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

Louisa BESSAD & Roberto MEDINA

4 mai 2014

Table des matières

1 Problème de concurence d'accès

1.1 Introduction

Lorsque nos véhicules ont commencés à adopter des architectures logicielles concernant l'automobile, ces dernières se contentaient seulement de respecter des contraintes au niveau du système embarqué, temporelles et de minimisation de coûts. De nos jours, l'évolution de nos besoins entrainent l'intégration de plus en plus de systèmes multimédia au sein des véhicules. Ces derniers sont basés sur Linux et Androïd et utilisent des calculateurs différents des systèmes temps-réel. Grâce à des processeurs multicœurs nous allons pouvoir diminuer les coûts. En effet nous souhaitons éviter l'utilisation de plusieurs processeurs : certains pour les tâches temps réel et d'autre pour les applications multimédia. Si l'on utilise la virtualisation sur de telles architectures, nous pourrons éxecuter en parallèle des systèmes temps-réels et des systèmes multimédias et cela sur un même processeur.

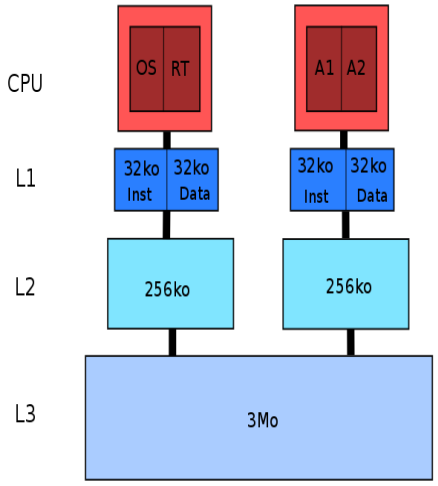
Néanmoins pour permettre ce parallèlisme il est important de garantir le respect des contraintes de sûreté et de sécurité, cela est d'autant plus important que le système doit gérer des tâches temps-réel. Pour obtenir cette garantie il faut assurer l'isolement spatial et temporel entre les différents systèmes, ce contrôle sera géré par un hyperviseur que nous appelerons scheduler. De façon générale, les hyperviseurs dévéloppés pour les systèmes embarqués fixent sur un ou plusieurs cœurs les différents OS virtualisés et ce de manière exclusive. De plus chaque système d'exploitation virtualisé se voit attribué une portion de mémoire statique, par ailleurs l'isolation est assurée par les mécanismes de protection mémoire. Ainsi le partionnement de la mémoire et du processeur sont assurés. Cependant certaines ressources restent partagées, c'est le cas du bus système et des caches. De fait si un cœur effectue de nombreux accès mémoires il peut provoquer une saturation du bus et des caches partagés. Cela empêche les autres cœrs d'agir correctement, ce qui est un problème pour les tâches temps-réel, il faut donc réussir à gérer l'accès concurent à ces ressources.

L'objectif de ce projet est de proposer une solution en mode user au problème de contention mémoire pour des applications temps-réel s'exécutant sur des processeurs multi-cœurs. Il faut également mesurer les ralentissements en-

gendrés sur les différentes tâches qui s'exécutent suite à la solicitation des différents éléments du système mémoire (Bus et Cache). Pour effectuer ces mesures nous utiliserons la librairie PAPI, nous pourrons uniquement faire des mesures gloables au niveau des accès mémoire et pas pour chaque cœur du processeur. Ainsi, nous mettrons d'abord en évidence le problème via une solution sans scheduler. Ensuite nous comparerons les mesures de la solution sans scheduler à celles d'une solution avec scheduler.

Notre plate-forme de tests faite pour ce projet a été testée sur une architecture Intel i3-330M, première génération pour ce type de processeurs, sur un système d'exploitation Ubuntu 14.04 avec le noyau Linux 3.13. Le processeur dispose de deux cœurs physiques donc quatre cœurs logiques avec hyper-threading activé. On dispose d'un cache de niveau L1 de 64ko pour les instructions et 64ko pour les données sur chaque cœur. Le cache de niveau L2 est aussi exclusif aux deux cœurs, mais les données et les instructions sont mélangées, sa taille est de 256ko. Enfin, le cache de niveau L3 fait 3Mo. Celui-ci est partagé par les deux cœurs.

La figure ci-dessous montre un schéma de l'architecture utilisée décrite auparavant. On peut aussi constater la disposition des tâches attaquantes et de la tâche temps-réel. Nous aborderons ce sujet dans la partie qui montre comment nous avons effectué les mesures.



Au début du projet, nous avons également réalisé des tests sur un processeur Intel i7-3517U : deux cœurs physiques et quatre logiques avec un L1 de 64ko pour les données et 64ko pour les instructions, un L2 de 256ko et un L3 partagé de taille 4Mo qui ne séparaient pas instructions et données. Cette machine utilisait un système Arch Linux avec un noyau 3.14. Les résultats n'étaient pas satisfaisant car ce processeur est utilisé sur des portables de type ultrabook. Quand le

processeur recevait trop de charge il activait son mécanisme de « Turbo Boost » pour répondre à la grande demande faite par l'utilisateur. La cadence pouvait passer de 1.9 GHz à 3.0GHz. Ce comportement faussait les mesures puisque la tâche temps-réel s'exécutait plus rapidement quand la charge était plus importante, le système voulait alors se débarrasser le plus rapidement de ce processus pour laisser de la place aux autres tâches. Le fait que la tâche temps-réel soit maintenue moins de temps sur le processeur faisait que le nombre d'accès aux caches et à la mémoire était inférieur aux nombre d'accès lorsque la tâche s'exécutait toute seule. Par conséquent les mesures présentées sur ce rapport correspondent seulement à la première architecture décrite.

