Temps-réel en multi-cœurs : problème de la contention mémoire

Louisa BESSAD & Roberto MEDINA

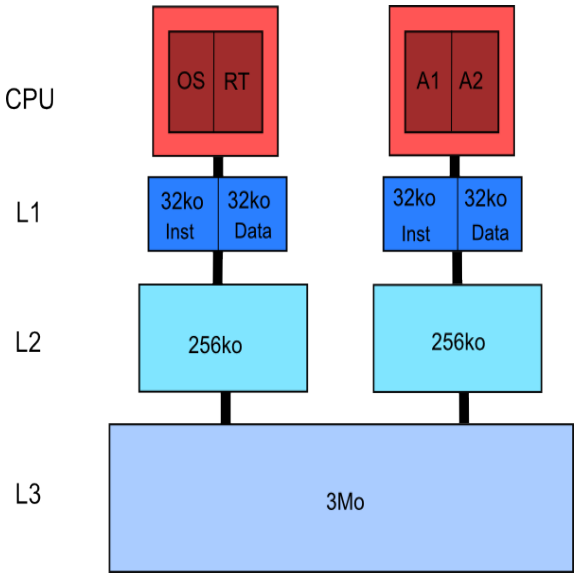
30 avril 2014

Table des matières

1 Problème de concurence d'accès

1.1 Introduction

- Description du sujet
- Problématique : géstion de l'accès concurrent aux caches et à la mémoire entre une tâche temps-réel et des tâches attaquantes en sachant qu'on ne peut faire que des mesures globales dans la mémoire et pas pour chaque cœur
- Objectif : proposer une solution en mode user en utilisant une librairie particulière (PAPI) pour les mesures
- Description de l'architecture : intel Core I3-330M Processor(2.13GHz, 3MB L3 Cache), approximation des accès mémoire par les MISS sur le L3 + schéma de la différence entre les machines qui ont fait les tests.

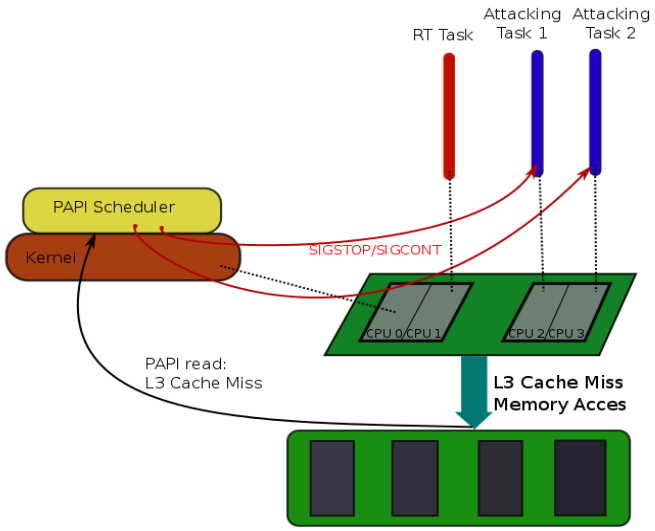


1.2 Les différentes tâches (attaquantes, temps réel)

- Description des 2 types de tâches et leur impact sur la mémoire :

  - on a une tâche temps-réelle non périodique et sans contrainte temporelle, on doit uniquement être sûre qu'elle se termine en un temps fini
  - les tâches attaquantes doivent effectuer un maximum d'accès mémorie afin de saturer le bus mémoire et les caches partagés entre les différents cœurs
- Evolution des choix face au prefetching, régime transitoire et permanent, les printf

1.3 Comment mesurer ?



Pour faire les mesures correspondantes à la tâche temps-réel on utilise la librairie PAPI (Performance Application Programming Interface) décrite au par avant. Notamment on utilise les fonctions de bas niveau pour avoir une meilleure granularité sur les données.

Dans un premier temps on utilise un seul « event set » qui va indiquer à PAPI les différentes mesures qu'on veut faire, c'est-à-dire le nombre de cache MISS et HIT pour les niveaux L1, L2 et L3. Ce qui nous permet donc avoir un taux de cache HIT pour les différents niveau, de même que le nombre d'accès à la mémoire (nombre de cache miss au niveau L3).

Le programme qui se charge de faire les mesures s'appelle « papi_wrapper » (src/papi_wrapper.c). Son fonctionnement consiste à initialiser PAPI avec toutes les options nécessaires : attacher l'événement à un seul CPU, changer le domaine de mesures et la granularité, initialiser l'event set ; il faut aussi utiliser un multiplexage pour arriver à faire toutes les mesures sur un seul event set, notamment avoir le nombre de cache MISS du L3 au même temps que les autres caches. Les

mesures se font sur la période stationnaire de la tâche temps réel. Les compteurs sont initialisés après que la tâche temps-réel est lancée et ils sont arrêtés avant que le processus se termine.

