UNIVERSITE MONTPELLIER II

SCIENCES ET TECHNIQUES DU LANGUEDOC

T H E S E

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITE MONTPELLIER II

Discipline : Microélectronique
Formation Doctorale : Systèmes Automatiques et Micro-électronique (SYAM)
Ecole Doctorale : Information, Structures, Systèmes (I2S)

Soutenue le :

04 décembre 2012

par

Rémi Busseuil

Exploration d'architecture d'accélérateurs à mémoire distribuée

Jury :

François PECHEUX	*Maitre de Conférences, UPMC – LIP6*	*Rapporteur*
Jean-Philippe DIGUET	*Directeur de Recherche CNRS, LAB-STICC*	*Rapporteur*
Yann THOMA	*Professeur HES, HEIG-VD*	*Examinateur*
Christophe JEGO	*Professeur des Universités, ENSEIRB-MATMECA*	*Examinateur*
Gilles SASSATELLI	*Directeur de Recherche CNRS, Université Montpellier 2*	*Co-directeur*
Michel ROBERT	*Président d'Université, Université Montpellier 2*	*Directeur de thèse*

Remerciements

La réalisation de cette thèse n'aurait pu avoir lieu sans le support physique, financier ou encore spirituel de nombreuses personnes.

Mon premier remerciement va à celle qui a été mon soutient le plus cher et qui m'a accompagné durant ces années de travail : mon amoureuse Jennifer. Compréhensive lors des nuits de labeur passées au laboratoire, réconfortante durant les moments de doute, sa présence a été primordiale dans l'accomplissement de ces travaux.

Je remercie également l'ensemble de ma famille, en commençant par mes parents Catherine et Jacques, responsables de ce que je suis devenu aujourd'hui : ils sont et resteront des modèles pour moi, une référence de la personne que je souhaite devenir. La gratitude envers l'aide qu'ils m'ont apportée, tout comme celui de mon frère Lucas et mes deux sœurs Coralie et Magali ne peut être résumée par les quelques lignes ici présentes, je ne peux leur rendre honneur qu'en disant qu'ils ont su être la famille idéale durant toutes ces années.

Merci également à l'ensemble de mes amis et camarades de thèse qui m'ont permis de passer d'inoubliables moments de joie et de rigolade. Grâce à eux les moments de stress et de tension ont été vite oubliés, et je garde aujourd'hui une majorité de bons souvenirs de ces années grâce à eux.

Mon dernier remerciement va à l'ensemble des personnes qui ont contribué professionnellement à la réalisation de cette thèse, en commençant par mes encadrants Gilles Sassatelli et Michel Robert. Par leur intermédiaire, j'ai pu découvrir le monde de la recherche, et agrandir mes connaissances et mes compétences. Merci également à l'ensemble des personnes du LIRMM, enseignants, chercheurs, administratifs, qui m'ont aidé et supporté durant toutes ces années. Merci enfin aux différents membres du jury pour leur intérêt pour mes travaux et leurs remarques pertinentes.

De manière générale, merci à tous ceux qui, de près ou de loin, m'ont aidé, supporté et soutenu durant cette tâche fastidieuse mais ô combien gratifiante qu'est la réalisation d'une thèse.

Sommaire

Introduction générale

Le développement de la microélectronique est aujourd'hui guidé par deux phénomènes majeurs : la diminution de la taille des éléments technologiques (transistors, connections), et l'augmentation du besoin en performance. Ces deux tendances entrainent l'apparition d'une problématique globale : comment optimiser l'espace disponible au sein d'une puce pour atteindre les performances (en calcul et en consommation) souhaitées.

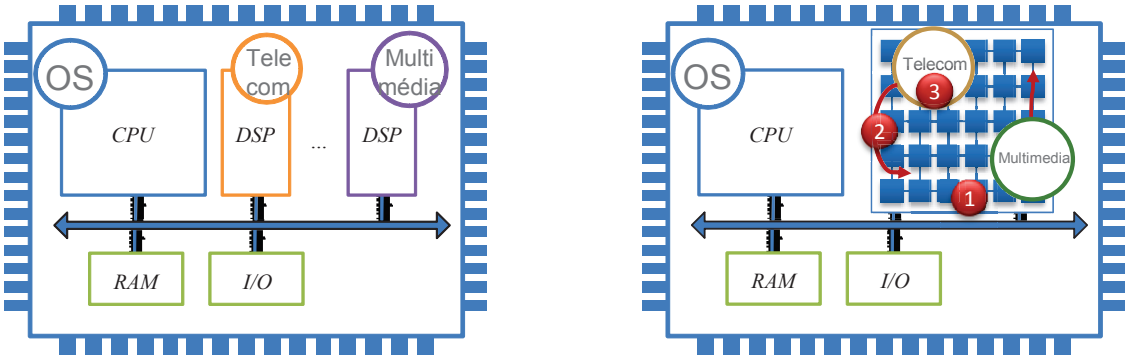
Dans le cadre des microprocesseurs, cette problématique s'est rapidement traduite par le développement de puces multiprocesseurs. En effet, l'utilisation de puces multiprocesseurs permet de cumuler les avantages de plusieurs éléments de calcul au sein d'une même puce, la rendant ainsi plus polyvalente et efficace. Ces architectures affichent un fort potentiel pour résoudre la majorité des défis actuels : optimisation de la consommation, à travers des mécanismes d'adaptation de chaque élément en fréquence et en tension, gestion de la fiabilité, grâce à la multiplication des éléments de calcul, rapidité de développement par la réutilisation de processeurs déjà existant, etc.

Ces architectures peuvent être séparées en deux catégories : les multiprocesseurs homogènes, possédant des éléments de calcul identiques, et des multiprocesseurs hétérogènes. Les multiprocesseurs hétérogènes sont spécialisés dans l'exécution d'un type de calcul bien défini. Ces accélérateurs optimisent les capacités maximales de calcul d'un appareil pour une consommation minimale : on dit qu'ils ont une bonne efficacité énergétique (rapport entre les performances de calcul et l'énergie consommée).

Cependant, ils possèdent également de nombreux désavantages : spécialisés, ils ne sont pas capables de réaliser facilement ni efficacement d'autres calculs que ceux pour lesquels ils ont été conçus. Ainsi, au sein d'une carte électronique classique, de nombreuses puces sont aujourd'hui nécessaires pour réaliser les différents calculs spécialisés (voir figure 1.a).

A contrario, la grande force des multiprocesseurs homogènes est leur flexibilité : par rapport à un accélérateur spécialisé, capable d'exécuter de façon optimale un type de programme précis, mais bien moins performant pour d'autres types de calcul, les multiprocesseurs homogènes sont capables d'exécuter un large panel de programmes avec une efficacité énergétique raisonnable. Ainsi, si l'on prend un large spectre d'applications différentes, les architectures homogènes peuvent avoir une efficacité énergétique supérieure à celle des architectures hétérogènes.

Les recherches effectuées au sein de l'équipe ADAC (cadre dans lequel cette thèse s'est développée) s'attachent à démontrer le potentiel d'un système non plus composé de plusieurs éléments spécialisées hétérogènes (figure 1.a), mais remplacé par une macro-puce multiprocesseur homogène (figure 1.b). Le grand avantage de ce type de puce, est qu'il permet de s'adapter en fonction de ses conditions d'utilisation. Ainsi, si les conditions d'utilisation de l'appareil changent (variation de la quantité de travail demandée, variation de la température, etc.), il est possible de répartir différemment les charges de calcul au sein de l'architecture pour optimiser ses performances (éteindre une partie des éléments de calcul, ou en faire fonctionner certains à vitesse réduite, etc.). Cet avantage permet d'obtenir de ce type de puce des performances et une efficacité énergétique globalement meilleures que pour un système basée sur des éléments spécialisés [1].



(a) **(b)**

Figure 1 : a. Exemple d'architecture classique d'un appareil électronique portatif, b. Architecture novatrice utilisant une puce multiprocesseur homogène comme accélérateur

Le but est de démontrer que malgré une performance en pointe inferieure à un système hétérogène, la flexibilité d'un système homogène va permettre d'obtenir des performances en moyenne au moins comparables, grâce à une meilleure utilisation des

ressources disponibles. De plus, l'utilisation de ce type d'architecture permet également un coût de conception plus faible (réplication d'un même élément de calcul), et d'une programmation plus facile (chaque élément a le même jeu d'instruction).

L'objectif de cette thèse est de montrer le potentiel des systèmes multiprocesseurs homogènes, à travers plusieurs optimisations exploitant la flexibilité de ceux-ci. Ces optimisations seront effectuées suivant trois axes :

- L'exploitation de la généricité de l'architecture, avec le développement d'une puce dont le nombre d'éléments de calcul est facilement modifiable, et dont les performances sont linéairement liées à ce nombre (item 1 sur la figure 1.b).
- La mise en place de mécanismes d'adaptation, avec différents protocoles de répartition des charges au sein de l'architecture permettant d'optimiser les performances dynamiquement (item 2 sur la figure 1.b).
- Le développement d'un environnement de programmation facilement utilisable, avec la mise en place de différents modèles de programmation de cette architecture (item 3 sur la figure 1.b).

Chapitre 1 : Contexte et Etat de l'art

1.1. Introduction

Le développement de systèmes embarqués a de tout temps dû faire face à de nombreux défis. Suivant la loi de Moore, les besoins en calculs et performances pour répondre aux exigences des utilisateurs ont toujours suivi une courbe exponentielle. La compétitivité mondiale dans ce domaine ainsi que l'arrivée de technologies nouvelles non basée sur silicium amènent même aujourd'hui plusieurs économistes à envisager un développement dans ce domaine encore plus rapide : on parle d'une loi « plus que Moore » (more than Moore law) [2].

Devant une telle demande, les développeurs doivent faire face à des problèmes en constante évolution (problèmes de fiabilité, problèmes de consommation énergétique, etc.). Cette thèse se focalise sur une des plus importantes de ces thématiques : la gestion de la complexité. En effet, devant la multitude des demandes de fonctionnalités d'une puce actuelle, et la quantité d'éléments aujourd'hui disponible au sein d'une puce, le problème de la conception d'une nouvelle puce a pris une dimension nouvelle.

Jusque dans les années 2000, deux méthodes principales permettaient d'augmenter les performances d'un calculateur. La première consistait à augmenter le nombre d'instructions que pouvait traiter un processeur en parallèle. Plusieurs instructions pouvant être exécutée en parallèle étaient concaténées pour former une macro-instruction. Cette concaténation pouvait être effectuée de manière logicielle, lors de la

compilation (on parle alors d'architecture *« Very Large Instruction Word », VLIW*), ou de manière matérielle lors de l'exécution (on parle d'architectures super-scalaires). Cependant, ce système possède ces limites, car la parallélisation d'instructions d'un code séquentiel est dans la plupart des applications très limité (on parle du *mur de Parallélisme au Niveau Instruction, ILP Wall*).

La seconde méthode consistait à augmenter le nombre d'étage de pipeline des processeurs : ainsi, il était possible d'augmenter la fréquence de ces processeurs qui pouvait donc potentiellement exécuter plus d'instructions par seconde. Ainsi, des architectures super-scalaires telles que le pentium 4 [3] possédant jusqu'à 20 étages de pipeline, ont vu le jour. Cependant, cette méthode s'est vue compromise à cause d'une barrière technologique : il s'agit de la puissance consommée. L'augmentation de la fréquence de ces architectures entraine également l'augmentation du nombre de transitions effectuées par seconde et donc la puissance consommée (on parle du *mur de fréquence, Frequency Wall*). Un processeur comme le Pentium 4 pouvait consommer plus de 100W en pic, soit une puissance surfacique de l'ordre de grandeur d'un réacteur nucléaire.

Aujourd'hui, la solution la plus communément utilisée pour augmenter les performances de calcul d'une puce est d'augmenter son nombre de processeurs. Contrairement au parallélisme d'instruction, le parallélisme au niveau tâche n'engendre pas de pénalité lors d'une prédiction de branchement. De plus, il permet d'exécuter plusieurs instructions en parallèle sans augmenter la fréquence d'exécution, contrairement à la technique d'augmentation du nombre d'étage de pipeline. La réalisation de puces multiprocesseurs a permis de passer cette barrière technologique, et a engendré l'apparition de nouveaux concepts de gestion de l'exécution et de la mémoire. Dans cette thèse, le terme d'**architecture multiprocesseur** ou de **système multiprocesseur** désignera de manière générale n'importe quelle puce possédant plusieurs processeurs de calcul indépendant.

Un second enjeu primordial dans la conception des processeurs actuels concerne le temps de développement. L'augmentation de la complexité des puces et la réduction des délais de production pour faire face à la concurrence amène les constructeurs à optimiser au maximum le temps de développement d'une nouvelle puce. Deux caractéristiques majeures permettent de déterminer la tendance d'une conception de microprocesseur à optimiser ce délai : la *réutilisabilité* et la *scalabilité*.

Définition : la *réutilisabilité* (*reusability*), détermine la part de la conception d'une puce de génération N qui pourra être réutilisé lors de la conception de la génération N+1.

Définition : la *mise à l'échelle*, ou la *scalabilité* (*scalability*), évalue l'aptitude d'un système à pouvoir augmenter ses capacités en fonction de ses ressources. Ainsi, un système est dit

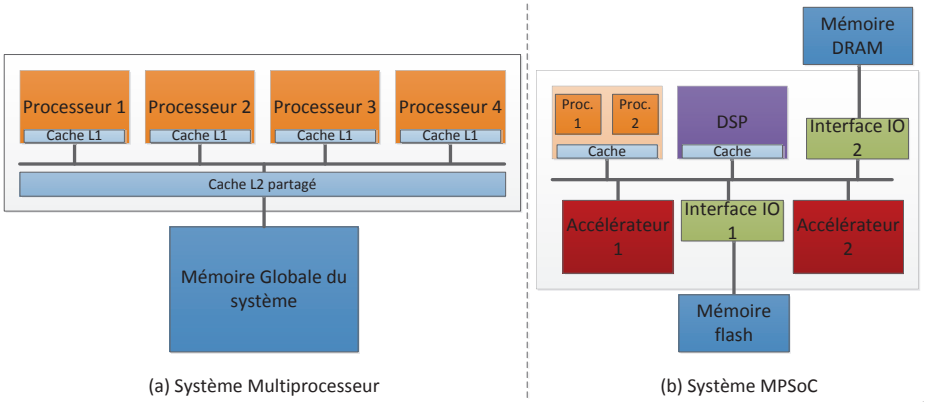
scalable en performance si pour doubler les performances de celui-ci, il suffit de doubler sa taille.

Dans cette thèse, une architecture multiprocesseur possédant les qualités de scalabilité et de réutilisabilité sera exposée, et différents concepts d'optimisation de celle-ci seront proposés, concernant notamment les communications, la gestion de la mémoire, ou encore le modèle de programmation. Ce chapitre a pour but d'exposer l'état actuel du développement dans le domaine des puces multiprocesseurs, et des différents concepts de programmation et d'optimisation relatifs à ce domaine. A partir de ces constatations, les différentes contributions de cette thèse seront amenées en conclusion de ce chapitre.

1.2. Evolution des architectures multiprocesseurs

Aujourd'hui, la conception d'architectures multiprocesseurs s'est généralisée à tous les domaines du calcul numérique (calcul haute performance, ordinateurs personnels, domaine de l'embarqué). On distingue parmi les puces multiprocesseurs deux grandes familles (voir Figure 2):

- Les systèmes homogènes : possédant des cœurs de calcul de même type. Ces architectures sont généralement utilisées dans le domaine des ordinateurs personnels. L'ensemble des cœurs possède la plupart du temps un cache partagé hiérarchique (cache L1 et cache L2), ainsi qu'une mémoire centrale de grande capacité et de forte latence. Afin de les distinguer, on nommera ces architectures **systèmes multicœurs**.
- Les systèmes hétérogènes: possédant des cœurs de calcul de type différent. D'une efficacité plus grande que les systèmes homogènes sur un type de calcul particulier, mais plus difficile à programmer, ces systèmes sont la plupart du temps utilisés dans le domaine de l'embarqué [4]. Ceux-ci consistent généralement en la connexion de plusieurs processeurs ou accélérateurs dédiés avec de nombreuses interfaces d'entrées/sorties diverses (capteurs, actionneurs, interface homme-machine, mémoire de tous types, etc.). Par la suite, on nommera ces architectures Systèmes sur Puce MultiProcesseur (**MultiProcessor System on Chip, MPSoC**).



(a) Système Multiprocesseur (b) Système MPSoC

Figure 2 : Architecture typique des deux grandes familles de multiprocesseurs : (a) Multicœur, (b) MPSoC

1.2.1. Systèmes Homogènes

Avant les années 2000, le développement de systèmes multicœurs homogènes n'avait vu le jour que dans quelques cas particuliers. Au milieu des années 90, la société Sun Microsystems a par exemple sorti le microprocesseur MAJC. Celui-ci était composé de 2 cœurs *VLIW*, qui permettait d'avoir à la fois une exécution multi-instruction et multitâche[5]. Cette architecture fut utilisée dans certains accélérateurs graphiques des stations Sun, mais fut surtout le point de départ du développement des multicœurs UltraSPARC. En 2001, la société IBM sort le premier multicœur pour ordinateur personnel : le POWER4 [6]. Cette architecture était composée de deux cœurs PowerPC 64-bits, et de trois niveaux de cache, dont le troisième se trouvait en dehors de la puce. Ce processeur fut utilisé dans les ordinateurs RS6000 et AS/400, principalement réservé au domaine professionnel du fait de leurs coûts.

L'apparition des premiers systèmes multicœurs grand public a eu lieu en 2004. Cette année là, les deux principaux constructeurs de processeurs pour ordinateur personnel ont annoncé le développement d'un système double cœurs : tout d'abord AMD, en aout, publie les premiers résultats concernant le développement d'une puce x86-64 bits double cœurs, gravé en 90 nm [7], puis Intel, en septembre, annonce également l'apparition de son premier processeur double cœurs, à l'occasion du *Intel Developper Forum* de San Francisco[8].

Bien que le marché des multicœurs soit aujourd'hui majoritairement celui des ordinateurs personnels, d'autres domaines possèdent ce type d'architecture sur leurs marchés. Dans le domaine des applications réseaux, la société *Cavium Networks* développe depuis 2004 sa famille de processeurs multicœurs OCTEON et OCTEON2 [9], permettant des traiter efficacement les demandes d'un réseau internet. Dans le domaine des serveurs de communication, la société *Oracle* développe la série des multicœurs UltraSARC. Leur dernier projet, le SPARC-T5 inclut 16 processeurs identiques capables chacun de faire tourner 8 tâches grâce à la technologie d'*Hyperthreading* (partage de l'exécution sur un cœur entre plusieurs tâches au niveau matériel). Les marchés du traitement audio et graphique, des transports, ou encore des simulations physiques et financières ont également été abordés par les multicœurs.

L'un des points fort de ces architectures, expliquant leur popularité, vient de leur généricité, et de leur historique. En effet, avant l'apparition de puces multicœurs, des ordinateurs multiprocesseurs étaient développés. De plus, le fait d'utiliser une structure homogène et un système de gestion de la mémoire centralisé permet une programmation plus simple de ces architectures par rapport aux MPSoCs. Cependant, cet avantage est compensé notamment par le manque d'efficacité sur des calculs spécifiques, mais aussi par la faible scalabilité de ce type de système. En effet, la centralisation de la mémoire rend la

gestion de celle-ci particulièrement complexe lorsque le nombre de cœurs augmente significativement (plus d'une dizaine de cœurs) [10].

1.2.2. Systèmes Hétérogènes

La conception d'architectures multiprocesseurs hétérogènes est particulièrement développée dans les domaines spécialisés. En effet, l'utilisation d'un cœur dédié permet d'optimiser le calcul dans un domaine spécifique, au détriment de la généricité.

Dans le domaine de la téléphonie mobile par exemple, la grande majorité des systèmes actuels est composée de puces multiprocesseurs hétérogènes, chacun des cœurs étant spécialisé dans une tâche donnée. Dans le cas de la plateforme Qualcom MSM7200 par exemple [11], 4 cœurs de calculs ainsi que plusieurs accélérateurs sont connectés au sein d'une même puce : 1 processeur ARM11 s'occupe de la gestion applicative, alors qu'un processeur ARM9 aidé de deux Processeurs de Traitement de Signal (*Digital Signal Processor, DSP*), traitent les applications audio et modem. Des accélérateurs 2D, 3D et Java viennent également compléter cette architecture.

Dans le cadre du décodage audio/vidéo, Texas Instrument introduit en 2011 la plateforme OMAP 5 [12]. Celle-ci repose sur quatre cœurs hétérogènes (2 ARM Cortex A15 et deux ARM M4) couplés à un accélérateur multicœur graphique POWERVR SGX544-MPx permettant de traiter des textures et des interfaces 3D. Une centaine d'autres blocks spécialisés tel que des processeurs audio ou vidéo ainsi que des interfaces I/O sont également présent sur cette plateforme (voir Figure 3). Cette architecture est capable de fournir le décodage d'un flux vidéo full HD en 3D.

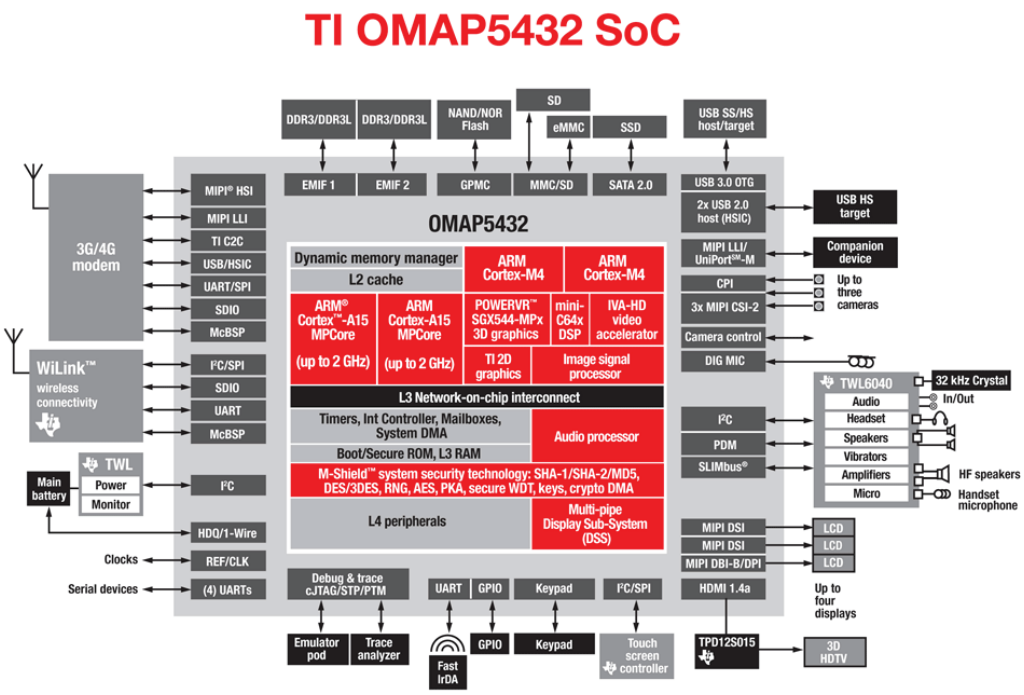


Figure 3 : Architecture de la plateforme OMAP 5432 (tiré de [12])

Le développement des MPSoCs est beaucoup plus ancien et diversifié que celui des multicœurs[13]. Cette diversité s'explique par la variété des domaines de lesquels ceux-ci sont utilisés : communication[11], réseau[14], traitement multimédia [12][14][15], transport[11][15], embarqué [12], etc. De ce fait, la programmation d'un MPSoC est souvent spécifique à celui-ci, et peu d'environnements de développement sont compatibles avec plusieurs MPSoCs différents[13].

1.2.3. Accélérateurs Graphiques

Un autre type de puces multiprocesseurs particulier existe également sur le marché : celui des accélérateurs graphiques (*Graphical Processing Unit, GPU*). Ces processeurs possèdent généralement un très grand nombre d'unités de traitement identiques, organisées autour d'un maillage hiérarchique symétrique. Ces multiples cœurs permettent d'exploiter le fort parallélisme des applications graphiques, mais sont également aujourd'hui utilisés pour le calcul dans d'autres domaines tel que le calcul Haute Performance (*High Performance Computing, HPC*), ou encore la modélisation.

La Figure 2 montre l'architecture d'un processeur graphique GTX280 de la société NVIDIA. Ce processeur fait parti de la gamme GTX, pouvant posséder jusqu'à 1536 cœurs pour la GTX680. Cette puce est composée de 240 cœurs identiques réunis sous forme de 10 clusters, eux-mêmes composés par 3 groupes de 8 cœurs identiques. Chaque groupe de 8 possède une mémoire locale partagée, et chaque cluster possède un cache partagé, connecté ensuite à une mémoire centrale (hors puce) par un bus très haut débit 512 bits.

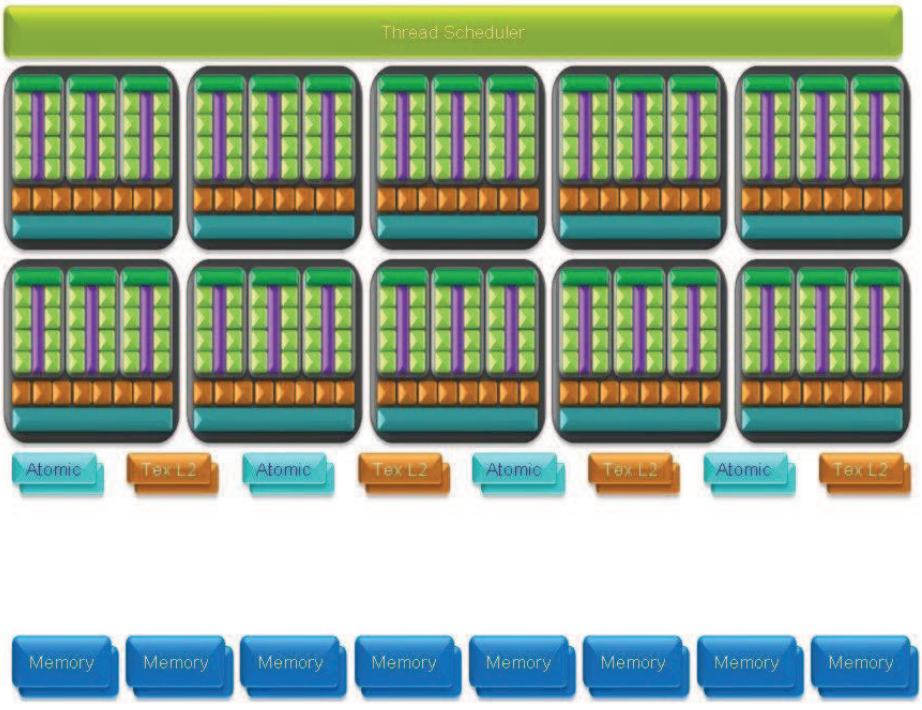


Figure 4 : architecture d'une puce graphique GTX280

Ces architectures permettent d'obtenir des performances en calcul parallèle particulièrement intéressantes (la GTX280 permet théoriquement d'exécuter 933 milliard d'opérations à la seconde), et ainsi une efficacité énergétique supérieure à celle des architectures multicœurs. Cependant, elles souffrent d'une programmation plus complexe que celles-ci.

1.3. Modèles de programmation

Dans le cadre d'une architecture multiprocesseur, la vision classique séquentielle de la programmation n'est plus représentative du fonctionnement de ces puces. Afin d'exploiter de manière adéquate ces architectures, des modèles de programmation parallèle doivent être utilisés.

Les concepts de la programmation parallèle ont commencé à être développés dans les années 70, afin de répondre aux besoins des premiers ordinateurs hautes performances (HPC), qui réalisaient un véritable réseau de calculateurs parallèles. Aujourd'hui, tous les systèmes d'exploitation (OS) commerciaux ont la possibilité de permettre une programmation parallèle à travers l'utilisation de librairies, ou même nativement implantée à l'intérieur de l'OS.

On peut distinguer deux grandes familles de programmation parallèle : la première consiste à créer plusieurs tâches indépendantes, communicant par l'intermédiaire de messages (on parle de programmation à passage de messages). La seconde consiste à créer des tâches ayant un ensemble de variables communes, accessibles par chacune (on parle de programmation à mémoire partagée).