1.2 Les différentes tâches (attaquantes, temps réel)

Pour représenter notre modèle automobile sur le processeur multi-c œur nous allons utiliser une tâche temps-réel et plusieurs taches attaquantes. Afin de gérer au mieux la concurrence d'accès pour ces deux types de tâches nous avons dû prendre en compte le "prefecthing" effectué par le processeur ainsi que l'optimisation faite par le compilateur, qui sont difficilement apréhandable.

Nous avons développé une tâche temps-réel (src/rt_task.c) paramétrable, non périodique dont le temps d'exécution est fini et qui accède à tous les caches ainsi qu'à la mémoire via le Bus. Notre première tâche temps-réel consistait à lire un élément au hasard dans un tableau de grande taille. Le choix aléatoire de l'élément lu évitait la lecture séquentille favorisée par le compilateur. Néanmoins cela ne permettait pas de saturer tous les caches, notamment le L3, nous n'évitions donc pas le prefetching du processeur. Nous avons donc rajouté un second tableau de même taille que le premier, ainsi nous choisissons au hasard le tableau dans lequel nous allons lire et quel élément sera lu. De cette façon les tableaux sont assez grands pour qu'ils soient mis sur la mémoire et pas seulement sur le cache L3 et le prefetching est évité. Ainsi nous maximisons les accès aux caches et à la mémoire donc au Bus également.

D'un autre côté la tâche attaquente (src/attack_task.c), qui représente le système multimédia, doit être la plus gourmande possible. Elle va donc faire un maximum d'accès mémoire et ainsi saturer le Bus mémoire et les caches partagés entre les différents coeurs du processeur. Pour cette tâche nous utilisons une liste doublement chaînée dont les éléments sont constitués d'une matrice à deux dimensions, remplie aléatoirement, et d'un index. En fonction du paramètre passé en argument on fera soit un parcours simple de la liste, soit un parcours de taille aléatoire (compteur et emplacement de départ tirés aléatoirement). Cependant quel que soit le parcours exécuté on aura un affichage de chaque matrice contenue dans tous les éléments parcourus de la liste doublement chaînée.

Le premier parcours utilisera fortement le prefetching du CPU et l'optimisation du compilateur à cause la localité spatiale. Dans le second parcours le prefecthing est ici évité ainsi que l'optimisation du compilateur de par la grande

taille des matrices contenues dans chaque élément et le nombre d'éléments contenus dans la liste doublement chaînée, empêchant la structure de tenir dans les caches partagés. La lecture séquentielle effectuée par le compilateur ne peut également pas être effectuée car on utilise des listes et non des tableaux. Pour l'affichage nous avons limité le nombre de printf car lors des mesures nous avons remarqué que cela les faussait au-delà d'un certain nombre.

1.3 Comment mesurer ?

Pour faire les mesures correspondantes à la tâche temps-réel on utilise la librairie PAPI (Performance Application Programming Interface) décrite ci-dessus. 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 d'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, ainsi qu'avoir le nombre de cache MISS du L3 en 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 soit lancée et ils sont arrêtés avant que le processus ne se termine.

Comme ces fonctionnalités vont être utilisées par le scheduler plus tard dans le projet, l'initialisation des éléments utilisés pour faire le benchmarking va être mise 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 ; pour à cela on utilise les fonctions du scheduler (sched_setaffinity, sched_setscheduler). On ne veut pas que le processus fasse une migration de CPU car les mesures ne correspondraient pas 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 pour mesurer les événements grâce à PAPI pendant une période de huit secondes. Ensuite il attend la fin de son fils et fait la destruction du set pour se terminer 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 en permanence. PAPI a besoin des droits de super-utilisateur pour réussir à lier l'event set utilisé à un seul CPU (PAPI_set_opt avec l'option PAPI_CPU_ATTACH).

Pour avoir une meilleure perception de la performance de la tâche temps-réel on va isoler les cœurs qui vont être utilisés par cette dernière, et pour les tâches attaquantes. 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 grace à la ligne de commande du noyau, on utilise donc l'option « isolcpus » passés au démarrage à travers GRUB 2.

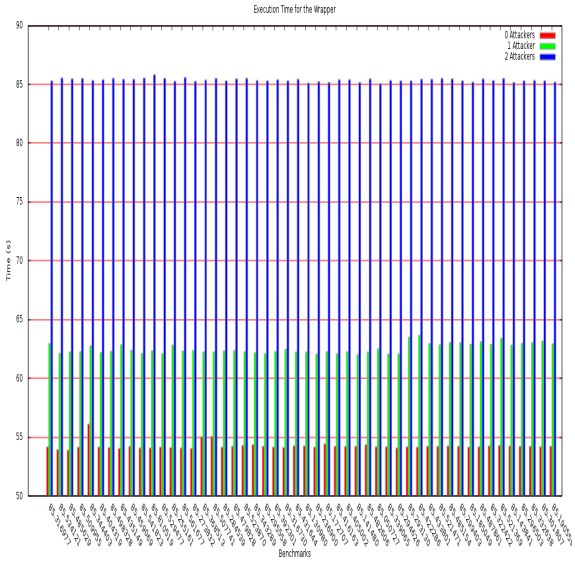
Lancer d'autres tâches attaquantes sur les CPUs déjà utilisés ne va pas influencer le temps d'exécution de la tâche temps-réel. On va avoir une stabilisation de la courbe quand il n'y a plus de CPUs disponibles.

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 .

