La hiérarchie mémoire

Sid Touati
Professeur à l'université Nice Sophia Antipolis

Le but de ces exercices pratiques est de vous montrer comment peut on évaluer les performances de la hiérarchie mémoire d'une machine (caches de divers niveaux, mémoire centrale, bande passante, TLB). Avant de débuter le TP, analysez rapidement la hiérarchie mémoire de votre machine en utilisant ce que nous avons appris dans un TP précédent. Ensuite, nous allons programmer ensemble des codes (micro-benchmarks) pour vérifier les performances de chaque niveau de la hiérarchie mémoire. Les exercices de ce TP doivent être effectués dans l'ordre pour mieux comprendre.

1 Tester les performances de la hiérarchie mémoire d'un CPU avec un micro-benchmark séquentiel

1.1 Détecter la taille d'un cache de niveau L1

Nous allons écrire un code (micro-benchmark) qui accède à des données (de type `int`) d'un tableau de façon répétée. Le but de ce code est d'évaluer la rapidité moyenne avec laquelle le CPU accède aux données de ce tableau : si toutes les données accédées sont dans le cache de niveau L1, alors le temps d'accès moyen à une donnée serait rapide (normalement). Afin de détecter par logiciel la taille du cache de niveau L1, le code doit tâtonner sur le nombre de données chargées dans le cache L1. Si la taille des données accédées dépasse largement la capacité du cache L1, nous devrions observer une augmentation brutale du temps d'accès moyen à une donnée. Le code du micro-benchmark ressemblerait au pseudo code suivant (à modifier ou à corriger selon besoin) :

```
#define MAX_TAILLE_DATA_KO 4096   // =4 Mo

int tab[MAX_TAILLE_DATA_KO * 1024];  // tableau des données accédées
int  i; // un compteur de boucle simple
int taille_data,   // qui correspond à la taille totale des données accédées en Ko
      nbdonnee; //  = (taille_data * 1024) / sizeof(int)
struct timeval t1, t2; // variables utilisées pour mesurer le temps d'exécution

float temps_acces_moyen; //en microsecondes

void main() {
for (taille_data=sizeof(int);  taille_data <= MAX_TAILLE_DATA_KO * 1024; taille_data++){

   nbdonnee= (taille_data * 1024) / sizeof(int);
   for (i=0; i < nbdonnee ; i++) { // boucle simple qui pré-charge les données dans le cache L1
         x=x+tab[i]
   } // end for i

   gettimeofday(&t1);   // récupérer ici la valeur de l'horloge juste avant la boucle
   for (i=0; i < nbdonnee ; i++) { // boucle qui accède aux données pré-chargées dans le cache
```

```
        x=x+tab[i]
    } // end for i
    gettimeofday(&t2);  // récupérer la valeur de l'horloge à la fin de la boucle

   // attention, la formule ci-dessous oublie de récolter le nombre des secondes écoulées
    // il faut corriger la formule en utilisant t2.sec et t1.sec
    temps_acces_moyen= (float) (t2.tv_usec - t1.tv_usec) / nbdonnees;

    printf("Le temps d'accès moyen est de \%f microsecondes lorsque la taille totale \\
              des données accédées est de \%d KO", temps_acces_moyen,  taille_data);

} // endfor taille_data
} // end main
```

1. Étudiez le pseudo-code du micro-benchmark ci dessus pour comprendre ce qu'il fait exactement.
2. Programmez un micro-benchmark en C en vous inspirant du pseudo-code ci-dessuss. Compilez le puis exécutez le sur votre machine de test. Attention, pour que vos expériences soient crédibles, il faut vous assurer que votre machine de test soit très peu chargée : il faut faire en sorte qu'il y ait très peu de processus qui tournent pour alléger la machine... Idéalement, la machine de test ne devrait exécuter que le micro-benchmark tout seul !

