RenPar'18 / SympA'2008 / CFSE'6
Fribourg, 11 au 13 février 2008

Gestion de la cohérence des caches dans les architectures MPSoC utilisant des NoC complexes

Hajer CHTIOUI†, Rabie BEN ATITALLAH‡, Smail NIAR‡, Mohamed ABID†, Jean-Luc Dekeyser‡

†CES, Ecole Nationale des Ingénieurs de Sfax, Tunisie,
‡INRIA-FUTURS, Projet DART, Lille, France,
chtioui_hajer@yahoo.fr, {rabie.ben-atitallah, smail.niar, jean-luc.dekeyser}@lifl.fr,
mohamed.abid@enis.rnu.tn

Résumé
Les unités mémoires jouent un rôle de première importance dans les architectures embarquées multiprocesseurs (ou Multi-Processor System-on-Chip MPSoC). Dans cet article, nous nous intéressons en particulier aux MPSoC à mémoires partagées où la gestion de la cohérence de données représente un point crucial de l'architecture. Notre étude montre que la mise en cache des données partagées permet de réaliser des gains en performance et en consommation d'énergie électrique. De ce fait, les architectures multiprocesseurs embarquées n'intégrant pas la gestion de la cohérence des caches ne sont pas efficaces. Cette étude montre aussi qu'un choix judicieux de l'implémentation de cette gestion est nécessaire pour offrir aux applications un gain en performances et en consommation d'énergie. Deux méthodes sont habituellement proposées pour résoudre ce problème, la première applique l'invalidation des données et la deuxième consiste à les mettre à jour. Ces deux méthodes comportent plusieurs inconvénients. D'un coté, leur mise en uvre dans le cas des MPSoC utilisant des réseaux d'interconnexion complexes engendre un coût important en consommation d'énergie et de l'autre coté elles ne prennent pas en compte les motifs des accès mémoires réalisés par les applications. Notre objectif est de proposer des solutions pour remédier à ces faiblesses et pour tirer profit des deux approches précédentes. Les résultats préliminaires ont montré que des gains intéressants en performance et en consommation d'énergie pourraient être obtenus grâce à ces solutions .

Mots-clés : MPSoC, mémoire partagée, cohérence, crossbar, énergie

---

1. Introduction

Du fait de l'évolution des technologies d'intégration sur silicium d'une part et l'apparition de nouvelles applications gourmandes en puissance de calcul et en capacité mémoire, la tendance dans les systèmes embarqués hautes performances s'oriente vers l'utilisation des architectures embarquées multiprocesseurs (ou Multi-Processor System-on-Chip MPSoC). En effet, ces dernières requièrent un temps de conception plus court que les ASIC, du fait de leurs structures modulaires. Elles offrent aussi un rapport performance/consommation plus intéressant que les architectures monoprocesseurs utilisant des fréquences d'horloge élevées. Elles permettent enfin, une meilleure exploitation des ressources du silicium.
Notre travail se situe dans ce contexte et vise à améliorer les performances de ce type d'architecture en minimisant les latences et le coût en consommation d'énergie des accès mémoires par une meilleure exploitation des ressources mémoires. Nous nous intéressons aux architectures MPSoC à mémoire partagée centralisée utilisant des réseaux d'interconnexion complexes, comme le crossbar ou les réseaux multi-étages. Ces architectures sont très attractives du fait qu'elles facilitent à la fois le développement des applications parallèles, grâce à leur modèle de programmation et aussi l'intégration d'un nombre important de processeurs dans la plateforme. En effet, l'architecture à bus partagé, couramment utilisée ces dernières années pour interconnecter les processeurs, arrive rapidement à saturation dès que le nombre de processeurs dépasse la dizaine. L'augmentation de la fréquence d'horloge au niveau du bus