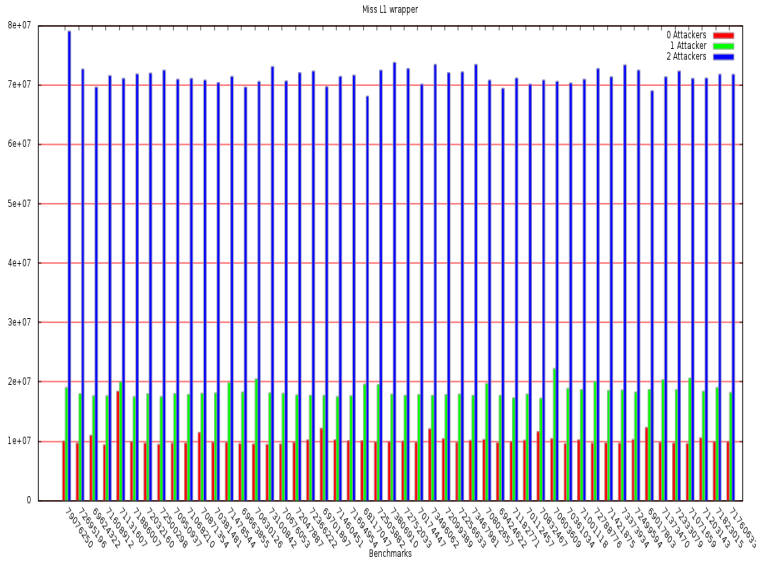
1.4 Résultats et analyse des courbes

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 et pour finir avec deux attaquants, on a 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 ajoute un attaquant et une augmentation de 35% avec deux attaquants. Pour la tâche temps-réel seule, le temps d'exécution est environ de 55 secondes, avec un attaquant on monte à 64 secondes et avec trois attaquants on arrive jusqu'à 89 secondes.

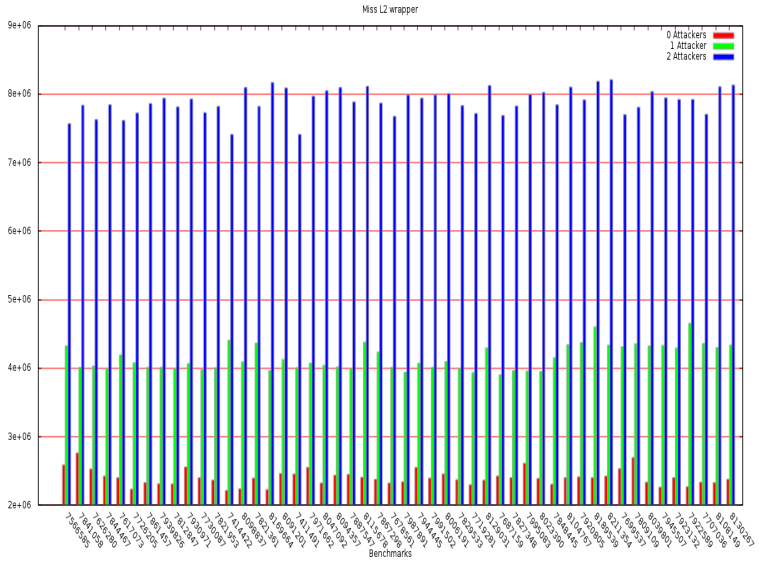


Le nombre d'accès aux caches de niveau inférieurs et à la mémoire vont augmenter aussi. Pour les miss de niveau L1 on a une augmentation de facteur 1.9 pour les cache MISS totaux (instructions et données) avec un attaquant. Puis une augmentation de facteur 3.5 avec deux attaquants par rapport à un seul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

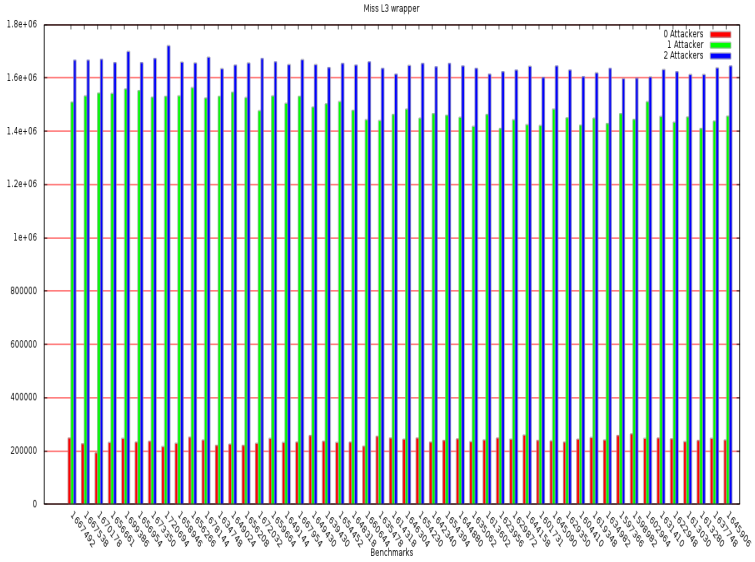


Le taux de MISS du L2 ne vas pas avoir de variantions aussi grandes puisque les tâches attaquantes ont leur propre cache L2, la taille du cache est aussi un

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 .

