TD 4 - Caches - Correction

1 Etiquette et Index de cache

Un processeur a 2 Gio de mémoire principale. Pour les différents caches ci-dessous :

1. cache de 2 Mio à correspondance directe et écriture simultanée avec des blocs de 16 octects
2. cache de 4 Mio à correspondance directe, réécriture et blocs de 32 octects
3. cache de 4 Mio associatif 4 voies (4 blocs par ensemble), réécriture et blocs de 32 octects

1.1 Quelle est la décomposition d'une adresse mémoire (nombre de bits des différentes parties) ?

Rappels :

- l'index indique d ans quelle ligne va la donnée.
- l'adresse dans le bloc (ou offset) indique où ets la donnée dans la ligne
- l'étiquette sert à comparer si la donnée que l'on cherche est bien celle qui est stockée actuellement dans le cache ou non.

Mémoire de 2 Gio $= 2 * 2^{30} = 2^{31}$. Les adresses font donc 31 bits.

1. Cache de 2 Mio $= 2 * 2^{20} = 2^{21}$ octets. Comme une ligne = 16 octets, il y a $2^{21}/2^4 = 2^{17}$ lignes. L'index étant la partir qui permet de savoir à quelle ligne est la donnée dans le cache, il faut 17 bits d'index.
   La ligne de cache étant de $16 = 2^4$ octets, pour savoir à quel endroit est la donnée dans la ligne, il faut 4 bits d'adresse dans bloc (offset).
   Enfin, l'étiquette fait $31 - 17 - 4 = 10$ bits
2. Même explication que précédemment.
   17 bits d'index.
   5 bits d'adresse dans bloc (offset).
   9 bits d'étiquette. $(31 - 17 - 5)$
3. Même explication que précédemment, sauf pour le nombre de lignes.
   15 bits d'index. En effet, comme le cache est associatif par 4 voies, $4 = 2^2$ lignes ont la même adresse. Donc les 2^{17} lignes sont par groupes de 2^2 soit $2^{17}/2^2 = 2^{15}$ groupes
   5 bits d'adresse dans bloc (offset).
   11 bits d'étiquette. $(31 - 15 - 5)$

1.2 Donner les différentes parties d'une ligne (bloc) de cache (nombre de bits des différentes parties). Quel est le surcoût lié aux bits de contrôle et d'étiquette (par rapport à la partie « données » du cache)

Rappels :

- l'étiquette est la même que celle de la question précédente.

- il y a 1 ou 2 bits de contrôles : 1 bits de validité (la données est-elle utilisée actuellement ?) et 1 bit de modification en case de réécriture (la donnée a-t-elle été modifiée sans avoir été modifiée aussi en mémoire principale ?).
- les lignes de données sont celle de la questions précédente.

1. Étiquette : 10 bits
   Contrôle : 1 bit (1 bit de validité)
   Données : $16 * 8$
   Surcoût : $11/139 : 7,91\%$
2. Étiquette : 9 bits
   Contrôle : 2 bits (Réécriture : 1 bit valide + 1 bit modifié)
   Données : $32 * 8$
   Surcoût : $11/267 : 4,12\%$
3. Étiquette : 11 bits
   Contrôle : 2 bits (Réécriture : 1 bit valide + 1 bit modifié)
   Données : $32 * 8 = 256$ bits
   Surcoût : $13/269 : 4,83\%$

Note : plus les lignes sont longues, plus les bits de contrôle sont négligeables en comparaison des données stockées. Cependant, il faut mettre ça en regard de la recopie des lignes de caches qui vont prendre plus de temps. L'espace versus le temps.

