***TIC – Filière Informatique***

**Jigé Pont**

**Nicolas Fuchs**

**Alan Sueur**

**Microprocesseur 3**

**Labo 2**



27/02/2018

# Configuration

Système d'exploitation : Kali Linux Rolling 64 bits  
Processeur : Intel Core i7-4510U CPU @ 2.00GHz

­

# Configuration cache

getconf -a | grep CPU

Commande :

\_POSIX\_CPUTIME 200809

\_POSIX\_THREAD\_CPUTIME 200809

Résultat :

cat /proc/cpuinfo | grep CPU

Commande :

model name : Intel(R) Core(TM) i7-4510U CPU @ 2.00GHz

Résultat :

cat /proc/cpuinfo | grep CPU

Commande :

[ 0.000000] smpboot: Allowing 128 CPUs, 127 hotplug CPUs

[ 0.000000] setup\_percpu: NR\_CPUS:512 nr\_cpumask\_bits:512 nr\_cpu\_ids:128 nr\_node\_ids:1

[ 0.004000] RCU restricting CPUs from NR\_CPUS=512 to nr\_cpu\_ids=128.

[ 0.052810] CPU: Physical Processor ID: 0

[ 0.052833] mce: CPU supports 0 MCE banks

[ 0.067613] smpboot: CPU0: Intel(R) Core(TM) i7-4510U CPU @ 2.00GHz (family: 0x6, model: 0x45, stepping: 0x1)

[ 0.068000] core: CPUID marked event: 'cpu cycles' unavailable

[ 0.068000] core: CPUID marked event: 'instructions' unavailable

[ 0.068000] core: CPUID marked event: 'bus cycles' unavailable

[ 0.068000] core: CPUID marked event: 'cache references' unavailable

[ 0.068000] core: CPUID marked event: 'cache misses' unavailable

[ 0.068000] core: CPUID marked event: 'branch instructions' unavailable

[ 0.068000] core: CPUID marked event: 'branch misses' unavailable

[ 0.068000] NMI watchdog: Perf event create on CPU 0 failed with -2

[ 0.076557] smp: Bringing up secondary CPUs ...

[ 0.076558] smp: Brought up 1 node, 1 CPU

[ 1.568969] ledtrig-cpu: registered to indicate activity on CPUs

Résultat :

cat /proc/cpuinfo | grep CACHE

Commande :

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 8

LEVEL2\_CACHE\_LINESIZE 64

LEVEL3\_CACHE\_SIZE 4194304

LEVEL3\_CACHE\_ASSOC 16

LEVEL3\_CACHE\_LINESIZE 64

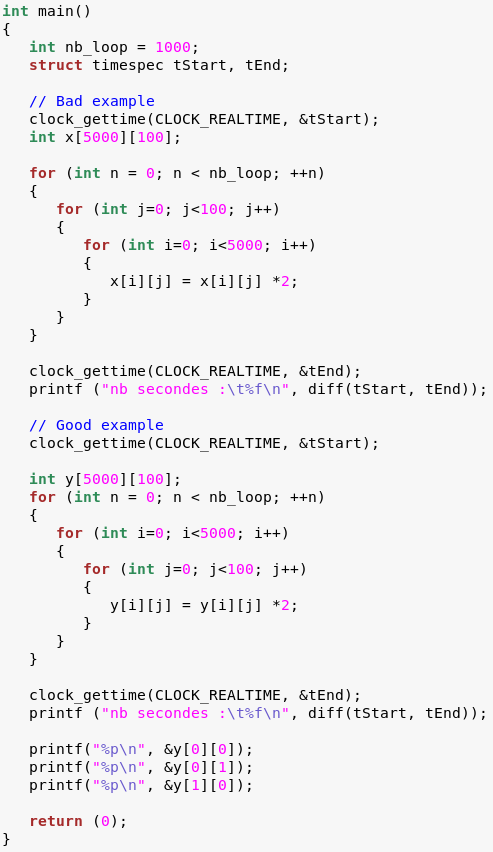
LEVEL4\_CACHE\_SIZE 0

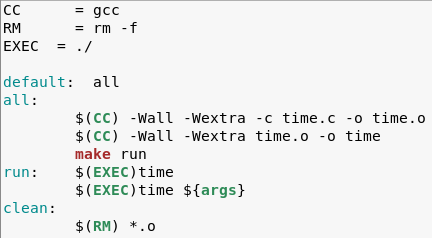
LEVEL4\_CACHE\_ASSOC 0

LEVEL4\_CACHE\_LINESIZE 0

Résultat :