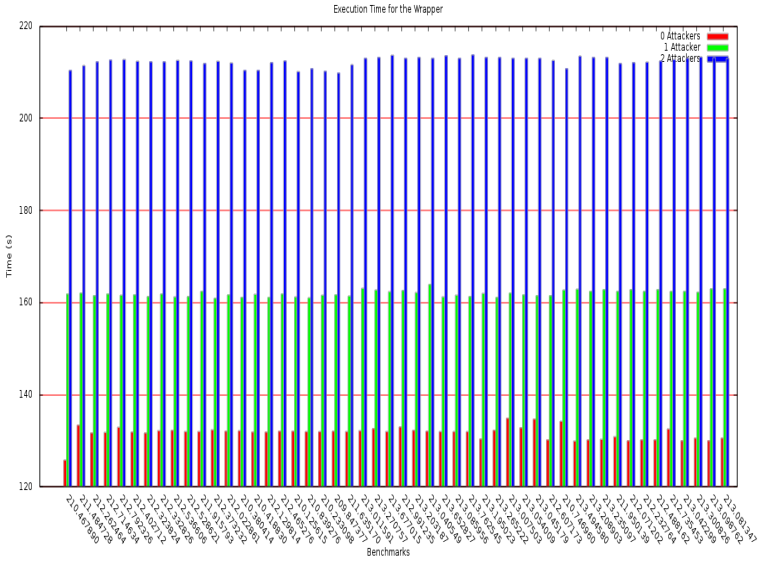


Enfin, le taux de MISS pour le niveau L3 va avoir des rapports plus élevés puisque qu'il est partagé avec les autres tâches attaquantes. On a un facteur de 6.23 pour un attaquant et un facteur de 1.13 pour deux attaquants, ceci est dû à la surcharge du cache L3. Avec un seul attaquant le cache est déjà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

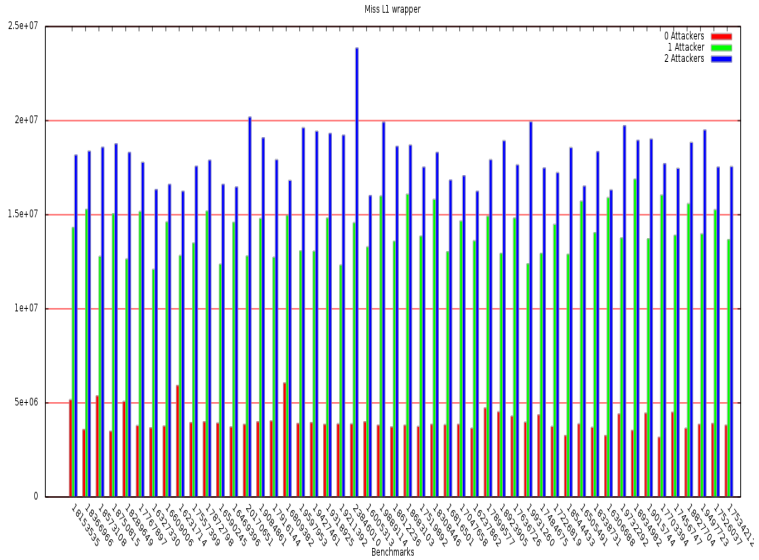


Le cache partagé (niveau L3) va être le plus pollué par les données et les instructions des tâches attaquantes. Ceci va avoir des influences sur les autres niveau de cache pour la tâche temps-réel : quand la donnée qui doit être chargée par le parcours aléatoire ne va pas se trouver sur le niveau du dessous il est fortement probable que la donnée se trouve dans la mémoire et pas dans le cache de niveau L3 car ce cache est pollué et saturé par les tâches attaquantes. De même les tableaux gérés par la tâche 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 assez souvent écrasées dans le cas du parcours aléatoire et un peu moins souvent quand les tâches utilisent un parcours linéaire.

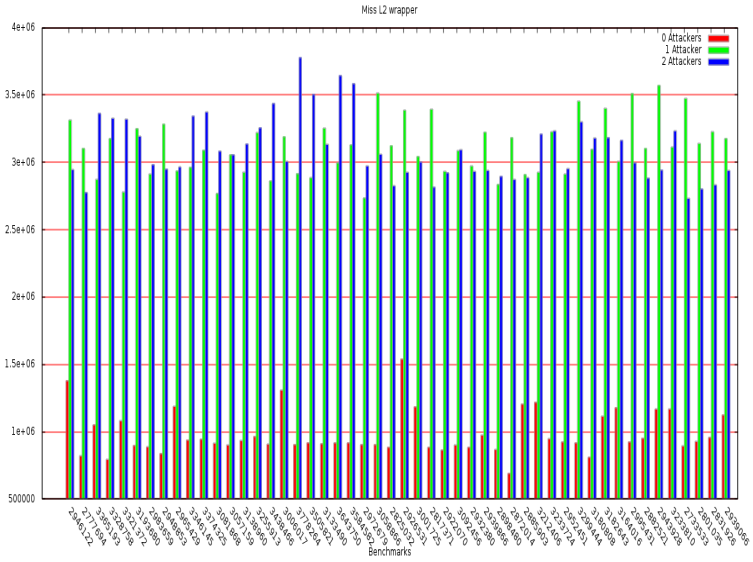
Dans un deuxième temps nous avons fait des mesures en utilisant le gouverneur « powersave ». Nous nous attendions à des augmentations dans le temps d'exécution pour la tâche temps-réel, ce qui est bien le cas. Nous avons plus ou moins doublé le temps d'exécution, c'est-à-dire aux alentours de 120 secondes pour la tâche temps-réel toute seule. Puis 162 secondes avec un attaquant et enfin 212 secondes pour deux attaquants. On a donc une augmentation de 35% pour un attaquant et 30% pour deux attaquants par rapport à un seul. Par rapport au gouverneur « performance » un attaquant entraîne une forte hausse sur le temps d'exécution.