1.3.1. Programmation par passage de messages

Historiquement la première utilisée, la programmation par passage de message est particulièrement adaptée (mais pas uniquement) aux architectures MPSoC. En effet, avec ce modèle de programmation, la communication entre les tâches s'effectue par l'intermédiaire de fonctions d'envoi ou de réception explicitement déclarées dans le code, qui permettent de faire transiter l'information d'une tâche à une autre. De ce fait, si deux tâches sont situées sur des cœurs de natures différentes, il suffit d'avoir une implémentation spécifique pour chaque cœur du même standard pour pouvoir effectuer facilement la communication. De même, dans un système à plusieurs mémoires distribuées, les données peuvent transiter facilement d'une mémoire à l'autre car les déplacements se font de manière explicite.

Parmi les nombreuses librairies développées pour la programmation par passage de message, on peut citer CORBA (Common Object Request Broker Architecture) [16], DCOM (Distributed Component Model) réalisé par Microsoft, SOAP (Simple Object Access Protocol) utilisant le protocole http pour transmettre ses messages, ou encore MPI

(Message Passing Interface) une norme définissant une bibliothèque de fonctions (*Application Programming Interface, API*). MPI a été défini en 1993 et est utilisable avec les langages C, C++ et Fortran. Cette norme a réussi à s'imposer comme le standard le plus utilisé pour réaliser un système gérant la programmation par passage de message. La bibliothèque MPI est constituée de fonctions bas-niveau permettant de réaliser de simples communications point à point, ainsi que de fonctions plus haut-niveau permettant par exemple l'envoi d'information à un groupe de tâches, ou encore la distribution de données d'un tableau à différentes tâches.

Dans le domaine des MPSoCs, de nombreux travaux ont été réalisés pour proposer une implémentation de MPI (souvent non complète) performante. On peut notamment citer le développement de TDM-MPI [17], une implémentation allégée de MPI pour architecture multicœurs basée sur FPGA. Dans [18], les auteurs présentent une sélection de 11 primitives implémentées (point-à-point et avancées) pour systèmes multicœurs basés sur des processeurs de signaux numériques (Digital Signal Processing, DSP). L'implémentation des primitives de communication point-à-point du standard MPI sur la plateforme développée dans cette thèse sera exposée chapitre 3.

1.3.2. Programmation à mémoire partagée

La programmation à mémoire partagée est basée sur le partage d'une zone mémoire entre plusieurs processus pour communiquer. Afin d'éviter les situations de compétition sur l'accès à une donnée partagée (*race condition* en anglais), les standards de la programmation à mémoire partagée se doivent d'inclure des barrières de synchronisations entre les tâches.

La programmation à mémoire partagée est également appelée programmation multithread (*multithread programming* en anglais). Cette appellation vient du fait de la distinction d'une **tâche**, processus indépendant possédant sa propre mémoire indépendante, et un **thread**, processus léger partageant son espace mémoire. On parlera donc de tâche lorsque l'on évoquera la programmation par passage de message, et de thread lorsqu'on évoquera la programmation à mémoire partagée.

Dans les années 90, sous l'impulsion du développement de machines multiprocesseurs, un très grand nombre d'implémentations de librairies de programmation à mémoire partagée ont vu le jour. Le protocole DCE/RPC (*Distributed Computing Environment* / *Remote Procedure Calls*) [19], créé en 1993, permet de réaliser une programmation à mémoire partagée sur une architecture de type réseau d'ordinateurs. La librairie GNU-Pth [20] a également été développée pour les systèmes Linux. Cette librairie permet d'utiliser la programmation à mémoire partagée au sein d'une application, sans faire appel à des fonctions du système d'exploitation : la gestion de la concurrence des threads est gérée par la librairie elle-même, lors de l'exécution de l'application. Cependant, le principe de ne

pas faire appel aux fonctions du système d'exploitation fait que cette librairie ne permet pas d'exécuter plusieurs threads sur des cœurs distincts (parallélisme artificiel).

En 1993, sous la concertation des développeurs de Sun et Linux, le standard POSIX thread (abrégé en PThreads) a été créé [21]. Ce standard défini une Interface de Programmation (API) permettant la création, la gestion et la destruction de threads, ainsi que des primitives de synchronisation. Parmi les autres standards adoptés par la majorité des systèmes d'exploitations grand public, on peut citer OpenMP [22], développé depuis 1997, qui aujourd'hui est en version 3. Les systèmes Windows NT possèdent également leur propre standard de gestion de programmation multithread, bien que des interfaces PThreads et OpenMP aient été réalisées, permettant de faire le lien entre celles-ci et les librairies standards de Windows.

La programmation à mémoire partagée est particulièrement adaptée aux architectures multicœurs. En effet, la structure à mémoire centralisée de ces architectures permet de facilement définir une zone mémoire partagée. Le concept de partage de mémoire sur ces architectures a mené à la création d'une gestion du parallélisme par le système d'exploitation appelée *SMP* (*Symetric MultiProcessing*). Cette méthode consiste à exécuter un unique système d'exploitation sur le système (par l'un des cœurs) qui va répartir lui-même l'exécution des tâches sur les différents cœurs. Cette méthode est particulièrement intéressante pour gérer à haut niveau la répartition des tâches sur le système, mais ne prend pas en compte le temps d'accès entre une mémoire et un cœur, ce qui peut s'avérer sous optimal dans une architecture à mémoires distribuées [23].

En ce qui concerne les MPSoCs, peu de travaux se focalisent directement sur l'implémentation d'une librairie multithread : la plupart des systèmes sont développés pour être utilisable avec des systèmes d'exploitations standard (SMP), sur lesquels ces librairies sont souvent déjà implémentées. Dans [24], les auteurs explorent les difficultés et les défis d'une implémentation OpenMP sur une architecture de type MPSoC. Ils exposent notamment les difficultés d'utilisation d'une programmation multithread sans gestion de cohérence de cache matérielle.

1.3.3. Programmation pour accélérateurs graphiques

L'architecture particulière des processeurs graphiques, composée d'un très grand nombre de cœurs de calcul basiques réalisant les mêmes opérations sur des données différentes, a entrainé la mise en place de concepts et de modèles de programmation propres à celles-ci.

De plus l'évolution de la demande en calcul sur ces architectures pour des applications non graphiques (calculs hautes performances, physique ondulatoire, mécanique des fluides, traitement multimédia, bioinformatique, etc.) a amené à la création de plusieurs langages développés spécifiquement pour fournir les outils nécessaires à une programmation

générique sur des processeurs graphiques : on parle de programmation *GP-GPU* (*General Purpose-Graphic Processing Unit*).

La Figure 5 donne une vue globale du principe de programmation GP-GPU. L'idée est d'avoir un processeur maitre (généraliste la plupart du temps), pilotant l'exécution du programme sur l'architecture graphique. Dans un premier temps, les données sont envoyées à la mémoire du GPU. Ensuite, les instructions sont données au GPU pour traiter ces données de manière parallèle. Enfin, le résultat de cette exécution est renvoyé depuis la mémoire du GPU vers la mémoire globale, accessible par le processeur maitre.

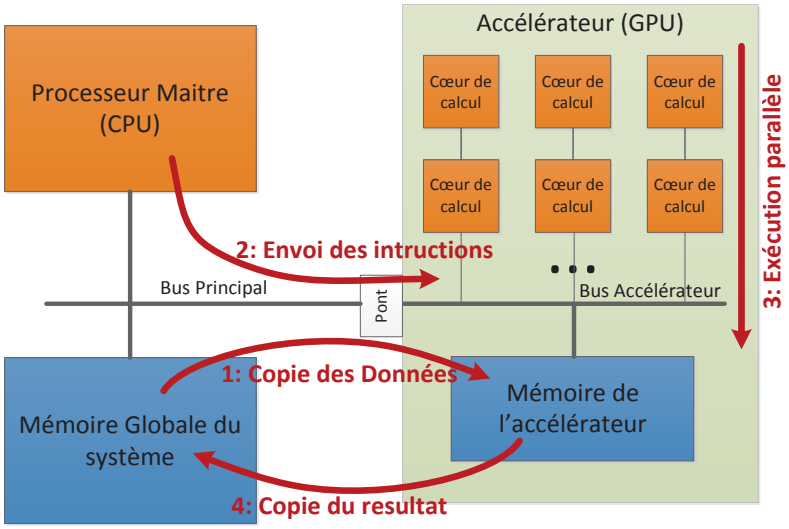


Figure 5 : Principe de la programmation GP-GPU

Parmi les principaux développeurs de langages GP-GPU, ont retrouve bien entendu les deux grands constructeurs de puces graphiques. AMD a pour sa part conçu AMD FireStream [25], associant à la fois un kit de développement logiciel (SDK) complet permettant de programmer, compiler et exécuter un programme GP-GPU et nommé AMD App SDK, mais aussi des plateformes composées de processeurs graphiques pour les utiliser. Cette association cible les utilisateurs souhaitant faire du calcul parallèle intensif, tel que les calculs scientifiques ou d'ingénierie. La société Nvidia a quant à elle développé le kit de développement CUDA [26] (Compute Unified Device Architecture) pour ses architectures. CUDA a connu une plus grande popularité que FireStream, et est aujourd'hui utilisé par de nombreux logiciels de calculs scientifiques, tels que Matlab par exemple.

Dans le domaine Open-Source, l'université de Standford a développé une librairie de programmation GP-GPU nommée Brook [27]. Celle-ci s'appuie sur les librairies graphiques OpenGL et DirectX pour apporter des fonctions avancées permettant la programmation de programmes génériques (i.e. non graphiques). L'avantage du développement utilisant Brook est le fait qu'il peut être utilisé par n'importe quelle architecture supportant soit

OpenGL, soit DirectX, contrairement à FireStream et CUDA, respectivement liés aux architectures AMD et Nvidia.

D'autres industriels se sont lancés dans le développement de librairies GP-GPU, notamment Microsoft, avec DirectCompute [28], inclus dans les librairies standards de DirectX11, Google avec RenderScript [29], utilisable sur le système d'exploitation Android, ou encore Intel, avec RapidMind[30].

Mais l'approche la plus populaire aujourd'hui est le développement du standard *Open Computing Langage* [31] (*OpenCL*). OpenCL inclus à la fois une interface de programmation utilisateur et un langage de programmation dérivé du C. Il est développé par le groupe Chronos, fondé par Apple, en collaboration avec les équipes d'AMD, Nvidia, Intel et IBM, dans l'objectif de fournir une plateforme de programmation pour architectures hétérogènes. De ce fait, ce projet est disponible sur de nombreuses plateformes et de nombreux systèmes d'exploitation différents. Bien que le développement des librairies et compilateurs pour une plateforme particulière soit souvent propriétaire, le standard a été placé sous licence ouverte, ce qui permet à tout développeur de produire sa propre librairie OpenCL pour une plateforme particulière.

Ainsi, plusieurs kits de développement OpenCL ont été réalisés pour différentes plateformes (AMD, Nvidia, Intel…) et différents systèmes d'exploitation (Windows, Linux, MacOS, etc.). Dans le domaine de l'embarqué, la société Ziilab a développé son propre kit pour les architectures ZMS [32], processeurs graphiques pour matériels embarqués. Les auteurs de [33] explorent les avantages et les inconvénients en termes de performance de l'utilisation du la programmation GPGPU sur une architecture embarquée. Pour se faire, ils développent un environnement de programmation basé sur OpenCL permettant notamment d'évaluer la puissance consommée par le système. Dans [34], les auteurs présentent l'implémentation partielle d'un kit de développement pour une architecture multicœur composée de plusieurs unités de calcul simples ayant chacune sa propre mémoire locale, privée et indépendante.

Dans le cadre des MPSoCs, l'utilisation de ce type de langage pour harmoniser le pilotage des accélérateurs par un processeur généraliste maitre semble être une solution prometteuse qui permettrait d'uniformiser l'environnement de développement des MPSoCs.

1.4. Adaptation

Le terme d'adaptation au sein d'une puce de calcul concerne la capacité de celle-ci à s'adapter rapidement et efficacement à un changement. On dit qu'un système est adaptatif s'il est capable de modifier son comportement en fonction de son état interne et/ou de son environnement extérieur. Les architectures multiprocesseurs, de par leurs

structures, offrent de nombreuses possibilités d'adaptation différentes de celles des processeurs classiques. On peut séparer celles-ci en deux catégories :

- Les techniques d'adaptation au niveau architecture : elles concernent les techniques bas niveaux, où des mesures de capteurs vont venir modifier les caractéristiques physiques de l'architecture (fréquence, tension, …).
- Les techniques d'adaptation au niveau système : elles concernent les techniques hauts niveaux, où l'analyse de l'état général du système permet de prendre des décisions concernant la gestion de l'exécution au sein des cœurs de calcul.

1.4.1. Au niveau architecture

Les techniques d'adaptation au niveau architecture s'attachent à modifier une caractéristique physique d'une portion du système : soit la fréquence (on parle alors de *Dynamic Frequency Scaling, DFS*), soit la tension (on parle alors de *Dynamic Voltage Scaling, DVS*), soit les deux ensemble (on parle *Dynamic Voltage and Frequency Scaling, DVFS*).

Dans les systèmes multicœurs (homogènes), la technique historiquement la plus utilisée industriellement aujourd'hui est le *clock gating* [35]: cette technique consiste à désactiver le signal d'horloge d'un cœur ne faisant aucun calcul. Depuis, d'autres techniques plus élaborées de DVFS ont également vu le jour pour ce type d'architecture [36].

Les techniques d'adaptation au niveau architectures sont extrêmement variées : elles peuvent avoir pour objectif de gérer la température [36] [37], la fiabilité [38] ou encore la puissance consommé du système [36] [37] [39]. Elles peuvent également découper le système de manière différente : au niveau des processeurs [36][38], au niveau de l'interconnexion [38], ou encore au niveau de grappes de processeurs [39][37], voire même sur le système tout entier [39] [37].

Dans [38], les auteurs lient les variations de tension et fréquence d'un cœur à sa fiabilité. Ils proposent un évaluateur de fiabilité capable de prédire la fiabilité d'un élément du système. A partir de celui-ci, ils construisent un modèle d'optimisation de fréquence et de tension permettant de tenir compte en même temps des performances, de la consommation d'énergie, mais également de la fiabilité du système.

Les travaux effectués dans [39] permettent de gérer la puissance globale du système en analysant soit la charge de chacun des processeurs du système, soit le débit imposé par le cahier des charges en sortie d'application de flux pour optimiser la fréquence et le tension de chaque élément de calcul du système.

Les auteurs de [37] combinent une technique de type DVFS avec une adaptation au niveau système : la répartition des tâches au sein des processeurs. Leur méthode consiste à

utiliser une double boucle de régulation : l'une à grain fin, utilisant le DVFS pour optimiser la consommation d'un processeur, l'autre à grain plus large, utilisant le système d'exploitation pour répartir les tâches au sein du système afin d'éviter l'apparition de points chauds (zone de température élevée au sein du circuit dû à une activité importante).

Au sein de l'équipe ADAC, plusieurs travaux d'adaptation ont été réalisés, dans le cadre notamment d'une thèse axée sur cette thématique [40]. Au niveau architecture, un algorithme de variation de la fréquence de chaque processeur en fonction de la charge de celui-ci a été proposé. Cette méthode a la particularité d'utiliser un contrôleur PID pour régler la fréquence adéquate. Au delà de l'indépendance de chaque contrôleur, cette méthode offre également l'avantage d'une bonne réactivité et d'une bonne précision.

1.4.2. Au niveau système

Les techniques d'adaptation au niveau système dans une architecture multiprocesseur consistent principalement en la répartition optimale de l'exécution des charges de calculs demandés par l'utilisateur, au sein des différents processeurs. Ces techniques peuvent être divisées en deux catégories :

- Les techniques de répartition globale des charges de calculs sur l'ensemble des processeurs à un moment donné : on parle de technique de *répartition de charges* (ou *répartition de tâches*, lorsque les charges sont découpés sous forme de tâches)
- Les techniques de déplacement d'une charge (ou d'une tâche) d'un processeur à un autre : on parle de *migration de tâche*.

Au sein des architectures multicœurs, pilotées par un système d'exploitation SMP, la répartition des tâches au sein du système est facilitée par la gestion centralisée de l'exécution de celles-ci. En effet, la présence d'un unique ordonnanceur sur le système, permet de maitriser la position de chaque tâche sur le système [41].

En ce qui concerne les MPSoCs, la répartition des tâches est plus complexe. De nombreuses techniques font appel à des patrons de répartition préétablis lors de la conception du système. Ce type de méthode permet de prendre en compte de manière fiable les contraintes d'échéances d'un système temps réel, en simulant le pire cas du scénario considéré. On peut donc ensuite naviguer d'un scénario à l'autre sans avoir le souci de dépasser une échéance. Cette méthode, proposé notamment dans [42], permet de maintenir une faible consommation du système lors de son fonctionnement. Dans ces travaux, un gestionnaire de répartition ayant une vue globale du système est utilisé pour minimiser la consommation.

Dans [43], les auteurs améliorent la répartition statique adaptative de tâche : les patrons de répartition préétablis ne concernent plus des cœurs physiques mais logiques. On peut

ainsi obtenir des répartitions différentes suivant un même patron, par des méthodes de rotation, translation ou symétrie entre les cœurs.

Il existe également d'autres méthodes de répartition de charge pour des applications avec contraintes temps réel. La stratégie développé dans [44] consiste à distinguer des cœurs principaux et des cœurs secondaires. Chaque cœur principal peut s'allouer un certain nombre de cœurs secondaires suivant sa charge. On évite ainsi une répartition des tâches entièrement statique, tout en conservant les contraintes de temps réel en utilisant le pire scénario (nombre de cœurs secondaires minimum).

Un des autres objectifs les plus populaires pour l'utilisation de la répartition dynamique de tâche concerne la gestion de la température. Plusieurs travaux se focalisent sur l'élimination de l'apparition de points chauds au sein de l'architecture [45] [46]. Le système d'exploitation décrit dans [45] utilise une nouvelle technique d'ordonnancement des tâches en fonction de la température. Un indicateur de la capacité de chaque tâche à augmenter la température d'un cœur est utilisé pour aider l'ordonnanceur dans le choix des tâches à exécuter sur chacun des cœurs. Les auteurs de [46] exposent une stratégie de migration de tâche qui prend en compte la diffusion thermique des cœurs voisins.

Devant l'augmentation importante du nombre de nœuds de calculs au sein d'un MPSoC, les travaux concernant des stratégies de migration distribuées deviennent de plus en plus populaires. En effet, celles-ci offre la possibilité de conserver leurs performances lors de l'augmentation du nombre de cœurs : elles permettent donc de conserver la scalabilité d'un système.

Dans [47] le réseau sur puce du MPSoC est utilisé pour fournir des informations sur la charge de chacun des nœuds du système, et ainsi répartir la charge en utilisant ces informations. Le système utilise une vision de la mémoire partagée, bien que chaque nœud prenne ses décisions de manière locale et autonome.

Parmi les travaux de l'équipe ADAC [40], des heuristiques distribuées de répartition de charge ont été proposées, permettant une répartition des charges tout en conservant la scalabilité du système. Ces travaux se basent sur des notions d'attrait ou de répulsion de la part de processeurs voisins : un processeur fortement chargé va chercher à donner une tâche à son voisin le plus faiblement chargé, alors qu'un processeur faiblement chargé va chercher à prendre une tâche de son voisin le plus chargé.

1.5. Contributions

Si l'on cherche à faire la synthèse des éléments exposés précédemment, on s'aperçoit que 3 grandes catégories de systèmes multiprocesseurs ressortent (les multicœurs, les MPSoCs et les accélérateurs graphiques), chacun possédant de manière générale des modèles de

programmation, des techniques d'adaptation et un potentiel de mise à l'échelle propre. Le Tableau 1 résume ces caractéristiques.

Ce tableau permet de mettre en avant les avantages et inconvénients de chaque type d'architecture. Si l'on prend tout d'abord le cas de la scalabilité, la centralisation de la mémoire ne permet pas d'obtenir une architecture scalable, ce qui rend le développement d'architectures multicœurs ou d'accélérateurs graphiques de plus en plus contraignant.

Secondement, l'hétérogénéité d'un MPSoC classique entraine le développement de techniques d'adaptation pour ce type d'architecture particulièrement complexe est souvent non optimisé. Cela entraine une sous utilisation des potentialités maximales de celui-ci lorsque la demande en calcul n'est pas maximale (pas d'adaptation à la charge demandée).

Type d'architecture	Type de cœur	Modèle de programmation	Architecture Mémoire	Scalabilité	Adaptation
Multicœur	Homogène	SMP	Mémoire centralisée	Faible	Gestion simple car centralisée
MPSoC	Hétérogène	Mémoire partagée	Mémoire centralisée	Faible	Gestion complexe et non optimisée
Accélérateur Graphique	Homogène	Mémoire privée + mémoire partagée	Mémoire centralisée	Faible	A la compilation et par l'ordonnanceur matériel
Architecture SHoP	Homogène	Passage de messages	Mémoire distribuée	Optimisée	Gestion simple optimisée
Architecture développée dans cette thèse	Homogène	Mémoire partagée et passage de messages	Mémoire distribuée	Optimisée	Gestion améliorée

Tableau 1 : caractéristiques typiques de chaque type d'architectures multiprocesseurs

Pour palier à ces deux problèmes, une architecture a été développée au sein de l'équipe ADAC depuis plusieurs années, basée sur la connexion de processeurs identiques possédant chacun une mémoire personnelle : l'architecture *SHoP* (*Self HOmogeneous*

Platform) [40]. Cette architecture avait pour objectif de concurrencer les performances d'un MPSoC hétérogène à travers le développement de mécanismes d'adaptation permettant de gérer les ressources disponibles pour coller au plus près de la charge demandée. C'est sur cette architecture que les différentes méthodes d'adaptation évoquées dans la section précédente (variation de fréquence par retour PID, heuristiques de répartition de charges distribuées) ont été développées.

Cependant, cette architecture souffrait néanmoins du manque de programmabilité intrinsèque à sa structure : le modèle de programmation par passage de messages ne permettait pas d'envisager une utilisation simple de cette architecture.

Les contributions apportées durant cette thèse concernent donc la reprise du concept de cette architecture avec pour objectif une amélioration concernant deux axes :

- La gestion des communications afin d'améliorer les possibilités d'adaptation de cette architecture
- La gestion de la mémoire, afin de permettre la mise en place d'un modèle de programmation de type mémoire partagée sur une architecture distribuée

Ce manuscrit sera donc organisé comme ceci :

- Le chapitre 2 exposera la structure de l'architecture SHoP, et les différentes modifications qu'il a été nécessaires de faire pour envisager le développement des améliorations présentées ici. La nouvelle plateforme sera alors décrite, avec plusieurs mesures de performances.
- Le chapitre 3 expliquera les différentes optimisations des stratégies d'adaptation réalisées grâce à la nouvelle architecture. Grâce à l'amélioration des communications et de la gestion de la mémoire, plusieurs nouvelles méthodes d'adaptation seront proposées.
- Le chapitre 4 s'attardera sur la programmabilité du système. Un nouveau modèle de programmation hybride sera présenté, à la fois capable d'effectuer du passage de messages et de la programmation multithread. Afin de valider ce modèle, le développement de la librairie PThreads sera exposé.

Du fait de la diversité des thèmes abordés durant cette thèse, et de la quantité des recherches présentes dans chacun de ces différents domaines, il sera exposé dans chaque chapitre un état de l'art spécifique afin de rappeler le contexte dans lequel ces recherches ont été effectuées.

Chapitre 2 : Une architecture scalable à gestion de mémoire hybride : OpenScale

2.1. Introduction

L'un des défis les plus importants dans la conception actuelle de puces multiprocesseurs concerne la pérennité de celles-ci : devant la complexité grandissante des systèmes et les temps de conception toujours plus court pour faire face à la compétitivité de ce domaine, il est nécessaire aujourd'hui de pouvoir réutiliser les composants des puces des générations précédentes pour en concevoir une nouvelle.

La méthode la plus efficace pour surmonter ce défi est l'utilisation d'une architecture multiprocesseur scalable : avec ce type de puce, l'augmentation du nombre de processeur engendre une augmentation de nombre de composants de manière linéaire, et donc n'augmente pas la complexité globale.

C'est dans cet objectif que le développement de l'architecture SHoP au sein de mon équipe de recherche a été réalisé au cours de ces dernières années. Ce type d'architecture s'appuie sur une grille d'éléments de calcul identiques possédant chacun sa propre

mémoire locale, interconnectés au moyen d'un Réseau sur Puce (*Network on Chip, NoC*). L'utilisation d'un NoC permet en effet d'assurer la scalabilité de l'architecture [48] : l'ajout d'un nœud (et donc d'un processeur) sur un Réseau sur Puce implique l'ajout de connexions vers d'autres nœuds, et donc un accroissement de la bande passante globale du système.

Ce type d'architecture a fait l'objet de nombreux développements dans la recherche autant au niveau académique qu'industriel. Ainsi, la société TILERA a annoncé en 2007 le développement du TILE64 [49], architecture multiprocesseur basée sur une grille de 8x8 processeurs homogènes, et capable d'effectuer jusqu'à 443 milliards d'opérations à virgule flottante par seconde (443 GFLOPs). Le projet de recherche TeraScale [10], développé par la société Intel, a pour but le développement d'architectures basées sur cette structure. Cette société a, dans le cadre de ce projet, proposé l'architecture POLARIS en 2007, capable d'atteindre le trillion d'opération par seconde (1TeraFLOP), ou encore plus récemment le SCC (*Single-Chip Cloud Computer*), promis pour posséder un nombre de cœurs dépassant la centaine. Ce dernier est composé de nœuds de calcul double-cœurs possédant chacun un cache L1 séparé, et un cache L2 unifié.

Dans le cadre de cette thèse, nous présenterons l'architecture OpenScale, une amélioration de l'architecture SHoP permettant de développer les différentes contributions au niveau communication, gestion de la mémoire et programmation proposées dans cette thèse.

Ce chapitre est organisé comme suit :

- Dans un premier temps, l'architecture SHoP sera présentée, et les choix structurels et logiciels réalisé seront expliqués.
- Ensuite, les différents objectifs d'amélioration seront exposés.
- Enfin, l'architecture OpenScale sera présentée, en se focalisant sur les différences par rapport à SHoP, et les performances de celle-ci.

2.2. Présentation de l'architecture SHoP

La plateforme est basée sur un réseau d'Eléments de Calculs (*Processing elements, PE*) identiques et génériques interconnectés à travers un Réseau sur Puce (NoC). La mémoire est physiquement distribuée sur chaque PE, qui possèdent chacun un système d'exploitation (micro-kernel) indépendant. L'ensemble des décisions globales et des communications est réalisé par l'envoi et la réception de messages à travers le NoC. Aucun maître global n'est considéré sur la structure. Un soin particulier a été porté à la compacité de chaque PE, que ce soit au niveau matériel ou logiciel, afin d'augmenter la compacité de la structure et ainsi favoriser le parallélisme. Chaque nœud, appelé Unité de Calcul Réseau (*Network Processing Unit, NPU*), est constitué de deux parties : une partie calcul contenant

notamment le microprocesseur, et une partie communication avec l'interface du NoC et le routeur. Une représentation schématique de cette architecture est donné Figure 6.

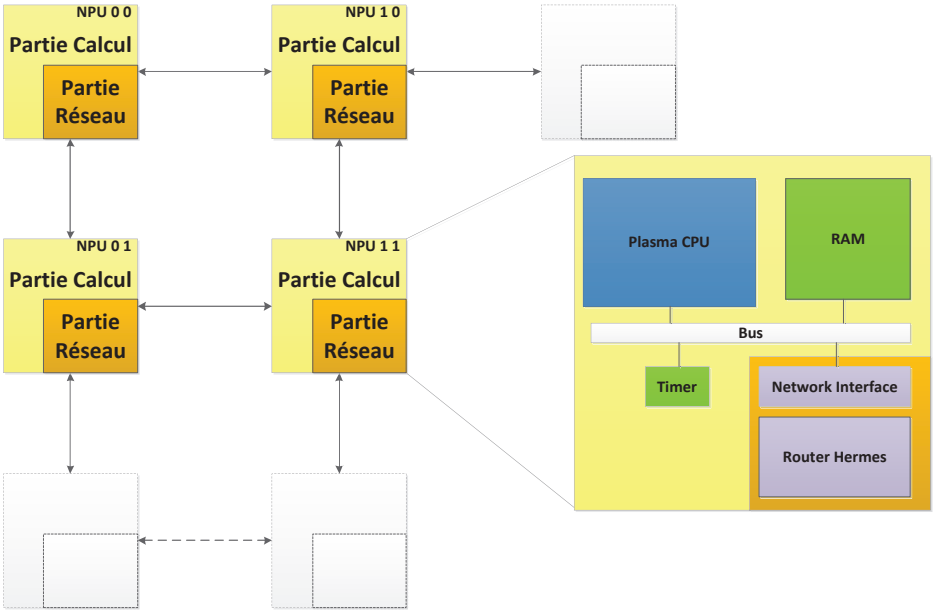


Figure 6 : Architecture matérielle générale

2.2.1. Matériel

Chacun des NPU possède une partie calcul et une partie réseau.