partagé pourrait être une solution, mais amènerait à une augmentation sensible de l'énergie consommée.
Malheureusement, les MPSoC à réseau d'interconnexion complexe posent le problème de la gestion des données partagées. La solution habituelle de l'espionnage du bus [2] n'est pas applicable à ce type de MPSoC. En effet, la diffusion des données n'est pas une solution efficace dans ce cas, en particulier lorsque le nombre de processeurs devient important et/ou lorsque la quantité de données partagées en écritures devient importante.
Une deuxième solution, appelée sans cohérence [1], consiste à ne pas mettre dans le cache les données partagées. Cette solution simplifie l'architecture, puisque celle-ci ne s'occupe plus de la gestion de la cohérence, mais charge l'utilisateur d'indiquer au système les données pouvant être mises en cache (ou *cachables*) et les donnes partagées ou *(non-cachables)*.
En plus de la surcharge qu'elle provoque sur le programmeur, nous montrons dans cet article que cette solution ne permet pas une utilisation efficace des mémoires caches. Par conséquent, nous obtenons des temps d'exécution et des consommations d'énergie importants pour les applications contenant une quantité importante de données partagées en écriture, comme c'est le cas des applications de traitement de signal intensif qui nous intéresse. Pour éviter ces deux inconvénients, la solution adoptée dans certaines architectures multiprocesseurs hautes performances consiste à intégrer dans l'architecture un répertoire centralisé [3]. C'est aussi cette technique qui est utilisée dans cet article mais nous l'adaptons aux cas des MPSoC.
Notre contribution comporte les trois points suivants. Nous proposons d'abord deux protocoles, par invalidation et par mise à jour des données, pour la gestion de la cohérence adaptée aux architectures MPSoC. Sur la base des résultats expérimentaux de ces deux protocoles, nous montrons dans un deuxième temps, la nécessité de disposer d'un nouveau protocole qui tire profit des avantages des deux premiers et qui s'adapte à la façon avec la quelle une donnée est utilisée. Enfin, le troisième point concerne la proposition d'un support matériel simple, mais qui faciliterait la mise en ouvre de ce dernier protocole.
Ce papier est constitué de quatre sections. Dans la première section, nous résumons les principaux travaux effectués sur la gestion centralisée de la cohérence de cache. Dans la deuxième section, nous présentons en premier lieu la plateforme MPSoC SoCLib que nous avons utilisée pour réaliser nos tests expérimentaux. Nous montrons aussi la façon avec laquelle la gestion de la cohérence est réalisée à travers les deux protocoles que nous avons implémentés et comparés. La troisième section présente les résultats expérimentaux obtenus pour ces deux protocoles. Enfin dans la section 4, nous présentons la structure générale d'un protocole hybride qui exploite les caractéristiques des applications et qui s'adapte automatiquement aux motifs d'accès des données. Enfin, nous donnons une conclusion et une liste des points qui restent à réaliser est dressée.

2. État de l'art : Gestion de la cohérence des données dans les MPSoC

La cohérence des caches dans les MPSoC, utilisant un autre moyen de communication que le bus partagé, reste encore un problème ouvert. À l'opposé, dans le domaine des architectures hautes performances, ce problème a donné lieu à un nombre important de travaux [12]. Dans ces travaux, le mécanisme du répertoire centralisé a été largement étudié pour assurer la cohérence de données dans les caches. Ce dernier se présente sous forme d'un tableau dont les colonnes correspondent aux mémoires caches des processeurs et les lignes aux blocs de données partagés entre les processeurs. Le répertoire permet de mémoriser l'état actuel des blocs de la mémoire (figure 1) [3].
Les différents mécanismes de cohérence de cache existants qui utilisent le répertoire centralisé diffèrent les uns des autres par les états du protocole utilisé. MSI [4] et le Standard du IEEE "Scalable Coherent Interface" (SCI) [5] sont parmi les plus anciens protocoles de cohérence. Ces protocoles ont donné lieu à un grand nombre d'extensions telles que MESI, MOSI, MOESI et le "GLOW coherence protocol" [6]. L'inconvénient majeur de ces protocoles, est que pour chaque accès au cache il est nécessaire de connaître à la fois l'état du bloc référencé et les processeurs qui le partagent avant chaque utilisation. Ce passage fréquent par le répertoire devient coûteux en termes de temps et de communication à travers le réseau surtout lorsque le nombre de processeurs est important.
Des études récentes [9] ont mis l'accent sur cet aspect. Les auteurs de l'article [9] ont mesuré les quantités de données transférées à travers le réseau pour réaliser les opérations de lecture et d'écriture dues à :

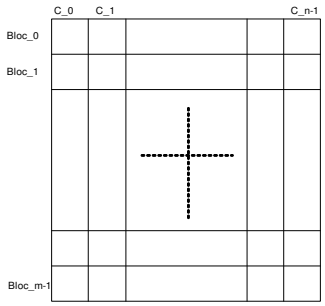

FIG. 1 – Structure du répertoire centralisé dans un MPSoC

1. L'exécution de l'application (les instructions chargement et stockage des données).
2. La gestion de cohérence centralisée.