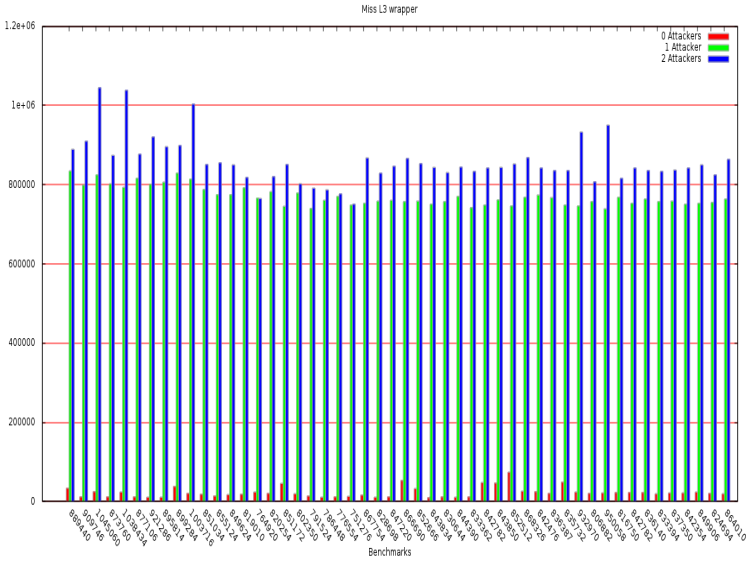
Concernant le nombre de MISS pour le niveau L1 on a un facteur d'augmentation de 3.64 quand on passe à un attaquant. Puis un facteur de 1.16 avec deux attaquants.



Le nombre de MISS pour le niveau L2 montre une augmentation de 1.52 puis de 1.24.



En comparaison le nombre d'accès mémoire augmente d'un facteur 3.16 avec un attaquant et 1.09 pour deux attaquants.



1.5 Conclusion

Les tests effectués sur l'architecture Intel, montrent à quel point la tâche temps-réel est influencée par des tâches attaquantes très gourmandes en mémoire. Indépendamment du gouverneur CPU qui est utilisé sur le système d'ex-

ploitation, l'influence reste la même. L'augmentation du temps d'exécution est similaire à celle du nombre d'accès mémoire sera un problème pour un processus de ce type car le pire cas d'exécution varie sensiblement avec ou sans attaquants. Ce problème est dû aux différentes ressources partagées, ajouté à nos mesure cela met en évidence le problème de contention mémoire.

2 Problème de la sous-réservation de BP mémoire :

2.1 Introduction

Grâce à notre premier modèle sans scheduler nous avons pu vérifier le fait que si on lance de nombreuses tâches attaquantes gourmandes en mémoire, deux dans notre cas, les contraintes de la tâche temps-réels peuvent ne pas être satisfaites, notamment au niveau du temps d'exécution ou la possibilité d'accès à la mémoire. Dans un système embarqué cela ne peut être possible.

Afin de gérer ces accès on va utiliser un mécanisme de sous-réservation de la bande passante mémoire. Pour cela nous allons mesurer la bande passante minimale que le contrôleur mémoire peut fournir en permanence, quelque soit l'état de la mémoire. Ensuite nous devons savoir quelle portion de cette bande pasante minimale doit être alloué à la tâche temps-réel afin que la contrainte temporelle soit respectée. La bande passante restante sera répartie équitablement entre les différentes tâches attaquantes. Ces allocations de bande passante se font pour une période P, à la fin d'une période qu'une tâche ait consommé toute sa bande passante ou non, sa consommation sera remise à 0. Chaque tâche peut ainsi utiliser sa bande passante à son rythme et quand une tâche aura atteint son quota elle sera bloquée jusqu'à ce qu'une nouvelle période ne commence.

Néanmoins ce mécanisme pose un problème : comment gérer les accès mémoires tout en gardant une certaine concurrence ? Une seconde solution est étudiée, on va utiliser un scheduler qui enverra des notifications aux différentes tâches lorsque toutes les tâches attaquantes auront consommées la bande passante qui leur est allouée et que la tâche temps-réel n'aura pas encore entièrement utilisé la sienne. C'est lui aussi qui préviendra de la remise à zéros des différents compteurs mesurant la bande passante utilisée sur chaque tâche.

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 également utiliser 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 ajouté au wrapper pour faire une sorte 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 afin de savoir combien d'accès mémoire ont été faits. Un event set spécifique (scheduler_eventset) pour les accès mémoire a été ajouté puisqu'il doit être réinitialisé après chaque fenêtre d'exécution.

Les fenêtres d'exécutions 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 correspond aux nombre d'accès faits par l'OS et la tâche attaquante qui tournent respectivement sur le premier et le deuxième cœur. Évidemment ce nombre peut varier à cause des interruptions ou à d'autres comportements du système d'exploitation. En fonction des architectures le nombre d'accès peut également varier.

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 arrêtées trop tard, ce qui provoquera une augmentation sur le temps d'exécution de la tâche temps-réel. Si cette période est trop longue il est possible 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 rapidement les tâches attaquantes ne vont pas pouvoir s'exécuter pendant que la tâche temps-réel s'exécute. Le problème se pose également dans l'autre sens ; si la fenêtre d'exécution est trop courte les tâches attaquantes ne vont pas réussir a être arrêtées et le temps d'exécution va être comparable à celui du wrapper.

