Perrine MOLINAS

Mathieu LESUR

INF-4101C

Évaluation des performances de la mémoire cache

ESIEE Paris

Octobre 2022

Sommaire

1. Introduction

2. L’utilisation de la mémoire cache et les premiers résultats

3. La performance selon le type de cache utilisé

4. Conclusion

1. Introduction

Ce TP n°4 comporte plusieurs objectifs.
Le premier est de pouvoir estimer les performances et l'importance de la mémoire cache. Pour cela, nous allons utiliser des outils qui permettent de simuler l'utilisation de différentes mémoires caches.
Le deuxième objectif est de comprendre les différents paramètres de la mémoire cache.
Quant au dernier, nous essayerons de comprendre comment le style de programmation peut agir sur l'efficacité de la mémoire cache.

Pour ce TP, nous utiliserons le langage de programmation C pour pouvoir comparer l'efficacité de différents programmes tout en utilisant un compilateur C nommé Apple clang version 12.0.5. Pour comparer l'efficacité des programmes, nous utiliserons un outil appelé "valgrind" qui est une commande à installer et à utiliser via le terminal du système d'exploitation. Valgrind est un outil de programmation libre pour déboguer, effectuer du profilage de code et mettre en évidence des fuites mémoires.

L'ensemble des codes utilisés sont disponibles sur le GitHub public ci-joint :
https://github.com/MathieuLsr/ESIEE-INF4101C-Optimisation-TP4/tree/main/TP4

2. L’utilisation de la mémoire cache et les premiers résultats

Dans un premier temps, nous allons programmer 3 programmes différents qui ont tous les 3 le même but : remplir un premier tableau de 0 puis remplir un deuxième tableau avec les valeurs du premier.
L’objectif de cet exercice est de faire un grand nombre d'appels vers différentes adresses mémoires et de stocker les informations pour comprendre comment fonctionne cette mémoire cache.

Le premier programme sera le plus basique possible et sans aucune optimisation. Nous créerons 2 tableaux de même taille et les remplirons en passant par toutes les cases du tableau 1 par 1. Nous n'utilisons aucun pointeur pour avoir un programme le plus basique possible et sans toucher à la mémoire.

```c
#define ROW 1000
#define COL 1000

int main(int argc, char const *argv[])
{

    int tab1[ROW][COL];
    int tab2[ROW][COL];

    for (int i = 0; i < ROW; i++)
        for(int j=0 ; j < COL ; j++)
            tab1[i][j] = 0;

    for (int i = 0; i < ROW; i++)
        for(int j=0 ; j < COL ; j++)
            tab2[i][j] = tab1[i][j];

    free(tab1);
    free(tab2);

    return 0;
}
```

```c
#define ROW 1000
#define COL 1000

int main(int argc, char const *argv[])
{

    int tab1[ROW][COL];
    int tab2[ROW][COL];

    for (int i = 0; i < COL; i++)
        for(int j=0 ; j < ROW ; j++)
            tab1[j][i] = 0;

    for (int i = 0; i < COL; i++)
        for(int j=0 ; j < ROW ; j++)
            tab2[j][i] = tab1[j][i];

    return 0;
}
```

Le deuxième consiste à inverser les boucles et à parcourir les tableaux lignes par lignes et non colonnes par colonnes.

Le dernier programme consiste à utiliser des fonctions mises à disposition par des librairies confirmées de la programmation en C. Nous déclarons nos tableaux en tant que pointeurs et utilisons des méthodes pour réserver les adresses mémoires avec malloc et calloc.
Calloc met tous les octets du bloc à la valeur 0 alors que malloc ne modifie pas la zone de mémoire. La fonction memcpy permet de copier un bloc de mémoire d'une adresse source jusqu'à une adresse destination. Donc ici, nous avons le même résultat que les autres programmes mais en utilisant des méthodes plus adaptées.

```c
#define ROW 1000
#define COL 1000

int main(int argc, char const *argv[])
{

    int SIZE = (ROW * COL) * sizeof(int) ;
    int* tab1 = calloc(ROW*COL, SIZE);
    int* tab2 = malloc(SIZE) ;

    memcpy(tab2, tab1, sizeof(SIZE)) ;

    free(tab1);
    free(tab2);

    return 0;
}
```

Pour connaître les performances de ces programmes, nous utilisons l'outil cachegrind de valgrind. L'outil cachegrind simule la façon dont le programme interagit avec la hiérarchie de cache d'une machine.

