LAB5 MEMORY OPTIMIZATION

1. Data Alignment

Compilación munge_vectors

Modifico el make para que funcione papi añadiendo la ruta de la librería y la del include

```
CFLAGS = -O2 -fno-inline -g -static -I/opt/install-arm/papi/include/
CFLAGS = -O2 -fno-inline -g -I/opt/install-arm/papi/include/
CC=gcc

all: munge_vectors8.2 munge_vectors16.2 munge_vectors32.2 munge_vectors64.2

munge_vectors8.2: munge_vectors8.c clock.c
	$(CC) $(CFLAGS) -o munge_vectors8.2 munge_vectors8.c clock.c -L/opt/install-arm/papi/lib/ -lpapi

.....
#Hacemos lo mismo con los demás ejecutables
```

Ejecución munge_vectors

Creo un script **execute.sh** que ejecuta los cuatro ejecutables y guarda los outputs en la carpeta Outs creada previamente

```
#! /bin/bash

./munge_vectors8.2 > out8
./munge_vectors16.2 > out16
./munge_vectors32.2 > out32
./munge_vectors64.2 > out64

mv out* Outs/
```

Además de un script **times.sh** para calcular los tiempos de cada ejecutables con **time** y guardarlos en una carpeta dentro de la carpeta Times

```
#! /bin/bash

/usr/bin/time -p -o time00 ./$1 > /dev/null
/usr/bin/time -p -o time01 ./$1 > /dev/null
/usr/bin/time -p -o time02 ./$1 > /dev/null
/usr/bin/time -p -o time03 ./$1 > /dev/null
/usr/bin/time -p -o time04 ./$1 > /dev/null

mkdir Times/$2
mv time0* Times/$2/
```

Enlaces a los times: **mv8** **mv16** **mv32** **mv64**

Enlaces a los resultados de cada ejecución: **out8** , **out16** , **out32** , **out64** :

En la siguiente tabla se puede ver algunos resultados

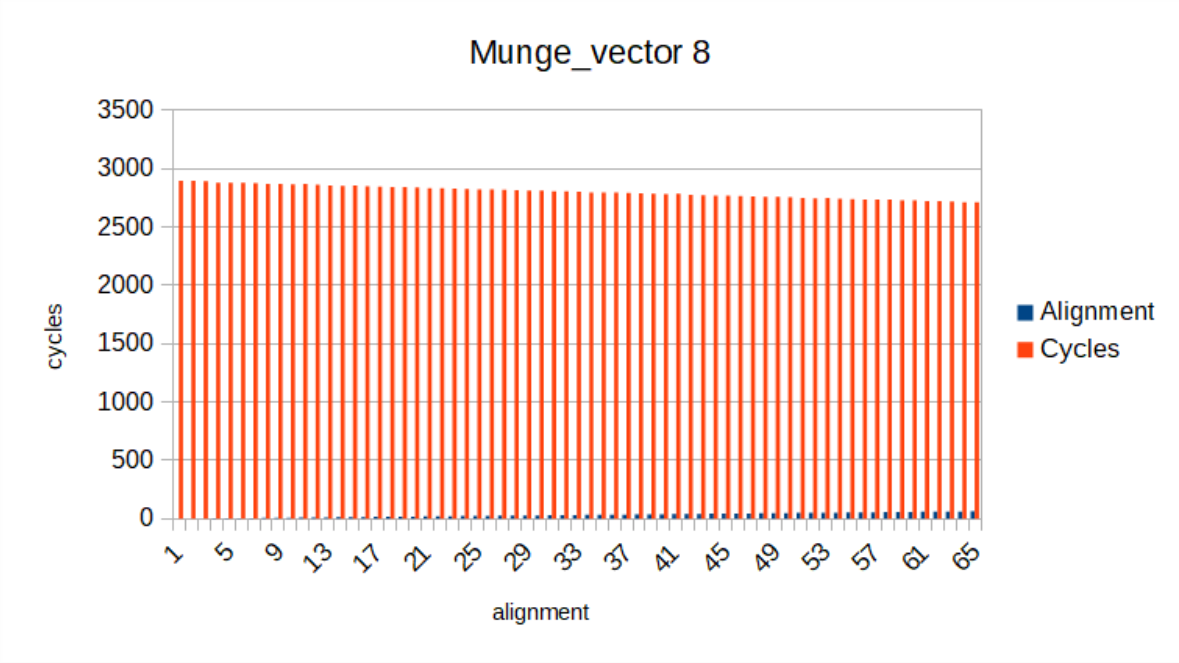
Alignment	8	16	32	64
0	2896.02	1461.58	745.39	661.58
1	2896.96	3481.37	2223.28	210035.63
2	2894.30	1465.30	2202.47	210107.59
3	2881.73	3347.97	2211.47	210120.79
4	2881.36	1455.73	739.59	673.42
5	2881.01	3466.35	2191.81	210021.70
6	2877.27	1453.16	2228.90	210118.56
7	2870.53	3352.16	2183.42	210113.30
52	2749.81	1390.95	706.85	642.25
53	2742.45	3300.27	2087.05	200176.68
54	2741.03	1385.26	2124.19	200172.93
55	2737.12	3171.94	2081.22	200179.16
56	2735.84	1381.05	705.20	632.12
57	2737.22	3285.19	2098.64	198551.48
58	2729.94	1377.64	2077.19	198530.07
59	2729.01	3163.20	2115.09	198525.45
60	2723.30	1377.19	697.50	634.77

