

Estructura de Computadores: Práctica 6

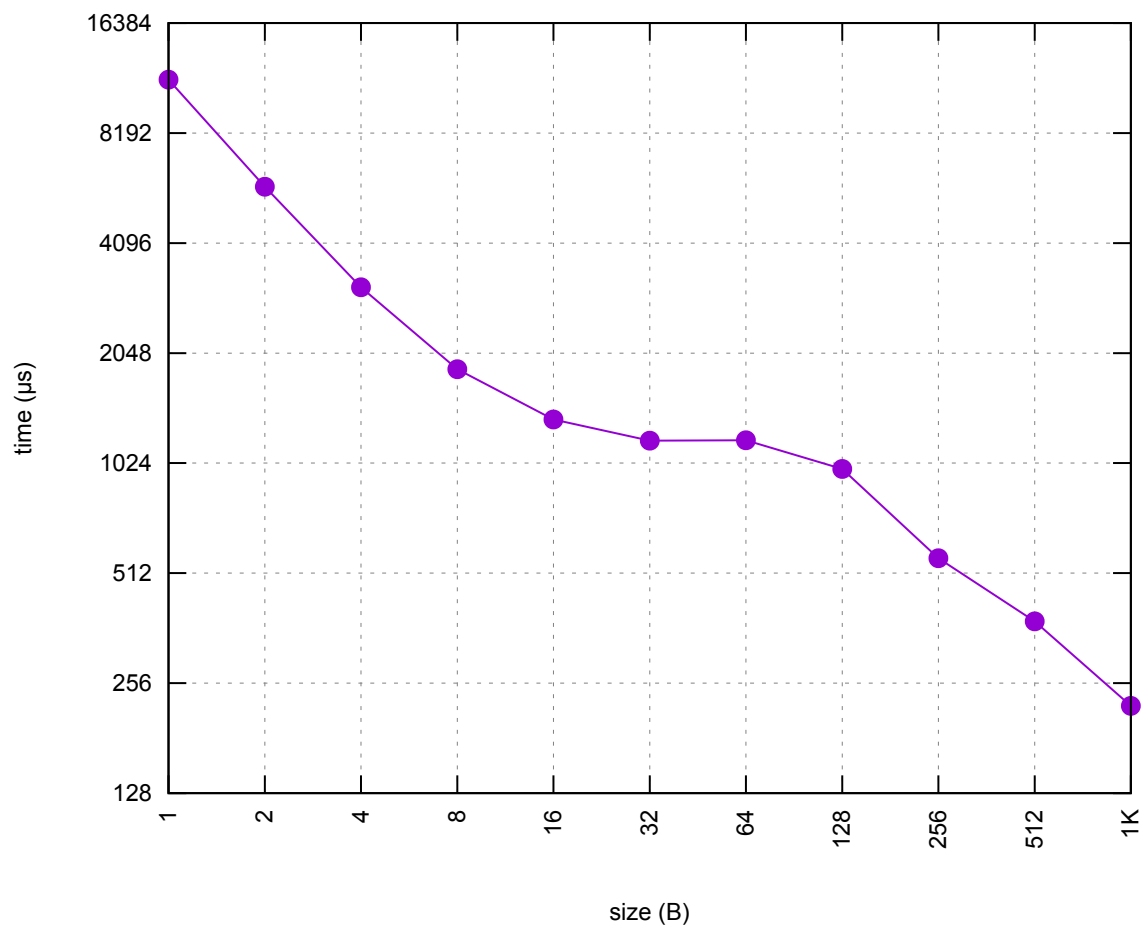
Memoria Caché

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, comencemos 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 líneas 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:

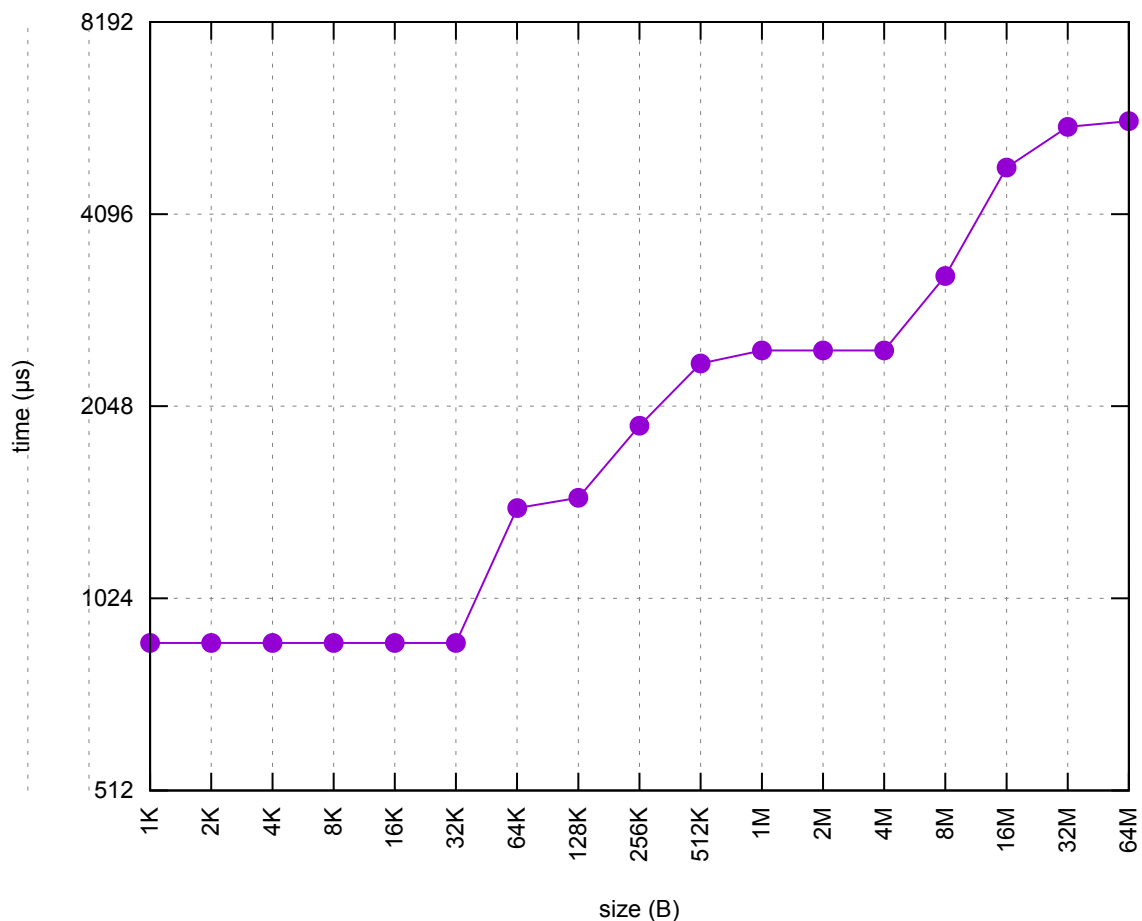


Efectivamente, vemos que a partir de los 64B los tiempos de acceso van disminuyendo drásticamente. Esto es un indicio de que el tamaño de línea es de 64B, ya que no tiene que ir buscando datos internos del bloque y por tanto, el acceso a posiciones múltiplo de ésta hace que disminuya este tiempo.

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`:

```
pablo@laptop: ~/Documentos/DGIIM2/EC/Pfácticas y Se...
pablo@laptop:~/Documentos/DGIIM2/EC/Pfácticas y Seminarios/Practica 6$ make info
line size = 64B
cache size = 32K/32K/256K/8192K/
cache level = 1/1/2/3/
cache type = Data/Instruction/Unified/Unified/
pablo@laptop:~/Documentos/DGIIM2/EC/Pfácticas y Seminarios/Practica 6$
```

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

Información de mi ordenador:

```
pablo@laptop: ~
pablo@laptop:~$ lscpu
Arquitectura: x86_64
modo(s) de operación de las CPUs: 32-bit, 64-bit
Orden de los bytes: Little Endian
Address sizes: 39 bits physical, 48 bits virtual
CPU(s): 8
Lista de la(s) CPU(s) en línea: 0-7
Hilo(s) de procesamiento por núcleo: 2
Núcleo(s) por «socket»: 4
«Socket(s)»: 1
Modo(s) NUMA: 1
ID de fabricante: GenuineIntel
Familia de CPU: 6
Modelo: 158
Nombre del modelo: Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz
Revisión: 10
CPU MHz: 800.017
CPU MHz máx.: 4000,0000
CPU MHz mín.: 800,0000
BogoMIPS: 4599.93
Virtualización: VT-x
Caché L1d: 128 KiB
Caché L1i: 128 KiB
Caché L2: 1 MiB
Caché L3: 8 MiB
CPU(s) del nodo NUMA 0: 0-7
Vulnerability Itlb multihit: KVM: Mitigation: Split huge pages
Vulnerability L1tf: Mitigation; PTE Inversion; VMX conditional
cache flushes, SMT vulnerable
Vulnerability Mds: Mitigation; Clear CPU buffers; SMT vulnerab
le
Vulnerability Meltdown: Mitigation; PTI
Vulnerability Spec store bypass: Mitigation; Speculative Store Bypass disabl
ed via prctl and seccomp
Vulnerability Spectre v1: Mitigation; usercopy/swapgs barriers and __
user pointer sanitization
Vulnerability Spectre v2: Mitigation; Full generic retpoline, IBPB co
nditional, IBRS_FW, STIBP conditional, RSB
filling
Vulnerability Srbds: Mitigation; Microcode
Vulnerability Tsx async abort: Not affected
Indicadores: fpu vme de pse tsc msr pae mce cx8 apic sep
mtrr pge mca cmov pat pse36 clflush dts ac
pi mmx fxsr sse sse2 ss ht tm pbe syscall n
x pdpe1gb rdtscp lm constant_tsc art arch_p
erfmon pebs bts rep_good nopl xtopology non
stop_tsc cpuid aperfmperf pni pclmulqdq dte
s64 monitor ds_cpl vmx est tm2 ssse3 sdbg f
ma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic
movbe popcnt tsc_deadline_timer aes xsave
avx f16c rdrand lahf_lm abm 3dnowprefetch c
puid_fault epb invpcid_single pti ssbd ibrs
ibpb stibp tpr_shadow vnmi flexpriority ep
t vpid ept_ad fsgsbase tsc_adjust bmi1 avx2
smep bmi2 erms invpcid mpx rdseed adx snap
```

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>