Cachegrind recueille les statistiques suivantes (les abréviations utilisées pour chaque statistique sont indiquées entre parenthèses) :

- Les lectures du cache I (Ir, qui est égal au nombre d'instructions exécutées), les lectures manquées du cache I1 (I1mr) et les lectures manquées des instructions du cache LL (ILmr).

- Les lectures du cache D (Dr, qui est égal au nombre de lectures de mémoire), D1 lectures manquées de cache (D1mr), et LL lectures manquées de données de cache (DLmr).

- Les écritures dans le cache D (Dw, égal au nombre d'écritures en mémoire), manque d'écriture dans le cache D1 (D1mw) et manque d'écriture de données dans le cache LL (DLmw).

- Branches conditionnelles exécutées (Bc) et branches conditionnelles mal prédites (Bcm).

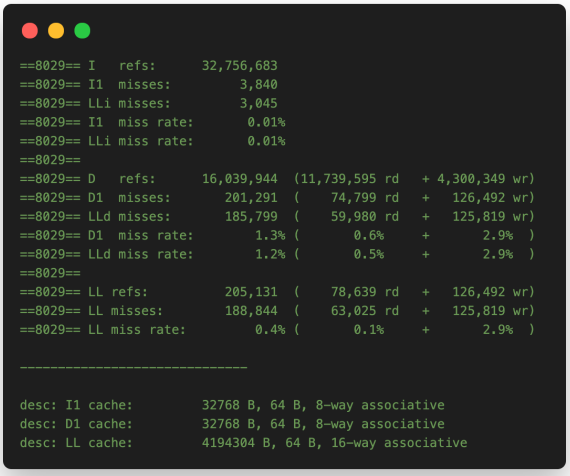
- Branches indirectes exécutées (Bi) et branches indirectes mal prédites (Bim).

À noter que le total des accès D1 est donné par D1mr + D1mw, et que le total des accès LL est donné par ILmr + DLmr + DLmw.

Sur une machine moderne, un ratage de L1 coûte généralement environ 10 cycles, un ratage de LL peut coûter jusqu'à 200 cycles et un branchement mal prédit coûte entre 10 et 30 cycles. Un profilage détaillé du cache et des branchements peut être très utile pour comprendre comment le programme interagit avec la machine et donc comment le rendre plus rapide.

En outre, étant donné qu'une lecture du cache d'instruction est effectuée par instruction exécutée, nous pouvons savoir combien d'instructions sont exécutées par ligne, ce qui peut être utile pour le profilage traditionnel.

Nous exécutons donc la commande suivante : “valgrind --tool=cachegrind ./main” et obtenons les résultats des 3 programmes. La commande nous affiche pas mal d’informations que nous exportons dans un fichier Excel pour que les données soient exploitables.



Nous pouvons tirer pas mal d'informations de ce tableau. La première chose que nous voyons, c'est la grande différence entre les 2 premiers programmes et le dernier sur le nombre d'accès au cache I. Cependant, le nombre de lecture raté est similaire aux 3 programmes.

Ensuite, nous voyons aussi que l'appel au cache D est divisé par 16 pour le dernier programme. Nous remarquons que le programme 2 possède un grand nombre d'appels raté sur le cache D1 par rapport au programme 1. Or, la seule différence entre les deux programmes est la façon dont on parcourt les tableaux. Nous en concluons qu'il existe une méthode plus efficace que l'autre pour parcourir l'ensemble du tableau.

	Programme 1	Programme 2	Programme 3
I refs	32 756 683	32 757 317	2 789 857
I1 misses	3 840	3 885	3 981
LLi misses	3 045	3 045	3 078
I1 miss rate	0,01%	0,01%	0,14%
LLi miss rate	0,01%	0,01%	0,11%
D refs	16 039 944	16 040 233	1 047 755
D1 misses	201 291	3 013 824	13 886
LLd misses	185 799	187 854	4 444
D1 miss rate	1,30%	18,80%	1,30%
LLd miss rate	1,20%	1,20%	0,40%
LL refs	205 131	3 017 709	17 867
LL misses	188 844	191	7 522
LL miss rate	0,40%	0,40%	0,20%