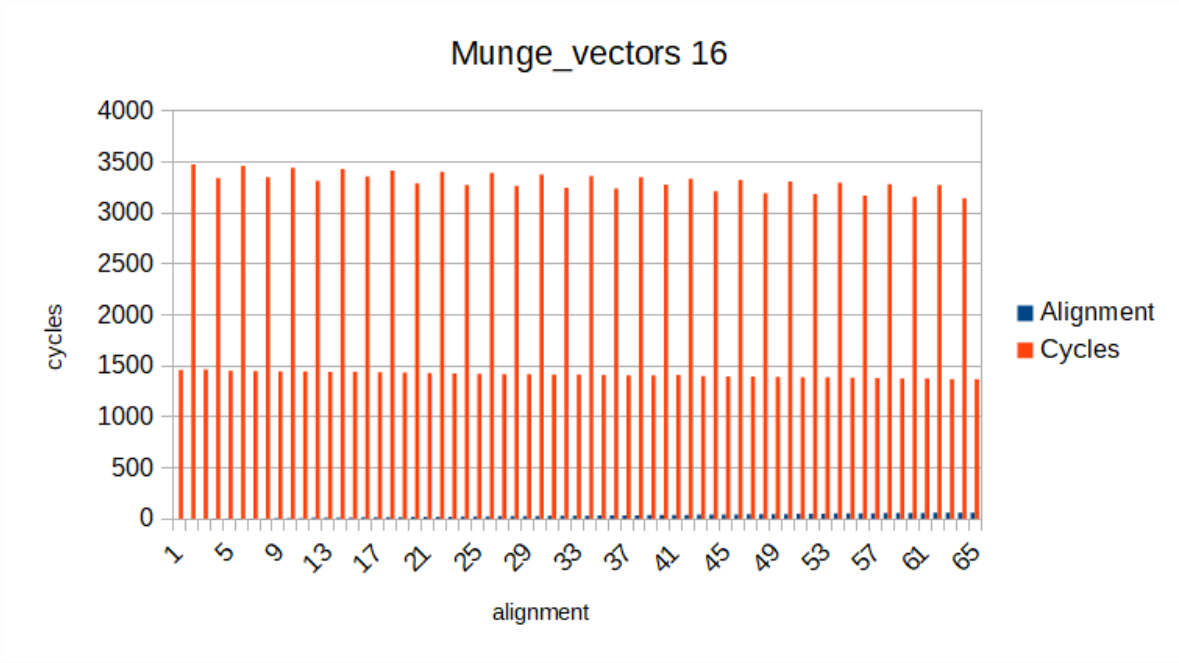
Comportamiento munge_vectors

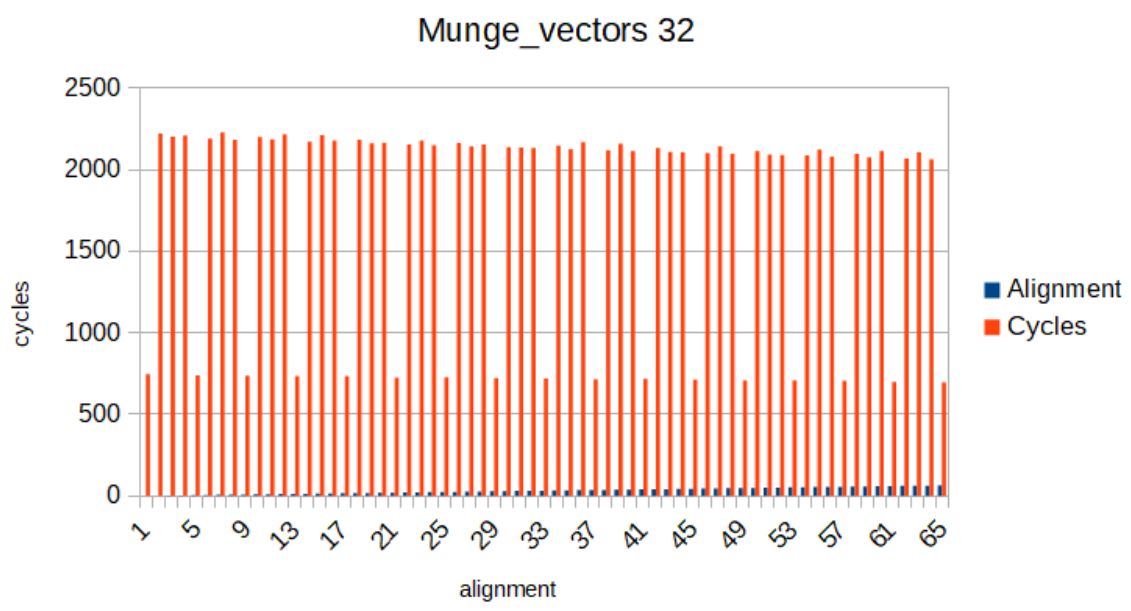
Dado un puntero al primer elemento de un vector y alienado a una dirección x, hacemos accesos de 1B, 2B, 4B y 8B, para munge8, munge16, munge32 y munge64 respectivamente. Y lo hacemos con un alienado diferente cada vez que recorremos todo el vector. El alineado puede ser de x + 0 hasta x + 64. Y por cada vez que recorremos todo los datos con un alineado en particular, obtenemos los ciclos que ha tardado