Après avoir initialisé toutes les variables qui vont être utilisées pour les mesures, via 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 autant de fois que l'utilisateur l'a demandé lors du lancement des processus attaquants sur des émulateurs de terminaux. Les émulateurs commencent à s'exécuter sur les CPUs trois et quatre mais l'OS les fait migrer la plupart du temps vers le CPU 1, ceci doit être un effet de bord du à l'utilisation de serveur X sur un seul c œur. Cependant les tâches attaquantes sont bien clouées aux CPUs trois et quatre. 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 l'état 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 » initialisée 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, il faut alors 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 répété jusqu'à que la tâche temps-réel se termine.

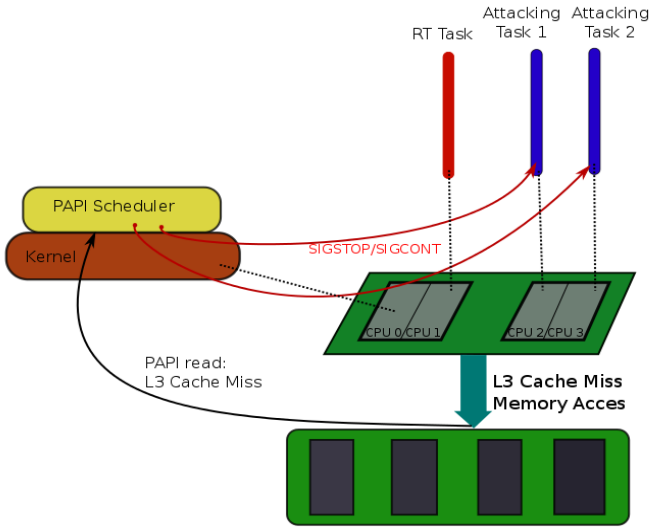
Algorithme Scheduler :

$quota\_memoire \leftarrow QUOTA\_DISPONIBLE$
$envoye \leftarrow faux$
$nouvelle\_fenetre \leftarrow NB\_APPELS\_TIMER$
function HANDLER_TIMER
  $consommation \leftarrow recuperer\_les\_valeurs\_ducompteur\_materiel\_avec\_PAPI$
  $quota\_memoire \leftarrow quota\_memoire - consommation$
  if $quota\_memoire \leq 0 ET envoye = faux$ **then**
    $envoyer\_SIGSTOP\_aux\_attaquants$
    $envoye \leftarrow vrai$
  end if
  $nouvelle\_fenetre \leftarrow nouvelle\_fenetre - 1$
  if $nouvelle\_fenetre = 0$ **then**
    $envoyer\_SIGCONT\_aux\_attaquants$
    $nouvelle\_fenetre \leftarrow NB\_APPELS\_TIMER$
    $quota\_memoire \leftarrow QUOTA\_DISPONIBLE$
    $envoye \leftarrow faux$
  end if
end function

On pourrait choisir d'utiliser les compteurs de MISS pour un niveau de cache L2 par exemple afin de savoir exactement quelle tâche est la plus gourmande et réussir à l'arrêter. Mais l'intérêt du projet était d'essayer de faire un ordonnancement en fonction des compteurs partagés entre les différents processus qui sont utilisés. Dans notre architecture c'est le cache de niveau L3 qui est partagé de fait nous utilisons le nombre de MISS de ce cache comme quota.

En plus de l'ordonnancement, le scheduler va faire les mesures de performance exactement de la même manière que le wrapper, mesurer le nombre de MISS sur les différents niveau pendant la période stationnaire de la tâche temps-réel. À la fin, le scheduler va écrire sur des fichier afin que gnuplot puisse les utiliser pour faire les graphes présentés sur ce rapport.

Ce schéma du scheduler donne une vision de l'exécution de ce programme :



2.3 Mesures, résultats et analyse

Concernant les tests du schedules les différentes tâches attaquantes et temps-réel vont avoir les même paramètres que celles utilisées pour le wrapper. Nous utilisons donc un parcours aléatoire pour les tâches attaquantes sur des listes de 900.000 éléments. Le processus temps-réel va faire 20'000.000 d'itérations et va être attaché au deuxième CPU logique. Les attaquants vont être attachés aux CPUs trois et quatre et l'OS va s'exécuter sur le premier.

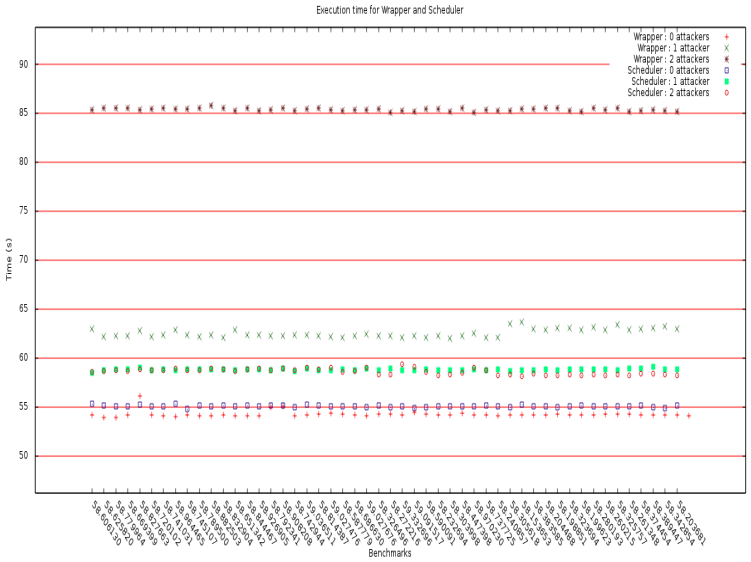
Le premier point à constater et également le plus intéressant est le temps d'exécution obtenu avec le mécanisme d'arrêt des processus attaquants. Comme pour le wrapper nous avons fait des tests en utilisant deux types de gouverneurs CPUs : « powersave » et « performance ». Les premiers résultats discutés correspondent au gouverneur « performance ».