La partie réseau consiste principalement en un routeur 2D, dérivé du projet Hermes [50], utilisant un routage Hamiltonien [51]. Celui-ci propose un échange de paquets de type « trou de vers » : chaque paquet est envoyé de manière monolithique, le trajet emprunté par celui-ci étant bloqué durant la durée du transfert. Le routage Hamiltonien (routage suivant la coordonnée X, puis Y) permet d'obtenir un trajet unique par couple (*NPU_expéditeur, NPU_destinataire*). Grâce à cette prédictibilité, il n'est pas nécessaire d'implémenter un processus de réordonnancement de paquet ou d'accusé de réception.

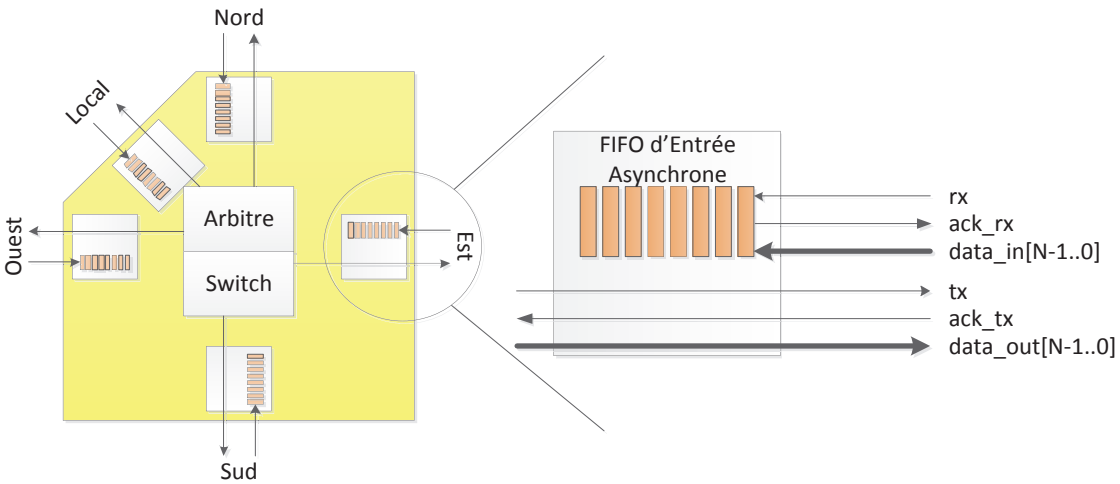


Figure 7 : Architecture du Routeur Hermes

Chaque routeur possède des files (FIFOs) asynchrones d'entrées dans chacune des directions permettant de mettre les paquets en mémoire tampon à l'intérieur du NoC. Ces files possèdent une largeur et une profondeur également paramétrables.

La partie calcul est constituée d'un microprocesseur Plasma [52], d'un timer, et d'une mémoire interne.

Le Plasma est un microprocesseur de type MIPS I 32 bits, disposant de manière configurable de 2 ou 3 étages de pipeline. Il s'agit d'une architecture de type Von Neumann, c'est-à-dire avec un seul accès à la mémoire (mémoire instruction et data unifiée). Il dispose d'une unité arithmétique et logique (*ALU*) classique, d'un multiplieur et d'un registre à décalage. Il est capable d'exécuter tout le jeu d'instruction MIPS I, à l'exception des accès mémoires non-alignés sur 32 bits. Ce microprocesseur, accessible sous licence GPL via le site OpenCores, est accompagné d'un timer, et d'une interface SRAM interconnecté par un bus interne. L'ensemble de ces éléments ont été repris sur SHoP pour réaliser la structure d'un NPU.

2.2.2. Logiciel

L'environnement logiciel de SHoP est basé sur la gestion indépendante de tâches utilisateurs grâce à un petit système d'exploitation de type microkernel. Chaque NPU possède son propre système d'exploitation, qu'il exécute de manière indépendante. Les décisions globales (mouvements de tâches, communication inter-NPU) se font au moyen de l'envoi de messages systèmes à travers le NoC.

Ce système d'exploitation est de type temps réel (RTOS), préemptif, et gérant la priorité des tâches. Celui-ci a été développé à partir du projet de Steve Roads [53]. Ce système est construit autour d'un ordonnanceur faisant tourner les tâches en fonction de leurs priorités, ou par un algorithme de type *Round-Robin* pour des tâches de priorités identiques.

Ce système est capable de réaliser un chargement dynamique de tâche, de gérer des sémaphores et des mutexes, de réaliser de l'allocation dynamique de mémoire (malloc) ou encore de faire communiquer les tâches entre elles.

En complément de ces fonctionnalités, des fonctions d'adaptation avancées telles que la migration de tâche ou la régulation de performance par correcteur PID.

Les librairies standard *libc* ainsi que *math.h* ont aussi été implémentée au sein du système.

2.2.3. Gestion des applications

La représentation des applications sur OpenScale se fait sous forme de graphe de tâches : chaque tâche représente une portion de code capable d'être exécutée en parallèle, une

tâche est représentée par un nœud du graphe. L'ensemble des données traitées par une tâche i venant d'une tâche j se représente par un arc allant du nœud i au nœud j. La Figure 8 représente un graphe de tâche classique.

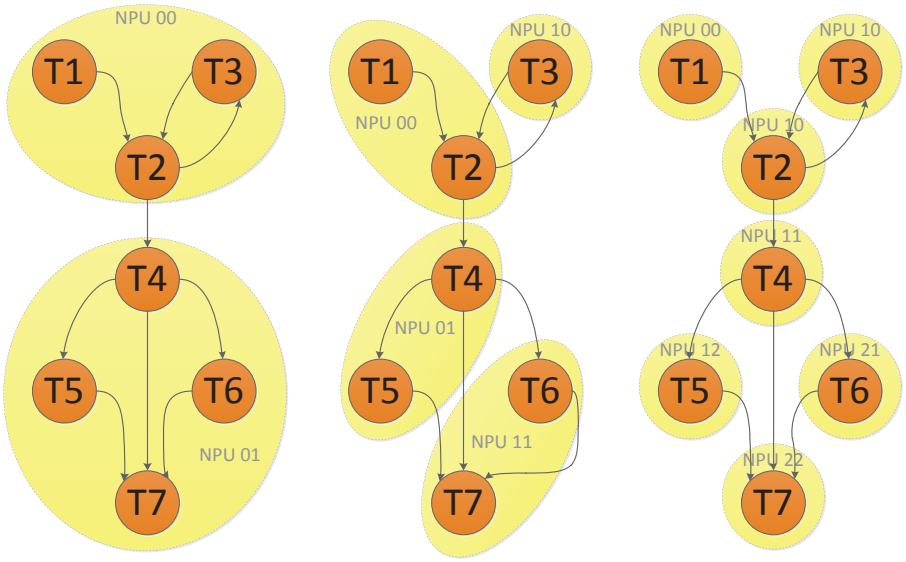


Figure 8 : Représentation en graphe des tâches et exemple de placement

Les tâches peuvent être placées de manière arbitraire au sein de l'architecture, et être déplacées pendant l'exécution pour faire de la répartition de charge. Le développeur possède ainsi deux types de granularité pour implémenter son application : le nombre de tâches et le nombre de NPU sur lesquelles les faire tourner. Les données transitent entre les tâches grâce à une interface de type passage de message. Le système d'exploitation offre deux primitives de communication haut niveau *MPI_Send()* et *MPI_Receive()*, basées sur le standard de communication MPI [17]. Lors d'une communication, la fonction *MPI_Send()*, non bloquante, envoie le paquet vers la tâche de réception. Celle-ci utilise la fonction *MPI_Receive()*, bloquante. Ces deux primitives de communications sont suffisantes pour offrir un support efficace à la majorité des applications orientées flux de données, pour lesquelles un formalisme de type *Khan Process Network* (KPN) est utilisé [54]. Ce formalisme permet notamment d'assurer le non blocage de l'application (de type deadlock) dans le cas d'un envoi non bloquant et d'une réception bloquante.

Ces fonctions de communication offrent un degré d'abstraction au programmateur, qui n'a pas à se soucier de savoir dans quel NPU se trouvent la tâche d'envoi et la tâche de réception. Ainsi, les prototypes de ces fonctions sont donnés ci-après :

MPI_Send(task$_{id}$, channel$_{id}$, data, sizeof(data))

MPI_Receive(channel$_{id}$, data, sizeof(data))

Le système d'exploitation prend ensuite en charge de faire transiter les données entre les bons NPU et les bonnes tâches.

2.3. Limites de la plateforme SHoP

Plusieurs éléments de la plateforme SHoP limitent le développement de fonctionnalités plus avancées. Afin de permettre de mettre en place un système d'adaptation évolué, une gestion de la mémoire approfondie, et une évolution du modèle de programmation, plusieurs améliorations sont nécessaires.

Sur la partie matérielle tout d'abord, la gestion des transactions mémoire a dû être repensée : l'architecture de Von Neumann du Plasma, possédant une mémoire instruction et une mémoire donnée unifiée ne permet pas de gérer séparément les problèmes spécifiques à une de ces mémoires. De plus, l'absence de mémoire cache ne donne pas la possibilité d'effectuer des accès à une mémoire à forte latence avec des performances acceptables. Enfin, l'absence de standardisation, notamment au niveau du bus, ne permet pas l'ajout direct d'éléments venant d'un tiers (IP commerciales ou universitaires, éléments disponibles sur internet, etc.) sur le NPU.

Au niveau logiciel, deux éléments principaux ont été revus. Le premier concerne la gestion des communications : afin de permettre la mise en place d'un système d'adaptation évolué, et le développement de nouvelles fonctionnalités pour faire évoluer le modèle de programmation, l'utilisation d'un système de communication modulaire, permettant plusieurs services différents (accusé de réception, ouverture et fermeture de canaux virtuels, etc.) est nécessaire. Dans le cadre du développement de SHoP, dont l'un des objectifs principaux est de rester un système scalable même avec un nombre de processeurs élevés, l'absence de pile de communication modulaire et efficace restreint sensiblement les possibilités d'évolution.

Le second point logiciel concerne la compacité du système d'exploitation. L'ajout des différentes fonctionnalités (notamment d'adaptation) sur la plateforme SHoP a considérablement augmenté la taille du système d'exploitation. Ainsi, le système d'exploitation de SHoP possèdent une taille pouvant aller jusqu'à 196 Ko, et nécessitant 256 Ko minimum pour fonctionner correctement. Sachant que ce système doit être présent sur l'ensemble des NPU, la taille totale de la mémoire nécessaire au fonctionnement de SHoP s'avère rapidement trop grande pour être supporté par un système réel (on parle ici de mémoire interne à une puce) : si l'on prend par exemple un système composé d'une centaine de processeurs, il est nécessaire d'avoir un minimum de 25 Mo de mémoire interne.

Afin de permettre la mise en place des optimisations détaillées dans cette thèse, la plateforme OpenScale a été mise en place.

2.4. Architecture de la plateforme OpenScale

L'objectif du développement d'OpenScale à partir de SHoP était de créer une plateforme performante et modulaire. L'idée était de pouvoir ainsi se rapprocher des contraintes des systèmes actuels et d'améliorer les possibilités d'évolution de la plateforme. De ce fait, le développement de cette architecture a été réalisé en langage VHDL synthétisable (niveau RTL), et validé sur plateforme FPGA.

2.4.1. Matériel

La grande majorité des changements matériels sur OpenScale concerne la partie calcul de l'architecture. Tout d'abord, le microprocesseur Plasma a été remplacé par le microprocesseur SecretBlaze, développé au sein de l'équipe ADAC [52]. Ensuite, le bus a été modifié pour suivre le standard Wishbone v4 [55]. Sur celui-ci sont également connecté un contrôleur d'interruption, une timer, et une mémoire interne.

Le SecretBlaze est un microprocesseur écrit en VHDL, synthétisable sur FPGA, de type Harvard et open source. Il possède une architecture RISC, à 5 étages de pipeline, avec le jeu d'instructions du processeur industriel MicroBlaze [56]. Ce processeur a été optimisé pour une efficacité maximale sur FPGA, bien qu'il puisse être implémenté sur ASIC. Le développement de ce microprocesseur a été conduit suivant une approche modulaire, permettant ainsi d'avoir de nombreuses possibilités d'optimisation : on peut ainsi optionnellement ajouter un multiplieur, un registre à décalage, un diviseur, ou encore une unité de prédiction de branchement (*Branch Target Predictor, BTC*). De plus, le SecretBlaze possède un cache instructions et un cache données. Ces caches sont à correspondance préétablie (*direct-mapped*), configurables en taille totale, en nombre de lignes, et en politique de remplacement (écriture directe ou différée). Il faut cependant noter que ce processeur est dépourvu d'unité de gestion mémoire (MMU).

Le SecretBlaze est connecté à une mémoire locale, de taille paramétrable, accessible uniquement par le processeur local. Le cache assure une cohérence mémoire avec celle-ci, mais n'assure pas de cohérence avec les mémoires des autres NPU (mémoires privées).

Le contrôleur d'interruption permet de gérer jusqu'à 8 interruptions en parallèle, avec gestion de priorité, masquage et armement.

Le timer et un compteur 32 bits permettant de générer une interruption après une durée paramétrable. Il est également possible d'ajouter optionnellement un contrôleur de port série (UART) pour créer une interface utilisateur.

L'ensemble de ces éléments, ainsi que l'interface réseau sont connectés via un bus utilisant le standard Wishbone v4 [55]. Ce standard est open-source, ce qui a permis de faire une implémentation ouverte de celui-ci.

2.4.2. Logiciel

Les changements au niveau logiciel concernent le système d'exploitation et la gestion des communications.

2.4.2.1. Système d'exploitation

Le système d'exploitation a été entièrement remanié afin tout d'abord d'augmenter la compacité du code, ensuite de le rendre plus modulaire pour rendre optionnelle les fonctions d'adaptations de haut niveau. La Figure 9 représente le schéma du nouveau système d'exploitation.

Le système d'exploitation est maintenant composé de plusieurs catégories bien distinctes : (i) les fonctions systèmes, (ii) les communications, (iii) les drivers et enfin (iv) les fonctions avancées.

Les fonctions systèmes permettent notamment de réaliser un chargement dynamique de tâches au sein du système d'exploitation à n'importe quel moment de l'exécution, et ce malgré le manque de gestionnaire mémoire matériel (MMU). Un gestionnaire mémoire logiciel permet néanmoins de gérer la pile et le tas de chaque tâche. Un gestionnaire d'interruption permet également de relancer l'ordonnanceur après chaque interruption du timer. Le système d'exploitation propose aussi l'utilisation de sémaphores, de mutexes, et d'allocation mémoire dynamique (malloc).

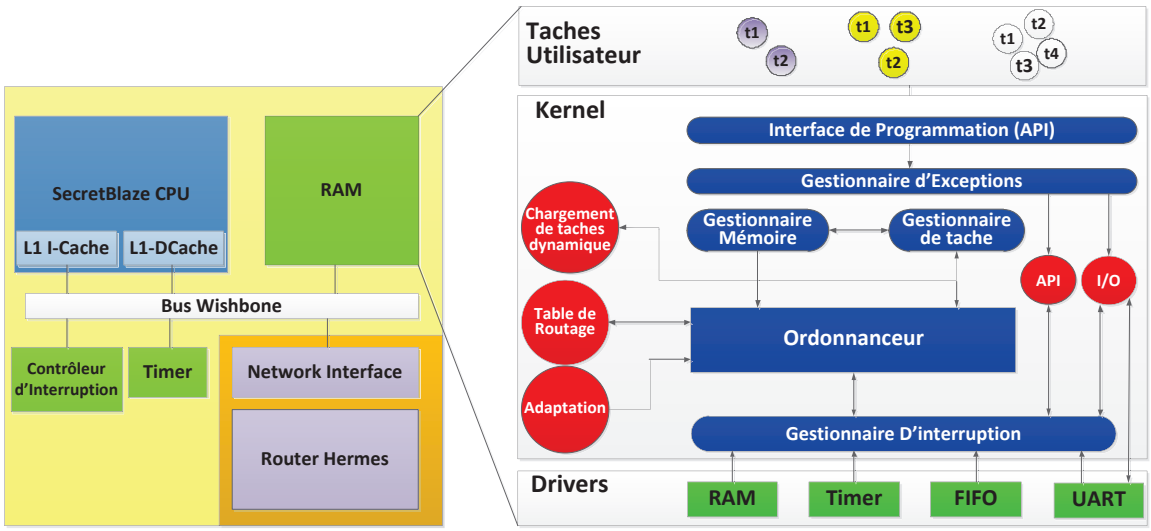


Figure 9 : Structure générale du système d'exploitation

Les communications entre tâches sont réalisées par l'intermédiaire de files (FIFOs) logicielles. Pour le cas de communications inter-NPU, une pile de communication de type TCP-IP optimisée a été réalisée.

L'interface réseau, le timer et l'UART possèdent des drivers logiciels. Une interface de programmation (API) permet aux tâches d'accéder à ces ressources. Ainsi, lorsqu'une tâche veut effectuer un accès système (printf par exemple), une exception est levée générant l'appel au driver correspondant.

Les fonctions d'adaptation avancées telles que la migration de tâche, la régulation de performance par correcteur PID ou encore celles développées dans cette thèse peuvent être rajoutée de manière modulaire.

Le système d'exploitation ainsi remanié possède une taille comprise entre 58 Ko et 96 Ko en fonction des modules d'adaptations ajoutés. Un système possédant 128 Ko de mémoire est capable de faire tourner celui-ci.

2.4.2.2. Gestion des communications

L'envoi des données d'une tâche à l'autre peut être effectué suivant deux cas particuliers : (i) les deux tâches communicantes sont sur le même NPU, et (ii) les deux tâches sont sur des NPU différents. La Figure 10 illustre ce protocole d'envoi, en fonction de ces deux cas.

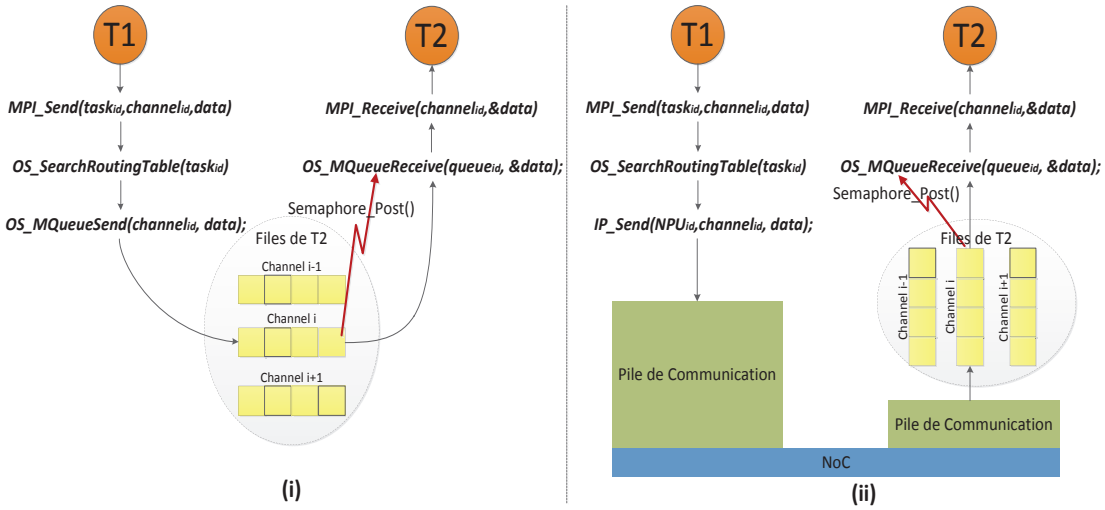


Figure 10: Protocole de communication entre tâches : (i) situées sur le même NPU, (ii) située sur deux NPU différents

Pour chaque tâche, le nombre de canaux entrants est défini à la création des tâches. Lors de l'envoi de paquets une fonction de recherche de l'emplacement de la tâche de réception est effectuée (*OS_SearchRoutingTable()*). Si les tâches communicantes sont situées sur le même NPU, les données sont directement envoyées à la file logicielle correspondante. Cet envoi provoque le réveil de la fonction de réception (par le biais d'un sémaphore), qui va ensuite fournir la donnée à la tâche de réception. Dans le cas où les tâches ne sont pas situées sur le même NPU, la fonction *MPI_Send()* va utiliser la pile de

communication du système d'exploitation pour transmettre les données via le réseau sur puce. A la réception, les données sont envoyées à la file logicielle correspondante, provoquant la même réaction du système que dans le cas précédent.

2.4.3. Exploration de l'espace de conception

La modularité est le maître mot de la plateforme OpenScale. Le but est de créer une plateforme capable de s'adapter aux ressources et aux besoins de l'utilisateur. Il est bien sûr possible de gérer le nombre de NPU, mais également les éléments composant chaque NPU : la quantité de mémoire, la taille des caches, la présence d'accélérateurs matériels au sein du processeur (multiplieur matériel, barreau de décalage, diviseur, unité de prédiction de branchement), ou encore la taille des canaux de transmission du réseau sur puce. L'objectif de cette section est d'évaluer les performances de la plateforme en fonction des différents éléments modifiables et des objectifs visées.

2.4.3.1. Calculs au sein d'un NPU

Afin d'évaluer les performances en calcul brut d'un NPU, plusieurs applications typiques de la programmation embarquée ont été exécutées :

- *Advanced Encryption Standard Coder* (*AES*): application de codage cryptographique utilisant le standard de chiffrement avancée [57]. Ce type d'application est communément utilisé lors d'un besoin de transmission sécurisée.
- *Transformée en cosinus discrète classique* (*DCT*): application de transformation de données par projection sur cosinus [58]. Ce type d'algorithme est très utilisé en traitement du signal.
- *Transformée en cosinus discrète par algorithme de Loeffler* (*Loeffler-DCT*) : il s'agit de la même application mais utilisant l'algorithme de Loeffler [59], qui optimise le nombre d'opérations de multiplication.
- *Décoder vidéo JPEG* (*MJPEG*) : application de décodage d'un flux vidéo couramment utilisé. Cet algorithme peut être parallélisée, mais nous utiliserons ici une version mono tâche de cette application. L'évaluation du temps de calcul se fera sur la base d'une image traitée.
- Application synthétique *Dhrystone* : il s'agit d'un programme de test de performance sur du calcul entier [60].

La première exploration consiste à faire varier le nombre d'accélérateurs matériel au sein du processeur. La Figure 11 donne le temps d'exécution de chacune des applications citées précédemment, ainsi que le nombre de LUTs et de registres utilisés pour synthétiser le processeur dans chacun des cas suivants : (i) le processeur sans aucun accélérateur matériel, (ii) le processeur avec multiplieur et diviseur matériel, (iii) le processeur avec multiplieur, diviseur et barreau de décalage, et enfin (iv) le processeur avec multiplieur, diviseur, registre à décalage et unité de prédiction de branchement (BTC).

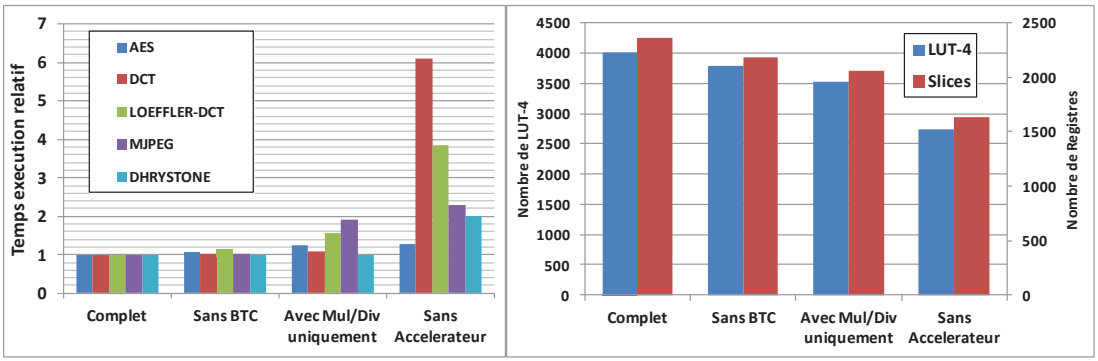


Figure 11 : Performance du SecretBlaze

La première remarque que l'on peut faire est que la variation du temps d'exécution suivant le nombre d'accélérateurs est très dépendante du type d'application : ainsi, les algorithmes de DCT, réalisant un grand nombre d'opérations de multiplication, ont un temps d'exécution jusqu'à 6 fois plus long dans le cas d'un processeur sans accélérateur. A contrario, l'AES, algorithme de cryptographie basée sur des masquages par fonction OU EXCLUSIF (XOR), ne se retrouve que peu impacté par la présence des accélérateurs (ralentissement de l'ordre de 1,2). On peut également noter que la présence de l'unité de prédiction de branchement (BTC) n'influence que très faiblement le temps d'exécution de ces applications. En ce qui concerne la taille du CPU, passer d'un SecretBlaze complet à un SecretBlaze sans accélérateur permet un gain de l'ordre de 45% en taille.

Dans notre cas, la taille de ce processeur, même avec tous ses accélérateurs, reste très faible face aux capacités des FPGA utilisés (30% de registres occupés pour une spartan3s1000, et seulement 5% pour une virtex 5VLX110T). C'est pourquoi l'ensemble des tests réalisés par la suite a été fait par défaut avec des processeurs complets, ce qui permet d'avoir les meilleurs résultats possibles en termes de performance.

2.4.3.2. Performances du Réseau sur Puce

Le Réseau sur Puce utilisé sur OpenScale est tiré du projet Hermes : il s'agit d'un réseau de type grille à deux dimensions. Ce Réseau sur Puce possède une certaine modularité, notamment concernant la taille des canaux de communication, ou encore la taille des files (FIFOs) en entrée de chacun des ports (nord, sud, est, ouest et local).

Le Tableau 2 présente le temps de transfert (en nombre de cycles) d'un paquet d'1Ko de données à travers le NoC, pour des routeurs distants de un, deux ou trois nœuds. La taille des files dans ces routeurs n'influence pas ce temps de transfert. L'ensemble de ces mesures a été réalisé en prenant en compte la latence de la pile de communication du système d'exploitation. Il est à noter qu'il s'agit de l'évaluation de la latence d'un paquet (temps de transfert d'un paquet), et non pas du débit du réseau (quantité de données transitant par unité de temps).

distance / canaux	1 nœud	2 nœuds	3 nœuds
4 bits	16670	16688	16706
8 bits	8465	8473	8481
16 bits	6157	6161	6165
32 bits	6155	6157	6159

Tableau 2 : Temps de transfert d'un paquet d'1Ko à travers le NoC

On remarque que le temps de transfert varie peu en fonction du nombre de nœuds traversés (moins d'1%), par contre, il peut varier substantiellement en fonction de la taille des canaux de communication. En effet, un canal deux fois plus petit va demander l'envoi du double de paquets à travers le NoC pour transférer une même quantité de données. Cependant, pour des canaux de 16 ou 32 bits, le temps est significativement le même. Ceci peut s'expliquer par le fait que le temps de traitement d'un mot de 32 bits à travers le NoC devient, pour ces tailles de canaux, prépondérant face au temps d'envoi ou de réception. En effet, quel que soit la taille des canaux de communication, le routeur Hermes traite les données par mots de 32 bits, afin d'être compatible avec la largeur du bus Wishbone. Ainsi, le routage de deux paquets de 16 bits va prendre environ le même temps que le routage d'un paquet de 32 bits.

2.5. Conclusion

La plate-forme OpenScale est le fruit de plusieurs années de recherche et de développement dans l'objectif de former une plate-forme innovante et performante. Les contraintes de scalabilité et de réutilisabilité ont entrainé la conception de l'architecture SHoP : une structure de type réseau de nœuds de calculs, connectés au moyen d'un NoC. Ce chapitre a montré le développement d'OpenScale, une plateforme inspiré de SHoP dont la modularité lui permet d'obtenir des performances collant au mieux au cahier des charges : que ce soit en faisant varier le nombre de NPU, la complexité de la partie calcul, ou encore la configuration de la partie réseau. Ces efforts d'innovation et de développement ont produit une architecture aujourd'hui mature et disponible sous licence GPL à l'adresse suivante : www.lirmm.fr/ADAC

OpenScale offre également, de par sa structure, de nombreuses opportunités d'améliorations. Elle permet tout d'abord de considérer de nouveaux mécanismes d'adaptation à son environnement : en fonction des applications qui lui sont demandés et des contraintes qui lui sont imposées (puissance consommée, température), celle-ci peut modifier, en temps réel, sa manière d'effectuer ses calculs. En effet, du fait de son

architecture multicœur homogène à mémoire distribuée, il existe plusieurs degrés d'actions supplémentaires pour optimiser ses performances : placement des données, répartition de l'exécution entre les NPU, variation de la fréquence de chaque NPU de manière indépendante, etc. Ces différents mécanismes seront abordés dans le chapitre 3.