   Remarque : si vous souhaitez que votre micro-benchmark soit toujours exécuté sur le même CPU (cœur), utilisez la commande Linux `taskset` pour imposer au noyau système de l'exécuter sur un CPU précis. Par exemple, la commande `taskset -c 3 ./monprog` exécute le programme `./monprog` sur le CPU numéro 3.
3. Faites en sorte que votre micro-benchmark imprime des données de performances sous la forme tabulée de couples `"taille_data : temps_acces_moyen"`, un couple par ligne. Utilisez un logiciel de votre choix (gnuplot, LibreOffice, R, etc.) pour dresser une courbe qui visualise le temps d'accès moyen d'une donnée par rapport à la taille des données accédées. Que déduisez vous ?
4. Rééxecutez votre micro-benchmark plusieurs fois. Obtenez vous une courbe similaire d'une exécution à une autre ?
5. Ajoutez une boucle de répétition dans le code de votre micro-benchmark : l'objectif de cette boucle de répétition et de rééxecuter plusieurs fois la la boucle qui évalue le temps d'accès moyen d'une donnée. Le pseudo-code du micro-benchmark deviendrait :

   ```
   ...
    gettimeofday(&t1);   // récupérer ici la valeur de l'horloge juste avant la boucle de répétition
    for (j=0; j < nbrepetition; j++)  // boucle de répétiton
      for (i=0; i < nbdonnee ; i++) { // boucle qui accède aux données pré-chargées dans le cache
            x=x+tab[i]
       } // end for i
    gettimeofday(&t2);  // récupérer la valeur de l'horloge à la fin de la boucle de répétition
    temps_acces_moyen= (float) (t2.tv_usec - t1.tv_usec) / (nbdonnees * nbrepetition);
    // attention la formule ci-dessus est à corriger en ajoutant les secondes écoulées
   ...
   ```

   Le nombre de répétitions est à choisir par l'utilisateur, l'objectif des répétition est d'avoir un comportement stationnaire (stable) des expériences. Exécutez maintenant votre micro-benchmarks plusieurs fois. Obtenez vous une courbe similaire d'une exécution à une autre ?

1.2 Détecter les tailles des caches des autres niveaux si présents sur votre machine (L2, L3, etc)

Dans l'exercice précédent, le micro-benchmark incrémente petite à petit (par tâtonnement) la taille des données accédées pour évaluer le temps d'accès moyen au cache L1. Tant que les données sont pré-chargées dans le cache L1, le temps d'accès moyen est rapide. Lorsque la taille des données accédées déborde sur la taille du cache L1 (précisément, lorsqu'elle dépasse le double de la taille du cache), le temps d'accès moyen augmenterait brutalement.

Afin d'évaluer les temps d'accès moyens à tous les niveaux de caches intermédiaires (L2, L3, etc), le micro-benchmark doit être paramétré pour que les tailles des données accédées soient de plus en plus grandes. Précisément, il faut que la taille maximale (paramètre `MAX_TAILLE_DATA_KO`) des données accédées soit au minimum égale au double de la capacité du dernier niveau de cache de votre CPU (L2 ou L3, tout dépend du dernier niveau de cache de votre machine de test). Par exemple, si votre dernier niveau de cache est L2 avec une capacité de C, la valeur du paramètre `MAX_TAILLE_DATA_KO` doit dépasser $2 \times C$. Modifiez les paramètres du micro-benchmark de l'exercice précédent pour évaluer le temps d'accès moyen à tous les niveaux de cache de votre machine. Dressez la courbe qui dessine le temps d'accès moyen par rapport à la taille des données accédées. Déduisez les tailles des divers niveaux de caches de votre machine de test.

2 Évaluation de la bande passante entre le processeur et la mémoire centrale

Maintenant, nous souhaitons évaluer le temps d'accès moyen d'une donnée stockée en mémoire centrale seulement. En d'autres termes, nous souhaitons évaluer le temps d'accès moyen à des données qui ne sont chargées dans aucun cache (d'aucun niveau). Le micro-benchmark de l'exercice 1 accède à des données consécutives en mémoire (`tab[i], tab[i+1], tab[i+2], tab[i+3], ...`, etc), le premier accès (par exemple `tab[i]`) serait éventuellement un défaut de cache qui chargerait une ligne mémoire dans le cache. Mais les autres accès successifs seraient des accès à des données présentes dans le cache.