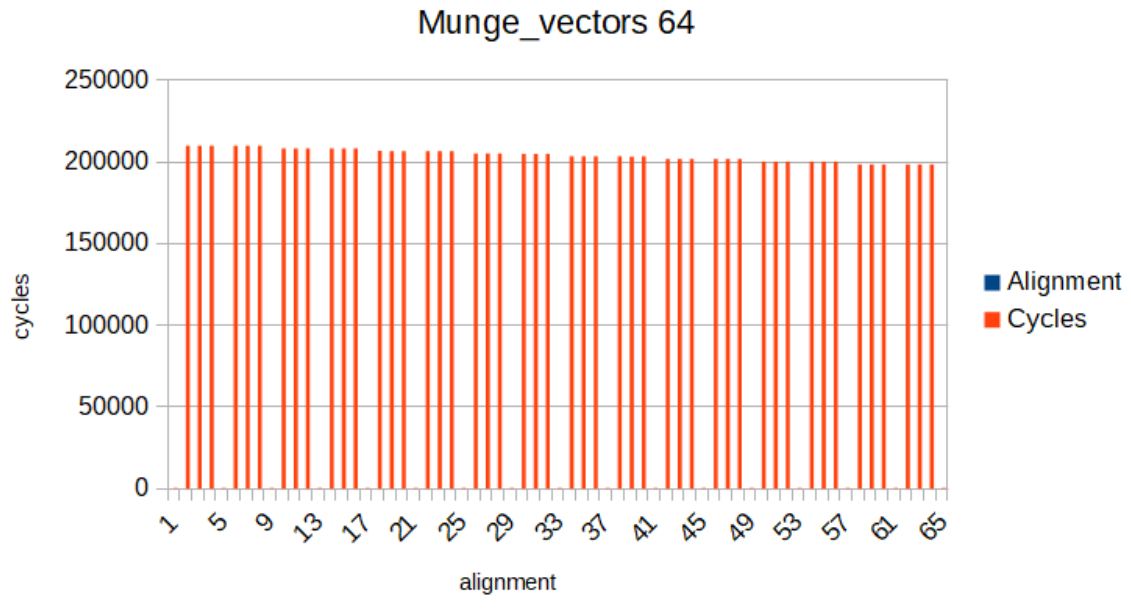
Gráficas

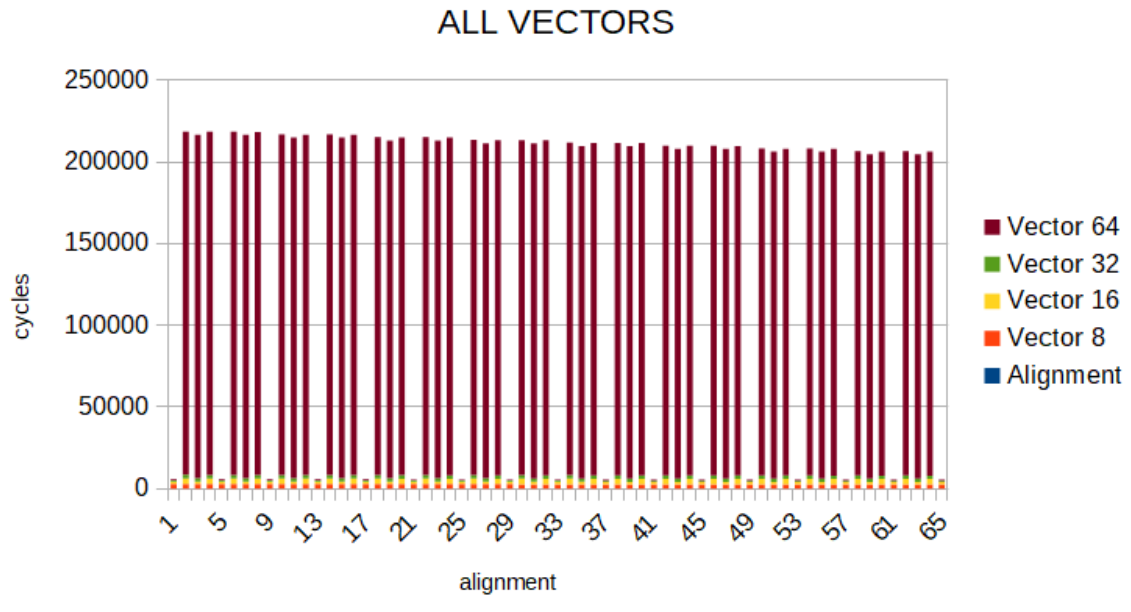
Las gráficas obtenidas como alignment como eje x y los ciclos como eje y, son las siguientes:






Munge_vectors 32
2500
2000
1500
1000
500
0
cycles
1
5
9
13
17
21
25
29
33
37
41
45
49
53
57
61
65
alignment
Alignment
Cycles


Munge_vectors 64
250000
200000
150000
100000
50000
0
cycles
1
5
9
13
17
21
25
29
33
37
41
45
49
53
57
61
65
alignment
Alignment
Cycles


ALL VECTORS
250000
200000
150000
100000
50000
0
cycles
1
5
9
13
17
21
25
29
33
37
41
45
49
53
57
61
65
alignment
Vector 64
Vector 32
Vector 16
Vector 8
Alignment

Análisis

Análisis Outputs

Como hemos visto que para el caso de munge_vectors64.2 pasa mucho tiempo de sistema, por lo que hemos visto en su comportamiento a los diferentes niveles de la jerarquía de memoria efectúa la mayor parte del tiempo en llamadas de sistema para la gestión de memoria. Por lo que lo tomaremos por un caso especial. La idea es observar los demás casos y ver lo que sucede. Como vemos, para accesos de 1 byte, el alineado es indeferente pues cualquier dirección es divisible por 1. Para accesos de 2 bytes, los accesos son más rápidos cuando estos se hacen en una dirección alineada a un número divisible por 2, es decir, todos los alineamentos pares. Finalmente, para 4 bytes, los accesos son más rápidos cuando los alineamentos son divisibles por 4.

Conclusión

Cuando los datos son de un tamaño 2^n y están alineados a una dirección divisible por 2^n, obtenemos mejor rendimiento. Y se debe a que cuando los datos no se alinean, un mismo elemento puede, por ejemplo,encontrarse en dos líneas de cache pudiendo estar en una. Y provocando más accesos a cache y generando peor rendimiento.

2. Memory Bandwidth

Munge_vectors

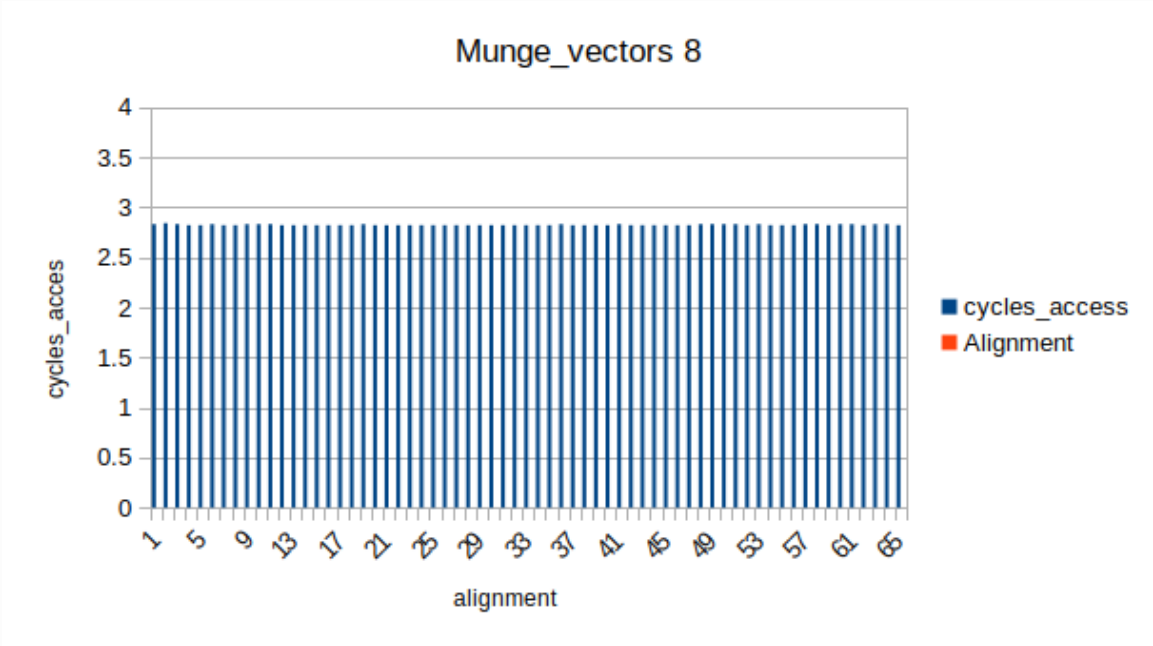
Descomentamos para cada fichero.c de munge_vectors la linea de código correspondiente, ejecutamos el script **execute.sh** después de compilar.