Elle permet également de repenser le modèle de programmation afin de prendre en compte la particularité du système : le modèle de programmation de type KPN, présenté dans ce chapitre, ne permet pas de prendre en compte les spécificités du système, et restreint le nombre de programmes exécutable sur la plate-forme. La création dynamique d'une tâche, par exemple, n'est pas disponible avec un tel modèle. Ces considérations seront abordées dans le chapitre 4, où d'autres modèles de programmation plus évolués, et utilisés communément dans le commerce seront décrits, ainsi que la manière dont ils ont été portés sur OpenScale.

Chapitre 3 : Exploration de stratégies d'adaptation

3.1. Introduction

Comme introduit précédemment, la motivation première du développement des plateformes SHoP et OpenScale est l'exploitation des potentiels d'une architecture scalable pour des applications multitâches. De nombreux éléments ont d'ailleurs déjà été développés au cours des ces dernières années sur ce type d'architecture, confirmant leur potentiel. On peut notamment citer la répartition dynamique de tâches, la migration de tâche ou encore la gestion indépendante de la fréquence de chaque NPU en fonction de la charge.

3.1.1. Répartition dynamique de tâche

La répartition dynamique de tâches consiste à placer les tâches sur la plateforme de manière optimale, en fonction des contraintes imposées par l'utilisateur (contrainte de performance, de consommation, ou encore respect d'échéances pour une application temps réel) [61]. De nombreux algorithmes de répartition de tâches ont été proposés au cours de ces dernières années [40]: celui du premier nœud vide (*First Empty*) qui consiste à

mettre les tâches sur la plateforme en prenant le premier nœud vide, ou le premier nœud le moins rempli ; celui du plus proche voisin (*Nearest Neighbour*), qui consiste à prendre le nœud le plus proche du dernier nœud affecté. D'autres heuristiques plus complexes ont aussi été réalisées : dans [62], un administrateur global va sélectionner un ensemble de cœurs pour y affecter une application, alors que dans [63], une heuristique basée sur des forces d'attraction (nœud vide) et de répulsion (nœud plein) est utilisée pour répartir les tâches. L'avantage de cette dernière approche est que la gestion de la répartition des tâches peut se faire de manière locale, se qui permet de conserver la scalabilité du système.

Cette répartition dynamique des tâches permet d'obtenir une distribution des charges au sein d'une plateforme multiprocesseur adaptée à la situation, et donc procure ainsi de meilleures performances qu'une répartition statique. Cependant, dans la plupart des applications, la charge de chaque tâche peut varier au fil du temps [64]. Dans de nombreux cas, ces variations ne peuvent être prédites à l'avance, notamment lors d'interactions avec des évènements asynchrones extérieurs non prédictibles (interface utilisateur par exemple) [65]. Dans ces cas, une répartition dynamique de tâches peut s'avérer sous-optimale pour réaliser un placement efficace des tâches. On peut alors utiliser des mécanismes de migration de tâches pour répartir en temps réel la charge du système.

3.1.2. Migration de tâches

De nombreux mécanismes de migration de tâches sont proposés dans la littérature. Cependant, la grande majorité de ces algorithmes, s'appuie sur une architecture à mémoire partagée. Dans [66] par exemple, les auteurs proposent un protocole de migration de tâches au sein d'une architecture basée sur NoC, mais possédant une mémoire partagée. D'autres travaux [67][68][69] se basent sur l'augmentation de la librairie MPI (*Message Passing Interface*) pour fournir un support au processus de migration de tâches.

Dans [70], les auteurs proposent une stratégie de migration qui exploite la mesure de la température de chaque cœur, ainsi que de la charge processeur, pour des applications de type flux de données. Dans leur approche, chaque nœud de calcul possède une réplique de toutes les tâches en mémoire. Lors d'une migration, le nœud source stoppe l'exécution de la tâche à migrer, et le nœud destination commence l'exécution de sa propre réplique. Cette technique est également décrite dans [71]. Ce mécanisme, bien que très performant, nécessite une quantité très importante de mémoire, à cause de la nécessité de réplication des tâches sur chaque nœud.

Dans [72], les avantages et inconvénients de nombreux mécanismes de migration de tâche sont abordés, ainsi que leurs performances associées.

Au sein de l'équipe ADAC, un protocole de migration de tache au sein d'une architecture distribuée, a été développé [40]. Celui-ci est basé sur le transfert du code de l'application par passage de message. Il sera décrit plus en détail dans la section 3.3.1.

3.1.3. Autres stratégies d'adaptation

D'autres méthodes de répartition des charges au sein d'une architecture multicœur non basée sur le positionnement des tâches existent. On peut notamment citer la variation de fréquence ou de tension d'un cœur indépendamment des autres. Cette technique est appelée *DVFS* (*Dynamic Voltage and Frequency Scaling*).

La plupart des mécanismes de DVFS utilisés actuellement sont définis lors de la conception du système, et sont généralement basés sur des schémas de profils standards prédéfinis [73][74].

Puschini et al. [75] propose une stratégie de modification de la fréquence basée sur l'utilisation de la théorie des jeux permettant de répondre à plusieurs objectifs à la fois : température locale, contrainte de délais, etc. D'autres stratégies utilisent des contrôleurs Proportionnel-Intégral-Dérivé (PID) pour modifier la fréquence ou la tension d'un processeur [76][77][78], ou encore du Réseau sur Puce (NoC) [79][80].

De nombreuses méthodes ont également pour objectif la diminution de la puissance consommée : dans [81] les auteurs proposent une stratégie de minimisation de la puissance consommée sur des applications de type flux de données possédant des contraintes de flux minimum ou de délais.

De nombreuses autres méthodes ont encore été proposées, comme par exemple l'adaptation en fréquence par suppression de périodes d'horloge [82], ou encore l'instanciation dynamique de cœurs dans une architecture reconfigurable (FPGA) [83]. Cependant, ces méthodes nécessitent la mise en place de modules spécifiques dépendants de la technologie et de l'architecture du système.

L'ensemble des ces méthodes, souvent prise en compte par notre équipe, permettent d'exploiter les ressources d'une architecture à mémoire distribuée. Cependant, de nombreuses optimisations restent encore à faire. Ce chapitre explore les différentes optimisations effectuées au sein de cette thèse concernant ces mécanismes de répartition de charge. On développera dans un premier temps la gestion adaptative des communications réalisée au sein d'OpenScale, qui a permis de réaliser plusieurs protocoles de migration de tâche, notamment un protocole de migration avec redirection de données, qui sera développé dans une deuxième partie. Enfin, une autre méthode de répartition des charges, appelée exécution distante, sera étudiée et comparée aux précédentes.

3.2. Gestion des communications

La gestion des communications au sein d'une architecture distribuée comme OpenScale est cruciale. En effet, l'ensemble des communications, échanges de données, synchronisation entre NPU, prise de décisions globales, va se faire par l'intermédiaire de passages de messages. Une pile de communication a donc été réalisée afin de pourvoir aux besoins du système d'exploitation et des applications en communication.

3.2.1. Une pile de communication adaptative

La variété des besoins en communication de la plateforme a entrainé la création d'une pile de communication adaptative, c'est-à-dire utilisant un protocole différent suivant les nécessités. On peut distinguer trois grands types de besoin en communication sur cette plateforme :

- L'envoi/réception de données au sein d'une application : le but est d'obtenir une bande passante la plus large possible afin que les communications ne soient pas le facteur restrictif concernant les performances de l'application.
- Les messages de contrôle ne nécessitant pas de précédence, c'est-à-dire ne nécessitant pas la confirmation de la prise en compte de ce message. Les messages d'ouverture de canaux de communication ou encore de déclenchement de migration font notamment partie de cette catégorie.
- Les messages de contrôle nécessitant une précédence, comme par exemple les messages de modification de l'emplacement d'une tâche avant migration. En effet, lors d'une migration, il est nécessaire d'être certain que l'ensemble des tâches communicantes avec la tâche migrante soient informées de cette migration avant d'effectuer celle-ci.

De fait, trois protocoles ont été mis en place : un protocole nommé RAW, composé d'un en-tête minimal, pour le transfert de données rapide, un protocole nommé UDP-lite, inspiré du protocole UDP permettant l'échange de messages systèmes sans confirmation d'arrivée, et un protocole nommé TCP-lite, inspiré du protocole TCP, permettant d'effectuer des envois de messages systèmes nécessitant une confirmation de l'arrivée du message.

Le schéma de la pile de communication est inspiré de la suite de protocole internet (TCP/IP). Elle est constituée de 4 couches : la couche liaison, la couche réseau, la couche transport et enfin la couche application. La Figure 12 présente les trois protocoles de communication.

On remarque que deux couches applications distinctes ont été réalisées : celle concernant le protocole RAW, uniquement utilisable par les applications au travers de la librairie MPI, et celle concernant les protocoles UDP-lite et TCP-lite, utilisée par les services systèmes.

Les protocoles UDP-lite et TCP-lite sont inspirés des protocoles UDP/IP et TCP/IP de la suite de protocole internet : ils possèdent donc les mêmes fonctionnalités. Ainsi, UDP ajoute la notion de connecteur réseau (socket) dans sa couche transport, lié à un port particulier. Le protocole TCP quant à lui inclut également la notion d'ordonnancement et d'acquittement.

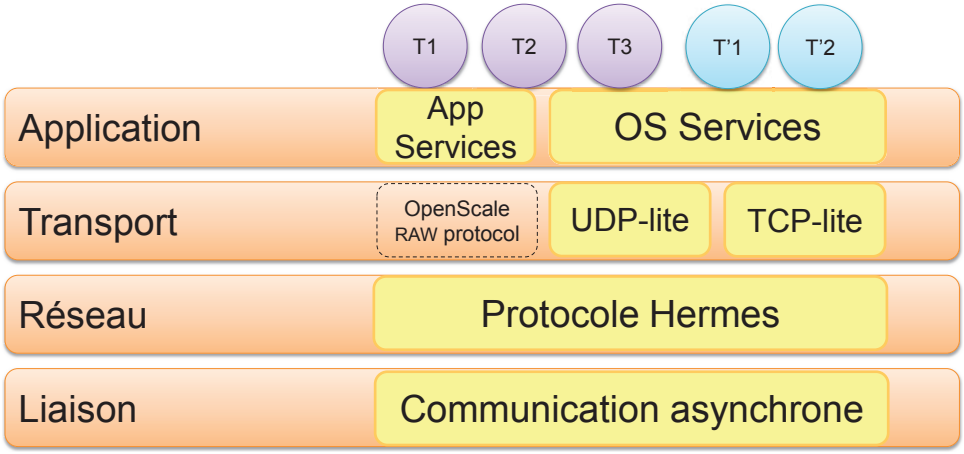


Figure 12: Pile de communication d'OpenScale

3.2.2. Performances

La pile de communication de la plateforme OpenScale devait être la plus compacte possible : en effet, chaque NPU possédant sa propre pile de communication, celle-ci ne devait pas encombrer la mémoire locale des NPU plus que nécessaire. Au final, la pile de communication ne dépasse pas les 10 kilo-octets. Il faut cependant remarquer que la majorité de l'espace utilisé par cette pile de communication (6Ko) concerne la couche application, à cause des très nombreux services procurés par la plateforme (chargement dynamique de tâches, migration, établissement de liens de communication entre tâches de manière dynamique, etc.).

Les tests de performance en communication ont été effectués en considérant plusieurs tailles de paquet, et plusieurs quantités de données. La Figure 13 donne le débit mesuré pour chacun des protocoles, en fonction de la taille des paquets et de la quantité de données transmises.

Le premier point à souligner sur cette figure concerne les valeurs de ces débits. Bien que le débit théorique d'un lien matériel du NoC puisse atteindre 250 Mo/s, on est ici limité à un débit maximum de 6 Mo/s. Cette limitation est dû à l'encapsulation/décapsulation du paquet, qui induit une double pénalité : tout d'abord par l'envoi de données supplémentaires (en-tête), mais aussi par le temps de calcul nécessaire pour effectuer l'encapsulation et la décapsulation.

Le second point que l'on peut noter sur cette figure concerne les variations de débit des différents protocoles. En effet, si la variation de débit est de l'ordre de 6% pour le protocole RAW, elle peut atteindre les 45% pour les protocoles UDP-lite et TCP-lite. Les

débits les plus faibles sont obtenus pour des petits paquets (100 à 200 octets). Ces différences sont liées à deux facteurs distincts. Tout d'abord, l'encapsulation des paquets provoque une baisse considérable du débit effectif pour des petits paquets : l'en tête UDP-lite étant de l'ordre de 20 octets, quant à celui de TCP-lite de l'ordre de 30 octets. On a donc pour des paquets de 100 octets l'envoi de paquets 20 à 30% plus gros. Le deuxième facteur concerne le temps de calcul de la pile de communication : pour chaque paquet, l'envoi et la réception entrainent l'exécution de l'ensemble des procédures d'encapsulation et de contrôle de chaque protocole. Ainsi, plus les données seront morcelées, plus le débit va faiblir. On peut noter cependant qu'après une taille optimale de paquet (environ 16 Ko), le débit faiblit à nouveau. Ceci est dû au fait que le temps des procédures d'allocation dynamique d'espace pour les données (malloc) devient prépondérant sur l'ensemble du temps de transfert.

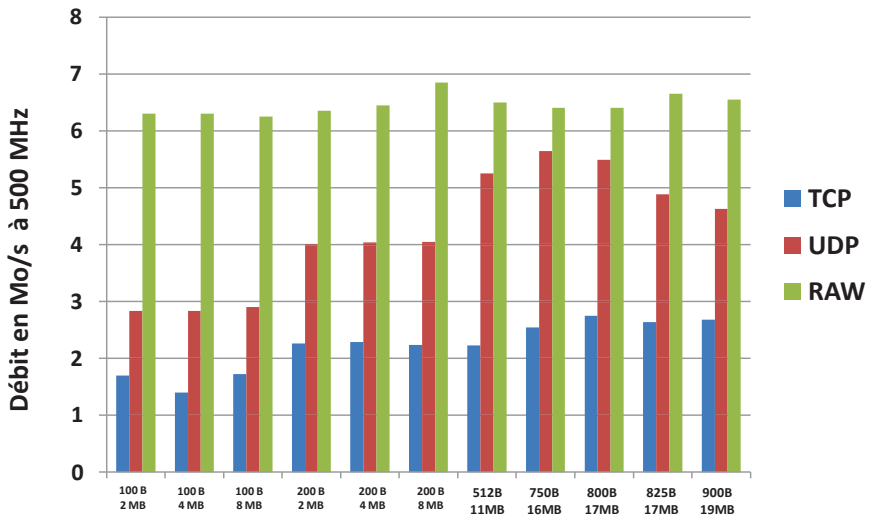


Figure 13 : Débit des différents protocoles en fonction de la taille des paquets

3.3. Optimisation de migration de tâche au sein d'une application à flux de données

L'un des objectifs principaux d'une architecture multicœurs telle qu'OpenScale concerne la possibilité de faire un équilibrage de charge de calcul au sein de l'architecture, à la fois efficace et scalable. Le premier mécanisme mis en place pour réaliser cette fonction fut la migration dynamique de tâche : celle-ci consiste à déplacer les données (mémoire instructions et mémoire données) d'un NPU vers un autre. Les travaux de recherche menés sur ce mécanisme, réalisé au sein de l'équipe, sont décrits en détail dans [40]. Ceux-ci seront brièvement rappelés dans un premier temps, pour ensuite exposer une optimisation de cette migration de tâche classique pour les applications à flux de données.

3.3.1. Migration de tâche classique

Afin de conserver la possibilité de mise à l'échelle de l'architecture, la migration de tâche se devait d'utiliser des mécanismes distribués. Ainsi, chaque NPU peut indépendamment prendre la décision de migrer l'une de ses tâches, suivant différentes heuristiques (charge du CPU, nombre d'éléments dans les files de communication, demande d'un NPU voisin…). Le mécanisme de migration est décrit Figure 14, en prenant un exemple particulier : dans cet exemple, une application est constituée de deux tâches (T_1 et T_2), T_1 communiquant avec T_2 (étape *i*). Pour des raisons de clarté, ce scénario ne fait mention d'aucune tâche en amont de T_1 ou en aval de T_2, car celle-ci n'influence pas le mécanisme de migration décrit. A un certain moment et pour des raisons dictées par une des heuristiques précédemment citées (non développées au sein de cette thèse), le NPU_{33}, possédant la tâche T_2, décide de faire migrer celle-ci vers le NPU_{31}. Le NPU_{33} va donc envoyer une demande de fermeture de connexion à l'ensemble des NPU possédant T_2 comme tâche avale (ici uniquement T_1), ainsi qu'au NPU principal du cluster, le NPU_{00} (étape *ii*).

Il est important de noter que le NPU_{00} est considéré comme principal seulement dans le sens où il va répertorier la position de l'ensemble des tâches de l'application donnée. Celui-ci est notamment utilisé lors de la création de canaux de communication entre deux tâches de l'application (création dynamique, ou au démarrage de l'application).

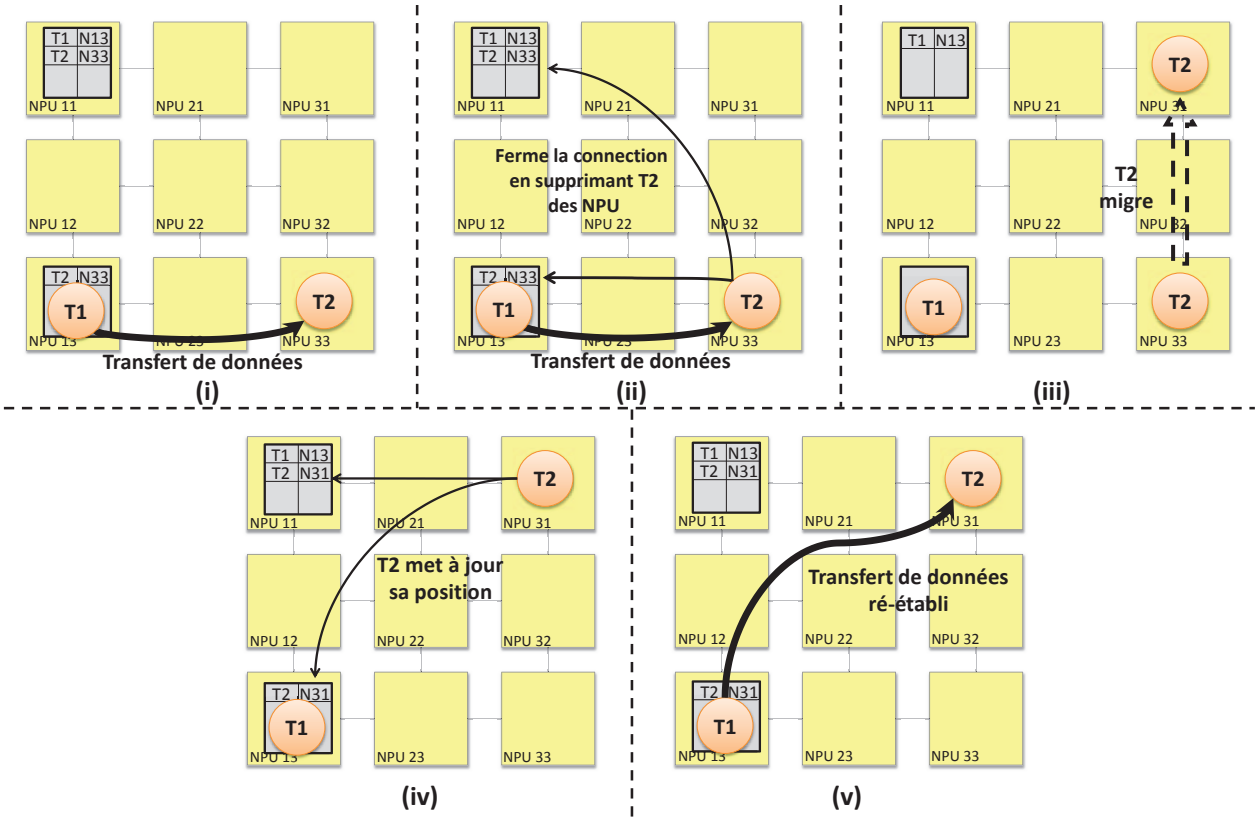


Figure 14: Protocole classique de migration de tache

Lorsque la connexion est finalement close et que la tâche T_2 a fini de traiter l'ensemble des paquets venant de T_1, celle-ci migre vers son NPU de destination, le NPU_{31} (étape *iii*).

Il est à noter que l'absence de MMU sur le système ne permet pas de réaliser une sauvegarde et une restauration du contexte d'exécution de la tâche migrante. Cependant, ce facteur est rarement limitant dans les applications à flux de données.

Après le transfert du code de la tâche T_2, le NPU_{31} envoie des messages de ré-établissement des canaux de communication, en mettant à jour sa position (étape *iv*). Finalement, T_1 se remet à communiquer avec T_2 (étape *v*).

Cette approche, contrairement à d'autres [71], permet d'effectuer une migration de tâche sans pertes de données. En effet, l'envoi de messages pour la fermeture des canaux de transfert, s'effectue grâce au protocole TCP-lite, ce qui assure le bon ordre d'exécution des étapes *ii* et *iii*. Ainsi, la fermeture des canaux est bien effective avant le transfert du code.

3.3.2. Migration avec renvoi de données

La technique de migration de tâche « classique » décrite dans le chapitre précédent possède un inconvénient : la fermeture des canaux de données des tâches amont. En effet, ce protocole de fermeture demande le traitement de l'ensemble des données présentes dans les files (FIFOs) logicielles et matérielles entrantes de la tâche migrante (afin de ne pas perdre de données). Cette étape est souvent la plus coûteuse en temps, car le NPU déclenchant la migration est souvent surchargé (puisqu'il déclenche une migration). De plus, durant cette étape, l'ensemble des tâches amont sont stoppées, diminuant encore d'avantage les performances.

Pour pallier à ce problème, un autre protocole de migration a été mis en place : la migration avec renvoi de données. Ce mécanisme est décrit Figure 15.

La situation initiale est la même que dans le cas précédent : le NPU_{33} déclenche le processus de migration de la tâche T_2 vers le NPU_{31} (étape *i*). Dès lors, une nouvelle tâche système est créée, qui va intercepter les données venant des tâches amont de la tâche T_2. La tâche T_2 peut donc effectuer sa migration directement, sans attendre la coupure des canaux de communication (étape *ii*).

Lorsque le code de la tâche T_2 est arrivé sur le NPU_{31}, celui-ci fait la demande d'envoi des paquets stockés au NPU_{33} (étape *iii*). Dans un même temps, il envoie des messages de ré-établissement des canaux de communication, en mettant à jour sa position comme dans le protocole précédant. Les paquets stockés sont donc redirigés vers la nouvelle destination de T_2, et traité en priorité avant les nouveaux paquets envoyés par T_1 (étape *iv*). Lorsque l'ensemble des paquets stockés a été redirigés, les données envoyées par T_1 sont traitées (étape *v*).

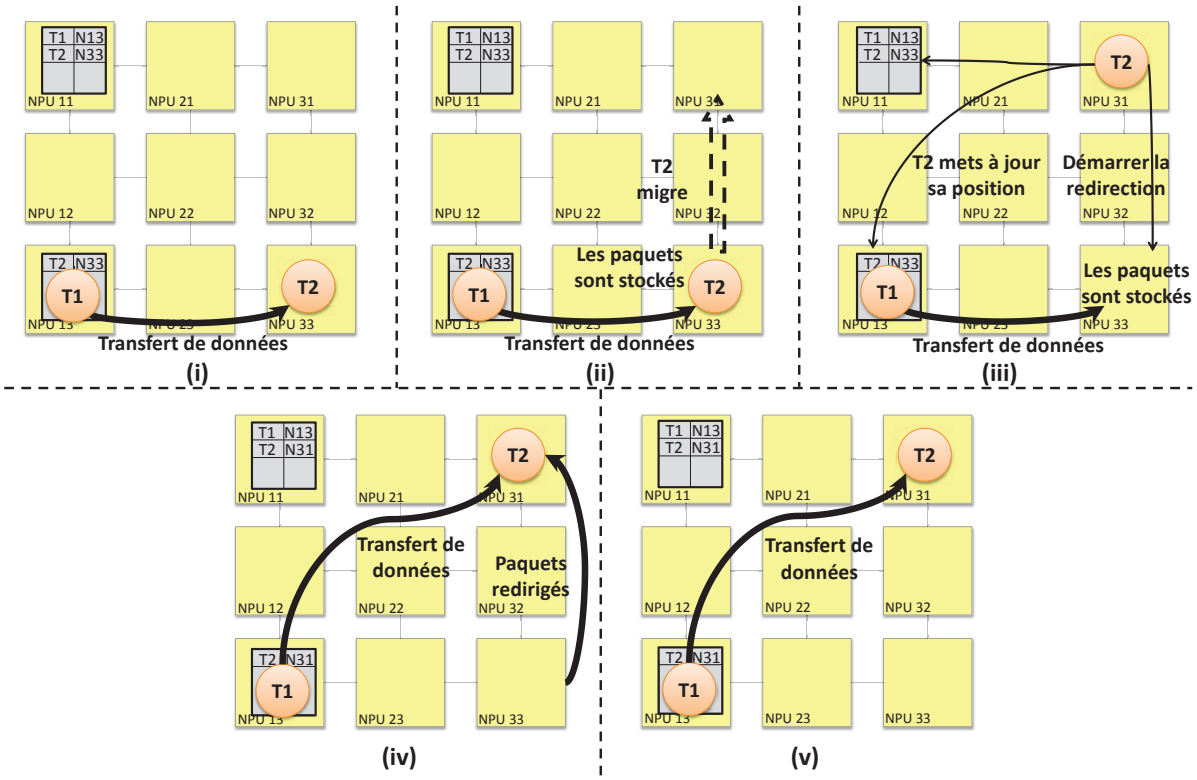


Figure 15 : Protocole de migration de tâche avec renvoi de données

Cette méthode entraine notamment un doublement ponctuel du débit de données vers la tâche T_2 (durant l'étape *iv*). Cela entraine une accumulation des données dans les files d'entrées de la tâche T_2 après migration. Cependant, si l'on se réfère au nombre total de données transmises, aucune donnée redondante n'est envoyée. De plus, cette accumulation de données est souvent vite absorbée par la tâche T_2 qui, si l'heuristique de migration est efficace, se retrouve plus performante sur le NPU_{31}.

3.3.3. Validations expérimentales

3.3.3.1. Cas d'étude

Afin d'évaluer les performances de ces deux protocoles, une application de décodage MJPEG (Motion JPEG) a été utilisée. Il s'agit d'une application caractéristique de traitement de flux de données. Ce type d'application est particulièrement adapté à la plateforme OpenScale, car elle doit faire l'objet d'une attention particulière concernant les mécanismes de répartition de charge tel que la migration de tâche. En effet, comme celles-ci possèdent des contraintes d'échéances d'une utilisation temps réel (débit minimum d'un flux audio ou vidéo par exemple), il est nécessaire d'évaluer l'impact d'un processus de répartition de charge, comme ici la migration de tâche, sur le flux de données de l'application.

Le graphe des tâches de l'application multitâche MJPEG utilisée ici est donnée Figure 16. Afin d'exposer des résultats les plus représentatifs possibles, il a été nécessaire d'évaluer en détail cette application. Le logiciel Valgrind [84] a permis de faire un profilage de

l'application, et ainsi d'évaluer la quantité de temps nécessaire à chaque tâche pour traiter un paquet par rapport au temps total de traitement d'un paquet. Cette indication est donnée en blanc sur la Figure 16. On remarque ainsi que la tâche ivlc est la plus coûteuse en temps de calcul, avec plus de 85% du temps.

