TP2 NMV : Topologie mémoire

Aymeric Agon-Rambosson

Lundi 25 janvier 2021

Exercice 1

Question 1

Initialiser la nouvelle zone mémoire permet de déclencher effectivement l'allocation. Si on n'avait pas touché la zone mémoire allouée, elle n'aurait pas vraiment été allouée (phénomène d'allocation paresseuse). De plus, la correspondance adresse virtuelle-physique est faite et est probablement gardée dans le TLB.

De cette manière, les potentiels bruits dans la mesure de la taille de la ligne sont éliminés.

Questions 2 et 3

La réponse à cette question dépend de la valeur de PARAM.

Le hit rate peut en fait être donné par la formule suivante : min((CACHE_LINE_SIZE - PARAM) / PARAM, 0)

Quand PARAM est à 1, le hit rate est maximal, proche de 1 (pour une valeur suffisemment grande de CACHE_LINE_SIZE).

Quand PARAM est légèrement inférieur à CACHE_LINE_SIZE, le hit rate est déjà très proche de 0.

Quand PARAM passe au-dessus de CACHE_LINE_SIZE, le hit rate est nul et le reste.

Question 4

Quand PARAM est très inférieur à CACHE_LINE_SIZE, les hit sont très majoritaires, donc la durée des cache hit déterminera beaucoup la durée d'exécution. Les cache miss ont beau être très minoritaire, leur durée étant plusieurs ordres de grandeur au-dessus de celle des hit, ils contribuent aussi beaucoup à la durée.

Quand PARAM est légèrement inférieur ou supérieur à CACHE_LINE_SIZE, la durée des miss est déterminante, ils sont majoritaires ou seuls.

Quand PARAM est inférieur à CACHE_LINE_SIZE, la durée d'exécution doit suivre une courbe inverse style 1/x + c, c étant une constante égale au rapport du temps d'un miss sur le temps d'un hit, et x une variable qui donne la proportion de miss multipliée par une constante. Quand on augmente la paramètre, x augmente, et donc la durée d'exécution diminue bien selon cette fonction. Cette tendance inverse est bien visibme

En revanche, quand on a passé CACHE_LINE_SIZE, tous les accès sont des miss, donc diviser par deux la quantité d'accès mémoire divisera presque exactement par deux "

Il n'y a pas vraiment trois étapes, il y en a deux :

- décroissance logarithmique en le paramètre (la courbe inverse est la dérivée du log) avant la taille de la ligne.

- décroissance linéaire en le paramètre après la taille de la ligne.

Donc le point de rupture entre les deux tendances de décroissance nous donnera la taille de la ligne.

Question 5

On pourrait avoir des variations de temps d'exécution liées à du bruit, potentiellement causé par les autres tâches qui s'exécutent sur la machine, ou n'importe quoi d'autre. Pour estomper l'effet de ce bruit, on peut faire une moyenne sur un grand nombre de boucles (ici un million).

La boucle de warmup permettrait de caler le système de prefetching du cache : en lui donnant un pattern d'accès mémoires sur un grand nombre de données d'entraînement, on lui permet de fonctionner de manière optimale pendant la mesure, plutôt que de manière aléatoire, ce qui permet encore une fois de supprimer le bruit.

Question 6

On verra la courbe dans le fichier pdf.

Même si l'échelle logarithmique des abscisses le cache un peu, la décroissance linéaire commence à partir de 64, ce qui est suffisant pour conclure que la ligne de cache de ma machine fait 64 octets (ce qu'on a pu vérifier par ailleurs).

Le plateau de 32 à 64 doit peut-être s'expliquer par des spécificités de mon matériel (Ryzen 3800x, processeur superscalaire), donc peut-être par la possibilité de paralléliser 2 accès au cache L1 en même temps, mais pas 4.

Exercice 2

Question 1

On admet que la zone mémoire dont il est question est contigüe : c'est une condition certes non nécessaire mais suffisante de ce qu'elle rentre en entier dans le cache L1 quand elle est de taille inférieure à la taille de ce dernier (on rappelle qu'on ne fait pas encore d'hypothèse d'associativité du cache).

Si PARAM est inférieur à la taille du cache L1, alors on aura (PARAM / CACHE_LINE_SIZE) miss lors de la première séquence d'accès, et 0 lors des suivantes, en admettant bien entendu que le cache ne se fait pas flush entre les séquences d'accès.