# Exemple bon et mauvais d'accès à la mémoire

Fonction main du fichier time.c

Pour lancer la compilation et le linkage, on utilise un makefile :

Résultat :

nb secondes : 2.985660

nb secondes : 1.795121

0x7ffdaffe3c40

0x7ffdaffe3c44

0x7ffdaffe3dd0

Remarques : Le code "bad" lancé 1000 fois dans une boucle nécessite presque 2 fois plus de temp que le code "good". Au vu de la théorie, cela s'explique par le nombre de recopies du cache nécessaire.

Le fait d'accéder au tableau d'entiers par ligne d'abord permet de charger le cache 1 seule fois par ligne. Si on accède au tableau par colonne d'abord, le cache doit être chargé plus souvent. Ce coefficient dépend du nombre de lignes qui tiennent dans le cache.

# Mémoire cache avec buffer complexe

gcc -Wall -Wextra -S time.c

Le fichier assembleur est généré grâce à la commande suivante :

objdump -d time

On le visualise ensuite avec la commande suivante :

Le programme de test est exécuté avec les structures suivantes:

typedef struct \_TEST2  
{  
 int a;  
 int c;  
 char b;  
 char d;  
} TEST2;  
static TEST2 test2 [1000];

Figure



typedef struct \_TEST2  
{  
 int a;  
 int c;  
 char b \_\_attribute\_\_ ((aligned (4)));  
 char d \_\_attribute\_\_ ((aligned (4)));  
} TEST2;  
static TEST2 test2 [1000];

Figure





typedef struct \_TEST2  
{  
 int a;  
 char b;  
 int c \_\_*attribute\_\_* ((packed));  
 char d;  
} TEST2;  
static TEST2 test2 [1000];

Figure

typedef struct \_TEST2  
{  
 char b;  
 char d;  
 int a \_\_*attribute\_\_* ((packed));  
 int c \_\_*attribute\_\_* ((packed));  
} TEST2;  
static TEST2 test2 [1000];

Figure



La procédure de test (une boucle qui affecte des valeurs aux 100 premiers éléments du tableau de 1000 structures) est elle-même dans une boucle qui tourne 1 000 000 fois (nb\_loop = 1 000 000).

for (int n = 0; n < nb\_loop; ++n)  
{  
 for (int i=0; i<100; i++)  
 {  
 test[i].a=1;  
 test[i].b=2;  
 test[i].c=3;  
 test[i].d=4;  
 }  
}

Figure

Résultats :

Les tailles des structures ont été obtenues via un sizeof.

|  |  |  |
| --- | --- | --- |
| Structure | Taille en octet | Temps de test (sec) |
| donnée | 16 | 0.32 |
| (figure 1) | 12 | 0.40 |
| (figure 2) | 16 | 0.32 |
| (figure 3) | 12 | 0.42 |
| (figure 4) | 10 | 0.42 |

On constate que l’utilisation des directives \_\_attribute\_\_ ne change pas la vitesse de manière souhaitée. Notre analyse montre que le compilateur aligne par défaut un char à 4 et qu’il s’agit de l’optimum pour l’accès aux valeurs de la structure. Quand nous changeons la taille de la structure via un packed, nous gagnons peut-être des recopies du cache mais nous perdons du temps par la complexité de l’accès à une valeur qui n’est pas alignée sur un int.

Nous le constatons très bien via la structure de la figure 2 qui, avec des alignements pour les chars, tourne à la même vitesse que la structure de la donnée.

Notre compilateur optimise donc par défaut les alignements dans les structures.

Le code de la figure 6 tourne lui en 0.27 secondes. Nous avons appliqué le principe de la sortie des invariants et un décalage par pointeurs. Nous n’avons pas réussi à trouver un code plus optimisé du point de vue vitesse pour l'instant.­­­­

TEST2 \*t;  
for (int n = 0; n < nb\_loop; ++n)  
{  
 t = &test2[0];  
 for (int i=0; i<100; ++i)  
 {  
 t->a=1;  
 t->b=2;  
 t->c=3;  
 t->d=4;  
 t += 10;  
 }  
}

Figure

Remarques :

On constate que la structure de la figure 1