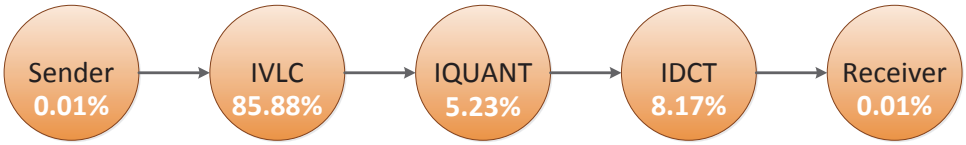
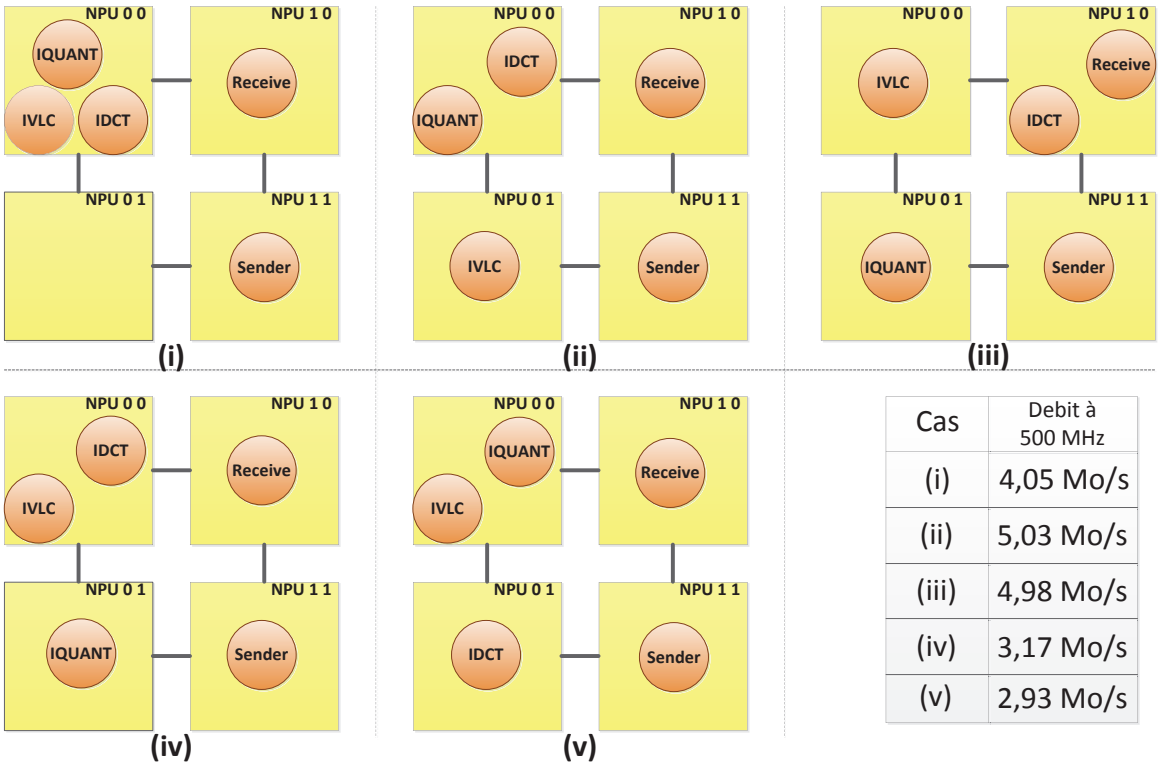


Figure 16: Graphe des tâches de l'application MJPEG

La Figure 17 donne quelques exemples de placement (statique) des tâches de cette application ainsi que le débit maximum atteint par l'application dans chacun de ces cas. Ces expériences ont été faites avec un réseau de 4 NPU, dans leur configuration standard (voir section 2.4.3.2).



Cas	Debit à 500 MHz
(i)	4,05 Mo/s
(ii)	5,03 Mo/s
(iii)	4,98 Mo/s
(iv)	3,17 Mo/s
(v)	2,93 Mo/s

Figure 17: Exemples de placement de l'application MJPEG et débits correspondants

On distingue clairement sur cette figure l'importance du placement de la tâche critique (ivlc). Il est en effet logique que le placement de cette tâche seule sur un NPU augmente le débit. Cependant, le détail exhaustif de l'ensemble des possibilités de placement des tâches au sein d'OpenScale n'est souvent pas faisable. En effet, le nombre de possibilités de placement est de N^T où N est le nombre de tâches à placer, et T le nombre de NPU. Dans le cas simplifié de notre application sur un réseau de 4 NPU, 256 possibilités serait déjà à prendre en compte. Dans un cas réel de plusieurs applications embarquées sur un

réseau de plusieurs centaines de NPU, ce chiffre dépasse facilement les milliards de possibilités.

3.3.3.2. Comparaison des deux protocoles

Afin d'effectuer une comparaison pertinente des deux protocoles de migration décrits précédemment, le scénario suivant a été réalisé : le placement initial des tâches est effectués suivant la configuration (*i*) de la Figure 17. Dans cette configuration, le NPU_{00} est surchargé : il va donc décider de migrer la tâche IVLC vers le NPU_{01}, pour se placer dans la configuration (*ii*). Cette opération est illustrée Figure 18.

Le débit de l'application a été mesuré en fonction du temps pour les deux protocoles de migration. Les résultats de ces mesures lors de la période de migration sont donnés Figure 19.

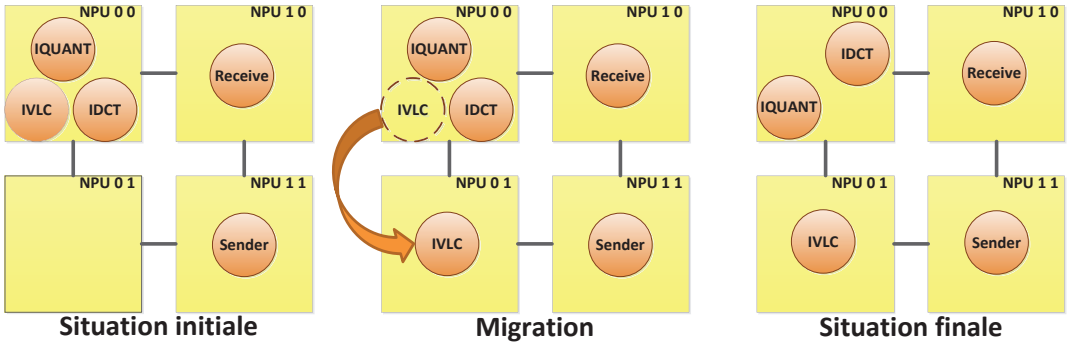


Figure 18: scénario de migration

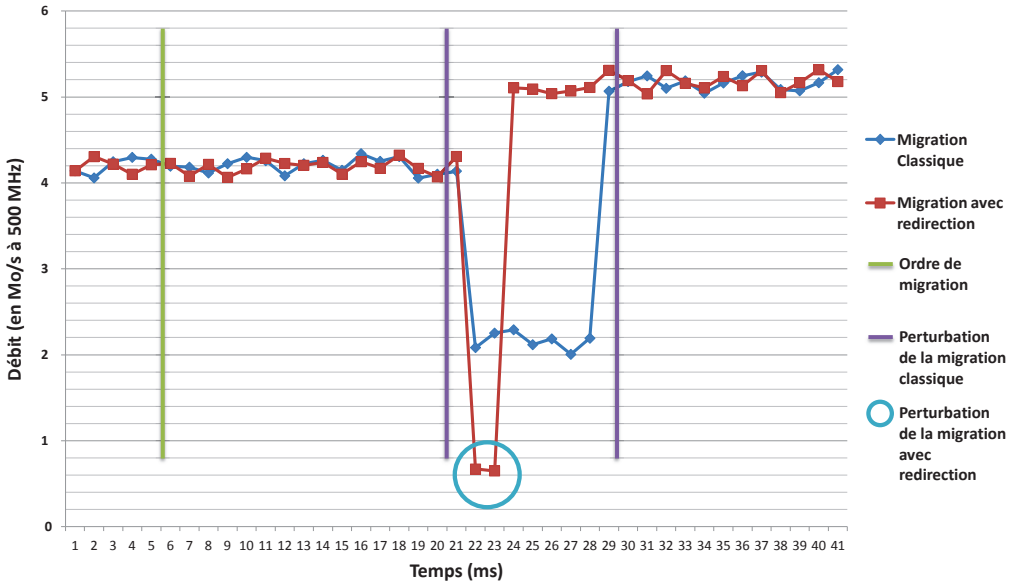


Figure 19: variation du débit de l'application MJPEG lors de la migration

L'ordre de migration est déclenché à 5 ms. Cependant, à cause des files de paquets déjà présents entre chaque tâche de l'application, les perturbations entrainées par la migration ne sont visibles qu'au bout de 19 ms. Avec la migration classique, la coupure du canal de

communication, le transfert du code, puis la réouverture du canal de communication entraine une baisse du débit, qui passe de 4,1 Mo/s à 2,1 Mo/s pendant 9 ms.

En ce qui concerne la migration avec redirection, la perturbation entrainée est plus importante, mais durant une période beaucoup plus faible (2 ms). Cette perturbation est due à la création et à la gestion des paquets à stocker et rediriger. On a donc un mécanisme plus stressant pour le système, mais d'une meilleure réactivité.

Ce facteur peut s'avérer important dans le cas d'application de flux dont le cahier des charges impose un débit minimum en sortie. Pour visualiser plus concrètement cette influence, un scénario pratique a été conçu. Celui-ci est décrit Figure 20.

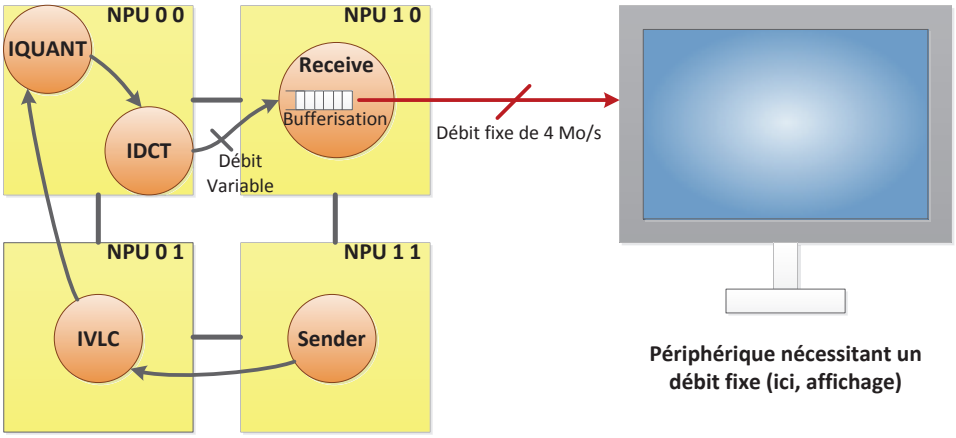


Figure 20 : utilisation de l'application MJPEG à flux de sortie constant

Dans ce scénario, la tâche de réception (Receive) envoie les paquets reçus sur un affichage demandant un débit fixe de 4 Mo/s. De ce fait, si le débit de l'application MJPEG est supérieur à ce débit fixe, les paquets seront stockés dans un buffer. A contrario, si le débit de l'application MJPEG est plus faible, les paquets stockés seront lus, diminuant le nombre de paquets dans le buffer.

La Figure 21 montre le nombre de paquet stockés par la tâche de réception qui n'ont pas encore été lus. Comme le débit considéré ici est de 4 Mo/s, lors de la situation initiale, le nombre de paquets reçu par la tâche de réception est sensiblement égal au nombre de paquets lus : on a donc un buffer qui varie peu (situation *1*).

Lors de la migration, le débit chute, ce qui entraine une diminution du nombre de paquets dans le buffer (plus de paquets sont lus que de paquets sont reçus). Il s'agit de la situation *2*. Enfin, le débit passe à sa valeur finale, ce qui entraine une augmentation continue du nombre de paquets dans le buffer (situation *3*).

Cette figure permet de mettre en avant les bénéfices de l'algorithme de migration avec redirection de paquet, car la diminution du buffer lors de la situation 2 est plus faible que pour l'algorithme de migration classique. Ainsi, alors qu'il est nécessaire d'avoir 15 paquets

dans le buffer pour effectuer une migration classique, il n'en faut que 9 pour le cas d'une migration avec redirection de paquet.

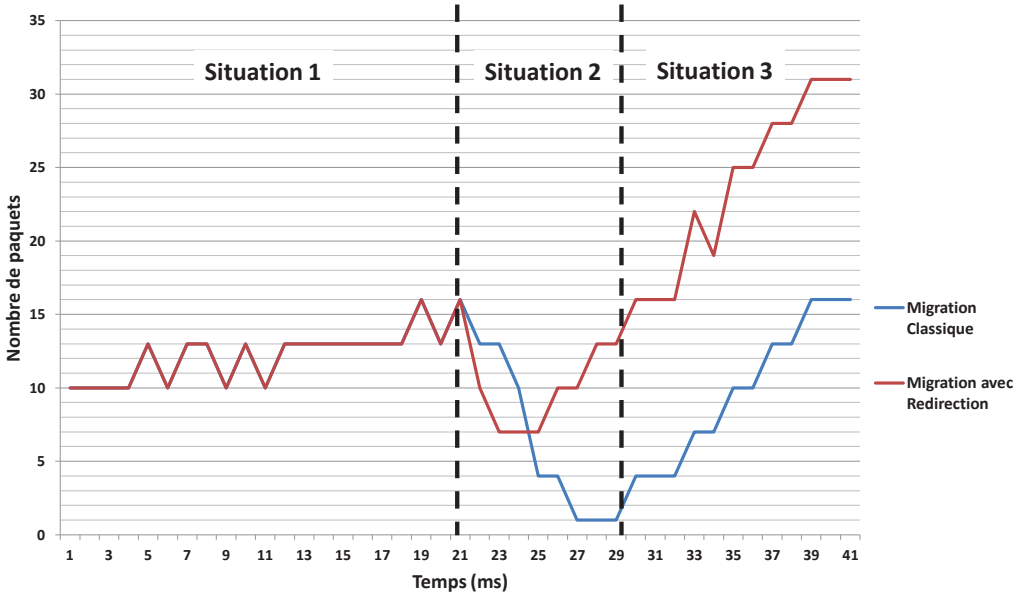


Figure 21 : Evolution du nombre de paquets dans le buffer de sortie lors d'une lecture de flux MJPEG

En conclusion, nous pouvons dire que l'algorithme de migration de tâche avec redirection permet une bonne exploitation de la bande passante élevée du Réseau sur Puce. En effet, en évitant la rupture du flux de communication amont de la tâche migrante, il impose un stress supplémentaire pour le CPU (création/destruction de la tâche de redirection), ainsi que pour le réseau (doublement du flux de transfert lors de la redirection). En revanche, il permet une plus grande réactivité de l'application, qui atteint plus rapidement les performances de sa situation finale.

3.4. Exécution distante

Dans la section précédente, le principe et l'utilisation de la migration de tâche pour effectuer une répartition de charge dynamique et scalable a été exposé. Ce principe permet d'obtenir une gestion globale de la charge d'une plateforme de manière distribuée et performante. Cependant, cette méthode souffre d'une latence importante et non compressible : le temps de transfert du code de l'application d'un nœud à un autre. Afin de pallier à ce phénomène, une nouvelle méthode a été développée sur la plateforme OpenScale : l'exécution distante.

3.4.1. Principe

Le principe de l'exécution distante repose sur le fait que le transfert d'une tâche n'implique pas forcement une migration des données, mais peut simplement être une « migration de l'exécution ». Il s'agit du principe inverse de celui développé dans [85], où les données sont migrées sans affecter l'exécution.

Ainsi, le principe de la migration classique, est illustré par la Figure 22, en prenant comme scénario la migration de la tâche T_1 du NPU_{X1Y1} vers le NPU_{X2Y2}. Ce mécanisme peut se décomposer en trois grandes étapes : tout d'abord, l'exécution de la tâche à migrer est arrêtée sur le NPU_{X1Y1}, et le code de celle-ci empaqueté pour être transmis. Ensuite, le code est transmis à travers le Réseau sur Puce. Enfin, le code est alloué dans la mémoire locale du NPU_{X2Y2} de destination, et la tâche est remise en route.

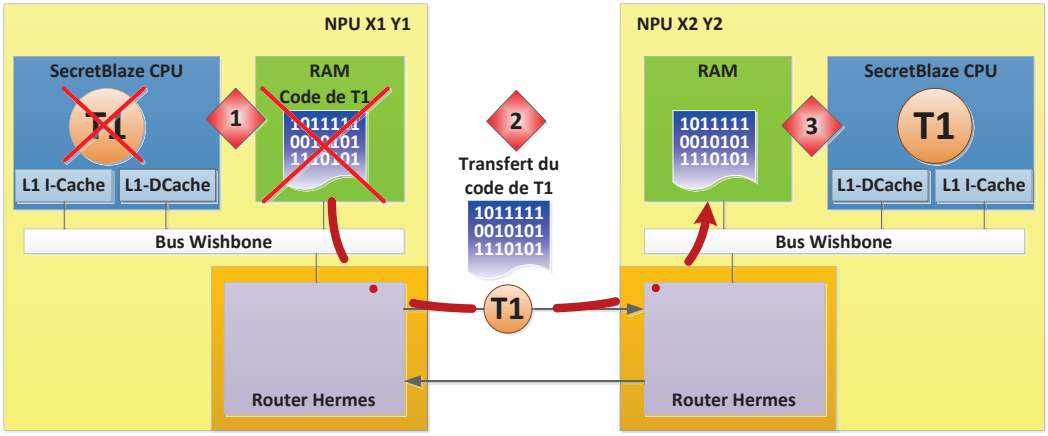


Figure 22: Principe général de la migration de tâche

Dans le cas d'une exécution distante, le code n'est pas transmis directement. Les trois étapes illustrant le mécanisme de l'exécution distante sont données Figure 23 : la tâche T1 va être exécutée de manière distante sur le NPU_{X2Y2} depuis le NPU_{X1Y1}. Là encore, le protocole débute par l'arrêt de l'exécution de la tâche. Cependant, ici, le code d'instructions et de données de la tache n'est pas explicitement déplacé, et seule une requête d'exécution est envoyée au NPU_{X2Y2}, de destination (étape 2). Le NPU_{X2Y2} va donc ensuite venir exécuter directement la tâche T_1 en accédant au code de celle-ci via le réseau sur puce. L'ensemble des instructions sont donc chargées depuis le NPU_{X1Y1} lors de l'exécution sur le NPU_{X2Y2}, par l'intermédiaire du cache instructions.

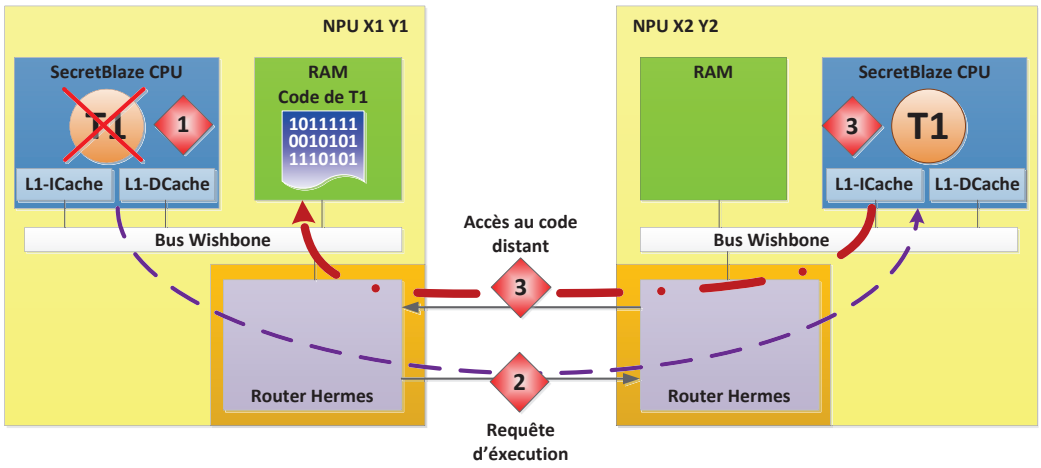


Figure 23: Principe de l'exécution distante

Le cache L1 d'instructions permet d'amoindrir la pénalité en performance de l'exécution distante due à la latence des requêtes sur le réseau sur puce.

Il faut noter ici que le modèle de programmation d'OpenScale reste de type mémoire distribuée avec passage de messages, et non mémoire partagée. Ainsi, seules les données statiques sont accédées de manière distante durant une exécution distante. La majorité des données à traiter est envoyée par passage de message. C'est pourquoi, dans cette configuration, les données statiques d'une tâche exécutée de manière distante sont non cachées. Ce choix a été fait en considérant que les données statiques sont peu nombreuses faces aux données à traiter dans une application de type flux de données, et que la latence d'un accès à la mémoire interne d'un NPU distant reste de plusieurs ordres de grandeur plus faible que pour un accès à une mémoire externe (de type DDR par exemple). De plus, cette méthode permet d'éviter la mise en place de mécanismes de cohérence de cache, difficile à implémenter sur ce type d'architecture [86].

3.4.2. Module d'accès à une mémoire distante

Afin de permettre l'utilisation du mécanisme d'exécution distante, deux types de développement ont dû être réalisés : le premier est logiciel, et consiste en la mise en place du protocole d'exécution distante, avec notamment la possibilité de création d'une tâche s'exécutant de manière distante par le système d'exploitation. La seconde concerne le mécanisme matériel permettant d'accéder, via le réseau sur puce, à la mémoire d'un NPU distant. Pour réaliser ce dernier point, un module supplémentaire a été créé : le module RMA (Remote Memory Access). Le module RMA est composé de deux principaux éléments : un sous-module RMA-Send et un sous-module RMA-Reply. Deux FIFO asynchrones permettent de relier ces modules au Réseau sur Puce. L'ensemble de ces éléments sont décrits Figure 24.

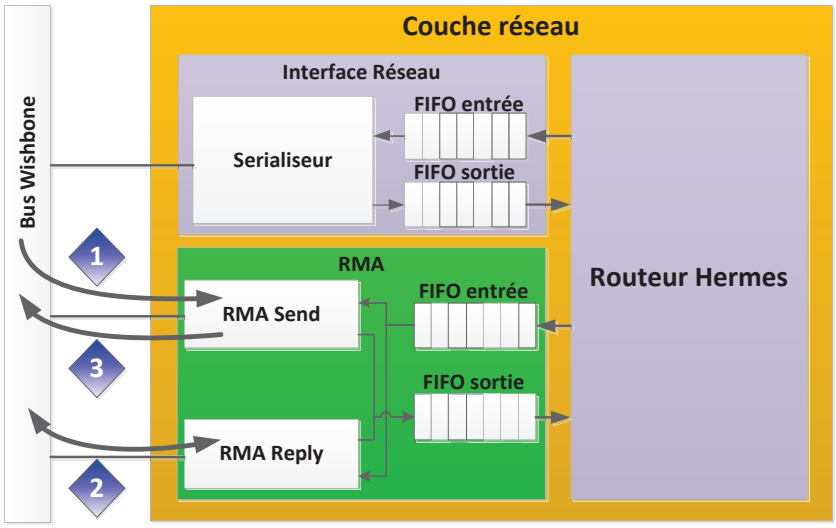


Figure 24 : Schéma du module RMA

Le but du module RMA-Send est de gérer les requêtes venant du NPU local vers le NPU distant, sur lequel se situe la donnée à lire ou à écrire. Le module RMA-Reply, quant à lui, s'occupe de répondre aux requêtes venant des NPU distants en lisant ou en écrivant dans la mémoire locale. Ainsi, une requête en mémoire distante peut se décomposer en trois étapes : tout d'abord, le module RMA-Send reçoit une requête de la part de la mémoire cache à travers le bus wishbone (étape 1 sur la Figure 24). Cette requête est empaquetée et envoyée via le réseau sur puce sur le NPU correspondant. Ensuite, le RMA-Reply du NPU distant reçoit la requête, et effectue une transaction sur le bus wishbone pour soit récupérer la donnée lors d'une lecture, soit écrire la donnée en mémoire lors d'une écriture (étape 2). S'il s'agit d'une lecture, le module RMA-Reply renvoie la donnée lue via le Réseau sur Puce au NPU ayant fait la demande. Enfin, le module RMA-Send récupère la donnée et la transmet à la mémoire cache à travers le bus wishbone (étape 3).

La deuxième modification matérielle à réaliser pour pouvoir développer l'exécution distante concerne la configuration de l'espace adressable. En effet, l'architecture originale d'OpenScale est telle que chaque NPU possède son propre espace mémoire indépendant : chaque mémoire locale est privée, et possède donc la même adresse. Ainsi, aucun conflit n'est possible car seul le processeur local peut accéder à sa mémoire. Cependant, pour développer l'exécution distante, la notion de mémoire partagée a dû être introduite dans l'architecture. Ainsi, il a été nécessaire d'implémenter une nouvelle configuration mémoire. Cette nouvelle configuration est donnée Figure 25. Afin de respecter le standard utilisé par le wishbone bus implantée dans OpenScale, les quatre bits de poids fort sont gardés en tant que sélection du périphérique local (UART, gestionnaire d'interruption ou encore mémoire). Dans le cas d'un accès à une mémoire distante (via le module RMA), les huit bits suivants sont utilisés pour déterminer les coordonnées X et Y du NPU concerné. Les vingt bits restant servent à déterminer l'adresse dans la mémoire.

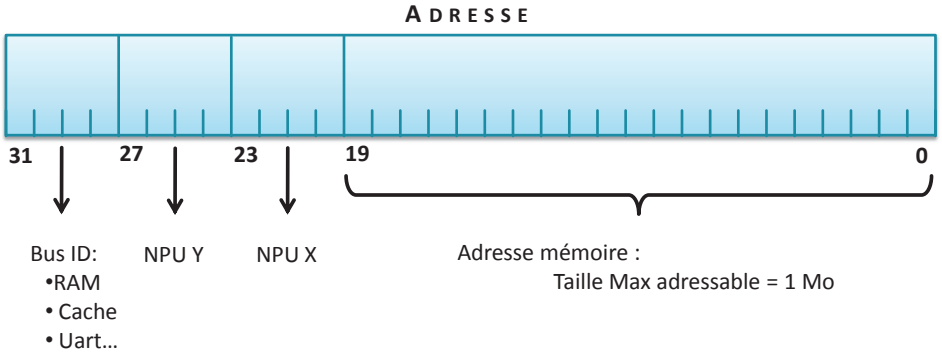


Figure 25 : Configuration de l'espace adressable

Une telle configuration permet d'obtenir des mémoires locales pouvant aller jusqu'à 1Mo, pour un réseau de 16 par 16 NPU.

3.4.3. Implémentation logicielle de l'exécution distante dans OpenScale

Les modifications sur le système d'exploitation pour utiliser l'exécution distante sont minimes comparées aux modifications matérielles. En effet, seule la création du protocole de déclenchement d'exécution distante (inspiré du protocole de migration de tâche classique) devait être réalisée.

Le mécanisme de déclenchement d'exécution distante suit donc les mêmes étapes que celui de la migration de tâche classique, à l'exception du transfert du code de la tâche (étape *iii* sur la Figure 14).

Ainsi, seul l'en-tête de description de la tâche est envoyé au NPU destinataire. Cet en-tête comprend notamment l'identifiant de l'application, l'identifiant de la tâche, la taille du cache de cette tâche, ou le nombre de communications entrantes (voir Figure 26). A cet en-tête de 12 éléments de 32 bits, est ajoutée l'adresse de la première instruction de la tâche. Ainsi, seul un total de 52 octets sont envoyés au NPU de destination.

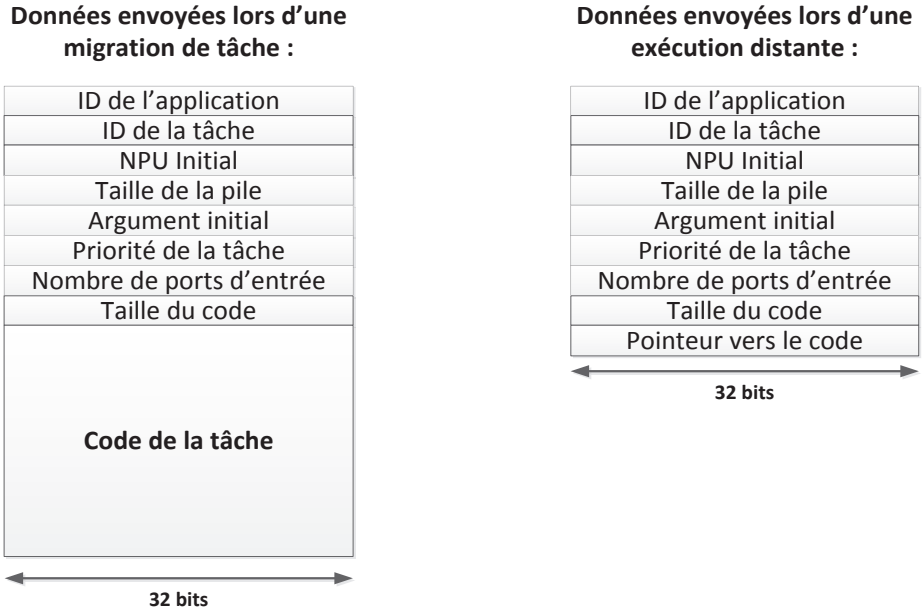


Figure 26 : Données envoyées durant une migration de tâche et lors du démarrage d'une exécution distante

3.4.4. Evaluation des performances

De nombreux éléments sont à évaluer pour déterminer la viabilité et les performances de ce protocole. L'objectif de ce paragraphe est de déterminer quels sont les paramètres qui vont influencer les performances de l'exécution distante, et dans quel(s) cas celle-ci va s'avérer préférable à une répartition de tâche statique ou à une migration de tâche.