Question 2

Si PARAM est supérieur à la taille du cache L1, alors on aura (PARAM / CACHE_LINE_SIZE) miss lors de la première séquence d'accès, et PARAM / CACHE_LINE_SIZE - max((2 * CACHE_SIZE - PARAM) / CACHE_LINE_SIZE, 0). Plus simplement : si PARAM est au moins le double de CACHE_SIZE, alors on fera (PARAM / CACHE_LINE_SIZE) miss à toutes les séquences. Si PARAM est entre CACHE_SIZE et 2 * CACHE_SIZE, alors un peu moins aux séquences suivantes.

Question 3

En donnant un nombre de séquences d'accès très grand, on cherche à rendre le coût des miss compulsifs (première séquence) négligeables devant celui des suivants.

Donc quand PARAM est inférieur à CACHE_SIZE, alors le coût moyen d'un accès mémoire converge vers le coût d'un hit.

Quand PARAM est (plus de deux fois) supérieur à CACHE_SIZE, alors le coût moyen d'un accès mémoire converge vers ((PARAM / CACHE_LINE_SIZE) * coût d'un miss) + (1 - (PARAM / CACHE_LINE_SIZE)) * coût d'un hit.

Question 4

La méthode de détection va donc se servir de la différence des coûts moyens.

La solution est toute simple, elle va consister en faire des accès mémoire séquentiels (avec un pas égal à la taille d'une ligne, pour se simplifier les choses) sur des zones mémoire de plus en plus grande, de manière à faire varier le paramètre PARAM des questions précédentes.

On veut donc récupérer, selon les variations de PARAM, le temps moyen d'exécution de N accès mémoire : pas besoin que N égale 1, il doit juste être constant en PARAM.

On est censé avoir une courbe plate, et une rupture autour de la taille du cache L1, puis une autre autour celle du cache L2, avant d'atteindre un nouveau plateau.

Si les accès au cache L1 sont parallélisés par le matériel (ce qui a l'air d'être le cas sur notre machine), on peut avoir sur des valeurs faibles de PARAM un début de courbe descendante plutôt que plate, la rupture sera remplacée par un minimum global.

Voilà notre solution :

```
#include "hwdetect.h"

#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>


#ifndef PARAM
#define PARAM 64
#endif

#define CACHELINE_SIZE      64
#define MEMORY_SIZE         (1ul << 21)
#define WARMUP              10000
#define PRECISION           1000000


static inline char *alloc(void)
{
        size_t i;
        char *ret = mmap(NULL, MEMORY_SIZE, PROT_READ | PROT_WRITE,
                         MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);

        if (ret == MAP_FAILED)
                abort();

        for (i = 0; i < MEMORY_SIZE; i += PAGE_SIZE)
                ret[i] = 0;

        return ret;
}

static inline uint64_t detect(char *mem)
{
        size_t i, p;
        uint64_t start, end;
```

```
        for (p = 0; p < WARMUP; p++)
                for (i = 0; i < PARAM; i += CACHELINE_SIZE)
                        writemem(mem + i);

        start = now();

        for (p = 0; p < PRECISION; p++)
                for (i = 0; i < PARAM; i += CACHELINE_SIZE)
                        writemem(mem + i);

        end = now();

        return ((end - start) / PARAM);
}

int main(void)
{
        char *mem = alloc();
        uint64_t t = detect(mem);

        printf("%d %lu\n", PARAM, t);
        return EXIT_SUCCESS;
}
```

On regardera les deux courbes dans le répertoire pdf.

On a bien une taille de cache L1 de 32 Ko, et une taille de cache L2 de 512 Ko.

Exercice 3

Question 1

On suppose que les sets sont fait de manière à minimiser les miss de conflit.

Le nombre d'emplacements dans le cache est donné par la quantité NB_EMPLACEMENTS = (CACHE_SIZE / CACHE_LINE_SIZE), soit ici 512.

Si le cache est à correspondance directe, il faut que deux lignes soient séparées de exactement 512 * 64 octets pour être en conflit.

Si on prend un cache de même taille que le précédent et qu'on le rend associatif 2 voies, alors deux lignes séparées de 512 * 64 / 2 octets seront dans le même set.

Si on a n le degré d'associativité du cache, alors deux lignes séparées de 512 * 64 / n octets seront dans le même set.

Le cas limite est le cache full associatif : il y a un seul set qui correspond à tout le cache, ce set est donc dans notre cas de 512, et donc deux lignes séparées de 64 octets (soient voisines) seront dans le même set. Tout se tient.