Nous rappelons que L1 coûte généralement environ 10 cycles, LL peut coûter jusqu'à 200 cycles et un branchement mal prédit coûte entre 10 et 30 cycles. Avec ces estimations, nous pouvons déduire le nombre de cycles par programme et pouvoir développer encore plus les résultats donnés. Sans réaliser les calculs, nous voyons par exemple que la ligne "LLd mises" est très importante vu qu'un LLd peut coûter jusqu'à 200 cycles donc nous voyons que le programme 1 et 2 font environ 185000*200 = 37 000 000 de cycles alors que le programme 3 n'en fait que 4500*200 = 900 000 cycles soit 41 fois moins.

Avec ces premiers résultats, nous en concluons que le développeur peut avoir un fort impact sur les performances du programme et qu'il est possible d'optimiser ses résultats si nous connaissons comment fonctionne notre mémoire cache.

3. La performance selon le type de cache utilisé

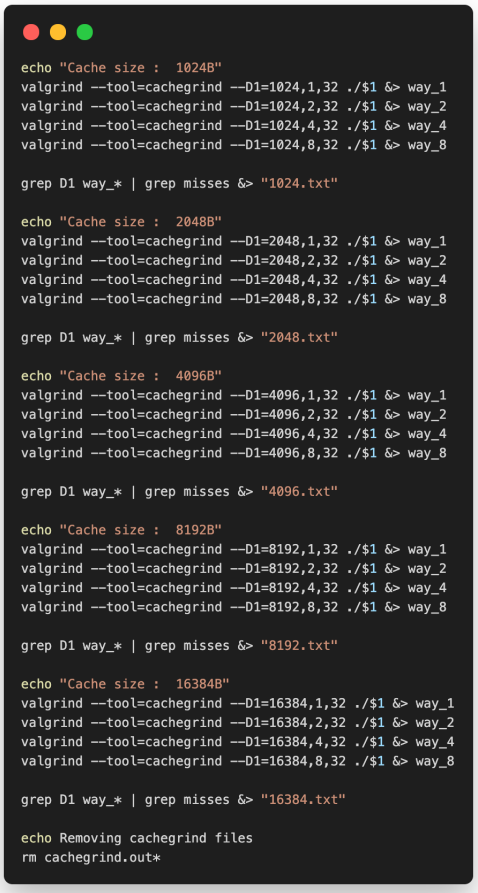
Pour continuer nos tests, nous allons procéder à des changements des paramètres du niveau D1 de la mémoire cache et analyser les conséquences sur les performances (exprimées en nombre total de "misses"). Pour ça, nous allons utiliser l'option --D1=<size>,<associativity>,<line size> de valgrind. Nous allons faire varier la taille (<size>) ainsi que l'associativité (<associativity>). L'argument <line size> sera fixé à 32.

La taille maximale théorique du cache est de 2 Go. La taille du cache que l'on peut spécifier est limitée par la quantité de mémoire physique et d'espace de pagination disponibles sur le système. Le cache de classe partagée est constitué de fichiers mappés en mémoire qui sont créés sur le disque et qui restent en place lorsque le système d'exploitation est redémarré.

Dans une mémoire cache entièrement associative, un bloc de données provenant de n'importe quelle adresse de la mémoire peut être stocké dans n'importe quelle ligne de la mémoire cache, et l'adresse entière est utilisée comme étiquette de la mémoire cache : par conséquent, lors de la recherche d'une correspondance, toutes les étiquettes doivent être comparées simultanément avec n'importe quelle adresse demander, ce qui exige un matériel supplémentaire coûteux.

Ces expériences seront effectuées sur 2 nouveaux programmes qui consistent à faire encore plus d'accès à la mémoire. Pour ça, la matrice B contiendra les valeurs moyennes de voisinage 3x3 pour chaque élément.

Désormais nous cherchons à modifier 2 paramètres sur 3 donc nous utilisons un script shell pour automatiser le lancement du programme avec la commande valgrind. Ce script lance le programme principal 20 fois au total en modifiant la taille de la mémoire cache 5 fois (1024B, 2048B, 4096B, 8192B, 16348B) puis en faisant varier l'associativité entre 1, 2, 4 et 8. Tous les résultats seront extraits dans un fichier texte.