Pour se faire, trois protocoles expérimentaux ont été mis en place :

- Le premier consiste à exécuter une application mono-tâche dont le code est situé sur une mémoire distante, afin d'évaluer les pénalités en performance d'une exécution distante face à une exécution locale.
- Dans le second protocole, une comparaison directe des protocoles de migration de tâche et d'exécution distante sera effectuée sur une application à flux de donnée.
- Enfin, une application multitâche dont plusieurs tâches exécutent le même code d'instruction sera considérée. On évaluera ainsi une exécution de type SIMD (*Single Instruction Multiple Data*) très utilisée dans les accélérateurs matériels et les GPU [87].

3.4.4.1. Performance lors de l'exécution

L'exécution d'une tâche dont les instructions sont situées sur une mémoire distante dépend directement du temps nécessaire pour accéder à ces instructions. Deux paramètres principaux vont donc jouer sur ces performances : la latence d'un accès à la mémoire distante et l'efficacité du cache.

3.4.4.1.1. Latence d'un accès distant

La latence d'un accès distant est intrinsèquement liée à la structure du réseau sur puce (par exemple sa topologie, ou encore la taille des liens entre deux nœuds). Dans OpenScale, le réseau sur puce possède une topologie de type réseau 2D (*2D-mesh*), liée par des connexions de largeur variable. Le Tableau 3 donne le nombre de cycle minimum nécessaire pour une lecture ou une écriture en fonction de la largeur des connexions et de la distance à parcourir (on défini la distance d'un accès par le nombre de routeurs à traverser entre le NPU source de la demande et celui possédant les informations). Ces mesures ont étés réalisées en utilisant un réseau de 2x2 NPU. Les instructions d'une application synthétique (instructions aléatoires) ont été placées sur le *NPU 00*, et sont accédés soit par le *NPU 10* (distance de 1), soit sur le *NPU 11* (distance de 2).

Pour réaliser ces expériences, les caches instructions et données ont été fixé à 8 Ko, avec une largeur de ligne de cache de 32 octets. Ainsi, lorsqu'une requête (un défaut de cache) est effectuée, elle concerne 32 octets. Les nombres de cycles mesurés ici vont de la requête de données jusqu'à son service complet lors d'une demande de lecture, et de la requête d'écriture jusqu'à la fin de la modification de la mémoire lors d'une demande en écriture.

Si la distance impacte peu le temps d'accès (13% à 16% pour une lecture, 5% à 8% pour une écriture), la largeur des connexions joue un rôle primordial. En effet, doubler la largeur

des connexions revient à doubler la bande passante du réseau sur puce : on a donc des gains de l'ordre de 40% à 46%.

	Lecture		Ecriture	
Largeur des connexions	Distance de 1	Distance de 2	Distance de 1	Distance de 2
4 bits	1052	1194	1308	1377
8 bits	560	638	702	748
16 bits	308	354	391	422
32 bits	182	212	231	251

Tableau 3 : Nombre de cycles minimum pour un accès distant en fonction de la largeur des connexions et de la distance

3.4.4.1.2. Exécution d'applications réelles

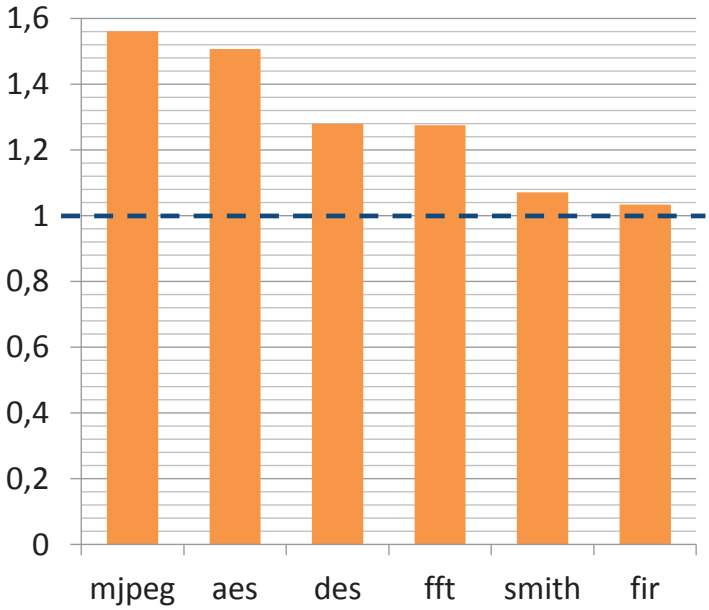
Afin d'évaluer les relations entre le cache et les performances de l'exécution distante, un ensemble d'applications mono-tâches a été utilisé. L'ensemble des ces applications ont été profilées sur OpenScale lors d'exécutions standards, afin de mettre en place des références quant aux performances de celles-ci.

Les applications considérées sont, à l'exception de l'application Smith-Waterman, des exemples typiques d'applications classiques utilisées dans l'embarqué :

- Application de décodage MJPEG : cette application consiste à décoder un flux vidéo encodé par un processus *Motion JPEG*. Contrairement à la section 3.3.3.1, il s'agit ici de la version mono-tâche de cette application.
- Application *Smith-Waterman* : cette application permet de trouver l'alignement optimal de deux séquences [88]. C'est un algorithme très utilisé notamment dans la bioinformatique, pour la recherche de séquences de protéines identiques.
- Application d'encryptage de type DES (*Data Encryption Standard*) et AES (*Advanced Encryption Standard*) [57]: ces applications sont utilisées par la plupart des communications cryptées, notamment lors de chargement de page web sécurisée, ou encore lors de la sécurisation de réseaux sans fil.
- Application de transformation matricielle de type FFT (*Fast Fourrier Transform*) ou FIR (*Finite Impulse Response*) : ces applications sont largement utilisées dans le traitement du signal.

Chacune de ces tâches a été exécutée sur un NPU distant d'une unité par rapport à son code. La plateforme utilisée contenait un réseau de 2x2 NPU relié par un NoC dont les liens faisaient 32 bits. Chaque NPU possédait 8 Ko de cache instruction et 8 Ko de cache data, et

l'ensemble des accélérateurs disponibles (voir section 2.4.3.1). La Figure 27 donne le temps d'exécution relatif de ces applications (1 étant le temps d'exécution de l'application en local). Le taux de cache-miss est également indiqué sur la Figure 27.



	MJPEG	AES	DES	FFT	SMITH	FIR
Taux de cache miss	3.97%	3.21%	1.74%	1.28%	0.645%	0.317%

Figure 27 : Temps d'exécution relatif d'applications exécutées de manière distante

Cette figure montre que l'exécution distante d'une application met entre 3% à 56% plus de temps que dans le cas d'une exécution locale. La Figure 27 permet également de souligner le lien direct entre le taux de cache-miss et les performances d'exécution.

3.4.4.2. Comparaison entre migration de tâche et exécution distante

Afin de comparer efficacement les performances d'une application à flux de données lors de l'utilisation de la migration de tâche et de l'exécution distante, le même scénario que dans la section 3.3.3.2 : l'application de décodage MJPEG à 5 tâches est utilisée (voir Figure 16). L'ensemble de ces tâches est tout d'abord placé de manière non-optimale sur la plateforme contenant 4 NPU. Afin d'obtenir une répartition des charges plus optimale, la tâche IVLC est déplacée soit avec un des protocoles de migration de tâche, soit avec le protocole d'exécution distante. La Figure 28 montre l'évolution du scénario lors de l'exécution distante (pour celui avec la migration de tâche, se référer à la Figure 18).

L'évolution du débit de flux de données de l'application MJPEG en fonction du temps est donnée Figure 29, pour l'ensemble des cas suivant : migration de tâche classique, migration de tâche avec renvoi de données et exécution distante. Les trois protocoles sont déclenchés au même moment dans les trois simulations. De ce fait, la migration ou l'exécution distante s'effectue au même moment, soit après 19 ms.

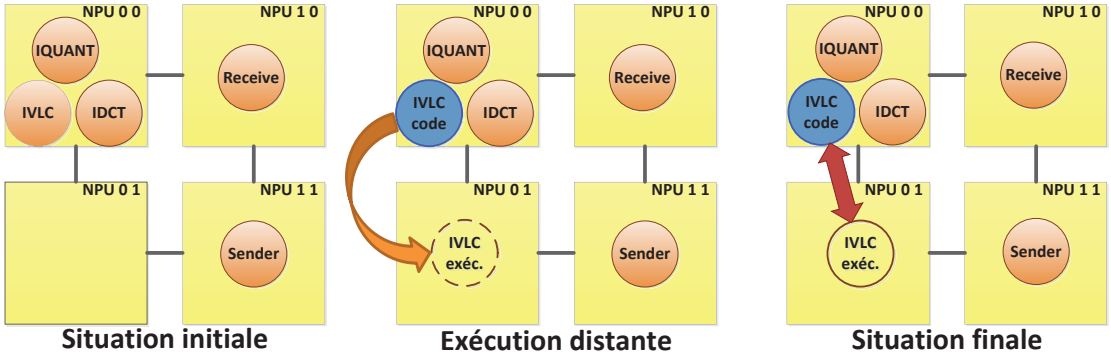


Figure 28 : Scénario d'exécution distante de l'application MJPEG

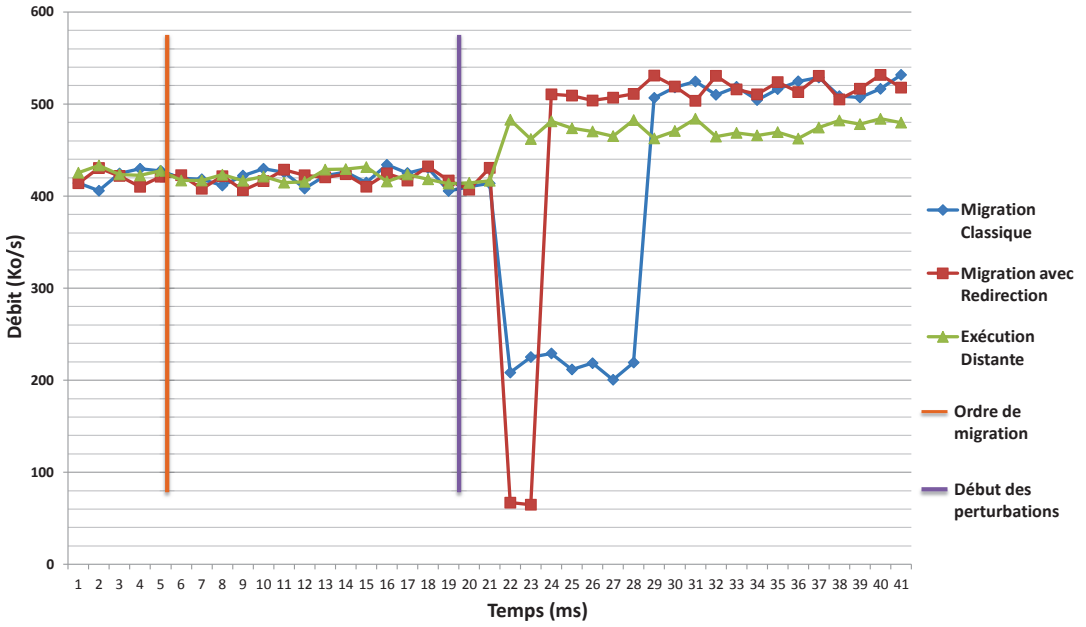


Figure 29 : Variation de débit de l'application MJPEG durant une migration ou une exécution distante

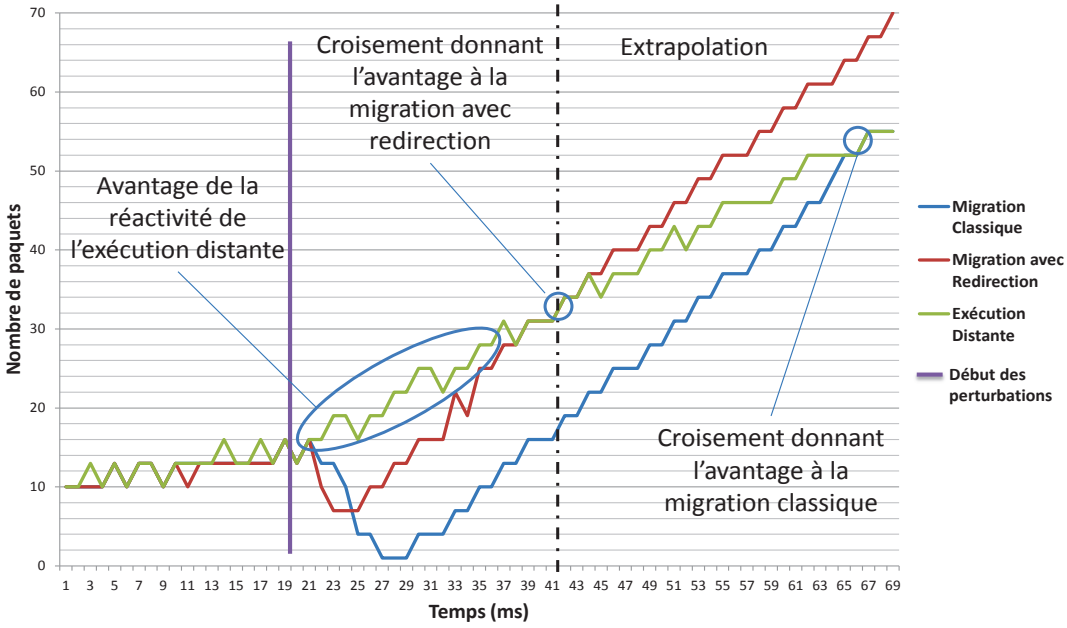


Figure 30 : Evolution du nombre de paquets dans le buffer de sortie lors d'une lecture de flux MJPEG

La Figure 29 souligne bien la réactivité de l'exécution distante. En effet, le débit de l'application MJPEG passe directement d'environ 410 Ko/s à 460 Ko/s sans période de transition, contrairement aux protocoles de migration. Par contre, l'exécution distante ne permet pas d'obtenir le débit final d'une migration de tâche, dont le code et l'exécution se font sur le même NPU.

Pour visualiser la réactivité de ces algorithmes, le scénario décrit Figure 20 a également été appliqué à l'exécution distante : en sortie de l'application MJPEG, un débit fixe est demandé. Un système de bufferisation des paquets est donc nécessaire dans la tâche de réception. La Figure 30 montre le nombre de paquets stockés par la tâche de réception qui n'ont pas encore été lus.

On considère ici un débit de données de sortie de 4 Mo/s. Ainsi, avant le début des perturbations, le nombre de paquets dans le buffer oscille entre 10 et 16. Ensuite, l'évolution du nombre de paquets dans le buffer va varier en fonction du protocole utilisé : la réactivité du protocole d'exécution distante va entrainer une augmentation directe du nombre de paquets, sans diminution comme dans le cas des protocoles de migration. Cependant, le débit final du scénario avec exécution distante étant plus faible que celui des scénarios de migration de tâche, l'augmentation du nombre de paquets va être moins significative. Ainsi le nombre de paquets dans les buffers des scénarios de migration de tâche vont finir par rattraper et dépasser celui du scénario d'exécution distante : au bout de 40 ms pour la migration avec redirection, et au bout de 68 ms pour la migration classique (données extrapolées en fonction du débit final des différents scénarios).

Cette réactivité de l'exécution distante peut être avantageuse dans le cas de traitement de d'application de télécommunication (VoiP, vidéoconférence, etc.). En effet, pour envisager une migration de tâche sans rompre le flux de communication, il faut un minimum de 9 paquets dans le buffer pour une migration avec redirection, et 15 paquets pour une migration classique. Dans le cas d'une exécution distante le stockage d'un seul paquet est suffisant. Au-delà de la pénalisation en terme d'espace mémoire, ce stockage de données va également entrainer une latence sur la communication, et donc un décalage entre l'émetteur et le récepteur.

On peut donc conclure sur l'utilité de l'exécution distante dans une application de traitement de flux de données : celui-ci permet de réaliser une modification rapide de la répartition des charges au sein du système sans subir le coût en latence et en performance d'une migration de tâche. Cependant, cette répartition sera moins performante que dans le cas d'une migration : elle est intéressante dans le cas de perturbations de haute fréquence du système (quelques millisecondes), ou encore dans le cas de télécommunication de type VoiP, là où la migration de tâche sera plus efficace sur des perturbations de basse fréquence ou continues.

3.4.4.3. Performance sur un scénario SIMD

Comme cela a été évoqué précédemment, la principale caractéristique de l'exécution distante est le fait de ne pas avoir à transférer le code : l'exécution se fait sur le code sans modification. On peut tirer également parti de cette caractéristique en créant plusieurs tâches exécutant le même code d'instruction (mais traitant des données distinctes). Ainsi, alors qu'avec une démarche classique de mémoire partagée indépendante, les instructions auraient dû être dupliquées pour chaque tâche, le même code peut être réutilisé ici par plusieurs tâches.

Cependant, cette méthode possède un inconvénient majeur : sa mise à l'échelle (augmentation du nombre de tâches utilisant le même code instruction) est faible, car chaque tâche accède à la même mémoire du même NPU. Ainsi, si le nombre de tâche augmente de manière trop importante, les performances de l'application vont être limitées par la bande passante maximum que peut offrir la mémoire contenant les instructions utilisées par toutes les tâches.

Le but de cette section est de montrer les performances possibles de l'utilisation de l'exécution distante dans le cas d'un fonctionnement où plusieurs tâches utilisent le même code d'instruction. Pour cela, on utilisera l'application multitâche de l'algorithme de ***Smith Waterman***. Cet algorithme a la particularité de pouvoir être divisé en un nombre quelconque de tâches effectuant toutes le même code sur des données différentes. En effet, celui-ci est composé d'une tâche d'envoi, suivie d'un nombre quelconque de tâches de comparaison (ces tâches seront exécutées de manière distante), puis d'une tâche effectuant le maximum des comparaisons renvoyées par la ou les tâches de comparaison.

La Figure 31 montre les 6 configurations qui ont été évaluées. Dans l'ensemble des cas, le code instruction de la ou des tâches de comparaisons est stocké sur le *NPU 1 1*. Afin de tester les limites de la mise à l'échelle du système, trois scénarios différents ont été réalisés : l'un avec une taille de cache de 16Ko pour chaque NPU, le deuxième avec une taille de 1Ko pour chaque NPU, puis enfin avec 512 octets par NPU.

Dans le premier scénario, la taille du cache permet à la tâche de comparaison de tenir entièrement en cache. Dans cette configuration, l'acquisition d'une ligne d'instruction en cache est définitive : celle-ci n'est jamais redemandée. Cela permet d'obtenir un pourcentage de défauts de cache très faible (de l'ordre de 0,1% maximum). Dans le second et le troisième scénario, l'ensemble des instructions d'une tâche de comparaison ne tient pas dans le cache : on a donc des acquisitions de ligne de cache tout au long de l'exécution.

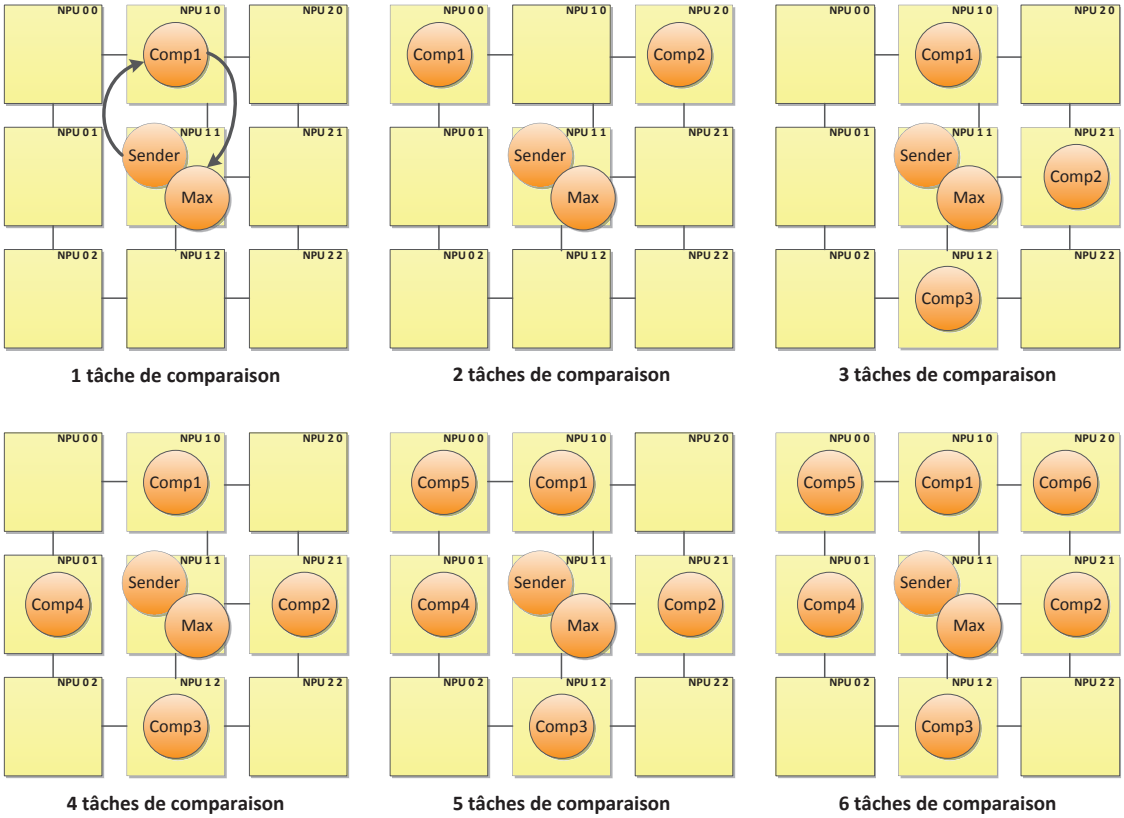


Figure 31 : Différentes configurations de l'application Smith-Waterman

La Figure 32 présente les performances d'exécution des trois scénarios en fonction du nombre de tâches de comparaison exécutées de manière distante (la référence à 100% correspondant à l'exécution de l'application avec une tâche de comparaison exécutée localement).

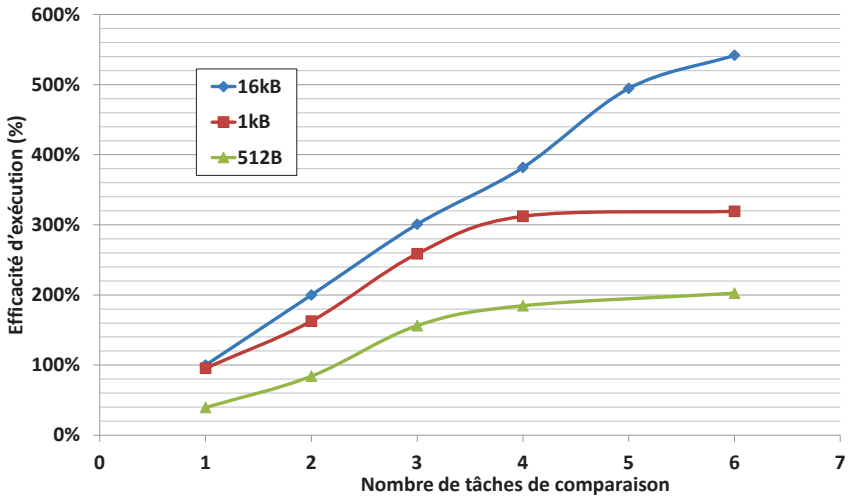


Figure 32 : Efficacité d'exécution en fonction du nombre de tâches de comparaison

Cette figure montre que dans le premier scénario, l'efficacité d'exécution augmente de manière quasi-linéaire. Cela est dû au fait qu'aucune instruction n'est chargée en cache plus d'une fois : ainsi, après la première boucle globale de l'algorithme de comparaison,

plus aucune instruction n'est chargée et le code s'exécute de manière locale (à partir du cache).

Cependant, pour le deuxième et le troisième scénario, l'efficacité tend à atteindre un plateau d'environ 325% pour le cas des 1Ko de cache, et 200% pour le cas des 512 octets de cache. Ces scénarios montrent les limites de la mise à l'échelle de cette méthode : à cause du nombre croissant de défauts de cache, et ainsi de la latence pour l'acquisition d'une ligne de cache, le temps d'exécution distante devient supérieur au gain résultant de la parallélisation des données, entrainant ainsi une stagnation, voire une baisse des performances.

La Figure 33 montre la latence moyenne d'une acquisition de ligne de cache en fonction du scénario et du nombre de tâche de comparaison. On remarque ainsi que la latence augmente plus lors du premier scénario que dans le second et le troisième. Ce phénomène est dû à la désynchronisation naturelle de l'exécution des tâches de comparaison. En effet, lors de l'acquisition de la première ligne de cache par l'ensemble des tâches de comparaison, toutes les requêtes arrivent environ au même moment au NPU central (*NPU 1 1*). Cependant, ces requêtes sont ensuite traitées séquentiellement l'une après l'autre. De ce fait, la première tâche servie pourra commencer son exécution avant les autres, et ainsi de suite. Ainsi, après plusieurs acquisitions, l'exécution des tâches devient désynchronisée, ce qui entraine une diminution de la latence d'une acquisition : durant l'acquisition d'une ligne par une tâche, les autres continuent leurs exécutions, et n'attendent pas une acquisition.

Comme dans le premier scénario le nombre de défauts de cache (et donc d'acquisitions) est plus faible que dans le scénario 2 et 3, la désynchronisation est plus rapide et donc la latence plus faible.

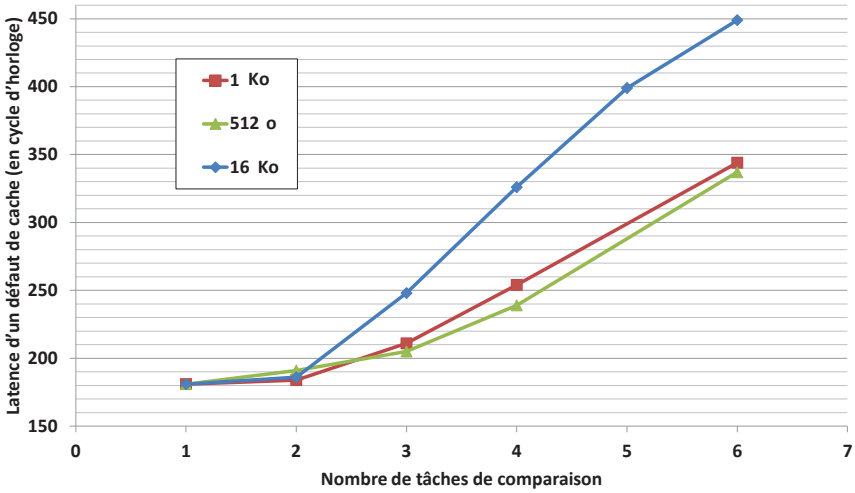


Figure 33 : Latence moyenne d'une acquisition de ligne distante

La Figure 34 illustre cette désynchronisation, en montrant le nombre de défauts de cache distants par unité de temps (ici, la période de cette unité de temps est de 1000 cycles), en considérant le scénario de 512 octets avec 3 tâches de comparaison. Cette figure montre également la latence moyenne par unité de temps d'une acquisition pour chaque tâche de comparaison.