On peut constater une légère augmentation du temps d'exécution quand on passe de zéro à un attaquant avec le scheduler. Avec un attaquant on a 6% d'augmentation, c'est-à-dire qu'on passe de 55 secondes à 58 environ. On a presque 0% d'augmentation quand on passe à deux attaquants.

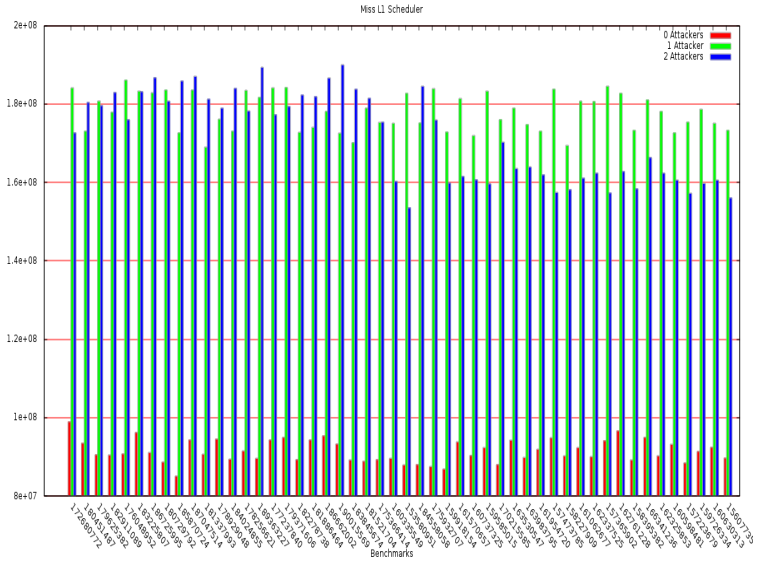
Comparé aux résultats obtenus avec l'attaquant qui n'est pas arrêté, on a un gain de 6 secondes en moyenne, mais avec deux attaquants le gain est de

31 secondes. Ce gain de 6 secondes peut être expliqué par la latence qui existe entre l'envoie des signaux à un processus et le traitement par ce programme. Effectivement le scheduler envoie le signal mais la tâche attaquante doit s'arrêter et ceci peut prendre un peu de temps. Une fois que la tâche attaquante doit à nouveau commencer son exécution elle va avoisiner le taux maximal d'usage de CPU. Tous ces aspects vont ralentir la tâche temps-réel et on n'aura pas exactement le même d'exécution que nous avions sans tâches concurentes.

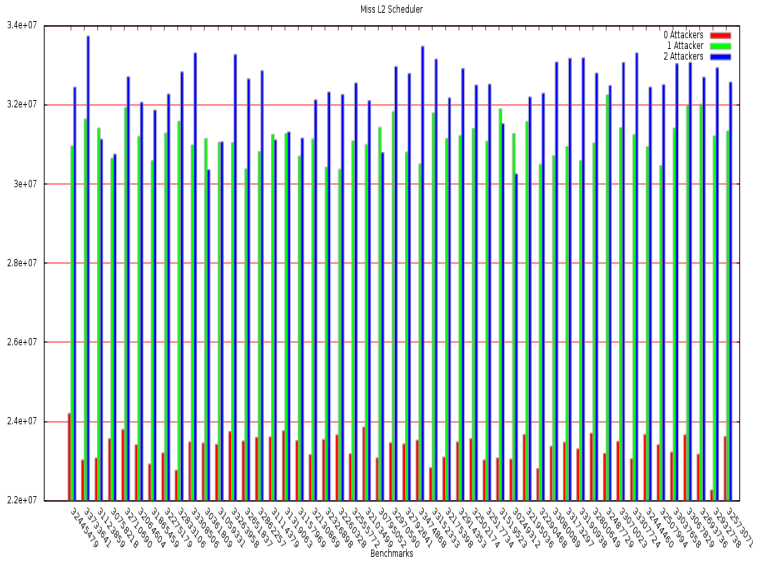
La courbe présentée ci dessous montre les différents temps d'exéction obtenus avec le wrapper et le scheduler :



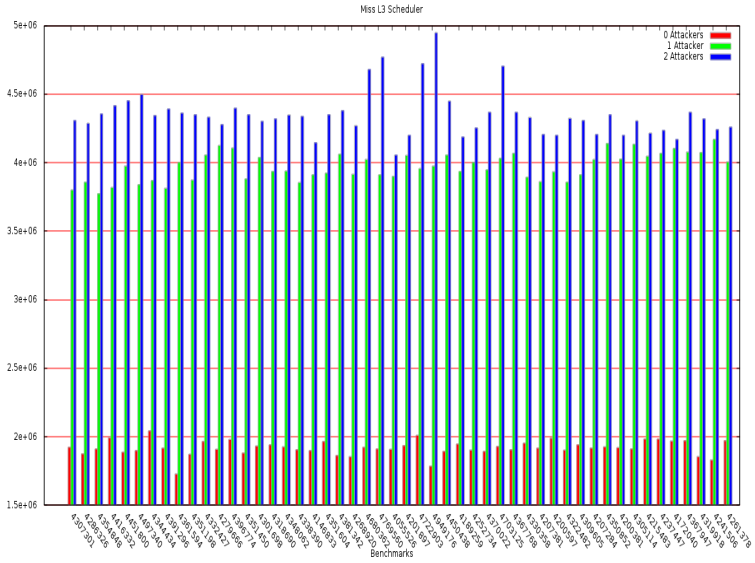
Le nombre de MISS au niveau L1 aura un comportement similaire au temps d'exécution, c'est-a-dire que nous avons une augmentation avec un attaquant, mais un deuxième attaquant n'influencer pas fortement le nombre de MISS. On peut remarquer que les valeurs sont plus variables et n'ont pas tendance à stagner sur une valeur bien précise. Une augmentation de rapport 1.94 sur le nombre de MISS pour un ou deux attaquants. Effectivement la différence entre un et deux attaquants est négligeable pour ces tests.



