados para ahorrar muchas llamadas a sistema de `write`, escribiendo todo el buffer con una sola llamada.
- Aprovechamiento de la localidad espacial y temporal: He creado un nuevo array, `empleatsOrd`, en el que almaceno un struct con el indice y el id de cada empleado antes de ordenar. De este modo, cuando ordeno, en vez de ordenar el vector de empleados, en el que cada elemento ocupa mucho, ordeno el vector `empleatsOrd`, que tiene un tamaño mucho más pequeño, de modo que aprovecho mejor la localidad espacial. Finalmente, cuando toca escribir, recorro el array `empleatsOrd` (que estará ordenado como toca), y voy escribiendo el empleado del array de empleados que me indica el indice del elemento de `empleatsOrd` en cuestión.