Il a été constaté que pour des caches de petite taille, le nombre de paquets transférés pour le maintien de la cohérence des données reste faible par rapport au nombre de paquets utilisés pour lire ou écrire les données. Néanmoins, la situation devient plus complexe lorsque la taille du cache augmente. En effet, pour des grandes tailles de cache les opérations de lecture et d'écriture diminuent du fait que le nombre d'échecs en cache diminue, alors que le nombre de paquets dus aux opérations de cohérence reste presque constant. Par conséquent, le coût de la gestion du répertoire, en terme de surcharge d'opérations de communication, dépend de la taille du cache.
Afin d'améliorer les performances de la cohérence dans les architectures multiprocesseurs à répertoire centralisé Eilsley et al. [7] ont proposé l'intégration du protocole de cohérence dans le réseau d'interconnexion. Ainsi, le répertoire n'est plus centralisé mais distribué dans les routeurs du NoC. Bien que les résultats obtenus montrent une réduction du temps d'accès, cette solution se base sur une conception matérielle coûteuse des routeurs. Dans ce même cadre, Bolotin et al.[8] ont proposé de réduire l'effet des paquets de maintient de cohérence sur les performances. Ils proposent pour cela de classer les messages qui transitent à travers le NoC en deux types : les messages courts, ceux utilisés pour envoyer une demande de lecture, d'écriture ou d'invalidation d'un bloc, et les messages longs qui correspondent aux messages nécessaires pour réaliser un transfert de données et qui nécessitent plus de temps de traitement lors de la réception du paquet par le processeur ou la mémoire. L'idée consiste à donner la priorité aux messages courts par rapport aux messages longs au niveau du NoC en cas de conflit. Cette technique permet d'une part de simplifier le trafic de données dans le réseau et de réduire les cycles d'attente des processeurs d'autre part. Là aussi, il est nécessaire d'ajouter des mécanismes matériels pour l'implémentation de la priorité.
À la lumière de cette étude, nous pouvons conclure que, malgré sa large utilisation dans les architectures hautes performances, le répertoire centralisé pose des problèmes de coûts d'implémentation dans les architectures embarquées. Dans cet article, nous présentons une solution pour permettre l'utilisation d'une telle gestion centralisée de la cohérence de données, mais qui génère une surcharge réduite en termes de temps d'accès, de consommation d'énergie et de nombre d'interconnexions supplémentaires.

3. Une gestion de la cohérence des données centralisée pour l'architecture SoCLib

3.1. Présentation de la plateforme SoCLib

La plateforme SoCLib [10] est une librairie des composants réutilisables qui permet de modéliser et de simuler des architectures MPSoC. Dans la version du simulateur SoCLib à laquelle nous avons eu accès, un crossbar est utilisé pour connecter les processeurs aux modules mémoires partagées (figure 2). Pour le partage des données, la solution implémentée ne permet pas de charger les données partagées dans les caches. Dans la version de SocLib que nous utilisée, il n'y a pas de gestion de la cohérence. Dans la suite de cet article, cette solution est appelée "sans cohérence". Plus exactement, les programmes exécutés par le simulateur comportent trois types de données :

1. Celles qui sont partagées en lecture et en écriture et qui ne sont pas mises en cache (*données non cachables*).
2. Celles qui sont partagées en lectures seulement et qui peuvent être stockées en cache.
3. Celles qui ne sont pas partagées (données privées) et qui peuvent aussi être stockées en cache.

Nous montrons dans ce travail que cette solution ne permet pas une utilisation efficace de la mémoire cache. L'objectif de ce travail consiste donc à permettre aux caches de stocker les données partagées de la catégorie 1.

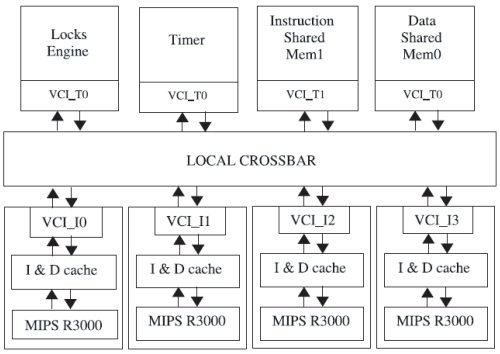

FIG. 2 – Architecture de la plateforme SoCLib

3.2. Présentation de deux protocoles simples basés sur le répertoire centralisé

Notons, avant de donner les détails des deux protocoles évalués, que la technique d'écriture dans le cache que nous avons employée est l'écriture simultanée (*write-through*). Dans ce mécanisme, tous les accès avec succès en écriture se font simultanément dans le cache et dans la mémoire partagée. Par conséquent, cette technique maintient une cohérence entre les données de la mémoire partagée et celles du cache. Contrairement à la technique d'écriture différée (ou *write-back*) qui n'écrit la donnée en mémoire partagée que lorsque le bloc modifié est éjecté du cache, l'écriture simultanée exige des mécanismes de cohérence moins complexes mais peut amener à une augmentation du nombre de paquets d'écriture qui transitent par le NoC.

En adoptant l'écriture simultanée, nous avons défini un protocole de gestion de la cohérence appelé ESI. Dans la littérature, il existe des protocoles similaires au protocole ESI tel que le protocole MSI [4].

Dans ce protocole, un bloc de données peut avoir trois états possibles dans le répertoire centralisé :

– E(Exclusif) : La valeur actualisée du bloc existe seulement dans le cache correspondant et dans la mémoire partagée.
– S(Shared) : La valeur actualisée du bloc existe dans le cache correspondant, dans d'autres caches et dans la mémoire partagée.
– I(Invalid) : Le cache ne contient pas la valeur actualisée du bloc.