Pour le cache de niveau L2 on a une augmentation de 1.32 du nombre de MISS pour un attaquant et une augmentation de 1.05 pour deux attaquants par rapport à ce dernier.



Enfin, le cache L3 va avoir des résultats semblables aux autres caches avec une augmentation de 2.03 pour un attaquant et 1.10 pour deux attaquants. Contrairement au wrapper la petite variance entre deux et un seul attaquant sur le nombre d'accès mémoire n'est pas dû seulement à une saturation du cache L3 mais plutôt au fait que les tâches sont arrêtées.

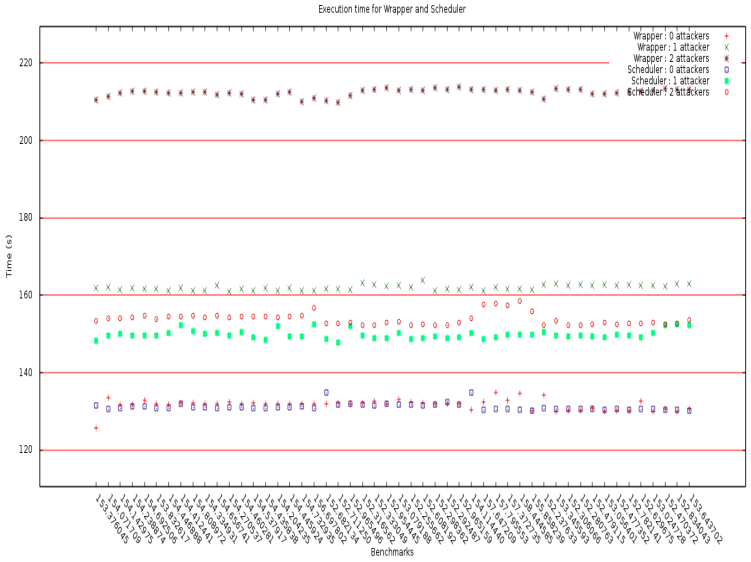


Les différents résultats montrent des nettes améliorations par rapport à l'exécution qui ne disposait pas du système d'arrêt pour les tâches attaquantes.

On peut voir qu'il y a une augmentation sur le nombre de MISS avec les attaquants, ce qui est normal puisque les attaquants arrivent à s'exécuter durant la fenêtre d'exécution du scheduler. Quand le quota d'accès mémoire est dépassé on arrête les tâches, les attaquants étant arrêtés séquentiellement cela peut expliquer la légère différence que l'on retrouve au niveau des temps d'exécution et sur le nombre de MISS quand on passe de un à deux attaquants.

En utilisant un gouverneur "powersave" l'écart entre un processus attaquant et deux attaquants est plus marqué, ceci est dû au temps d'exécution qui augmente. Les processus vont rester plus longtemps sur le CPU rendant l'écart plus visible.

Pour le temps d'exécution on a une augmentation de 12% avec un attaquant et 7% avec deux attaquants. On passe de 131 secondes sur la tâche temps-réel toute seule à 148 secondes avec un attaquant puis à 152 avec deux attaquants. Avec ce type de gouverneur l'arrêt des tâches attaquantes est plus lent. Il se fait aussi avec un effet d'escalier, on ne passe pas de 100% d'usage de CPU à 0% de manière linéaire, le gouverneur s'arrête vers la moitié pour enfin s'arrêter complètement.



2.4 Conclusion

Le mécanisme d'arrêt des tâches attaquantes, en utilisant PAPI comme plateforme de mesure, montre qu'on peut avoir des bons résultats avec des simples envois de signaux et en utilisant des mesures déclenchées avec un timer POSIX.

Sur les deux types de gouverneur CPU on a environ 10% d'augmentation sur le temps d'exécution et ce temps d'exécution reste plus ou moins stable après avoir lancé une seule tâche attaquante.

Le temps d'échéance de la tâche temps-réel et le pire schéma d'exécution vont varier mais d'une manière beaucoup moins importante que quand il n'y avait pas de mécanisme d'arrêt.

3 Conclusion face au sujet proposé :

Face aux deux stratégie étudiées, le problème de contention mémoire est bien mis en évidence. Un point négatif de la solution utilisant le scheduler serait la latence dans l'arrêt des différentes tâches quand les quotas sont atteints. Néanmoins cette dernière apporte une réelle évolution dans les mesures au niveau du temps d'exécution ou du nombre de cache MISS, indépendamment du nombre de tâches attaquant la tâche temps-réel. L'effet le plus marquant est le passage de 212 secondes sans scheduler à 152 secondes quand le mécanisme d'attente est mis en place pour deux attaquants, soit une amélioration de 32% grâce au scheduler.

Malheureusement nous n'avons pas pu avoir une architecture de test avec plus de cœurs. Cela nous aurait permis de voir si le temps d'exécution tend à se stabiliser en fonction du nombre de tâches attaquantes. Les résultats obtenus semblent montrer cette stabilisation, dans ce cas cette limite, qui est très liée au matériel, ne serait-elle pas le temps d'échéance de la tâche temps-réel ?