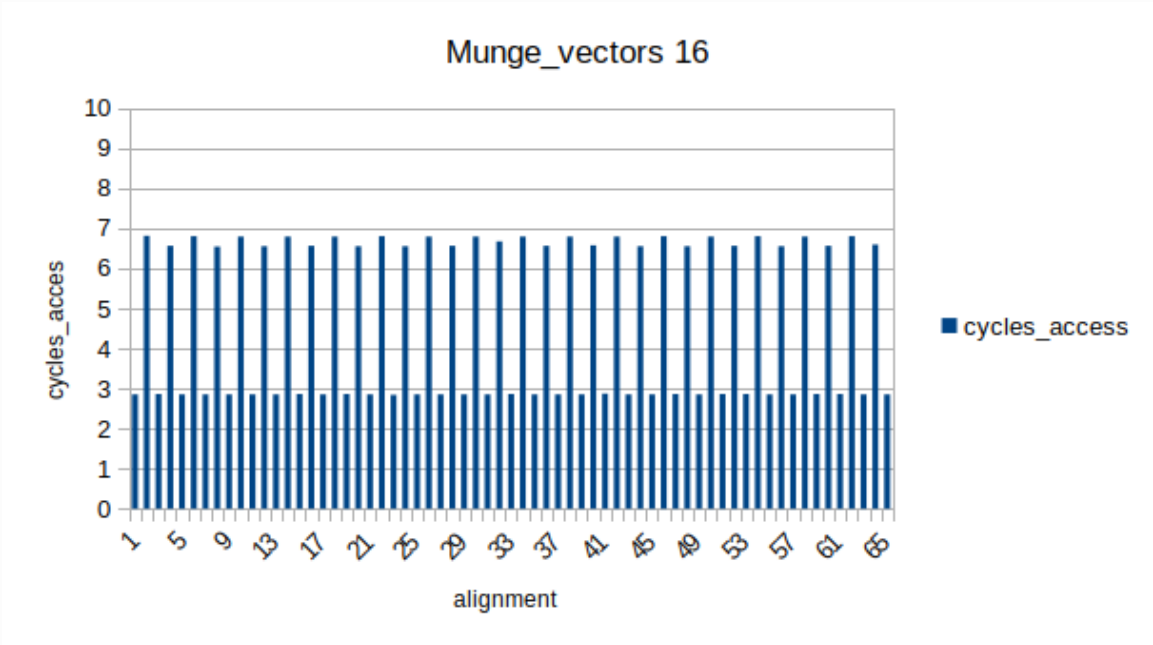
Análisis Output

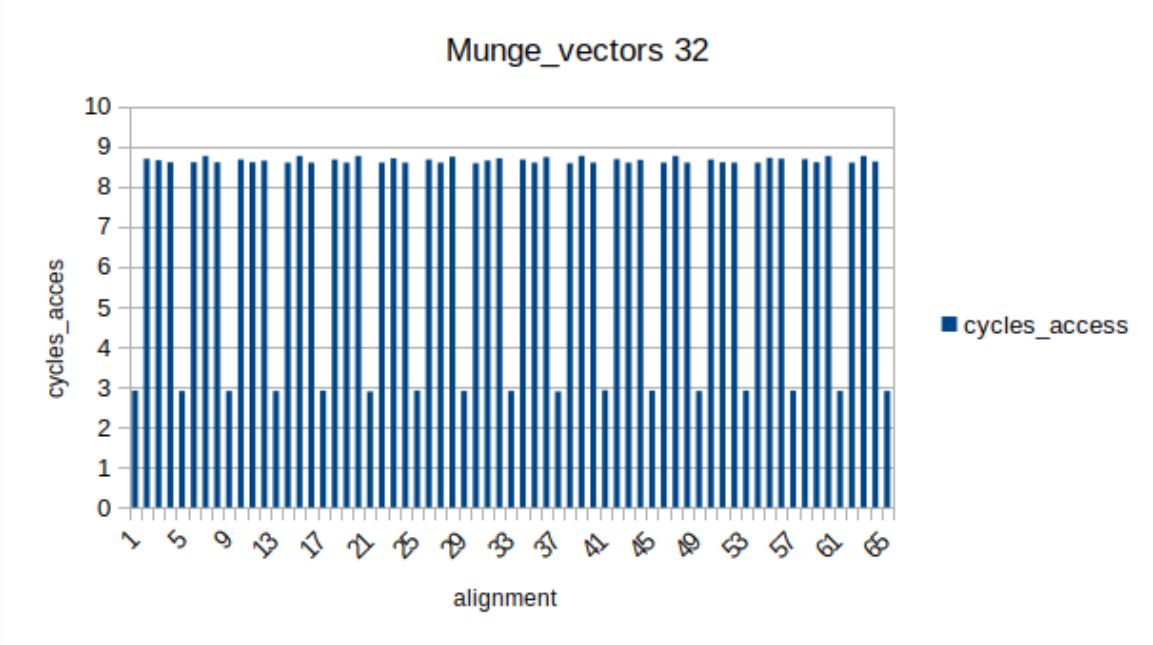
Enlaces Outputs Cycles_access: **out8** **out16** **out32**