Si un processeur modifie un bloc partagé à l'état "E" ou "S", il doit les informer selon deux possibilités :

1. Envoyer la nouvelle valeur du bloc à partir de la mémoire partagée aux caches contenant une copie de ce bloc.

   Ce protocole est appelé par conséquent "protocole avec mise à jour". Les figures 3 et 4 représentent le fonctionnement de l'automate pour maintenir la cohérence respectivement dans le répertoire et dans le cache du protocole avec la mise à jour. Les symboles utilisés correspondent à :
   – Ro (Read other) : Lecture de la donnée par un autre processeur.
   – RL (Read Local) : Lecture de la donnée par le processeur.
   – Wo (Write other) : Écriture de la donnée par un autre processeur.

– WL (Write Local) : Écriture de la donnée par le processeur.
– V et I représentent respectivement les états valide et invalide que peut avoir un bloc du cache.

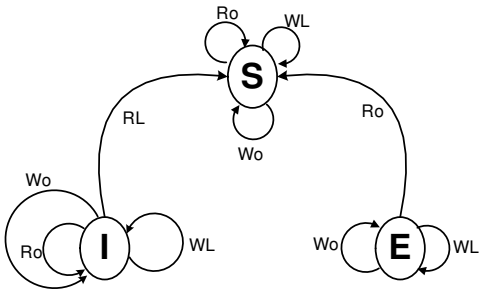

FIG. 3 – Automate du répertoire avec mise à jour

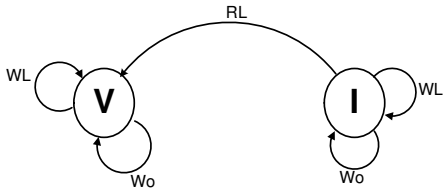

FIG. 4 – Automate du cache avec mise à jour

Comme nous pouvons le voir, suite à une opération RL ou Ro, le bloc modifié maintient son état valide. Le bloc peut donc être réutilisé lors d'un futur accès. Ce protocole permet généralement une diminution du nombre d'échecs du cache et par la suite une réduction du temps d'exécution.

Néanmoins, lorsque les données sont faiblement utilisées par les autres processeurs entre deux opérations d'écriture, les mises à jour deviennent inutiles. Ce qui entraîne l'augmentation du temps d'exécution et la consommation d'énergie.

2. Invalider les copies des blocs dans les autres caches.

   Avec ce deuxième protocole, appelé "par invalidation", dès qu'une opération d'écriture est réalisée sur un bloc, le bloc devient invalide dans les autres caches. Par la suite, si de nouveau ce bloc est demandé par un processeur, il sera chargé depuis la mémoire partagée. Par conséquent, ce protocol peut donner lieu à un nombre d'échecs en cache relativement plus grand, en particulier dans le cas où un nombre important de processeurs se partagent une grande quantité de données en lectures/écritures.

   Contrairement au protocole avec mise à jour, le protocole avec invalidation est relativement performant lorsque les données écrites sont faiblement utilisées par les processeurs. Ce deuxième protocole peut réduire la consommation d'énergie, puisqu'il n'y a pas de transfert de données pour réaliser les mises à jour. Les figures 5 et 6 représentent respectivement l'automate dans le répertoire centralisé et dans le cache pour le protocole par invalidation.

D'après la discussion précédente, nous pouvons conclure que les deux protocoles se complètent puisque les points forts de l'un correspondent aux points faibles de l'autre. Ceci est d'ailleurs confirmé par les résultats expérimentaux présentés dans la section suivante.

Afin d'exploiter le fait que la mémoire partagée est actualisée après chaque opération d'écriture dans le cache par l'un des processeurs (écriture simultanée), nous avons intégré le répertoire centralisé dans le module mémoire partagée. De cette façon, dès qu'un paquet écriture arrive à ce module, l'opération de mise à jour du répertoire est déclenchée. Ce mécanisme simplifie le maintient de la cohérence puisqu'il n'exige pas des paquets dédiés pour la mise à jour du répertoire. En terme de surface, pour les tailles de caches que nous avons expérimenté (entre 1 et 32KO), le répertoire ne représente que 3% de la surface de la mémoire partagée au maximum. Pour réaliser les modifications de l'état d'un bloc du cache (mise à jour ou invalidation), nous proposons l'utilisation d'un bus unidirectionnel, de la mémoire partagée vers les processeurs. Bien que son rôle consiste simplement à actualiser les caches selon le protocole (mise à jour ou invalidation) utilisé, ce bus permet de soulager le réseau d'interconnexion et réalise les opérations de maintien de la cohérence de cache. Cette architecture malgré sa simplicité, exige un nombre d'interconnexions et une complexité réduite. La figure 7 représente l'architecture du système globale. Dans cette figure, le signal CMD sert à informer les caches des modifications qui se passent sur