1. Comment s'assurer qu'une boucle accède à des données d'un tableau qui ne sont chargées dans aucun cache ? Étudiez des solutions qui utilisent des pas d'accès pour sauter des données en mémoire (sans oublier le mécanisme de pré-chargement matériel s'il est présent dans votre machine de test). Pour vous aider, le pseudo-code du micro-benchmark de l'exercice 1 doit être modifié comme suit :

   ```
   // la variable "pas" ci-dessous permettrait d'accéder à des données non consécutives en mémoire
   for (i=0; i < nbdonnee ; i=i + pas) { // boucle qui accède à des données non consécutives
           x=x+tab[i]
   } // end for i
   ```

   Votre objectif ici est de calculer une valeur adéquate pour le "pas" afin que la boucle accède à une donnée stockée en mémoire centrale à chaque itération.

2. Programmez un micro-benchmark qui évalue le temps d'accès moyen d'une donnée stockée en mémoire centrale.

3. Déduisez un micro-benchmark qui évalue passante entre le CPU et la mémoire centrale. Attention, lorsque le CPU accède à une donnée en mémoire centrale, il charge une ligne de cache entière et non pas uniquement une donnée.

4. Que se passe-t-il si la valeur du pas est trop grande ? (qui dépasse la taille d'une page mémoire par exemple ?)

3 Évaluation des performances du TLB

Le TLB (*translation lookaside buffer*) est une table matérielle présente au sein du CPU qui contient des traductions d'adresses de pages virtuelles vers des pages physiques. A chaque accès mémoire effectué par un programme, le CPU accède au TLB pour faire rapidement une traduction d'une adresse virtuelle vers une adresse physique. Le TLB est une table associative, sa taille est limitée. Elle ne peut pas contenir les adresses de toutes les pages virtuelles des processus qui tournent dans le système. Ainsi, seulement quelques adresses virtuelles de quelques pages sont stockées dans le TLB. Le nombre d'entrée de TLB est une décision architecturale.

Si un processus effectue un accès mémoire, et que l'adresse de la page virtuelle accédée n'est pas présente dans le TLB, un **défaut TLB** se produit. Cela veut dire que le CPU va d'abord accéder à la mémoire centrale (à la table des pages maintenue par le noyau système) pour récupérer la traduction d'adresse virtuelle-physique de la page accédée. Cette traduction d'adresse est mise dans le TLB (en éjectant une autre entrée du TLB si pas de place). Une fois la traduction d'adresse effectuée, le CPU peut enfin accéder à la donnée demandée par le processus.

Le but de cet exercice est de tester les performances du TLB.

1. Comment s'assurer qu'une boucle accède à une page virtuelle différente à chaque itération ? Étudiez des solutions qui utilisent des pas d'accès pour sauter des données en mémoire. Pour vous aider, une partie du pseudo-code du micro-benchmark de l'exercice 1 doit être modifié comme suit :

   ```
   // la variable "pas" ci dessous permettrait de sauter des données en mémoire
   for (i=0; i < nbdonnee ; i=i + pas) { // boucle qui accède à des données non consécutives
           x=x+tab[i]
   } // end for i
   ```

   Votre objectif ici est de calculer une valeur adéquate le "pas" pour que la boucle accède à une page virtuelle différente à chaque itération.

2. Programmez un micro-benchmark qui accède à `NB_PAGES` pages virtuelles différentes, en utilisant la boucle de la question précédente. `NB_PAGES` est un paramètre du micro-benchmark qui peut être défini à volonté (soit avec une directive de compilation `#define`, soit comme paramètre d'entrée au programme). Si vous estimez que la taille du tableau `tab` devient trop grande pour une allocation statique globale, je vous suggère de faire une allocation dynamique du tableau en utilisant la fonction `calloc`.
3. En faisant varier la valeur de `NB_PAGES`, et en mesurant les performances des accès mémoire avec votre micro-benchmark, essayez de déduire le nombre d'entrées du TLB de votre machine de test.