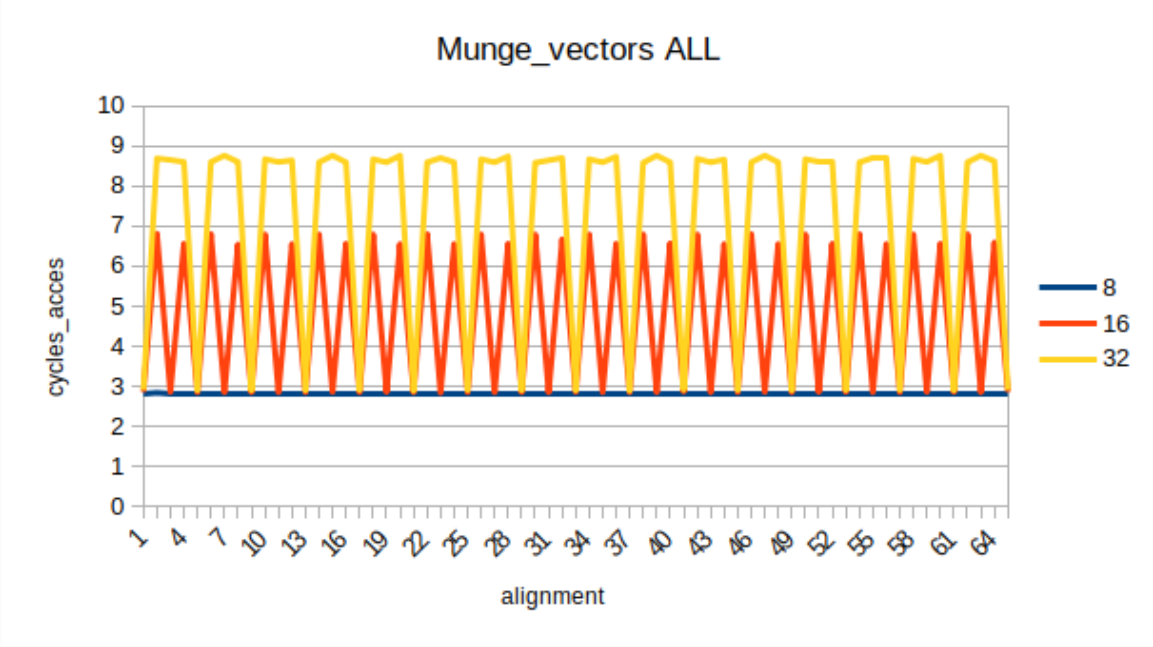
Obtenemos que para los casos en los que los datos están alineados correctamente, el tiempo de ciclo por acceso es de 2,30 - 2,80 para los casos de munge8, munge16, munge32 (munge64 no lo tomo en cuenta), pero cuando no están alineados correctamente, vemos que tarda más ciclos por acceso. Aun así, cuando mayor son los datos accedidos, obtenemos un mejora de ancho de banda, pues en beneficio hacemos menos accesos. Para el caso de munge64 sería contraproducente siendo la Cortex A9 una arquitectura de 32 bits, pues para hacer accesos de 8B habría que acceder 2 veces.

Gráficas


Munge_vectors 8
cycles_acces
4
3.5
3
2.5
2
1.5
1
0.5
0
1 5 9 13 17 21 25 29 33 37 41 45 49 53 57 61 65
alignment
cycles_access
Alignment


Munge_vectors 16
cycles_acces
10
9
8
7
6
5
4
3
2
1
0
1 5 9 13 17 21 25 29 33 37 41 45 49 53 57 61 65
alignment
cycles_access


Munge_vectors 32
cycles_acces
10
9
8
7
6
5
4
3
2
1
0
1 5 9 13 17 21 25 29 33 37 41 45 49 53 57 61 65
alignment
cycles_access



Conclusión

Vemos que si el acceso es de un tamaño mayor (sin pasarnos) y además estos datos están alineados, obtenemos un mejor rendimiento aprovechando un ancho de banda óptimo, debido a la disminución de los ciclos por acceso. De que depende esto? De la arquitecturas

La mejor opción es munge32 pues siendo una arquitectura de 32 bits, es la que mejor aprovecha el ancho de banda con un alineamento adecuado.

Swap

Archivo Optimizado: **swap.c**

Partiendo de lo visto hasta ahora, el read del swap tendría que leer 4B en vez de 1B a cada iteración, aprovechando así el ancho de banda. Utilizo un file.txt con <file.txt para el fread

```
1  /*READ VALUES*/
2  if ((fread(&buff,1,512,stdin)) < 512) panic("read");
3
4  /* WRITE VALUES INT BY INT */
5  for(int i=0; i<512; i+=4)
6  {
7  esc = *((unsigned int*) &buff[i]); //Load de 4 bytes
8  esc = ((esc << 8) & 0xFF00FF00) | ((esc >> 8) & 0x00FF00FF); //Swap bit hack
9  if (fwrite(&esc,1,4,stdout) < 0) panic("write1"); //Write del swap generado
10 }
11
12 return 0;
```

3. Spatial Locality

Pi

Teniendo en cuenta que no esta bien el unroll del lab4 :) la estructura de datos creo que ya la hice bien (espero), el código es este: **pi2Fusion.c**

Y el fragmento de mis estructuras esta aquí:

```
struct uDATA{
   unsigned q;
   unsigned r;
};
struct uDATA div5[50];
struct uDATA div25[250];
struct uDATA div239[2390];
void create_hashTable(){
   unsigned i=0;
   for(int u=0;u<50;u+=5){
      div5[u].q = i;
      div5[u+1].q = i;
      div5[u+2].q = i;
      div5[u+3].q = i;
      div5[u+4].q = i;
      ++i;
      div5[u].r = 0;
      div5[u+1].r = 1;
      div5[u+2].r = 2;
      div5[u+3].r = 3;
      div5[u+4].r = 4;
   }
   int u=0;
   for(i=0;i<10;i++){
      int N=25+(25*i);
      unsigned res=0;
      for(u;u<N;u++){
         div25[u].q=i;
         div25[u].r=res;
         ++res;
      }
   }
   u=0;
   for(i=0;i<10;i++){
      int N=239+(239*i);
      int res=0;
      for(u;u<N;u++){
         div239[u].q=i;
         div239[u].r=res;
         ++res;
      }
   }
}
```

Empleats

CPW

Cálculo los CPW con la ayuda del script **timeEmpleats.sh** sabiendo que la frecuencia es de 1 Ghz. Lo podía haber calculado en el propio script pero me petaba aún siguiendo tutoriales. Así que después de obtener los time elapsed de cada valor de N, he calculado la frecuencia en el calc de LibreOffice (aprovechando para hacer la gráfica). He calculado hasta 400000 para que no me saliera ningún error en algunos casos y en intervalos de 20000, para tiempos más diferentes de un número a otro, con menos resultados.