Comme ses fonctionnalités vont être utilisées par le scheduler plus tard dans le projet, l'initialisation des éléments utilisés pour le faire le benchmarking vont être mis dans un fichier utilitaire appelé « papi_util » (src/papi_util.c).

Pour arriver à faire le benchmarking le wrapper va se forker, lancer la tâche temps-réel en la clouant à un seul CPU, avec la plus haute priorité FIFO ; on utilise les fonctions du scheduler (sched_setaffinity, sched_setscheduler) pour arriver à ceci. On ne veut pas que le processus fasse une migration de CPU parce que les mesures ne vont pas correspondre à la bande passante de la tâche temps-réel. Dans notre cas on utilise le deuxième CPU pour la tâche temps-réel. Une fois forké, le wrapper va attendre la période stationnaire de la tâche temps-réel et mesurer les événements grâce à PAPI pendant une période de huit secondes. Après il attend la fin de son fils et fait la destruction du set pour se finir en affichant le temps d'exécution et en ajoutant les données qui seront utilisés par gnuplot sur des fichiers texte.

Le wrapper peut prendre n'importe quel programme pour le lancer et faire des mesures de performance. Cependant il va clouer la tâche passée en paramètre au deuxième cœur tout le temps. PAPI va nécessiter les droits de super-utilisateur pour arriver à attacher l'event set utilisé à un seul CPU (PAPI_set_opt avec l'option PAPI_CPU_ATTACH).

Pour avoir une meilleure perception sur la performance de la tâche temps-réel on va isoler les cœurs qui vont être utilisés pour la tâche temps-réel, et pour les attaquants. Dans l'architecture utilisée pour les tests on dispose de quatre cœurs : on va utiliser un pour l'OS, un pour le temps-réel et les deux autres pour les attaquants. L'isolation se fait avec la ligne de commande du noyau. On utilise l'option « isolcpus » passés au démarrage à travers GRUB 2.

Lancer d'autres tâches attaquantes sur les CPUs déjà utilisés ne va plus influencer le temps d'exécution de la tâche temps-réel. On va avoir une stabilisation de la courbe une fois qu'on n'a plus de CPUs disponibles.

En fonction du gouverneur du CPU les résultats peuvent varier, on a fait les mesures en utilisant le gouverneur « powersave » et « performance ». On peut considérer que dans un système embarqué on va préférer un gouverneur de type « powersave ». L'ensemble des tests ont été faits sur des noyau récents : 3.13.x .

Un facteur qui est pris en compte pour le benchmark c'est le prefetching fait par le compilateur et par le CPU. C'est difficile de savoir comment va réagir le CPU ou le compilateur pour les tâches attaquantes et la tâche temps-réel. On a donc deux modes de parcours de liste chaîné pour les attaquants. Un mode où la liste est parcourue linéairement donc le prefetching du compilateur et du CPU devraient être utilisés fortement dû à localité spatiale. Un autre mode avec un parcours aléatoire où on ne veut pas que le prefetching soit utilisé, on ne veut pas que le compilateur où le CPU connaissent la prochaine instruction ou donnée qui va être sollicitée.

La tâche temps-réel va faire un parcours aléatoire de deux tableaux. Les tableaux sont assez grands pour qu'ils soient mis sur la mémoire et pas seulement

sur le cache L3. Comme pour les tâches attaquantes le parcours aléatoire des deux tableaux va rendre le prefetching impossible ou minimal.

Effectivement le but est de solliciter le plus possible le bus mémoire du processeur pour toutes les tâches.les tests on dispose de quatre cœurs : on va utiliser un pour l'OS, un pour le temps-réel et les deux autres pour les attaquants. L'isolation se fait avec la ligne de commande du noyau. On utilise l'option « isolcpus » passés au démarrage à travers GRUB 2.

1.4 Résultats et analyse des courbes

Les premiers résultats ont été faits avec un gouverneur « performance » et en utilisant un parcours aléatoire pour les tâches attaquantes. On a lancé cinquante fois le wrapper sans tâches attaquantes au début, après avec une tâche attaquante pour finir avec deux attaquants, donc un total de cent cinquante exécutions pour le wrapper. Les tâches attaquantes vont faire un parcours aléatoire en se lançant sur un autre terminal pour que l'OS puisse forcer le pourcentage d'utilisation du CPU. Ces processus vont contenir 900.000 éléments. La tâche temps réel va faire vingt millions d'itérations avant de se terminer. (Script lancer_benchmark.sh).