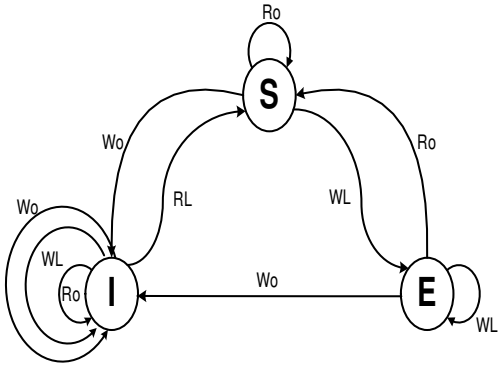

FIG. 5 – Automate du répertoire avec invalidation

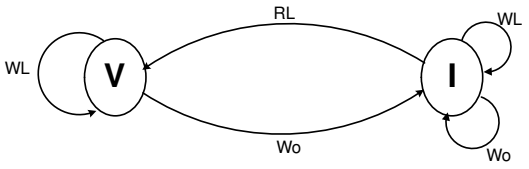

FIG. 6 – Automate du cache avec invalidation

les blocs. ADR, respectivement DATA, correspond aux bits d'adresse du bloc à modifier, respectivement à la nouvelle valeur du bloc dans le cas du protocole avec mise à jour. Finalement, NUM sert à identifier le numéro du processeur qui a causé la modification. Cette dernière information, est nécessaire pour invalider ou mettre à jour toutes les copies du bloc sauf la copie dans le cache du processeur qui a causé la modification.

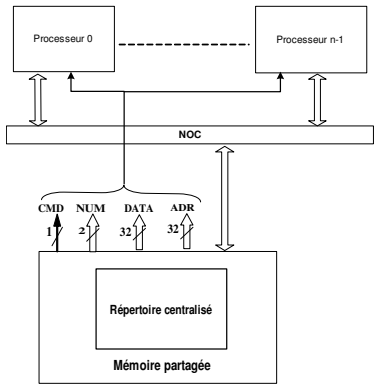

FIG. 7 – Architecture générale du système avec le bus de cohérence

4. Résultats expérimentaux

Plusieurs applications ont été parallélisées et portées sur la plateforme SocLib. Dans le programme de multiplication de matrices 100x100, nous avons utilisé quatre processeurs. Chaque processeur calcule les éléments des quatre sous-matrices carrées de la matrice résultat. Les résultats obtenus sont illustrés par la figure 8. Cette figure donne le temps d'exécution mesuré en millions de cycles en fonction de la taille du cache d'instructions et de données.

Les résultats illustrés par la figure 8 montrent une amélioration des performances du système en temps d'exécution par l'utilisation du répertoire centralisé avec le protocole de mise à jour. Cette amélioration augmente avec la taille du cache et atteint 38% avec un cache de 32Ko. En effet, lorsque la taille du cache augmente, la quantité de données partagées cachées augmente, ce qui amène à une réduction du trafic

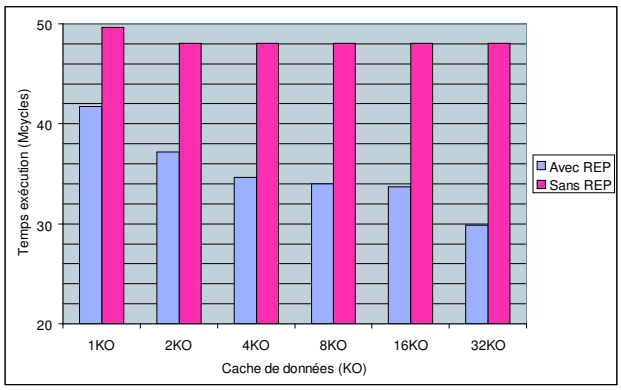

FIG. 8 – Temps d'exécution pour la multiplication de matrices sur 4 processeurs avec mise à jour pour des caches de 1Ko à 32Ko

sur le NoC.
Nous remarquons que la courbe sans cohérence (sans REP) est presque constante. En effet, après une légère diminution due à une réduction du nombre de défauts en cache d'instructions, le temps d'exécution au delà de 4KO reste constant. Ceci est dû au fait que la majorité des données traitées sont partagées. Il y a une légère diminution au niveau de cette courbe lorsque la taille du cache augmente du fait de la diminution du nombre d'échecs pour les données privées.