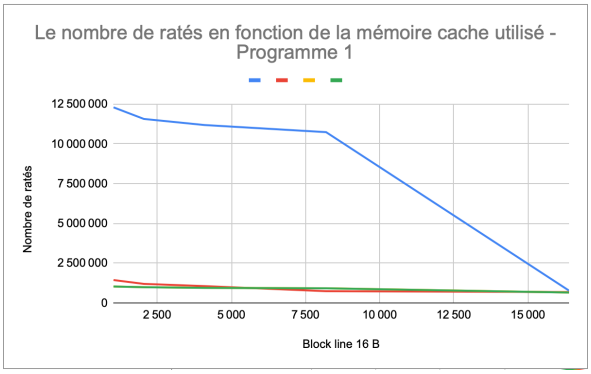
```
echo "Cache size :  1024B"
valgrind --tool=cachegrind --D1=1024,1,32 ./$1 &> way_1
valgrind --tool=cachegrind --D1=1024,2,32 ./$1 &> way_2
valgrind --tool=cachegrind --D1=1024,4,32 ./$1 &> way_4
valgrind --tool=cachegrind --D1=1024,8,32 ./$1 &> way_8

grep D1 way_* | grep misses &> "1024.txt"

echo "Cache size :  2048B"
valgrind --tool=cachegrind --D1=2048,1,32 ./$1 &> way_1
valgrind --tool=cachegrind --D1=2048,2,32 ./$1 &> way_2
valgrind --tool=cachegrind --D1=2048,4,32 ./$1 &> way_4
valgrind --tool=cachegrind --D1=2048,8,32 ./$1 &> way_8

grep D1 way_* | grep misses &> "2048.txt"

echo "Cache size :  4096B"
valgrind --tool=cachegrind --D1=4096,1,32 ./$1 &> way_1
valgrind --tool=cachegrind --D1=4096,2,32 ./$1 &> way_2
valgrind --tool=cachegrind --D1=4096,4,32 ./$1 &> way_4
valgrind --tool=cachegrind --D1=4096,8,32 ./$1 &> way_8

grep D1 way_* | grep misses &> "4096.txt"

echo "Cache size :  8192B"
valgrind --tool=cachegrind --D1=8192,1,32 ./$1 &> way_1
valgrind --tool=cachegrind --D1=8192,2,32 ./$1 &> way_2
valgrind --tool=cachegrind --D1=8192,4,32 ./$1 &> way_4
valgrind --tool=cachegrind --D1=8192,8,32 ./$1 &> way_8

grep D1 way_* | grep misses &> "8192.txt"

echo "Cache size :  16384B"
valgrind --tool=cachegrind --D1=16384,1,32 ./$1 &> way_1
valgrind --tool=cachegrind --D1=16384,2,32 ./$1 &> way_2
valgrind --tool=cachegrind --D1=16384,4,32 ./$1 &> way_4
valgrind --tool=cachegrind --D1=16384,8,32 ./$1 &> way_8

grep D1 way_* | grep misses &> "16384.txt"

echo Removing cachegrind files
rm cachegrind.out*
```

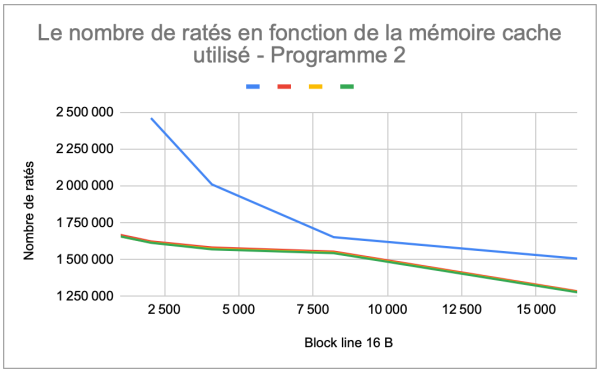
Afin d'exploiter et d'analyser au mieux ces données, nous les mettons dans un tableau excel et réalisons les courbes correspondantes.