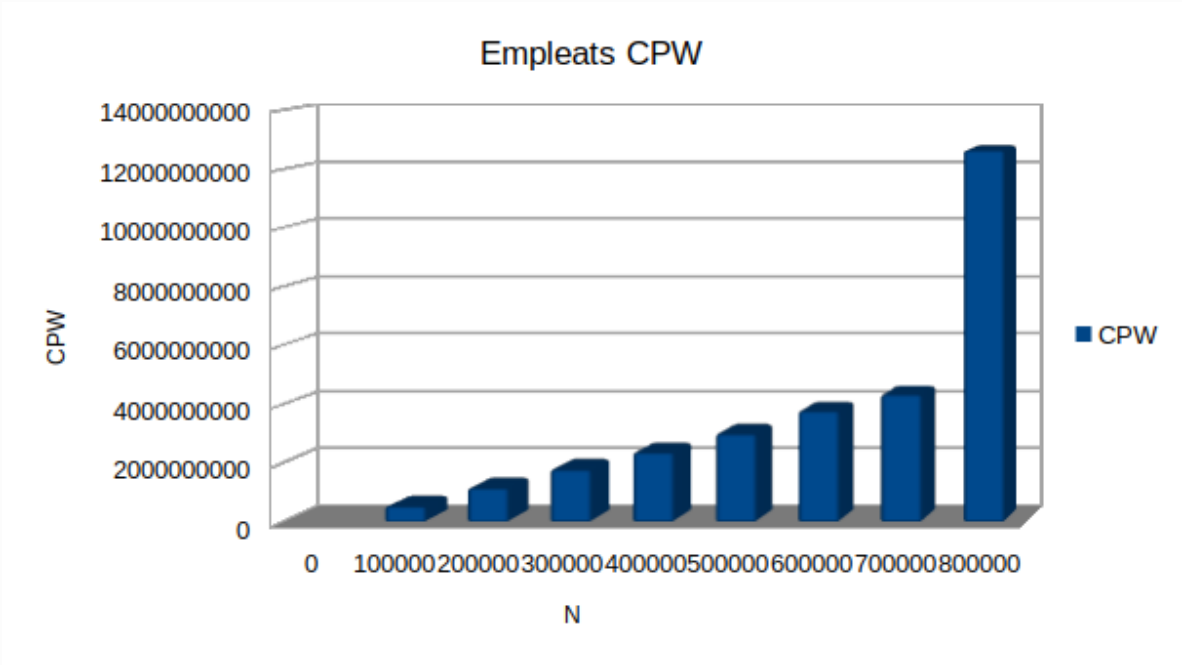
Código Script **timeEmpleats.sh**:

```
#!/bin/bash

for i in {0..400000..20000}
do
	/usr/bin/time -f "%e" ./empleats.3 $i > /dev/null
done
```

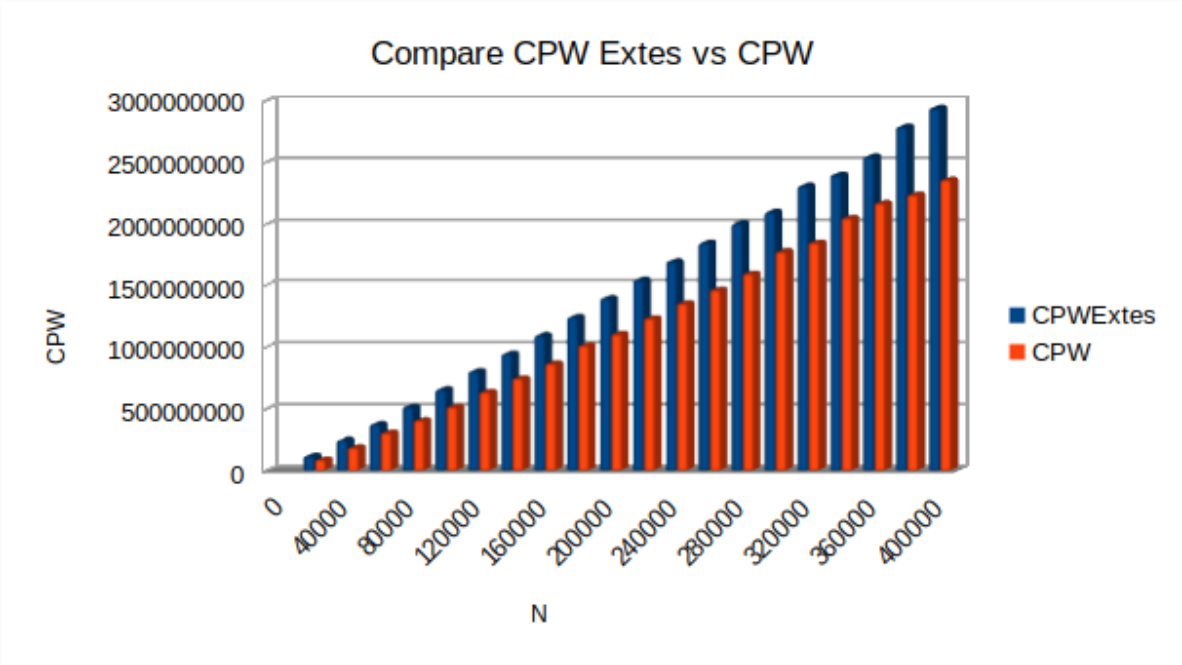
Todos los resultados: CPW Extes y CPW compilados con -O0 y los demás con -03

CPWExtes	CPW	CPW OPTI	CPW 3
0	0	0	0
0.12	0.09	0.06	0.09
0.25	0.19	0.12	0.18
0.38	0.31	0.18	0.29
0.52	0.41	0.25	0.39
0.66	0.52	0.31	0.5
0.81	0.64	0.38	0.61
0.95	0.75	0.45	0.72
1.1	0.87	0.52	0.84
1.25	1.02	0.59	0.95
1.4	1.11	0.66	1.06
1.55	1.24	0.73	1.18
1.7	1.36	0.8	1.29
1.85	1.47	0.86	1.41
2.01	1.6	0.94	1.53
2.1	1.78	1.01	1.65
2.31	1.85	1.08	1.78
2.4	2.05	1.15	1.89
2.55	2.17	1.23	2.02
2.79	2.24	1.29	2.14
2.94	2.36	1.36	2.26

Gráficas



He cambiado el rango de N para comparar los resultados con #define EXTES porque sino salia fuera de rango.



Como era de esperar, al guardar más datos para cada empleado, tarda más en ordenarlos debido a que realiza más accesos a cache , pues a mayor datos mayor son las líneas de cache distintas utilizadas a la hora de acceder a los elementos de un empleado.