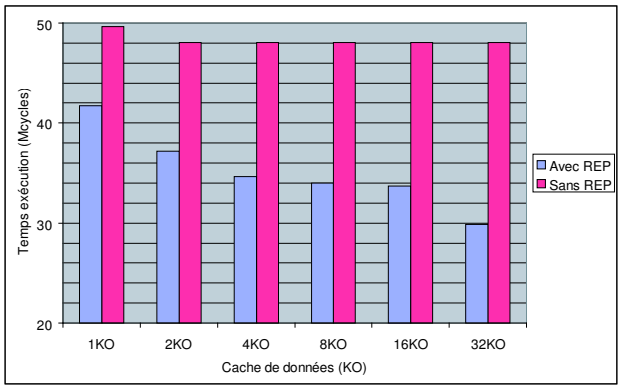

FIG. 9 – Temps d'exécution pour la multiplication de matrices sur 4 processeurs avec invalidation pour des caches de 1Ko à 32Ko

Les résultats obtenus avec le protocole d'invalidation prouvent que l'intégration d'un protocole de cohérence (avec REP) dans notre plateforme réduit le temps d'exécution. Là aussi, l'amélioration arrive jusqu'à 38% pour un cache de 32Ko (figure 9).
À partir de ces résultats, nous constatons que bien qu'il y a une réduction du temps d'exécution, les gains des deux méthodes sont presque similaires. Ceci s'explique du fait que dans cette application le partage de données est en lecture plus qu'en écriture. Ainsi, nous pouvons dire que cette parallélisation à favoriser le protocole avec invalidation. Mais ceci ne se voit pas en terme de temps d'exécution, car les mises à jour inutiles se font par le bus et pas par le crossbar. La deuxième application, que nous avons testé est le module d'initialisation des matrices dans l'application transformée de Fourier rapide (FFT). La parallélisation de toute l'application FFT est en cours de réalisation.

Le noyau de cette partie d'initialisation de la FFT est présenté ci-après :

```
for(i=0;i<MAXSIZE;i++) /*loop 2*/
{
    RealIn[i]=0;
    for(j=0;j<MAXWAVES;j++) /*loop 1*/
        RealIn[i]+=coeff[j]*cos(amp[j]*i);
}
```

La constante MAXSIZE vaut 32768 alors que la constante MAXWAVES vaut 8.
Cette application a été parallélisée de deux façons, suivant que c'est la boucle 1 ou la boucle 2 qui est distribuée sur les 4 processeurs. Dans la première version (loop 1), les éléments du vecteur RealIn sont lus et écrits par tous les processeurs. En effet, chaque processeur calcule de la valeur finale de RealIn[i] qu'il va additionner à la valeur actuelle. Dans version 2 (loop 2), à l'opposé, les données en écritures ne sont pas partagées, du fait chaque processeur se charge de calculer une partie des MAXSIZE éléments du vecteur RealIn. Nous pouvons donc imaginer que la version 1 favorise la mise à jour alors que la version 2 favorise l'invalidation.
Les résultats expérimentaux sont donnés dans la figure 10 qui représente le temps d'exécution en millions de cycles de la version 1 pour les solutions : sans cohérence, avec cohérence et mise à jour et enfin avec cohérence et invalidation. Cette figure donne aussi le gain en termes de réduction du temps d'exécution avec les deux protocoles. Des tailles de caches de données et d'instructions de 1 Ko jusqu'à 32 Ko sont considérées.

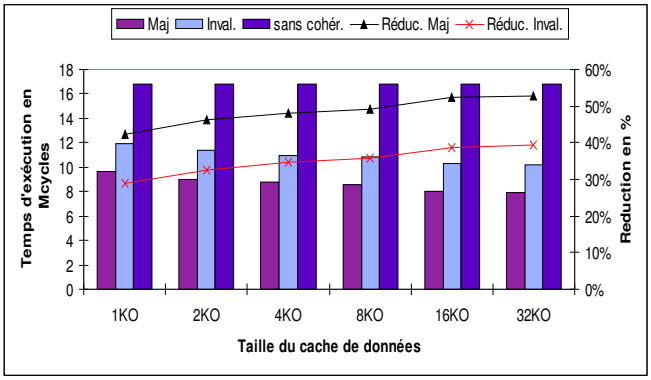

FIG. 10 – Temps d'exécution et rapport de réduction en fonction de la taille du cache pour la la FFT sur 4 processeurs et une parallélisation sur loop 1

Les résultats illustrés par cette figure montrent que le temps d'exécution diminue avec l'utilisation des deux protocoles. Cette réduction est plus importante avec le protocole de mise à jour qu'avec celui d'invalidation. En effet, la réduction du temps d'exécution dépasse les 55% pour des grandes tailles de cache. Cette variation, selon la taille du cache, s'explique par le fait qu'en augmentant la taille du cache le nombre d'échecs diminue, ce qui réduit le temps d'exécution. Avec le protocole de mise à jour le système est deux fois plus rapide. Le protocole avec invalidation donne des résultats moins intéressants que ceux obtenus avec la mise à jour.
Nous avons réalisé une estimation de la consommation d'énergie pour cette version de la FFT. En effet, notre plateforme MPSoC intègre des modèles de consommation pour chaque type de composants (processeur, mémoire, réseau d'interconnexion, cache, etc.). Ces modèles sont détaillés dans [11]. Ils permettent ainsi d'estimer l'énergie totale consommée par le système pour exécuter l'application. La figure 11 présente la consommation d'énergie en micro-joules en fonction de la taille des caches.