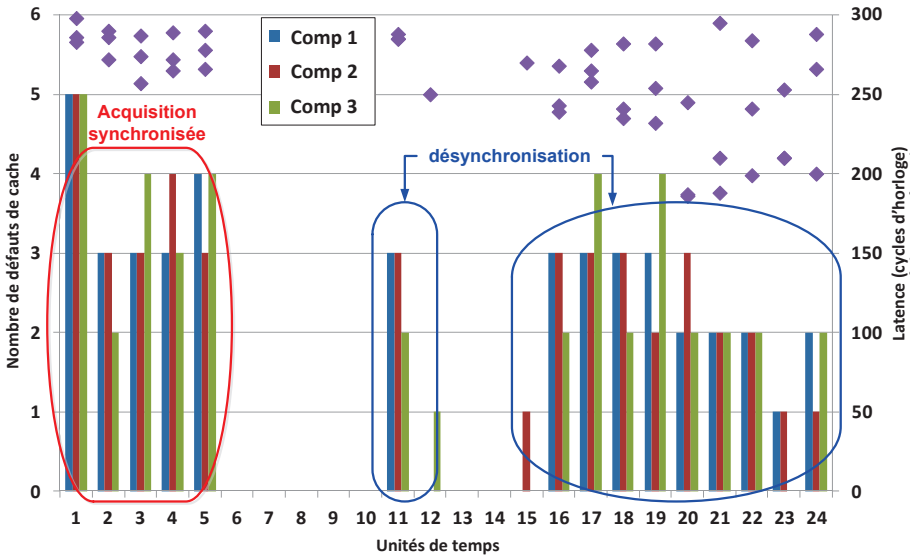


Figure 34 : Nombre de défauts de cache et latence de ces défauts par unité de temps (1000 cycles)

La structure localement synchrone, globalement asynchrone (GALS) d'OpenScale, provoque un décalage naturel des exécutions de chaque tâche de comparaison. Ainsi, l'exécution parallèle de code instruction identique diffère de l'exécution *Simple Intruction Multiples Données (SIMD)* classique que l'on retrouve dans les accélérateurs graphique par exemple. Cette exécution se rapproche plus d'une exécution *Multiples Instructions Multiples Données* (*MIMD*), du fait que chaque NPU puisse exécuter une instruction différente. Une telle structure permet ainsi d'éviter la sous-utilisation des ressources que l'on peut avoir dans des systèmes SIMD lors notamment de branchements : chaque exécution peut diverger sans pénalité.

Comme chaque NPU ne possède qu'un unique canal par direction pour transférer instructions et données, l'efficacité de l'exécution distante peut également être affectée par un transfert de donnée. Afin de quantifier cette perturbation, un nouveau scénario a été mis en place : l'exécution de 6 tâches de comparaisons est considérée avec un flux de données de perturbation, partant d'une tâche d'envoi située sur le NPU 0 1, et une tâche de réception située sur le NPU 2 1. Ce scénario est illustré Figure 35.

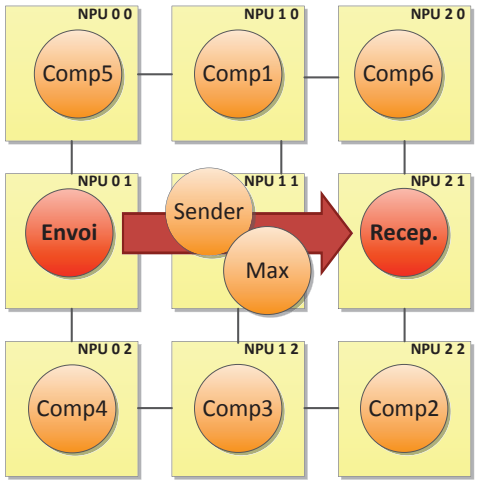


Figure 35 : Configuration du scénario de perturbation de l'exécution distante avec Smith-Waterman

L'expérience a été réalisée pour différentes valeurs de taille de cache ainsi que pour différentes valeurs de débit entre les tâches d'envoi et de réception. Il est important de noter que ce débit est un débit théorique, qui ne prend pas en compte la possible saturation du Réseau : la tâche d'envoi va essayer d'envoyer à ce débit, les paquets étant stockés en cas de trop forte demande. Dans notre cas, un débit théorique de 8Mo/s correspond à un envoi continu.

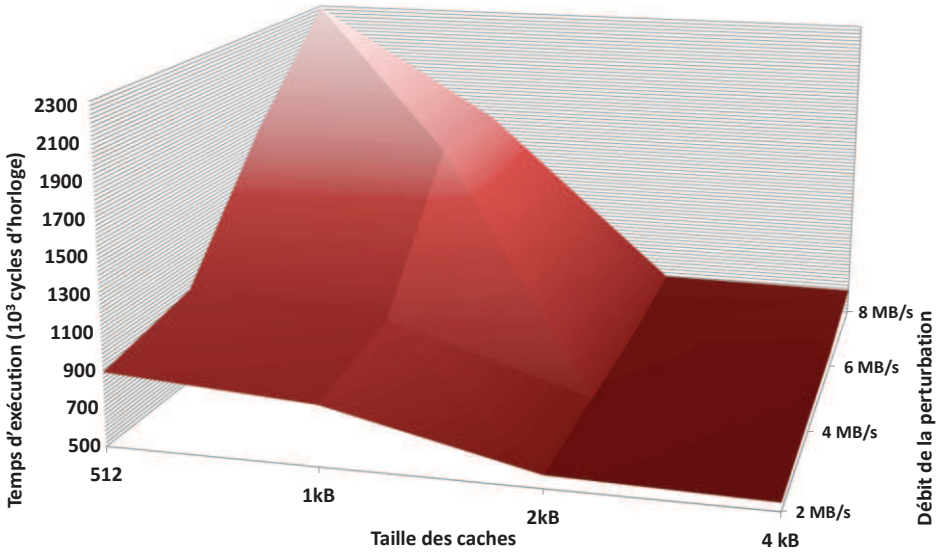


Figure 36 : Temps d'exécution de l'application Smith-Waterman avec 6 tâches de comparaison en fonction de la taille des caches et du débit d'une perturbation

La Figure 36 souligne le fait que l'exécution distante est particulièrement dépendante de la bande-passante disponible du Réseau sur Puce : pour une perturbation de plus de 6 Mo/s (proche de la saturation du Réseau sur Puce), le temps d'exécution augmente de 70% à 240% lorsque la taille du cache est divisée par 2. Pour le cas de cache de taille supérieure à

2Ko cependant, le nombre de défauts de cache est suffisamment faible pour que la perturbation n'influence l'exécution que de manière minime (inférieure à 10%).

3.4.5. Conclusion

En résumé, l'exécution distante possède un comportement permettant d'obtenir des performances intéressantes en fonction de plusieurs paramètres. Le facteur principal reste le nombre de défauts de cache : pour des tâches ne provoquant pas plus d'1% de défauts de cache, une exécution distante va entrainer des performances en temps d'exécution quasi-équivalente à une migration de tâche, avec une réactivité bien supérieure. Au-delà, c'est la latence de ces défauts de cache qui vont donner les performances. Ces latences peuvent être dues à plusieurs facteurs : le nombre d'accès à la mémoire distante, comme cela a été vu sur l'application Smith-Waterman, où les performances atteignent un plateau au delà de 4 tâches ; ou encore la présence d'autres transferts sur le Réseau sur Puce, où la saturation d'un canal de transmission va entrainer de fortes chutes de performances.

3.5. Conclusion

Dans ce chapitre, plusieurs méthodes permettant d'exploiter le potentiel de la plateforme OpenScale ont été développées : différents protocoles de migration de tâches, notamment avec une optimisation pour les applications de flux de données, ont été décrits.

Un nouveau protocole de répartition de charge, intitulé exécution distante a été développé. Ces différents outils peuvent être utilisés séparément ou en commun pour augmenter l'efficacité de la répartition des charges sur la plateforme. La spécificité de chacun (généraliste, axé flux de données, plus réactif…) permet de faire d'OpenScale une plateforme pouvant s'adapter à un nombre croissant de situations.

Par ailleurs, le protocole d'exécution distante a nécessité de faire des modifications à la fois matérielles et logicielles de la plateforme. OpenScale a ainsi évolué vers un système de mémoire non plus privatif, mais partagé, bien que distribué spatialement. Cette évolution ouvre également de nouvelles perspectives quant à la programmabilité de la plateforme. En effet, la potentielle utilisation d'outils utilisés pour les systèmes à mémoire partagés est maintenant envisageable.

Chapitre 4 : Evolution du modèle de programmation

4.1. Introduction

Les différentes évolutions de la plateforme OpenScale pour augmenter son accessibilité et ses performances ont amené aux travers des chapitres précédents la création au niveau matériel d'un système de gestion mémoire partagée. Les choix amenant à cette transformation sont également justifiables par les avantages d'un système à mémoire partagée : ce modèle de programmation bénéficie d'un historique plus conséquent que celui de la programmation par passage de message, de même que d'une popularité accrue de par son aisance d'utilisation.

Si l'accès à une mémoire distante a été discuté dans le chapitre 3, l'utilisation d'un modèle de programmation à mémoire partagée nécessite également la mise en place de systèmes permettant de fournir les différentes fonctionnalités de ce modèle, notamment la gestion de la cohérence mémoire. Ce chapitre décrit le développement de ce modèle de programmation, et les choix réalisés pour conserver les caractéristiques de scalabilité de la plateforme.

La première partie de ce chapitre s'attache à exposer l'historique du développement de la programmation à mémoire partagée, et souligne les différents problèmes que cette programmation peut entrainer. Dans un second temps, la solution adoptée sur OpenScale sera décrite, puis un certain nombre de résultats en performance seront décrit.

4.2. Programmation multithread

Dans ce chapitre, la programmation à mémoire partagée et la programmation multithread seront considérées comme équivalentes. La programmation multithread consiste à distinguer une tâche, portion d'une application autonome, possédant sa propre zone mémoire, et un thread, portion d'une application, crée par celle-ci, et partageant le même espace mémoire. Cette distinction est utile dans le cas d'architecture possédant une unité de gestion mémoire (*Memory Management Unit, MMU*). L'architecture d'OpenScale ne possédant pas de MMU, cette distinction est inutile.

4.2.1. Introduction

La programmation multithread a commencé à émerger commercialement dans les années 90, bien que les concepts utilisés aient été établies dans les années 70 [89][90]. Durant cette décennie, trois librairies se sont imposées : PThreads, et des librairies pour les systèmes OS/2 et Win32. Bien que très similaires dans leur fonctionnement, la librairie PThreads [21] a su s'imposer dans un standard (POSIX), qui lui permet aujourd'hui d'être utilisable par la très grande majorité des systèmes d'exploitations (autant ceux des ordinateurs personnels comme Mac, Linux ou Windows, que dans l'embarqué avec µClinux, Net-BSD ou eCOS).

4.2.1.1. Programmation multiprocesseur symétrique (SMP)

Cependant, l'ensemble des librairies PThreads développées font appel à une gestion de la part du système d'exploitation d'un système multicœur appelé programmation *multiprocesseur symétrique* (Symetric MultiProcessor, *SMP*). Ce type de programmation est basé sur une vision de la mémoire du système monolithique : l'ensemble de la mémoire est partagée et l'accès à un emplacement mémoire renvoi systématiquement la dernière valeur écrite dans celle-ci (on parle de mémoire cohérente). Cette vision, très proche des architectures classiques à mémoire partagée, ne colle pas avec une architecture à mémoire distribué comme OpenScale. Deux éléments viennent rendre la programmation *SMP* sous-optimale dans un système à mémoire distribué. Le premier concerne la gestion de la cohérence de la mémoire (notamment de la cohérence de cache), qui nécessite des aménagements souvent très pénalisant en termes de surcout matériel ou en performance. Le second concerne le fait qu'aucune notion de distance entre mémoire et processeur n'est considérée : il s'agit d'une programmation symétrique, ce qui implique qu'un programme va pouvoir être exécuté sur n'importe quel cœur, sans distinction. Afin d'éviter les pénalités en performance dues à une mémoire trop distante du cœur (i.e. dont

la latence d'accès est grande), des mécanismes matériels comme une hiérarchie de cache sont mis en place.

Les systèmes à mémoire distribuée, tel qu'OpenScale, voulant utiliser une programmation à mémoire partagée sont appelé système à *mémoire distribuée partagée* (Distributed Shared Memory, *DSM*).

4.2.1.2. Systèmes à Mémoire Distribuée Partagée

Les systèmes DSM sont très présents pour de la *programmation haute performance* (High Programming Computing, *HPC*). C'est dans ce domaine qu'ont été développées de nombreuses méthodes permettant la gestion de données partagées au sein d'une architecture à mémoire distribuée. Ont peut notamment citer le développement de la librairie Rthread [91], qui possède une syntaxe proche de celle de PThreads, et qui peut être utilisé sur un réseau quelconque de calculateurs hétérogènes. Dans [92], les auteurs présentent la librairie Adsmith, qui permet à l'utilisateur de programmer avec une vision mémoire partagée alors que le système d'exploitation gère une mémoire distribuée. L'ensemble de ces méthodes font cependant appel à un traitement lourd de la part du système d'exploitation afin de transformer les appels à une mémoire partagée en un mécanisme complexe de gestion de la cohérence mémoire et de l'accès aux données, via l'envoi et la réception de messages à travers le réseau.

Dans le domaine des MPSoC, d'autres types de mécanismes moins couteux en développement matériel tout comme en logiciel doivent être réalisés. Dans [93] par exemple, les auteurs proposent un gestionnaire de mémoire (nommé SoCDMMU) compatible avec une gestion partagée d'une mémoire distribuée. La principale caractéristique de l'utilisation de SoCDMMU est que l'allocation de mémoire se fait de manière rapide et déterministe. Son principal défaut est que chaque transaction mémoire doit obligatoirement passer par le module SoCDMMU, ce qui en fait un système non scalable. De La Luz et al. [94] décrivent dans leurs travaux le développement d'un gestionnaire logiciel complexe de la mémoire partagée au sein d'une architecture distribuée. Dans [95], les auteurs proposent une version matérielle moins complexe de ce gestionnaire, appelé HwMMU. Cependant, cette gestion de la mémoire reste centralisée au sein d'un seul module HwMMU, même si l'ensemble des transactions ne passe pas forcement par celui-ci : cela en fait un système que partiellement scalable.

Le but de cette section est de présenter une gestion de mémoire partagée distribuée au sein de l'architecture OpenScale. L'objectif de cette gestion est d'être distribuée au sein de chaque nœud afin d'en faire un système scalable. Elle doit également être d'une complexité faible pour éviter une augmentation trop significative de la taille d'un NPU.

4.2.2. Présentation de la gestion mémoire

Deux éléments principaux sont à prendre en compte lors de la gestion d'une mémoire partagée : le modèle de consistance mémoire et la cohérence de cache. Ces deux paramètres conditionnent très fortement le développement de l'architecture, que ce soit au niveau matériel comme logiciel. Dans cette section, sera exposé les principaux modèles de consistance mémoire utilisée puis les différentes cohérences de caches disponibles dans la littérature. Enfin, la solution adoptée dans notre cas sera exposé.

4.2.2.1. Modèle de consistance mémoire

La problématique d'un modèle de consistance mémoire apparait lorsque deux processeurs cherchent à accéder à un même emplacement mémoire. En effet, au sein d'un programme constitué de plusieurs tâches parallèles, le programmeur s'attend à avoir ses écritures et ses lectures effectuées dans un certain ordre. Prenons l'exemple de l'Algorithme 1, où un programme constitué de 2 tâches exécutées sur deux processeurs différents se synchronise à l'aide d'une variable drapeau (*flag*). Ce programme a pour but de faire afficher la variable A ayant la valeur 1 sur le processeur 2.

Hypothèse : A = flag = 0 à l'initialisation

Processeur 1 :	Processeur 2 :
A = 1	**while** flag = = 0 **do**
flag = 1	*skip*
	end while
	print A

Algorithme 1 : Exemple de programme nécessitant une consistance mémoire

Dans cet exemple, le programmeur s'attend à ce que la variable A soit passée à 1 **avant** la variable *flag*. Cependant, dans un système à accès mémoire non uniforme, cette hypothèse n'est pas forcément vraie : il faut définir un ordre, une séquence des lectures et écritures du programme. On fait donc appel à un modèle de consistance mémoire.

Il existe de nombreux modèles de cohérence mémoire, permettant chacun d'avoir plus ou moins de liberté sur l'ordre des lectures et écritures au sein d'un programme. Un modèle sera dit plus *relâché*, ou plus *faible* si il permet plus de réordonnancement, et donc plus de liberté sur les accès en mémoire partagée. Nous présenterons ici les principaux modèles du plus contraint au plus relâché.

Le modèle de consistance mémoire le plus contraint a été formellement énoncé par Lamport [96], et s'appelle *Consistance Séquentielle*. Dans un modèle de consistance séquentielle, l'ensemble des opérations d'accès mémoire sont exécutés dans un ordre séquentiel, vu de manière identique par tous les processeurs. La Figure 37 illustre la vision

de ce modèle : une seule et unique mémoire partagée existe, et l'ensemble des opérations sur cette mémoire se font de manière séquentielle. Ce modèle permet notamment à l'Algorithme 1 de s'exécuter de manière cohérente (l'écriture de A sera vue par tous les processeurs comme antérieure à l'écriture de flag).

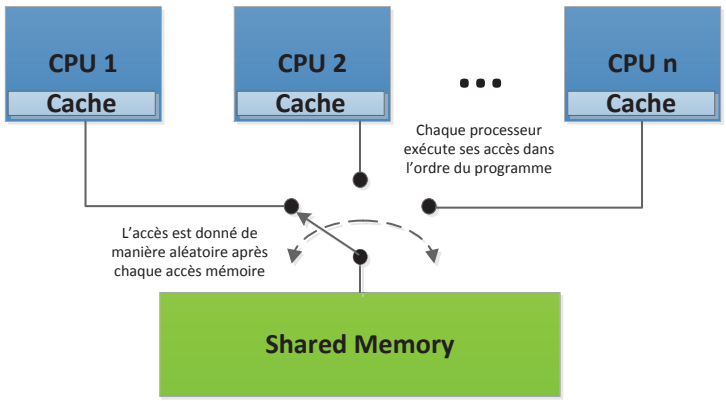


Figure 37 : Vision abstraite du modèle de consistance séquentielle

Un second modèle de consistance mémoire, plus relâché, est décrit dans [86][97][98] : la consistance processeur. Celle-ci impose seulement que l'ensemble des opérations mémoire effectuées par un processeur soient vu par un autre processeur dans le même ordre. Par contre, contrairement à une consistance séquentielle, tous les processeurs n'ont pas à voir les séquences d'accès mémoire dans le même ordre. Si l'on prend le cas de l'Algorithme 2 par exemple, celui-ci est assuré de se comporter de manière cohérente avec une consistance séquentielle, mais peut s'avérer défaillant avec une consistance processeur.

Hypothèse : A = B = 0 à l'initialisation

Processeur 1 :	Processeur 2 :	Processeur 3 :
A = 1	**while** A = = 0 **do**	**while** B = = 0 **do**
	skip	skip
	end while	**end while**
B = 1	print A	

Algorithme 2 : Exemple de programme nécessitant une consistance séquentielle

Ainsi, dans le cas d'un modèle de consistance processeur, le processeur 2 peut voir l'écriture de A à la valeur 1 avant le processeur 3. Il peut donc écrire B à 1, et cette écriture peut être vue par le processeur 3 avant l'écriture de A à 1. De ce fait, le processeur 3 peut lire et afficher la valeur A à 0.

Un modèle encore plus relâché de consistance mémoire est la consistance faible [99]. Ce type de modèle se base sur le fait d'utiliser les points de synchronisation pour ordonner les requêtes mémoires. Le principe est de considérer qu'en dehors de points de synchronisations, les accès mémoires peuvent être faits de manière quelconque, seul

compte le fait que l'ensemble des accès amonts à un point de synchronisation soient fait avant tout accès avals. La théorie derrière ce modèle de consistance est que tout accès mémoire (venant de tâches différentes) nécessitant d'être exécuté suivant une certaine séquence nécessite un système de synchronisation. Ainsi, si l'on prend l'Algorithme 2, les boucles d'attentes (''***while*** *A == 0*'' et ''***while*** *B == 0*'') peuvent être considérées comme des barrières de synchronisation.

Dans la même lignée que la consistance faible, la consistance libérée [97] utilise des points de synchronisation. Cependant, distingue différentes manière de se synchroniser : il s'agit de considérer l'instant où l'on a besoin d'acquérir la donnée partagée (barrière d'acquisition) et le moment où l'on souhaite relâcher la contrainte sur cette donnée (barrière de libération). Pour le bon déroulement d'un système possédant une consistance mémoire libérée, il faut suivre ces trois règles :

- Tout accès mémoire ne peut être effectué que si l'ensemble des barrières d'acquisitions précédentes ont été exécutées (et sont visibles par l'ensemble des processeurs).
- Avant qu'une barrière de libération ne soit exécutée, il faut que toutes les opérations précédentes sur ce processeur aient été exécutées (et soient visibles par l'ensemble des processeurs).
- Les barrières d'acquisition et de libération utilisent un modèle de consistance processeur.

Dans le domaine des applications de traitement de flux de données, J.W. van den Brand et al. ont proposé un modèle de consistance mémoire relâché, appelé consistance de flux [100]. Le principe est de considérer que l'ensemble des données partagées sont accessibles à travers des files (FIFO). Ainsi, chaque écriture va s'effectuer via un envoi dans la file, et chaque lecture via la sortie d'un élément de la file. De la même façon que pour la consistance libérée, des processus d'acquisitions et de libérations de ces files sont nécessaires. L'avantage principal de la consistance de flux est que l'on a plus de liberté concernant le réordonnancement de chaque section critique (section commençant par une acquisition d'une file et finissant par sa libération). L'inconvénient de ce modèle est que l'ensemble des données ne peuvent être considérées que sous forme de files.

La Figure 38 donne un résumé de l'ensemble des contraintes d'ordonnancement des accès en fonction des consistances mémoires vues dans cette section.

Les flèches de la figure représentent les contraintes du modèle : ainsi, une flèche allant d'une transaction à une autre indique que l'ensemble des processeurs doivent voir l'opération à la base de la flèche être effective avant l'opération pointée par celle-ci.

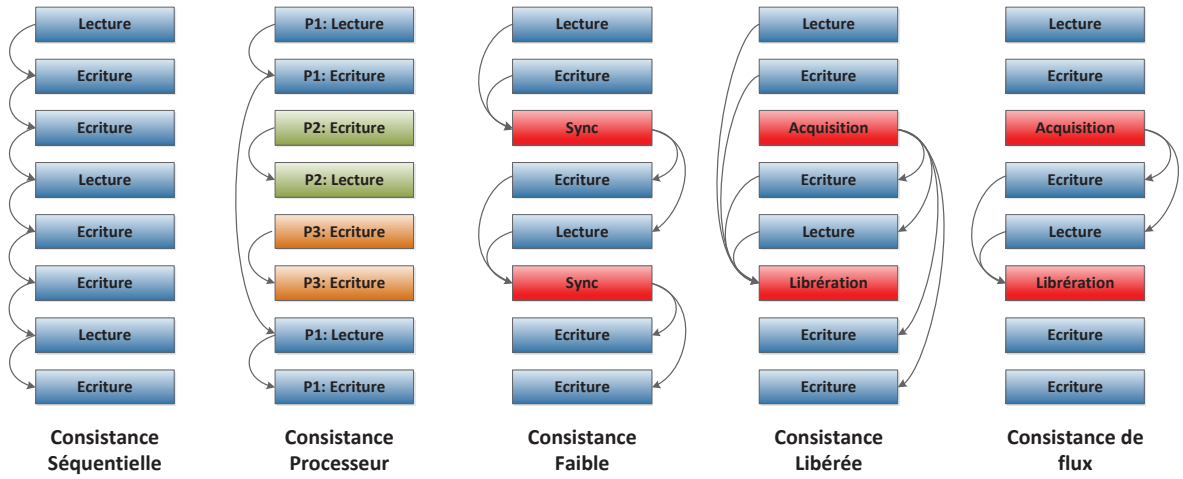


Figure 38 : Contraintes de visibilité suivant les modèles de consistances mémoire

Dans le cas de l'architecture OpenScale, on utilisera un modèle de consistance mémoire libérée. Ce modèle de consistance a pour avantage de posséder des contraintes plus facilement réalisables matériellement et de manière logicielle que les contraintes des modèles de consistance plus forte. De plus, ce modèle est suggéré dans la plupart des développements de programmes multitâches pour les accélérateurs actuels : ainsi, Intel, bien que supportant un modèle de consistance processeur, suggère à ces programmeur d'utiliser un modèle de consistance plus faible, ou encore le langage de programmation parallèle CUDA stipule l'utilisation d'un modèle de consistance relâchée proche d'un modèle de type libéré pour l'utilisation sur des processeurs graphiques.

4.2.2.2. Gestion de la cohérence de cache

Le deuxième point important à prendre en compte pour la réalisation d'une plateforme à mémoire partagée distribuée, est la gestion de la cohérence de cache. Le problème de la cohérence de cache peut apparaitre lorsqu'il existe plus d'une seul cache de même niveau dans une architecture. Dans ce cas, des lignes de cache venant de caches différents peuvent représenter la même zone mémoire : il faut donc une méthode pour que la modification d'une de ces lignes puisse être visible par les autres.

La Figure 39 illustre ce problème de cohérence de cache par un scénario simple se déroulant sur une plateforme à mémoire partagée où aucune gestion de cohérence de cache ne serait mise en place.

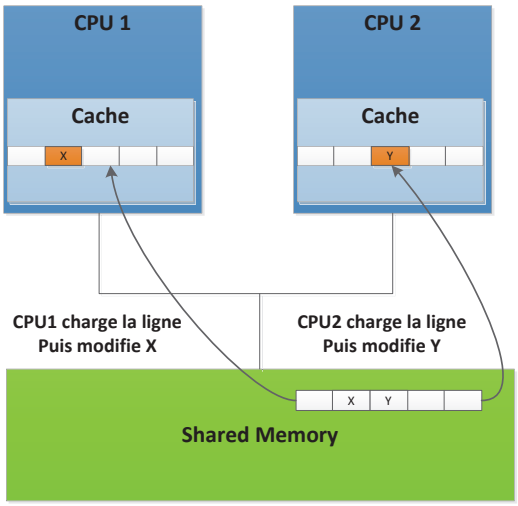


Figure 39 : Scénario mettant en avant un problème de cohérence de cache

Dans ce scénario, le CPU1 possède une tâche utilisant la donnée X, et la modifiant. On a donc, à un moment donnée, la ligne contenant X qui est chargée dans le cache du CPU1. Sur le CPU2, une tâche a besoin d'utiliser la même ligne de cache que sur le CPU1, soit pour accéder à la valeur X, soit pour accéder à une valeur Y située sur la même ligne. Plusieurs problèmes peuvent apparaitre alors :

- Si le CPU1 modifie la valeur de X, et que celle-ci est lue par le CPU2, alors la valeur lue sera fausse, car la ligne de cache chargée par le CPU2 l'a été avant cette modification.
- Si le CPU1 modifie la valeur de X, et que le CPU2 modifie la valeur de Y, alors aucune des deux lignes de cache ne possèdent l'ensemble des valeurs X et Y correctes : suivant l'ordre dans lequel elles vont être remise en mémoire, soit la valeur de X, soit la valeur de Y sera incorrecte.

Par pallier ce problème, des mécanismes de cohérence de cache ont été mis en place. Ceux-ci peuvent être divisés en deux catégories [101]: les mécanismes matériels, qui permettent de rendre la cohérence de cache transparente pour le logiciel, et les mécanismes logiciels, où des commandes logicielles (par exemple l'invalidation ou le vidage d'une ligne) viendront aider les mécanismes matériels à traiter la cohérence de cache.

Il faut noter que le problème de la cohérence de cache est intrinsèquement lié au problème de consistance mémoire. En effet, une consistance mémoire forte va entrainer le besoin de mise à jour des données en cache plus régulière, pour respecter les contraintes de ce modèle.

4.2.2.2.1. Mécanismes de cohérence de cache matériels

Parmi les solutions les plus utilisées actuellement pour réaliser une cohérence de cache de manière matérielle on retrouve la cohérence de cache de type espion (*snooping cache*

coherency). Le principe de ce mécanisme est, pour l'ensemble des caches, de venir « espionner » les opérations des autres caches, afin de se mettre à jour en cas de besoin. Ce mécanisme, bien qu'utilisable pour différents types d'architectures, est particulièrement pratique avec une architecture organisée autour d'un bus mémoire. En effet, pour que chaque cache puisse espionner les transactions, deux conditions sont à respecter : tout d'abord, il faut que toutes les transactions soient visibles par l'ensemble des caches, ensuite, il faut que l'ordre de ces transactions soit le même pour toutes.

Dans son principe le plus simple [102], le protocole espion utilise des caches en mode écriture directe (*write-through*) venant espionner chaque transaction d'écriture sur le bus. Si une écriture se fait sur une donnée située dans un des caches, alors soit cette donnée est modifiée (un protocole de type mise à jour), soit la ligne de cache est invalidée (un protocole de type invalidation).

D'autres protocoles plus complexes, tel que les protocoles MSI, MESI, MOESI de Intel, ou encore le protocole Dragon, ont été élaboré afin d'éviter d'avoir des caches à écriture directe [102]. Les protocoles MSI, MESI et MOESI par exemple, utilisés par Intel dans leurs architectures de processeurs généralistes, font appel à plusieurs types d'états d'une ligne de cache (*Mutual, Owned, Exclusive, Shared, Invalidate*), permettant de savoir si d'autres caches possèdent ces données.