Optimización

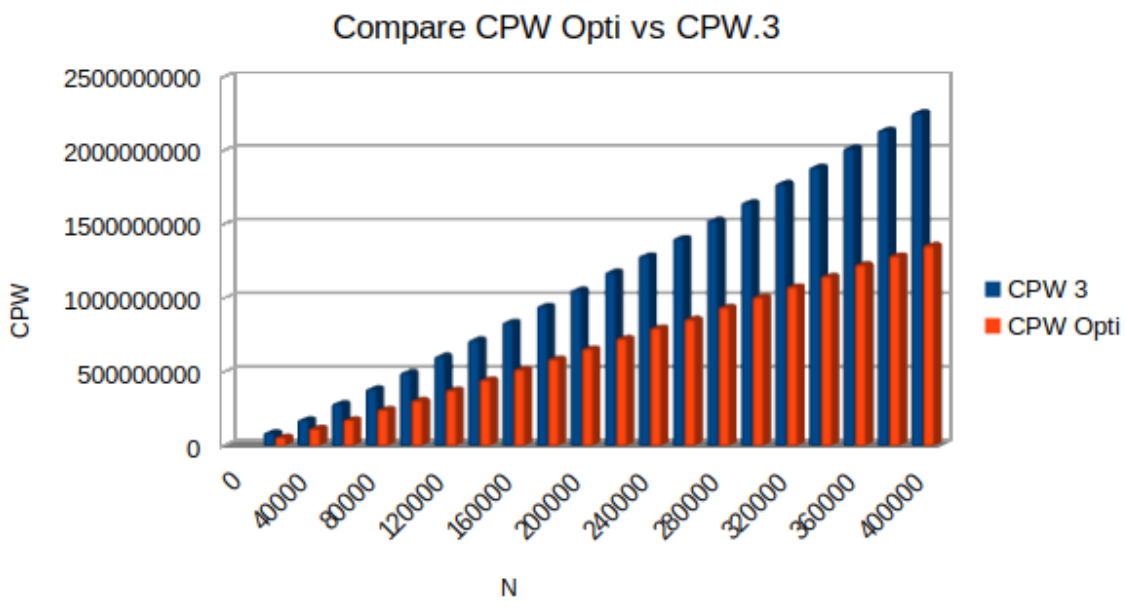
Hemos declarado una nueva estructura con el NID del empleado y un puntero a este. Esta nueva estructura es la que se utilizará para reordenar los empleados y con ella sabremos el orden para escribirlos.

Archivo del código optimizado: **empleatsOpti.c**

Código añadido:

```
...
typedef struct {
  long long int NID;
  Templeat *empls;
} IDEmpl;
...
IDEmpl *idenEmpleats;
...
idenEmpleats = (IDEmpl *) malloc(N*sizeof(IDEmpl));
if (idenEmpleats == NULL) { fprintf(stderr, "Out of memory 2\n"); exit(0); }
memset(idenEmpleats, 0, N*sizeof(IDEmpl));
...
for
...
    idenEmpleats[i].NID = empleats[i].NID;
    idenEmpleats[i].empls=&empleats[i];
...
qsort(idenEmpleats, N, sizeof(IDEmpl), compare);
for (i=0; i<N; i++){
        write(1,idenEmpleats[i].empls,sizeof(Templeat));
}
```

Gráfica Obtenida (con Optimización -03) para el código original y el optimizado:



4. Temporal Locality

El timing del código original y los del optimizado con diferentes BS lo he calculado con el script **timesBS.sh** y los outputs están aquí **Original.txt 2.txt 4.txt 8.txt 16.txt 32.txt 64.txt 128.txt 256.txt 512.txt**

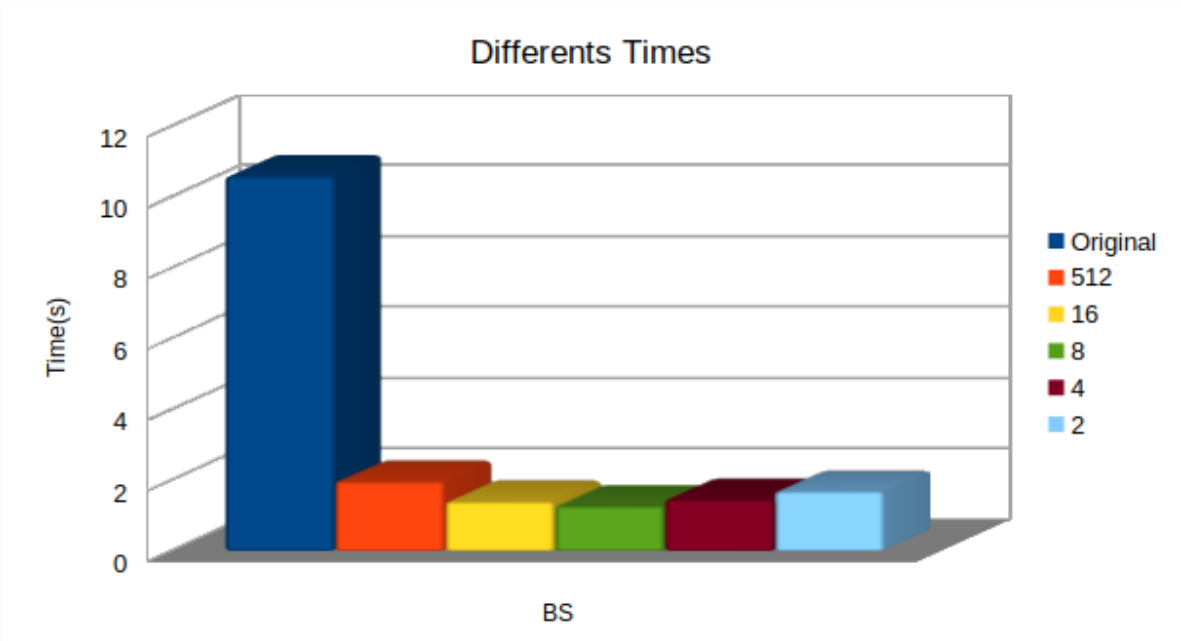
Script **timesBS.sh**:

```
#! /bin/bash

gcc -DBS=$1 -g3 -fno-inline mult1.c -O3  -o m$1


/usr/bin/time -p -o time00 ./m$1 > /dev/null
/usr/bin/time -p -o time01 ./m$1 > /dev/null
/usr/bin/time -p -o time02 ./m$1 > /dev/null
/usr/bin/time -p -o time03 ./m$1 > /dev/null
/usr/bin/time -p -o time04 ./m$1 > /dev/null

mkdir Times/$1
mv time0* Times/$1/
```

Las medias de algunos resultados

Código O3	Original	512	16	8	4	2
Timing	10.6	1.97	1.40	1.28	1.46	1.69

Gráfica totalmente opcional de la evolución de BS (me gustan)



Básicamente he añadido tres bucles más que iteran por cada Block Size para aprovechar la localidad temporal y efectuar todas las operaciones necesarias a un bloque pequeño de A en el momento de acceder. Para ello pongo como bucles exteriores la i,j y ii, jj dentro del acceso por bloque. Pues la idea es que se accedan a esas posiciones el mínimo de veces. Como observamos cuando BS va

bajando, el tiempo se mejora, hasta el punto óptimo que es BS=8, a partir de ahí vuelve a empeorar. De ahí deducimos, que el mejor BS es el 8, porque deduzco que se llena una línea de cache entera para ese valor.