2 Cache Données

On considère une architecture possédant un cache de données de 8 Kio octects organisé en blocs de 32 octects. On considère deux cas : correspondance directe et associativité par ensembles de 2 blocs, avec le LRU (Least Recently Used) comme algorithme de remplacement (on remplace la ligne la moins récemment utilisée). Les tableaux suivants de 4096 floats (la taille d'un float est de 32 bits) sont aux adresses suivantes :

`X`	`Y`	`Z`	`X1`	`Y1`	`X2`	`Y2`
`0x0001 0000`	`0x0001 4000`	`0x0001 8000`	`0x0001 C000`	`0x0002 0000`	`0x0002 4000`	`0x0002 8000`

2.1 Quels sont les éléments des tableaux X et Y qui peuvent occuper le premier mot du premier bloc du cache ?

Correspondance directe. : `X[0]`, `X[2048]` et `Y[0]`, `Y[2048]`
L'adresse de `X` est `0x0001 0000`, soit en binaire `0000 0000 0000 0001 000|0 0000 000|0 0000`
Sachant que l'offset est de 5 et l'index de 8 (faites les calculs pour vous entraîner), on observe que l'offset et l'index sont remplis de 0 $\Rightarrow$ `X[0]` est le premier mot, sur la première ligne.
S'il y a une autre valeur de `X` à cet endroit, il aura nécessairement que des 0 en index et offset. La prochaine adresse est `0000 0000 0000 0001 001|0 0000 000|0 0000` (soit `0x0001 2000`), soit 2000_{16} octets après le début de `X`. $2000_{16} = 8192_{10}$, soit le $8192/4 = 2048^e$ élément du tableau.
On observe que Y est 2000_{16} octets plus loin donc les mêmes calculs sont valables.

Associativité par 2 : `X[0]`, `X[1024]`, `X[2048]`, `X[3072]` et `Y[0]`, `Y[1024]`, `Y[2048]`, `Y[3072]`
Les mêmes calculs que précédemment sont valables, mais avec 2 fois moins de bits d'index. A noter que ces valeurs peuvent aussi occuper le premier mot de la deuxième ligne selon si la première ligne est déjà occupée ou non.

2.2 Pour chaque boucle, quel est le nombre de défauts de cache données par itération (on suppose que les variables scalaires sont toujours en registre) ?

Correspondance directe
Pour toutes les boucles, tous les accès font un défaut de cache car le chargement de `Y` chasse celui de `X` et inversement à l'itération suivante. On a donc 2 défauts de cache par itération pour le 1ère et 3^e kernel, 4 pour le 2^e et 4^e.

Associativité par 2
Kernel 1 et 3 : 2 défauts de caches (`X` et Y) toutes les 8 itérations (défaut de chargement).
Kernel 2 : Même problème que la correspondance directe, `X2` chasse `X1` et `Y2` chasse `Y1` : 4 défauts par itération.
Kernel 4 : Avec le LRU, `Z` chasse `Y` et inversement. On a 1 défaut de cache toutes les 8 itérations pour `X` et 2 défauts de cache par itération pour `Y` et `Z`. On a donc 17 défauts de cache toutes les 8 itérations (2,125 par itération).

4 Caches Données (Bonus)

Soit le programme suivant, qui effectue la normalisation des colonnes d'une matrice `X[8][8]` : chaque élément de la colonne est divisé par la moyenne des valeurs de cette colonne.

```
float X[8][8]; float sum = 0.f;
for (size_t j = 0; j < 8; ++j)
{
    for (size_t i = 0; i < 8; ++i)
    {
        sum += X[i][j];
    }
    float average = sum / 8.f;
    for (size_t k = 8; k >= 1; --k)
    {
        X[k - 1][j] /= average;
    }
}
```

On suppose que l'on a un cache de 128 octets avec des blocs de 16 octets (soit 8 blocs pour le cache). L'adresse de `X[0][0]` est `0xF000` (sur 16 bits).

4.1 En supposant la correspondance directe, définir dans quelles lignes du cache vont chaque élément de la matrice. En déduire le nombre de défauts de cache pour l'exécution du programme. Quel serait le nombre de défauts de cache en écrivant la seconde boucle interne sous la forme : `for (size_t k=0; k<8; ++k)` ?

Le cache contient 8 blocs de 16 octets, donc on a 4 bits pour l'offset et 3 bits pour l'index.
Attention : Un `float` occupe 4 octets (32 bits), donc un bloc contient 4 éléments du tableau.