4 Compteurs matériels de performances (sur Linux/x86)

Dans les exercices précédents, vos micro-benchmarks effectuent des traitements simples pour tester et évaluer les performances micro-architecturales d'une machine. Il est parfois très difficile d'analyser et comprendre les performances obtenues en examinant uniquement les temps d'exécution mesurés en micro-secondes. Lorsqu'un processus s'exécute, il déclenche le fonctionnement de divers composants micro-architecturaux du CPU, qui sont souvent complexes à analyser avec des mesures d'expériences simple comme le temps d'exécution.

Afin d'aider un peu plus l'analyse et la compréhension des performances observées, les compteurs matériels de performances (registres spéciaux), si présents dans le CPU, peuvent être exploités. Ces compteurs enregistrent plein d'événements matériels relatifs aux performances qui sont provoqués par l'exécution de processus : nombre de cycles d'horloge, nombre d'accès mémoire, nombre d'accès au cache L1, nombre de

défauts de cache, etc. La nature des événements enregistrés par les compteurs matériels de performances dépend d'une architecture de CPU à une autre. La portabilité n'est donc pas assurée, mais les mesures sont extrêmement précises.

Notons deux façons pour utiliser les compteurs matériels de performances :

1. **Analyser les événements matériels d'un CPU qui exécute plusieurs programmes en concurrence**. Par exemple, l'outil `Likwid` installé sur vos machines de l'université récupère les valeurs des compteurs matériels par cœur (et non pas par processus). Il peut agréger les compteurs matériels de tous les cœurs. Cette façon d'utiliser les compteurs matériels de performances permet d'avoir une vue globale du fonctionnement et des performances d'un cœur ou d'un système entier qui exécute plusieurs processus en même temps.
2. **Analyser les événements matériels provoqués par un seul processus qui s'exécute sur une machine**. C'est la façon à utiliser pour analyser et éventuellement optimiser les performances d'un programme particulier qui s'exécute sur une architecture matérielle précise.

Dans cet exercice nous allons utiliser la deuxième méthode ci-dessus. Là aussi, notons deux façons pour utiliser des compteurs matériels de performances pour un code donné :

1. Soit l'utilisateur est intéressé par l'analyse des performances d'un bout son programme (par exemple étudier les performances d'une boucle précise ou d'une fonction particulière). Dans ce cas, il faut détenir le code source pour l'instrumenter (le modifier). Le code modifié contiendrait des instructions utilisant des librairies d'accès aux compteurs matériels de performances. La librairie `LibPFM` permet par exemple à un programme d'accéder aux valeurs des compteurs matériels de performances.
2. Soit l'utilisateur est intéressé par l'analyse des performances d'un processus entier (pas uniquement un bout du programme). Dans ce cas de figure, l'utilisateur n'a pas besoin d'instrumenter le code source. Il peut utiliser des outils en ligne de commande comme `perf` pour évaluer les performances du code binaire.

A titre d'illustration, utilisons la commande `perf` pour analyser les performances des micro-benchmarks des exercices précédents.

1. Listez les événements matériels pouvant être enregistrés par la commande `perf` sur votre machine de test. À cet effet, la commande `perf list` peut vous aider. Intéressons nous uniquement aux événements matériels liés à la hiérarchie mémoire : nombre de *loads* et de *stores* exécutés, nombre de défauts de cache, nombre d'accès au TLB, nombre de défauts TLB, etc.
2. Reprenez les codes des micro-benchmarks des exercices précédents. Nettoyez les des diverses instructions inutiles aux expériences (par exemples les instructions d'affichage et de collecte des valeurs d'horloge ne sont pas utiles pour les expériences de cet exercice).
3. Pour enregistrer les événements matériels provoqués par un micro-benchmark, utilisez la commande `perf record` suivie du micro-benchmark binaire à exécuter. Lorsque le programme binaire finit de s'exécuter, un fichier `perf.data` est généré. La commande `perf report` permet de lire ce fichier `perf.data` et d'afficher un rapport.
4. Utilisez la méthode décrite ci-dessus pour :
   (a) Analyser les nombres de défauts de cache L1 du micro-benchmark de l'exercice 1. Comment est ce que ces mesures vous permettraient de déduire la taille du cache L1 ?
   (b) Analyser les nombres de défauts de TLB du micro-benchmark de l'exercice 3. Comment est ce que ces mesures vous permettraient de déduire le nombre d'entrées dans le TLB ?