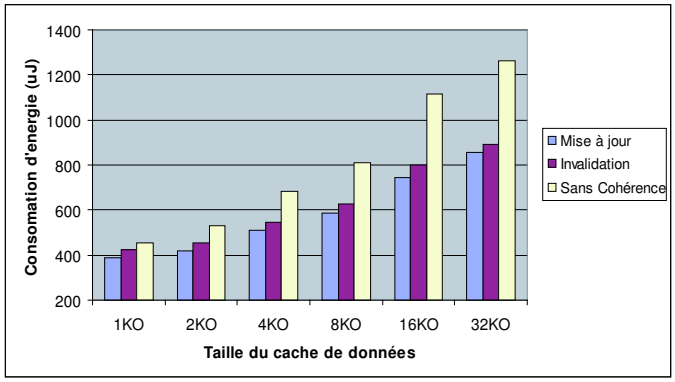

FIG. 11 – Consommation de l'énergie en fonction de la taille de cache de l'application FFT (4 processeurs) et une parallélisation sur loop 1

À l'aide des résultats présentés par la figure 11, nous pouvons déduire que les deux protocoles que nous avons implémentés, en plus de leurs performances en termes de temps d'exécution, permettent de réduire la consommation d'énergie. Là aussi, la réduction est plus significative pour le protocole avec mise à jour. Puisque le nombre d'échecs de cache diminue avec ce protocole, le nombre d'accès à la mémoire partagée à travers le réseau se réduit aussi. Ainsi, la consommation d'énergie diminue. Nous pouvons constater aussi que le bus de cohérence que nous avons ajouté ne dégrade pas la performance du système en termes de consommation d'énergie. Enfin avec cette version de la FFT nous avons pu montrer que pour un type de traitement de données, le protocole avec mise à jour est assez performant en termes de temps d'exécution et de consommation d'énergie.
La figure 12 représente les résultats de la deuxième version parallèle de la FFT (parallélisation sur la boucle 2). Nous donnons dans cette figure le temps d'exécution en fonction de la taille du cache, exprimé en millions de cycles pour les solutions : sans cohérence, avec cohérence et mise à jour et avec cohérence et invalidation. Les réductions du temps d'exécution sont aussi indiquées dans cette figure pour les deux protocoles.

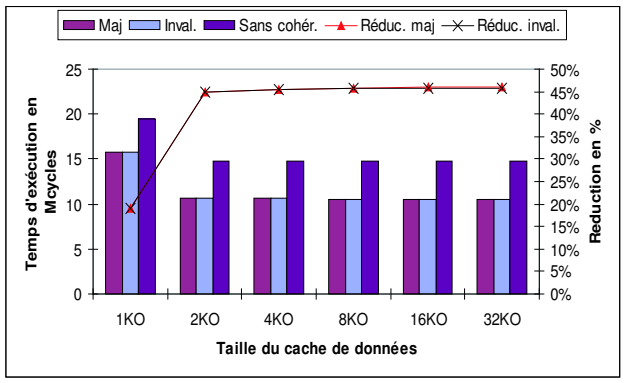

FIG. 12 – Temps d'exécution et réduction du temps d'exécution en fonction de la taille du cache de l'application FFT (4 processeurs) et une parallélisation sur loop 2

Nous remarquons qu'avec cette version de la FFT, la réduction du temps d'exécution pour les deux protocoles est quasi identique et atteint 45% (à partir de 2Ko). En effet, comme dans les deux protocoles le

crossbar est utilisé pour réaliser les écritures, les temps d'exécution sont identiques. La figure 13 indique que la réduction de la consommation d'énergie est meilleure pour le protocole avec invalidation. Ceci est dû au fait que le coût des mises à jour inutiles sur le bus fait augmenter sensiblement la consommation d'énergie totale.

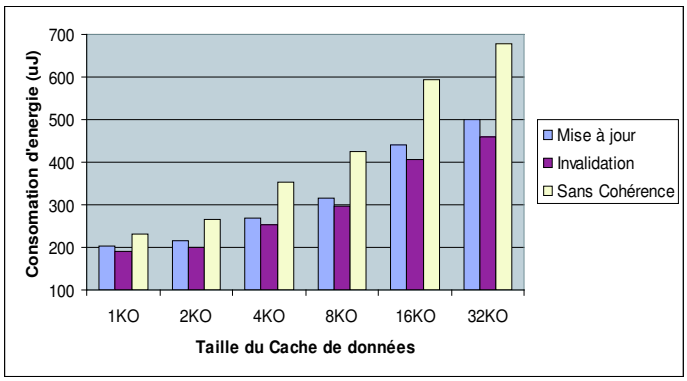

FIG. 13 – Consommation de l'énergie en fonction de la taille du cache de l'application FFT (4 processeurs) et une parallélisation sur loop 2

En conclusion, nous pouvons dire que selon la méthode suivie pour paralléliser l'application, tel ou tel protocole donnera de meilleures performances en termes de temps d'exécution et de consommation d'énergie.
À travers ces constatations, nous proposons un nouveau protocole qui tire profit des avantages de deux protocoles précédents.