Les résultats montrent une augmentation de 14% sur le temps d'exécution pour la tâche temps-réel quand on rajoute un attaquant. Et une augmentation de 35% avec deux attaquants. Pour le processus temps-réel seul, le temps d'exécution est aux alentours de 55 secondes, avec un attaquant on monte à 64 secondes et avec trois attaquants on arrive jusqu'à 89 secondes.

Le nombre d'accès au caches de niveau inférieur et à la mémoire vont augmenter aussi. Pour les miss de niveau L1 on a une augmentation de facteur 1.9 pour les cache MISS totaux (instructions et données) avec un attaquant. Puis une augmentation de facteur 3.5 avec deux attaquants par rapport à un seul attaquant.

Le taux de MISS du L2 ne vas pas avoir des variances aussi grandes puisque les tâches attaquantes ont leur propre cache L2, la taille du cache est aussi un facteur qui influence beaucoup les mesures. Le cache utilisé par la tâche temps-réel est celui qui est partagé avec l'OS. Entre zéro et un attaquant la différence est 1.57 taux de MISS. Et entre un et deux attaquants le rapport est de 1.81 .

Enfin, le taux de MISS pour le niveau L3 va avoir des rapports plus élevés puisque qu'il est partagé avec les autres tâches attaquantes. On a un facteur de 6.23 pour un attaquant et un facteur de 1.13 pour deux attaquants. Ceci est dû à la surcharge du cache L3. Avec un seul attaquant le cache est déjà saturé puisqu'on utilise autour de 200 Mo de mémoire pour chaque attaquant. De même les mesures faites par PAPI concernent toutes les tâches : attaquants et temps-réel. Donc les accès mémoires sont mélangés.

Le cache partagé (niveau L3) va être le plus pollué par les données et les instructions des tâches attaquantes. Ceci va avoir des influences sur les autres niveau de cache pour la tâche temps-réel : quand la donnée qui doit être chargée par le parcours aléatoire ne va pas se trouver sur le niveau de dessous c'est très

probable que la donnée se trouve dans la mémoire et pas dans le cache de niveau L3 puisque ce cache est pollué et saturé par les tâches attaquantes. De même les tableaux gérés par le processus temps-réel ont une taille plus grande que tout le niveau L3 du cache c'est-a-dire plus de 2Mo de données. Les données et les instructions vont être remplacées assez souvent dans le cas du parcours aléatoire et un peu moins souvent quand les tâches utilisent un parcours linéaire.

1.5 Conclusion

Le comportement des caches et de la mémoire avec des attaquants explique très bien comment les ressources partagées vont influencer le temps d'exécution de la tâche temps-réel.

Les benchmarks faits sur l'architecture Intel nous permettent de conclure que les tâches attaquantes vont être en concurrence avec la tâche temps-réel, surtout au niveau de la mémoire. Ceci a des conséquences directes sur le temps d'exécution de la dernière, ce qui peut va impacter sur le pire cas que doit assurer la tâche temps-réel.

2 Problème de la sous-réservation de BP mémoire :

2.1 Introduction

- Explication du problème de sous-réservation de BP mémoire
- Les problèmes que cette solution peut générer : comment gérer les accès mémoire tout en gardant une certaine concurrence
- Une solution proposée : Création du scheduler envoyant des signaux aux différentes tâches lorqu'elles ont consommées toute la BP accordé et que la tâche temps-réel n'a pas encore utilisé sa BP et remise à zéros des différents compteurs de BP dans ce cas

2.2 Le scheduler, la tâche temps-réel et les tâches attaquantes

La solution que nous proposons en mode utilisateur s'appelle « papi_scheduler » (src/papi_scheduler.c). Ce scheduler va reprendre les fonctionnalités de base de « papi_wrapper » et va utiliser aussi le fichier utilitaire « papi_util », en clouant une tâche temps-réel au deuxième CPU et en faisant des mesures de performance pour ce processus.

Ce qui a été rajouté au wrapper pour faire un type d'ordonnancement est un timer POSIX qui se déclenche toutes les 0.025 secondes, pour que le scheduler vérifie combien d'accès mémoire ont été effectués. On ne dispose pas d'interrupteurs matériels qui peuvent détecter si on dépasse un quota d'accès mémoire. Ce timer utilise un signal temps-réel (SIGTRMIN) et le handler qui a été défini pour ce singal va se charger de consulter les valeurs combien d'accès mémoire ont été faits. Un event set spécifique (scheduler_eventset) pour les accès mémoire a été rajouté puisqu'il doit être réinitialisé toutes après chaque fenêtre d'exécution.