	Programme 1				
Block line 16 B	**1 024**	**2 048**	**4 096**	**8 192**	**16 384**
1 way	3 319 188	2 458 886	2 008 943	1 650 888	1 505 357
2 way	1 666 094	1 622 053	1 580 747	1 552 887	1 283 848
4 way	1 656 145	1 614 646	1 572 284	1 545 913	1 279 346
8 way	1 655 949	1 612 918	1 568 423	1 542 587	1 277 024
	Programme 2				
Block line 16 B	**1 024**	**2 048**	**4 096**	**8 192**	**16 384**
1 way	12 294 138	11 565 253	11 182 005	10 735 950	773 300
2 way	1 437 642	1 195 346	1 055 239	736 600	682 818
4 way	1 030 831	989 347	947 172	921 682	654 844
8 way	1 031 040	987 627	944 058	917 696	652 555

Nous pouvons observer grâce à nos résultats de notre programme 1, que plus la mémoire cache est grande ou plus notre associativité est élevée, plus notre taux d'échec sera réduit. Dans le meilleur des cas, nous pouvons mettre des performances illimitées mais ceci est impossible et tout ce qui est parfait en informatique se paye autrement. Nous remarquons aussi que lorsque nos composants et nos options sont bien équilibrées, nous avons un rendu du taux d'échec acceptable. Par exemple, si nous prenons 4k de cache disponible et une associativité correcte, nous avons les mêmes résultats que si nous prenions une taille 4 fois plus élevée.

Cependant pour les résultats du programme 2, nous remarquons quelques anomalies. Nous remarquons que lorsque nous avons un cache associatif à 1 seul voie, notre nombre d'échecs est très important. Nous en déduisons que c'est à cause des fonctions "memcpy" ainsi que l'allocation de mémoire dynamique. Nous voyons aussi que le nombre d'échecs diminue instantanément lorsqu'on a plus de places dans notre mémoire cache.
Ce phénomène est dû à notre nombre de mémoire que possèdent nos tableaux. Nous avons instancié 2 tableaux de 1000*1000 soit 1 million d'entier. Si nous diminuons cette quantité à 500 000 par exemple, le résultat pour 8k de mémoire cache sera aux alentours de 800 000 et non 10 millions vu qu'il sera possible de mettre en cache l'ensemble du tableau et non que certaines parties.





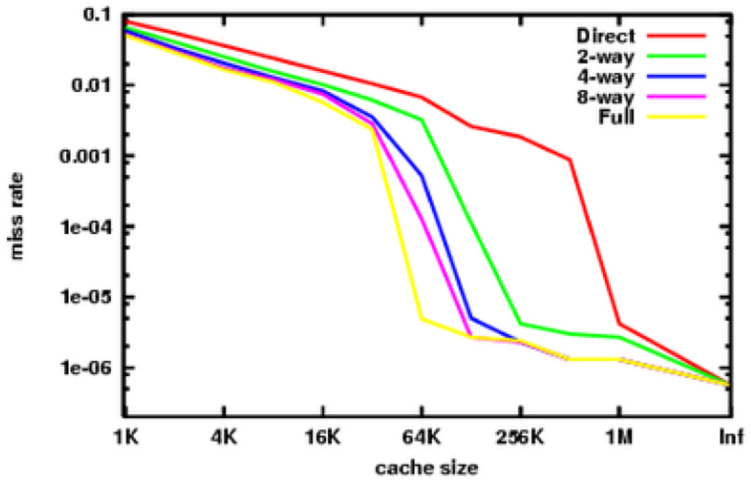


Figure 1 cache performance seen on the Integer portion of the SPEC CPU2000 benchmarks [1]

Nous obtenons des graphiques qui nous montrent que plus nous avons de place dans la mémoire cache et plus notre cache associatif possède de voies, plus nous aurons un taux d'échec se rapprochant de 0. Si nous avions continué nos tests en utilisant des tailles de cache encore plus grand, nous aurions obtenu le même graphique que la figure 1. Nous voyons une nette amélioration lorsqu'on a atteint une taille de mémoire cache acceptable pour un ordinateur.

4. Conclusion

Pour conclure, nous avons vu que le développeur avait un impact important sur l'optimisation de la mémoire cache et qu'il est important de connaître le matériel utilisé pour savoir quel type de fonction et quel type de programmation utilisé.

De plus, il est possible d'améliorer les performances du cache en utilisant une taille de bloc de cache plus importante, une associativité plus élevée, une réduction du taux d'échec, une réduction de la pénalité d'échec et une réduction du temps d'accès au cache.