5. Un protocole hybride et dynamique pour la gestion de la cohérence

En se basant sur l'étude que nous venons de réaliser sur les deux protocoles avec mise à jour et avec invalidation, nous avons entamé la conception d'un nouveau protocole qui exploite les avantages des deux protocoles précédents. L'idée que nous tentons d'exploiter consiste à déterminer dynamiquement le protocole de gestion de cohérence le plus adéquat pour un bloc donné. Cette détermination est réalisée en utilisant un tableau d'historique des opérations réalisées sur le bloc. Ainsi dés qu'un nombre seuil de mise à jour à été réalisé sans qu'aucun processeur n'a réalisé de lecture de la donnée mise à jour, il y un changement dans le protocole et passage vers une cohérence par invalidation. A l'opposé, lorsque plusieurs défauts sont constatés sur le même bloc dans un laps de temps, c'est le protocole par mise à jour qui est choisi.

6. Conclusion

Dans ce papier, nous avons implémenté une gestion centralisée de la cohérence de cache pour une plateforme MPSoC. Dans la littérature très peu de travaux ont été dédiés à ce type d'architectures. Pour cela nous avons implémenté deux protocoles simples en utilisant le répertoire centralisé. Le premier protocole utilise la mise à jour des données alors que le deuxième utilise l'invalidation des blocs modifiés. Les simulations réalisées avec la plateforme SoCLib ont montré que la solution du répertoire centralisé est assez performante et donne des réductions intéressantes du temps d'exécution et de la consommation d'énergie. Sur la base de ces résultats expérimentaux, nous avons proposé un nouveau protocole qui tire profit des avantages de ces deux protocoles. Nous envisageons la réalisation de ce nouveau protocole dans la prochaine étape du projet. Ce protocole permettrait d'obtenir des gains en temps d'exécution et en consommation d'énergie.

Bibliographie

1. Susan Owicki and Anant Agarwal, "Evaluating the Performance of Software Cache Coherence", 3rd international Conference on architectural Support for programming languages and operating systems, 1989.
2. Loghi, M., Poncino, M., "Exploring energy/performance tradeoffs in shared memory MPSoCs : snoop-based cache coherence vs. software solutions", Design, Automation and Test in Europe, 2005.
3. Lucien M. Censier and Paul Feautrier. "A New Solution to Coherence Problems in Multicache Systems", IEEE Transactions on Computers, c-27(12) :1112-1118, December 1978.
4. Resit Sendag, Ayse Yilmazer, Joshua J. Yi, and Augustus K. Uht, " Quantifying and Reducing the Effects of Wrong-Path Memory References in Cache-Coherent Multiprocessor Systems ", Parallel and Distributed Processing Symposium, April 2006.
5. IEEE. "IEEE Standard for Scalable Coherent Interface", IEEE, 1993.
6. Stefanos Kaxiras and James R. Goodma, " The GLOW Cache Coherence Protocol Extensions for Widely Shared Data ", Proceedings International Conference on Supercomputing ; May 1996.
7. Noel Eisley, Li-Shiuan Peh, Li Shang, " In-Network Cache Coherence ", International Symposium on Microarchitecture (MICRO'06), 2006.
8. Evgeny Bolotin, Zvika Guz, Israel Cidon, Ran Ginosar and Avinoam Kolodny, "The Power of Priority : NoC based Distributed Cache Coherency", Networks-on-Chip (NOCS), May 2007.
9. Gustavo Girao, Bruno Cruz de Oliveira, Rodrigo Soares, Ivan Saraiva Silva ; "Cache Coherency Communication Cost in a NoC-based MPSoC Platform", SBCCI'07, Rio de Janeiro, Brazil, 2007.
10. http ://soclib.lip6.fr.
11. Rabie Ben Atitallah, Smail Niar, Alain Greiner, Samy Meftali, and Jean Luc Dekeyser. Estimating energy consumption for an MPSoC architectural exploration. In Architecture of Computing Systems (ARCS'06), Frankfurt, Germany, March 2006.
12. David E. Culler, Jaswinder Pal Singh and Anoop Gupta, Parallel Computer Architecture : A HARDWARE/ SOFTWARE APPROACH, Morgan Kaufmann Publishers, August 1998.