El código optimizado es este: **mult1Opti.c**

```
#define BS
...
int i,j,k;
 int ii,jj,kk;

  for ( i=0 ; i < n; i+=BS )
  {
    for ( k=0 ; k < n ; k+=BS )
    {
      for ( j=0 ; j < n ; j+=BS )
      {
        // BLOCKING
        for ( ii=i ; ii < i+BS; ii++ )
        {
          for ( kk=k ; kk < k+BS ; kk++ )
          {
            for ( jj=j ; jj < j+BS ; jj++ )
            {
              C[ii][jj]+= A[ii][kk]*B[kk][jj];
            }
          }
        }
      }
    }
  }
}
```

Análisis: Perf

Ejecución Perf

Veamos con perf los accesos (miss) a las caches L1 y la TLB de L1 que tiene la zedboard. Para ello creo un script **caches.sh** . Este generará outputs de los resultados de perf con cada evento

```
#! /bin/bash

mkdir perf/$1

perf record --event cache-misses -F 500 ./$1 > /dev/null
perf report --stdio -n --header > perf/$1/cache_misses.txt

perf record --event L1-dcache-load-misses -F 500 ./$1 > /dev/null
perf report --stdio -n --header > perf/$1/L1D_load_misses.txt

perf record --event L1-dcache-store-misses -F 500 ./$1 > /dev/null
perf report --stdio -n --header > perf/$1/L1D_store_misses.txt

perf record --event dTLB-load-misses -F 500 ./$1 > /dev/null
```

```
perf report --stdio -n --header > perf/$1/TLBd_load_misses.txt

perf record --event dTLB-store-misses -F 500 ./$1 > /dev/null
perf report --stdio -n --header > perf/$1/TLBd_store_misses.txt
```

Ficheros con los resultados del perf

Original **cache_misses.txt** **L1D_load_misses.txt** **L1D_store_misses.txt** **TLBd_load_misses.txt** **TLBd_store_misses.txt**

Optimizado **cache_misses.txt** **L1D_load_misses.txt** **L1D_store_misses.txt** **TLBd_load_misses.txt** **TLBd_store_misses.txt**

TLB load misses

Para el original obtenemos esto **TLBd_load_misses.txt**

```
# Samples: 5K of event 'dTLB-load-misses'
# Event count (approx.): 69718196
#
# Overhead       Samples  Command    Shared Object      Symbol
# ........  ............  ........  ..................  ....................
#
    99.98%         5286  mOriginal  mOriginal          [.] mult1
     0.02%            1  mOriginal  [kernel.kallsyms]  [k] do_page_fault
     0.00%            1  mOriginal  [kernel.kallsyms]  [k] extract_entropy
     0.00%            4  perf       [kernel.kallsyms]  [k] perf_event_exec
```

Optimizado **TLBd_load_misses.txt**

```
# Samples: 306  of event 'dTLB-load-misses'
# Event count (approx.): 49674
#
# Overhead       Samples  Command  Shared Object      Symbol
# ........  ............  .......  ..................  ........................
#
    66.01%          279  m8       m8                 [.] mult1
    27.48%            1  m8       [kernel.kallsyms]  [k] page_add_new_anon_rmap
     3.91%           15  m8       [kernel.kallsyms]  [k] __do_softirq
     0.81%            1  m8       [kernel.kallsyms]  [k] __vma_link_list
     0.68%            2  m8       [kernel.kallsyms]  [k] __sync_icache_dcache
     0.34%            1  m8       [kernel.kallsyms]  [k] remove_vma
     0.29%            1  m8       [kernel.kallsyms]  [k] run_rebalance_domains
     0.25%            1  m8       [kernel.kallsyms]  [k] _raw_spin_unlock_irq
     0.21%            1  m8       [kernel.kallsyms]  [k] rcu_process_callbacks
     0.02%            1  perf     [kernel.kallsyms]  [k] perf_event_comm
     0.01%            3  perf     [kernel.kallsyms]  [k] perf_event_exec
```

Como esperabamos, Obtenemos con el optimizado mucho menos Samples misses de load. Porque aprovechan la localidad temporal y además, espacial pues guardamos exactamente en una línea de cache todos los datos, haciendo las operaciones correspondientes con datos que ya tenemos en la misma línea!

TLB store misses

Original **TLBd_store_misses.txt**

```
# Samples: 5K of event 'dTLB-store-misses'
# Event count (approx.): 69628641
#
# Overhead       Samples  Command    Shared Object      Symbol
# ........  ............  .........  .................  ...................
#
    99.98%         5089  mOriginal  mOriginal          [.] mult1
     0.02%            1  mOriginal  mOriginal          [.] init
     0.00%            1  mOriginal  [kernel.kallsyms]  [k] load_elf_binary
     0.00%            4  perf       [kernel.kallsyms]  [k] perf_event_exec
```

Optimizado **TLBd_store_misses.txt**

```
# Samples: 297  of event 'dTLB-store-misses'
# Event count (approx.): 54414
#
# Overhead       Samples  Command  Shared Object      Symbol
# ........  ............  .......  .................  .....................
#
    65.65%          281  m8       m8                 [.] mult1
    31.60%            1  m8       [kernel.kallsyms]  [k] handle_mm_fault
     0.88%            4  m8       [kernel.kallsyms]  [k] _raw_spin_unlock_irq
     0.83%            1  m8       [kernel.kallsyms]  [k] load_elf_binary
     0.19%            1  m8       [kernel.kallsyms]  [k] l2c210_inv_range
     0.19%            1  m8       [kernel.kallsyms]  [k] ip_route_input_noref
     0.17%            1  m8       libc-2.23.so       [.] __GI___libc_write
     0.16%            1  m8       [kernel.kallsyms]  [k] __do_softirq
     0.16%            1  m8       ld-2.23.so         [.] do_lookup_x
     0.15%            1  m8       [kernel.kallsyms]  [k] _clear_bit
     0.02%            1  perf     [kernel.kallsyms]  [k] strrchr
     0.01%            3  perf     [kernel.kallsyms]  [k] perf_event_exec
```

Más de lo mismo