Les fenêtres d'exécution peuvent correspondre aux échéances de la tâche temps-réel. Par exemple, dans notre cas, on veut assurer que toutes les cinq secondes le nombre d'accès mémoire ne dépasse pas les 129.000 accès. Ce nombre d'accès correspond aux nombre d'accès faits par l'OS et la tâche attaquante qui

tournent sur le premier et le deuxième cœur. Évidemment ce nombre peut varier dû à des interruptions ou à d'autres comportements du système d'exploitation. En fonction des architectures le nombre d'accès peut varier aussi.

Plusieurs aspects doivent être pris en compte pour que le scheduler ait le comportement voulu : si le timer se déclenche pour des périodes qui sont en dessous de 0.01 secondes, PAPI va mesurer 0 accès mémoire et les compteurs vont être saturés. Si on prend une période plus longue on perd en précision pour savoir quand est-ce que le quota d'accès mémoire a été dépassé et les tâches attaquantes vont être arrêtes trop tard, ce qui va provoquer une augmentation sur le temps d'exécution de la tâche temps-réel.

Si cette période est trop longue aussi ça peut arriver que les processus attaquants n'arrivent pas à s'arrêter complètement puisque la nouvelle fenêtre d'exécution va arriver plus tôt. Si le quota d'accès mémoire est dépassé très vite les tâches attaquantes ne vont pas pouvoir s'exécuter pendant que la tâche temps-réel s'exécute. Le problème va dans l'autre sens aussi, si la fenêtre d'exécution est trop courte les tâches attaquantes ne vont pas réussir a être arrêtés et le temps d'exécution va être comparable à celui du wrapper.

Le schéma général d'exécution du scheduler est le suivant :

Après avoir initialisé toutes les variables qui vont être utilisés pour les mesures en utilisant les fonctions de « papi_util », le scheduler initialise le timer qui va être utilisé en définissant le handler pour celui-ci. Ce programme prend en paramètre le nombre d'attaquants qui vont être lancés avec la tâche temps-réel. Le scheduler se forke le nombre de fois que l'utilisateur a demandé en lançant les processus attaquants sur des émulateurs de terminaux. Les émulateurs commencent à s'exécuter sur les CPUs 3 et 4 mais l'OS les fait migrer la plus part du temps vers le CPU 0, ceci doit être un effet de bord du fait d'utiliser le serveur X sur un seul cœur. Cependant les tâches attaquantes sont bien clouées aux CPUs 3 et 4. Par défaut les processus attaquants vont utiliser un parcours aléatoire et vont créer 900.000 éléments. Les conditions sont donc les mêmes que pour le wrapper. Il faut attendre la période stationnaire des tâches attaquantes pour pouvoir lancer la tâche-temps réel puisqu'au début les attaquants font l'allocation des éléments de la liste.

Une fois que cette étape a été atteinte, le scheduler va se forker une dernière fois pour lancer la tâche temps-réel sur le deuxième CPU, exactement pareil que le wrapper.

Pendant que la tâche temps-réel va s'exécuter, le scheduler va recevoir les SIGTRMIN du timer. Il consulte donc la valeur du compteur d'accès mémoire. Il va décrémenter une variable « rt_quota_l3 » initialisé avec le nombre d'accès mémoire définis dans une macro. Si la variable atteint zéro ou une valeur négative, le scheduler va envoyer un signal SIGSTOP aux attaquants. Pour ne pas renvoyer les signaux on a une autre variable qui sert de boolean. Chaque fois que le handler va être exécuté une variable « new_window » va être décrémentée. Quand la variable atteint zéro, alors il faut commencer une nouvelle fenêtre d'exécution. Le scheduler va envoyer les SIGCONT aux tâches attaquantes et réinitialiser toutes les variables globales pour le quota, les signaux et pour la fenêtre d'exécution. Ce procédé est répeté jusqu'à que la tâche temps-réel se

termine.

En plus de l'ordonnancement, le scheduler va faire les mesures de performance exactement de la même manière que le wrapper, en écrivant sur d'autres fichier pour que gnuplot puisse les utiliser après pour faire les graphes.

2.3 Mesures, résultats et analyse

- Changement de la méthode de mesure
- Analyse des courbes obtenues avec la solution proposée

2.4 Conclusion

- La solution résout-elle le problème de contention mémoire, si oui pourquoi ?

3 Conclusion face au sujet proposé :

- Avantages et inconvénients de l'utilisation du scheduler et comparaison avec la première partie
- Ouverture : Existe-t-il une solution plus optimale ?