Les lignes du tableau sont stockées dans la mémoire successivement à partir de l'adresse `0xF000`. On décompose l'adresse de `X[0][0]` :

hex	tag	index	offset
0xF000	1111 0000 0	000	0000

On en conclut que `X[0][0]` est le premier mot du premier bloc (bloc 0) du cache. Dans le même bloc on aura aussi les éléments suivants `X[0][1]`, `X[0][2]` et `X[0][3]`. Les 4 éléments suivants (`X[0][4]` — `X[0][7]`) correspondent au bloc 1 du cache, et ainsi de suite. On obtient alors le schéma suivant, où on indique pour chaque bloc du tableau `X`, sur quel bloc du cache il sera stocké :

	0	1	2	3	4	5	6	7
0	bloc 0				bloc 1			
1	bloc 2				bloc 3			
2	bloc 4				bloc 5			
3	bloc 6				bloc 7			
4	bloc 0				bloc 1			
5	bloc 2				bloc 3			
6	bloc 4				bloc 5			
7	bloc 6				bloc 7			

Défauts de cache
Première colonne (`j=0`) :

- En descendant (boucle sur `i`) on a un défaut de cache par itération. (8 défauts au total)
- En remontant (boucle sur `k`), on n'a pas de défauts de cache pour les lignes du tableau 7,6,5,4, les blocs correspondants étant déjà présents dans le cache. On a 4 défauts de cache pour les lignes du tableau 3,2,1,0.

Colonnes `j=1,2,3` :

- En descendant, les éléments des lignes 0,1,2,3 sont déjà dans le cache grâce à l'itération précédente. 4 défauts de cache pour les lignes du tableau 4,5,6,7.
- En remontant, 4 défauts de cache pour les lignes du tableau 3,2,1,0.

Colonne `j=4` :

- En descendant, un défaut de cache par itération. (8 défauts au total)
- En remontant, 4 défauts de cache pour les lignes du tableau 3,2,1,0.

Colonnes `j=5,6,7` :

- En descendant, 4 défauts de cache pour les lignes du tableau 4,5,6,7.
- En remontant, 4 défauts de cache pour les lignes du tableau 3,2,1,0.

Au total, $12 + 8 + 8 + 8 + 12 + 8 + 8 + 8 = 72$ défauts de cache.

Boucle sur `k` dans le sens inverse
On perd l'avantage qu'on avait grâce à la localité temporelle. On a un défaut de cache par chargement d'élément de `X`. Au total, $8 \times 16 = 128$ défauts de cache.

4.3 Pour un cache associatif par ensemble 2 voies, définir dans quels ensembles vont les éléments de la matrice. Quel est le nombre de défauts de cache pour le programme initial en utilisant le LRU comme algorithme de remplacement ?

Avec 2 blocs par ensemble, le cache contient 4 ensembles de 2 blocs, donc on a 4 bits pour l'offset et 2 bits pour l'index. Voici le schéma indiquant dans quel ensemble du cache va chaque bloc du tableau :

	0	1	2	3	4	5	6	7
0	ensemble 0				ensemble 1			
1	ensemble 2				ensemble 3			
2	ensemble 0				ensemble 1			
3	ensemble 2				ensemble 3			
4	ensemble 0				ensemble 1			
5	ensemble 2				ensemble 3			
6	ensemble 0				ensemble 1			
7	ensemble 2				ensemble 3			

Défauts de cache
Première colonne (`j=0`) :

- En descendant, 1 défaut de cache par itération.
- En remontant (boucle sur `k`), 4 défauts de cache pour les lignes du tableau 3,2,1,0.

Colonnes `j=1,2,3` :

- En descendant, 4 défauts de cache pour les lignes du tableau 4,5,6,7.
- En remontant, 4 défauts de cache pour les lignes du tableau 3,2,1,0.

Colonne `j=4` :

- En descendant, 1 défaut de cache par itération.
- En remontant, 4 défauts de cache pour les lignes du tableau 3,2,1,0.

Colonnes `j=5,6,7` :

- En descendant, 4 défauts de cache pour les lignes du tableau 4,5,6,7.
- En remontant, 4 défauts de cache pour les lignes du tableau 3,2,1,0.

Au total, $12 + 8 + 8 + 8 + 12 + 8 + 8 + 8 = 72$ défauts de cache.