Question 2

Prenons le cas du cache à correspondance directe : si on fait deux écritures dans le même cache set (mais sur des lignes de cache distinctes), alors on aura des miss compulsifs et des miss de conflit lors de la première séquence, et seulement des miss de conflit lors des suivantes.

Dans un pareil cas, avec un cache 2-associatif, on aurait eu seulement des miss compulsifs sur la première séquence et seulement des hit sur les séquences suivantes. Pour forcer des miss de conflit, on

aurait dû faire quatre écritures dans le même cache set.

Et ainsi de suite...

Avec le n le degré d'associativité du cache, il faut faire N = n*2 écritures dans le même cache set (sur des lignes toutes distinctes deux à deux, cela va sans dire) pour forcer des miss de conflit à chaque accès mémoire.

Question 3

On a donc notre expérience cruciale.

On rappelle les caractéristiques déjà connues de notre cache L1 : lignes de 64 octets pour une taille totale de cache de 32ko

Dans les différents programmes, on va faire des accès mémoire alignés sur des lignes de cache de la manière suivante. On se donne mem, un pointeur aligné sur 64 octets :

- un accès à mem, à mem + 64, etc..., jusqu'à mem + 32704, soient 512 accès en tout. Quelle que soit la conception du cache, on doit avoir des hit sur les séquences suivantes. Ce premier programme nous donnera la performance "ligne de base".
- un accès à mem, à mem + 32ki, à mem + 64, à mem + 32ki + 64, etc... jusqu'à mem + 16320, mem + 32ki + 16320. On note qu'on fait toujours un total de 512 accès. D'après notre protocole expérimental, les accès suivants sont censés donner uniquement des miss de conflit dans le cas d'un cache à correspondance directe, et uniquement des hit dans le cas d'un cache 2-associatif
- et ainsi de suite en augmentant le degré d'associativité jusqu'à 512, le maximum théorique sur mon matériel (512 emplacements donc full associative signifie 512-associative). Dans les faits on va s'arrêter à 32, aller plus loin serait irréaliste.

Tant que le degré d'associativité n'est pas atteint, on est censé avoir le même temps d'exécution, donc un plateau. Quand on atteint dépasse le degré d'associativité, le temps d'exécution est censé exploser (pas forcément atteindre un second plateau, forcer des degrés d'associativité très hauts peut avoir des effets différenciés sur les caches suivants).

Voilà donc notre méthode :

On a bien tenu compte des recommendations de la consigne : l'effet du tampon d'écritures postées est bien masqué par notre batterie de 512 accès mémoire par séquence.

```
#include "hwdetect.h"

#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>

#ifndef PARAM
#define PARAM 1
#endif

#define CACHELINE_SIZE      64
#define CACHE_SIZE          (1ul << 15)
#define NB_EMPLACEMENTS     512
#define MEMORY_SIZE         (1ul << 20)
#define WARMUP              10000
#define PRECISION           1000000

static inline char *alloc(void)
```

```
{
        size_t i;
        char *ret = mmap(NULL, MEMORY_SIZE, PROT_READ | PROT_WRITE,
                         MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);

        if (ret == MAP_FAILED)
                abort();

        for (i = 0; i < MEMORY_SIZE; i += PAGE_SIZE)
                ret[i] = 0;

        return ret;
}

static inline uint64_t detect(char *mem)
{
        size_t i, j, p;
        uint64_t start, end;

        for (p = 0; p < WARMUP; p++) {

                for (i = 0; i < NB_EMPLACEMENTS / PARAM; i += 1) {
                        for (j = 0 ; j < PARAM ; j += 1) {
                                writemem(mem + (j * CACHE_SIZE) + (i * CACHELINE_SIZE));
                        }
                }
        }

        start = now();

        for (p = 0; p < PRECISION; p++) {

                for (i = 0; i < NB_EMPLACEMENTS / PARAM; i += 1) {
                        for (j = 0 ; j < PARAM ; j += 1) {
                                writemem(mem + (j * CACHE_SIZE) + (i * CACHELINE_SIZE));
                        }
                }
        }

        end = now();

        return end - start;
}

int main(void)
{
        char *mem = alloc();
        uint64_t t = detect(mem);

        printf("%d %lu\n", PARAM, t);
        return EXIT_SUCCESS;
}
```

On verra dans le répertoire pdf les courbes.

On voit très bien le temps d'exécution qui explose après 8, ce qui correspond bien à notre matériel.

Exercice 4

L'exercice 4 sera fait si on a la possibilité de soumettre une nouvelle version du TP.