L'inconvénient majeur de ces protocoles est qu'il nécessite une visibilité de l'ensemble des transactions par l'ensemble des caches, et que celle-ci soient ordonnées, ce qui est une contrainte majeure dans le cas de l'utilisation d'un Réseau sur Puce. En effet, de par sa structure, la diffusion des transactions à l'ensemble des nœuds va à l'encontre des avantages de l'utilisation d'un Réseau sur puce, notamment la scalabilité.

Le second type de protocole matériel le plus connu est le protocole basé sur dossier (*folder-based protocol*). Le principe de ces protocoles est qu'un dossier va venir tenir l'historique des accès à la mémoire. Ainsi, lorsqu'une lecture ou une écriture en mémoire doit être faite par l'un des caches, celui-ci va d'abord lire ce dossier afin de savoir quels autres caches possèdent ces données, et va ensuite agir en conséquence.

Les protocoles basés sur dossier son plus avantageux en terme de scalabilité par rapport aux protocoles d'espionnage. En effet, les dossiers peuvent être distribués en fonction des mémoires, et on peut ainsi limiter le nombre d'accès à un même dossier. De nombreux projets exploitant un protocole de cohérence de cache basé sur dossier ont vu le jour : on peut notamment citer le projet DASH [103], faisant appel à une version amélioré du protocole basé sur dossier, où une couche logicielle de gestion permettait d'augmenter les performances, ou encore celui du projet TSAR [66], mettant en place un protocole de cohérence basé sur des dossiers distribués, mixé avec une méthode d'espionnage.

Cependant, ces protocoles possèdent d'autres défauts, comme la nécessité d'attendre la réponse du dossier, ainsi que potentiellement celle des différents caches possédant la donnée partagée avant d'effectuer une modification, ce qui peut entrainer de très longues latences.

4.2.2.2.2. Mécanismes de cohérence de cache logiciels

Le principe de la cohérence de cache logicielle est d'utiliser des indications dans le code (soit fournies par le programmeur, soit à travers un mécanisme de pré-compilation) pour invalider et/ou vider une ou plusieurs lignes de cache. De nombreux protocoles de cohérence de cache logiciels ont été réalisés au cours des années 90, dans le cadre de développement de plateformes hautes-performances [104]. Depuis lors, peu de papiers font état de nouveaux protocoles de cohérence de cache logiciels, l'accent étant mis sur des protocoles matériels.

On peut toutefois citer les travaux de F. Pétrot [105], qui propose une solution logicielle permettant d'éviter les problèmes de cohérence de cache, en mettant les données partagées dans une zone mémoire non cachée. Les travaux de J.W. Van Den Brand [106] peuvent également être cités, proposant un protocole logiciel simple pour un modèle de consistance mémoire de flux de données. Tartalja et al. [107] propose plusieurs protocoles de cohérence de cache logiciels basés sur un gestionnaire de version des données partagées. Dans [108], un système de zones d'accès à la mémoire partagée est proposé. Avec ce protocole, le programmeur doit définir des zones d'entrée et de sortie d'accès à une mémoire partagée (zones mutuellement exclusives). Lors de l'entrée ou de la sortie de ces zones, un protocole de mise à jour/invalidation des caches est effectué pour maintenir la cohérence.

L'inconvénient majeur de ce dernier protocole est que pour chaque région partagée, il faut maintenir une table du statut de cette région pour chaque processeur : soit la région est en cache et valide, soit elle n'est pas en cache ou invalide. Ce statut qu'il est nécessaire de garder pour chaque région et chaque processeur ajoute un coût en terme d'espace mémoire non-négligeable. De plus, ce système est peu scalable.

4.2.3. Implémentation sur OpenScale

4.2.3.1. Gestion de la cohérence

La méthode de gestion de la cohérence de cache choisie par notre système s'inspire du principe de zones d'accès à une mémoire partagée. Ce protocole s'inspire de celui proposé dans [109] : le principe est d'utiliser les barrières d'acquisition et de libération de la mémoire libérée pour gérer le problème de cohérence de cache. Les protocoles mis en place lors d'une acquisition ou d'une libération sont donnés dans l'Algorithme 3.

```
Acquisition(A) :
    while not(Bloquer(A)) do
        skip
    done
    Invalider_zone(Addresse(A),Taille(A))

Libération(A) :
    Vider_zone(Addresse(A),Taille(A))
    Débloquer(A)
```

Algorithme 3 : Protocoles d'acquisition et de libération d'une zone partagée

Pour gérer l'exclusivité de l'accès à une mémoire partagée, une mémoire verrou a été implémentée. Cette mémoire a pour particularité de ne pas posséder un mécanisme de *lecture* et *d'écriture*, mais un mécanisme de *blocage* et de *déblocage*. L'équivalent d'une lecture pour cette mémoire est un blocage : cela consiste à lire le bit présent en mémoire, le mettre à un, et retourner la valeur qui a été lue (avant la mise à un). L'équivalent d'une écriture est un déblocage : cela consiste à remettre la valeur du bit adressé à 0. Ce système, est utilisé dans la plupart des architectures de processeur pour entrer dans une section critique. Celui-ci permet d'assurer qu'un seul élément peut accéder aux données partagées. Pour assurer le bon fonctionnement du système, la mémoire verrou est non-cachable. Après l'entrée en section critique, la zone du cache qui va être utilisée pour A doit être invalidée, afin que les données soient chargées depuis la mémoire.

Lors de la libération, la zone du cache utilisée pour A doit être vidée, afin de mettre à jour la mémoire. Lorsque cette mise à jour est effective, on peut alors débloquer le verrou associé à A, afin de sortir de la section critique.

Cette méthode diffère de celle proposé dans [109] par le fait de n'utiliser qu'une invalidation lors de l'acquisition, contrairement à une invalidation et un vidage, permettant ainsi un gain en performance. De plus, le contexte dans lequel cette méthode était appliquée était légèrement différent : l'architecture proposée dans [109] était une architecture composé de deux cœurs ARM9 possédant un cache de niveau 1 à 4 voies et un gestionnaire mémoire (MMU), connecté à travers un Réseau sur Puce à deux mémoires partagées.

Un tel protocole assure d'avoir une zone partagée toujours à jour après l'acquisition et la libération de celle-ci. C'est alors au programmeur d'assurer la cohérence de son programme, en mettant des barrières de synchronisation (acquisition ou libération) aux bons endroits dans son code. De plus, celui-ci a la liberté de la granularité de chaque zone partagée : une zone partagée de grande taille va permettre de diminuer le nombre de verrous et la complexité du code, mais va en contrepartie augmenter le nombre d'acquisitions/libérations d'une celle-ci (problème de faux partage). La granularité minimum d'une zone reste la ligne de cache : en effet, OpenScale ne permet pas de ne réécrire qu'une partie d'une ligne de cache, et donc d'éviter le problème de cohérence de cache décrit Figure 39.

4.2.3.2. Coût de l'implémentation

L'implémentation d'un système de gestion de cohérence comme proposé précédemment demande des modifications à la fois matérielles (création d'une mémoire verrou), et logicielle (implémentation des barrières d'acquisition et de libération).

L'ajout du module de mémoire verrou est présenté Figure 40. Afin d'assurer le bon ordre des modifications entrainées par une barrière de libération (voir Algorithme 3), il faut que les effets du vidage du cache soient effectifs avant la libération du verrou. Pour se faire, chaque mémoire verrou d'un NPU sera associée à la mémoire partagée de ce même NPU, et placé sur le même bus. Ainsi, comme ni le réseau sur puce, ni le bus ne permettent de réordonnancement des requêtes, on peut assurer que les modifications sur la mémoire partagée (demandée avant) seront effectuées avant celle sur le verrou.

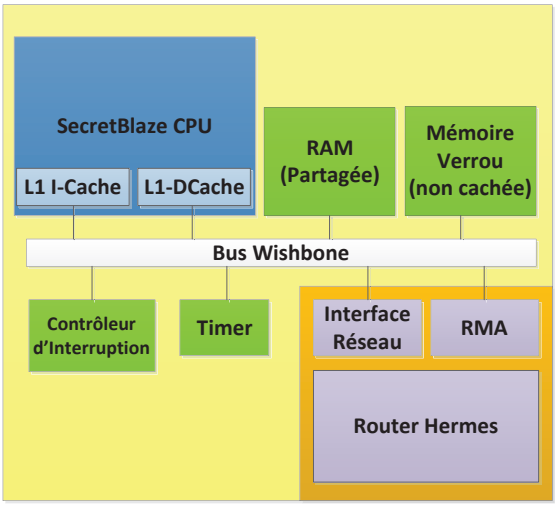


Figure 40 : Architecture d'un NPU supportant une mémoire partagée cohérente

Dans l'architecture d'OpenScale, la taille de cette mémoire verrou est personnalisable. Cependant, du fait qu'une zone de mémoire partagée fait au minimum la taille d'une ligne de cache, on peut en déduire une taille maximale de la mémoire verrou par l'équation 1 :

$$S_{verrou} = \frac{S_{mémoire\ partagée}}{8 * nb\_octets_{ligne\ de\ cache}} \quad (1)$$

Ainsi, pour une mémoire partagée de 128 Ko et une ligne de cache de 4 mots de 32 bits (configuration standard prise lors des mesures de performance), la taille de la mémoire verrou est de 512 octets. En implémentant la mémoire verrou, on a donc une augmentation de la taille de la mémoire utilisée de l'ordre de 0,4%.

En ce qui concerne le coût logiciel, l'implémentation en termes de nombre d'instructions est très faible : les barrières d'acquisition et de libération ne comportant que peu d'instructions. En ce qui concerne le coût en temps d'exécution, celui-ci dépend de la taille de la zone mémoire. La Figure 41 donne un aperçu du nombre de cycles minimum

nécessaire pour exécuter une barrière d'acquisition ou de libération, en fonction de la taille de la zone mémoire partagée concernée.

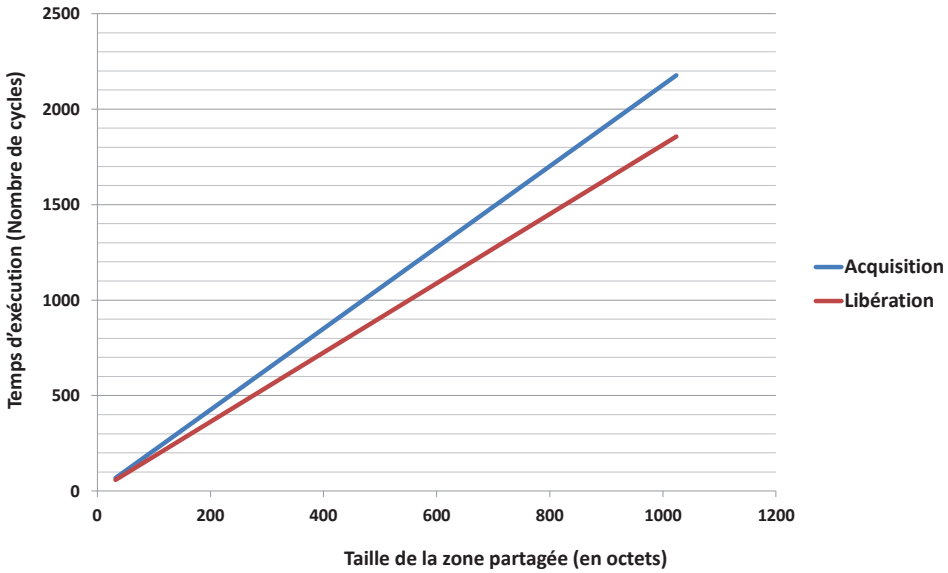


Figure 41 : Temps d'exécution d'une acquisition ou d'une libération en fonction de la taille de la zone partagée

Ce temps d'exécution est linéaire en fonction de la taille de la zone partagée : en effet, il est constitué d'un temps fixe lié à l'acquisition du verrou, puis d'un temps proportionnel à la taille de la zone partagée, lié au vidage et à l'invalidation de celle-ci.

4.2.4. Implémentation de la libraire POSIX-Thread

Le standard POSIX-Thread, aussi appelé PThreads est un des plus utilisés pour la programmation multithread [21]. Celui-ci propose une Interface de Programmation (*Application Programming Interface, API*) standard, qui a donné lieu à la création de librairies, appelée par abus de langage librairies PThreads, compatibles avec ce standard, pour de nombreuses architectures. Du fait de sa notoriété, l'implémentation de la librairie PThreads au sein d'OpenScale s'est avéré un choix évident pour faire la démonstration des capacités de cette architecture dans un contexte de programmation multithread.

Dans le standard PThreads, aucune indication n'est faite quant à la consistance mémoire à utiliser. Seules quelques indications nous laissent supposer qu'utiliser une consistance faible ou libérée est préférable [110] :

« Les applications doivent être réalisées telles que l'accès à emplacement mémoire, quel qu'il soit, par plus d'une tâche soit restreint, de manière à ce qu'aucune tâche ne puisse lire ou modifier cet emplacement mémoire durant le temps où une autre tâche peut potentiellement la modifier. Cette restriction d'accès est réalisée à l'aide des fonctions qui synchronisent l'exécution des tâches, mais aussi qui synchronisent la mémoire. Les

fonctions suivantes synchronisent la mémoire : pthread_mutex_lock(), pthread_mutex_unlock(), … »

Ainsi, dans le standard PThreads, les barrières de synchronisations peuvent se faire sous forme de *mutex*. On peut donc considérer les fonctions de *verrouillage* d'un mutex comme une *acquisition*, et les fonctions de *déverrouillage* d'un mutex comme une *libération*. L'inconvénient de cette méthode et que, dans le standard PThreads, aucune indication sur l'emplacement mémoire protégé par le mutex n'est donnée : on ne sait donc pas quelle zone mémoire nécessite l'application de l'algorithme de cohérence de cache. Deux possibilités s'offrent donc :

- Soit l'on considère que toute la mémoire peut être concernée par le mutex : il faut donc vider et invalider l'ensemble du cache lors de tout verrouillage (i.e. d'acquisition), et vider l'ensemble du cache lors de tout déverrouillage (i.e. de libération). Cette méthode a l'avantage de respecter entièrement la norme PThreads, et donc d'être compatible avec tous les programmes suivant cette norme. Cependant, cette méthode est très coûteuse en termes de temps de calcul, car tout le cache est vidé (et invalidé dans le cas d'un verrouillage) pour tout appel à un mutex.
- La seconde méthode consiste à modifier légèrement la librairie PThreads, en ajoutant dans la structure d'un mutex l'adresse et la taille de la zone mémoire concerné par celui-ci. Cette méthode, bien que demandant une modification de la librairie PThreads, et donc des programmes utilisant cette librairie, permet d'obtenir des performances en termes de calcul bien supérieures qu'avec la solution précédente. En effet, le fait de ne vider (et potentiellement invalider) qu'une zone spécifié du cache au lieu de tout le cache permet d'obtenir des temps de calcul bien inférieur (voir Figure 41).

La deuxième solution a donc été celle adoptée pour l'implémentation de la librairie PThreads sur OpenScale.

4.3. Performances

4.3.1. Coût d'exécution des fonctionnalités PThreads

La méthode de conception des fonctionnalités PThreads utilisée ici, basée sur une implémentation matérielle et logicielle de la gestion de la cohérence mémoire, permet de souligner les compromis entre performance et implémentation. En effet, pour une gestion optimale de la cohérence, il est nécessaire d'évaluer, à chaque appel à une fonction PThreads, quelles lignes de cache doivent être invalidées et/ou vidées. Plus ces informations sont précises, plus les performances seront bonnes. En revanche, pour avoir ces informations, il faut soit les fournir au niveau utilisateur, soit les obtenir à travers une

implémentation (logicielle ou matérielle) complexe. Les choix d'implémentation des principales fonctions PThreads concernant ce compromis sont donnés ci-après :

- ***pthread_create*** : lors de la création d'un nouveau thread, l'ensemble des données du thread créateur peuvent potentiellement être utilisé par le thread créé. Il est donc nécessaire de vider l'ensemble du cache de données du thread créateur, et d'invalider l'ensemble du cache de données du thread créé.
- ***pthread_join*** : lors du retour de l'exécution des threads créés vers le thread principal, il est nécessaire de vider l'ensemble du cache du thread finissant son exécution, et d'invalider l'ensemble des données partagées du cache du thread principal.
- ***pthread_mutex_lock*** : lors du verrouillage de données par un mutex, il est nécessaire d'invalider les lignes correspondantes à ces données, afin de s'assurer la lecture de la dernière version des données concernées. Pour éviter l'invalidation de tout le cache, un mutex sera lié à une zone de données précise (voir section 4.2.4).
- ***pthread_mutex_unlock*** : lors du déverrouillage de données par un mutex, il est nécessaire de vider les lignes correspondantes à ces données, afin de s'assurer la mise à jour en mémoire principale des données partagées. Pour éviter de vider tout le cache, un mutex sera lié à une zone de données précise (voir section 4.2.4).
- ***pthread_barrier_wait*** : la notion de barrière permet de créer une séquentialité dans l'exécution. Afin d'assurer la cohérence de la mémoire entre chaque point d'arrêt, il est nécessaire de vider et d'invalider les lignes correspondantes à toute la zone mémoire qui aurait pu être modifié par les threads concernés. Pour éviter de vider et d'invalider tout le cache, une barrière sera liée à une zone de données précise.

Il est à noter que les fonctions permettant de vider ou d'invalider le cache ont été réalisées de manière à éviter les problèmes de cohérence, à savoir : une ligne de cache est effectivement vidée uniquement si celle-ci a été modifiée, et une ligne de cache est invalidée si celle-ci correspond effectivement à la zone mémoire que l'on souhaite invalider (le tag de la ligne correspond à celui de l'adresse donnée pour invalidation).

Pour évaluer les coûts de ces différentes fonctions, plusieurs applications synthétiques ont été exécutées, en variant la taille des données partagées ainsi que la taille du cache de données de chaque NPU. Les temps d'exécution de chacune de ces fonctions sont données Figure 42.

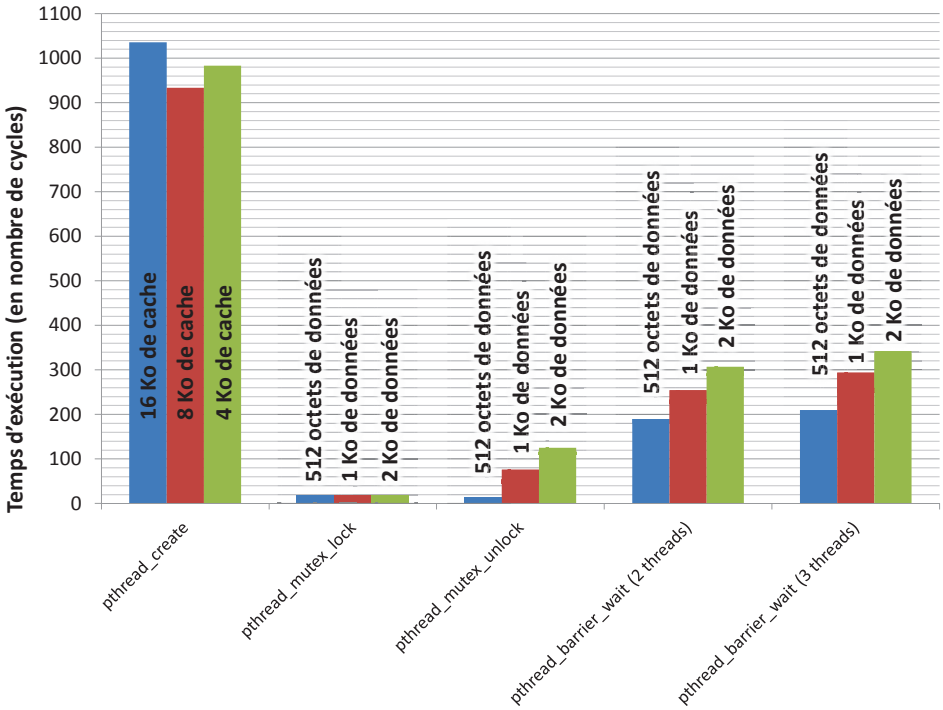


Figure 42 : Temps d'exécution des différentes primitives Pthreads

Plusieurs remarques peuvent être faites concernant cette figure. Tout d'abord, l'exécution de ***pthread_create*** demande un temps d'exécution bien supérieure à n'importe quel autre primitive Pthreads. Ceci peut s'expliquer par deux phénomènes particulièrement chronophages :

- le fait de vider et d'invalider tout le cache
- le protocole de création de la tâche, passant par l'envoi et la réception de nombreux paquets de contrôle à travers le réseau

De ce fait, l'évolution du temps de création d'un thread n'est pas monotone en fonction de la taille du cache : avec un cache de 16 Ko, le vidage et l'invalidation de celui-ci rend la création plus lente, alors qu'avec un cache de 4 Ko, c'est le temps de création qui pénalise les performances. Cette dépendance à la taille du cache reste tout de même faible, avec une variation de l'ordre de 10% au maximum.

L'exécution de verrouillage et déverrouillage de **mutexes** est particulièrement rapide (quelques dizaines à une centaine de cycles). Cependant, lors du verrouillage, ce temps ne prend pas en compte la pénalisation de l'exécution due à la nécessité de recharger les données invalidées.

La synchronisation par **barrière** est plus lente que l'utilisation des mutexes. Ceci est dû au temps nécessaire pour envoyer des messages de synchronisation entre les threads. Comme l'envoi massif de messages au travers d'un réseau sur puce (*broadcast*) est moins efficace que sur une architecture centralisée (de type bus par exemple), l'utilisation de barrières va afficher des performances plus humbles que celles des mutexes. Cette

pénalisation se confirme sur la Figure 42 par la comparaison des temps d'exécution d'une barrière concernant 2 threads, et 3 threads : une barrière de 3 threads aura un temps d'exécution de l'ordre de 12% plus élevé que barrière de 2 threads.

4.3.2. Expérimentation sur des algorithmes classiques

Afin de tester les performances de l'implémentation PThreads sur OpenScale, différentes applications ont été utilisées. L'ensemble de ces applications ainsi que leurs définitions sont donnés Tableau 4.

Nom de l'application	Définition	Temps d'exécution en mono-tâche (16 Ko de cache)	Ratio calcul/chargement de données	Taille du code d'instructions d'un thread
MJPEG	Décodage d'un flux vidéo utilisant la norme MJPEG	5.303.645	5,34	52 Ko
Smith Waterman	Algorithme de détection de séquence	13.243.883	7,08	3,8 Ko
LU	Transformation matricielle	12.520.226	2,48	2,4 Ko
FFT	Transposition en fréquence d'un signal numérique	1.943.406	4,18	4,9 Ko

Tableau 4 : Présentation des applications PThreads testées sur OpenScale

Ces applications ont été choisies pour leur capacité à pouvoir exécuter plusieurs threads identiques en parallèle sur des données différentes. Le temps d'exécution de chaque application en mono-tâche est également donné. Un point important quant aux performances de ces applications lors d'une exécution en parallèle avec mémoire partagée est le ratio entre le nombre d'instruction de calcul et le nombre d'instructions de chargement de données. En effet, les données partagées étant à la fois distantes, et nécessitant une gestion de la cohérence, ce ratio va influencer les performances de l'exécution en parallèle de l'application. Enfin, le nombre d'instructions d'un thread est donné. Ce nombre permet de déterminer la compacité de l'application : l'application MJPEG, par exemple, possède 13.292 instructions, pour un temps d'exécution de 5 millions de cycles. On a donc ici une application possédant peu de récurrences au niveau instructions, et de nombreux appels à la mémoire instructions seront effectués lors de l'exécution. Inversement, les applications Smith Waterman et LU possèdent un nombre

d'instruction très réduit pour un temps d'exécution conséquent : la majorité des instructions vont donc rapidement se retrouver dans le cache.

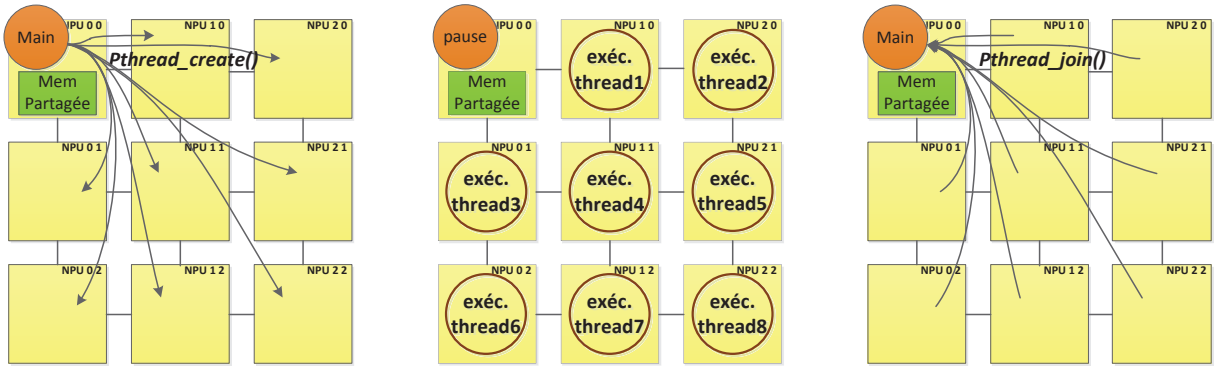


Figure 43 : Schéma du protocole des expériences d'évaluation des performances PThreads

Chaque application a été exécutée sur la plateforme OpenScale avec un réseau de 3x3 NPU, avec un nombre de thread variant de 1 à 8. Ces expériences ont étés réalisées avec des tailles de cache variables, afin de visualiser l'influence de la taille du cache sur les performances du système. Sur la Figure 43, on peut voir le placement de chaque tâche ainsi que de l'emplacement de la mémoire (instructions et données) partagée.

4.3.2.1. Performance du parallélisme à l'exécution

Afin de mesurer les performances de mise à l'échelle de notre implémentation, le temps d'exécution de chaque application en fonction du nombre de thread, et pour différentes tailles de cache, a été calculé.

Pour comparer ces résultats à un système plus classique, les applications de nos tests on également été exécutées sur un architecture multicœur centralisée autour d'un bus. L'architecture de ce système est donné Figure 44.

Ces processeurs exécutent un noyau linux avec une gestion multithread de type SMT (*Symetric Multi-Threading*), avec une cohérence de cache par espionnage des transactions du bus (voir section 4.2.2.2.1). Ce système a été simulé pour 1 à 8 processeurs sous l'environnement GEM5 [111]. Les processeurs utilisés pour cette simulation sont de type Cortex A9, possédant chacun un cache L1 d'instructions et un cache L1 de données de taille paramétrable. Afin de respecter la même configuration mémoire que sur OpenScale, aucun cache L2 n'a été ajouté, et les caches L1 ont été paramétrés pour posséder les mêmes caractéristiques que ceux de notre plateforme.

La Figure 45 montre l'accélération relative de chaque application en fonction du nombre de threads utilisés. Afin de s'abstraire des performances intrinsèques du jeu d'instruction de l'architecture, cette accélération est calculée en fonction du temps d'exécution mono-tâche de l'application, suivant la configuration matérielle considérée (type de processeur, taille des caches, etc.).

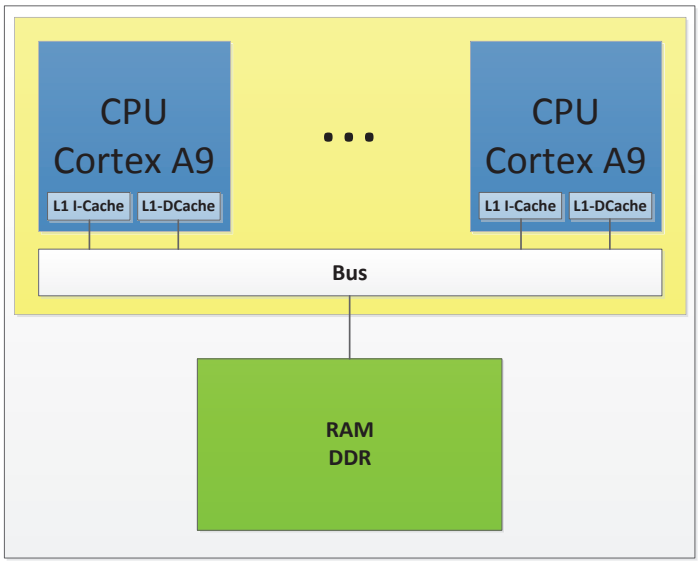


Figure 44 : Système centralisé utilisé en comparaison

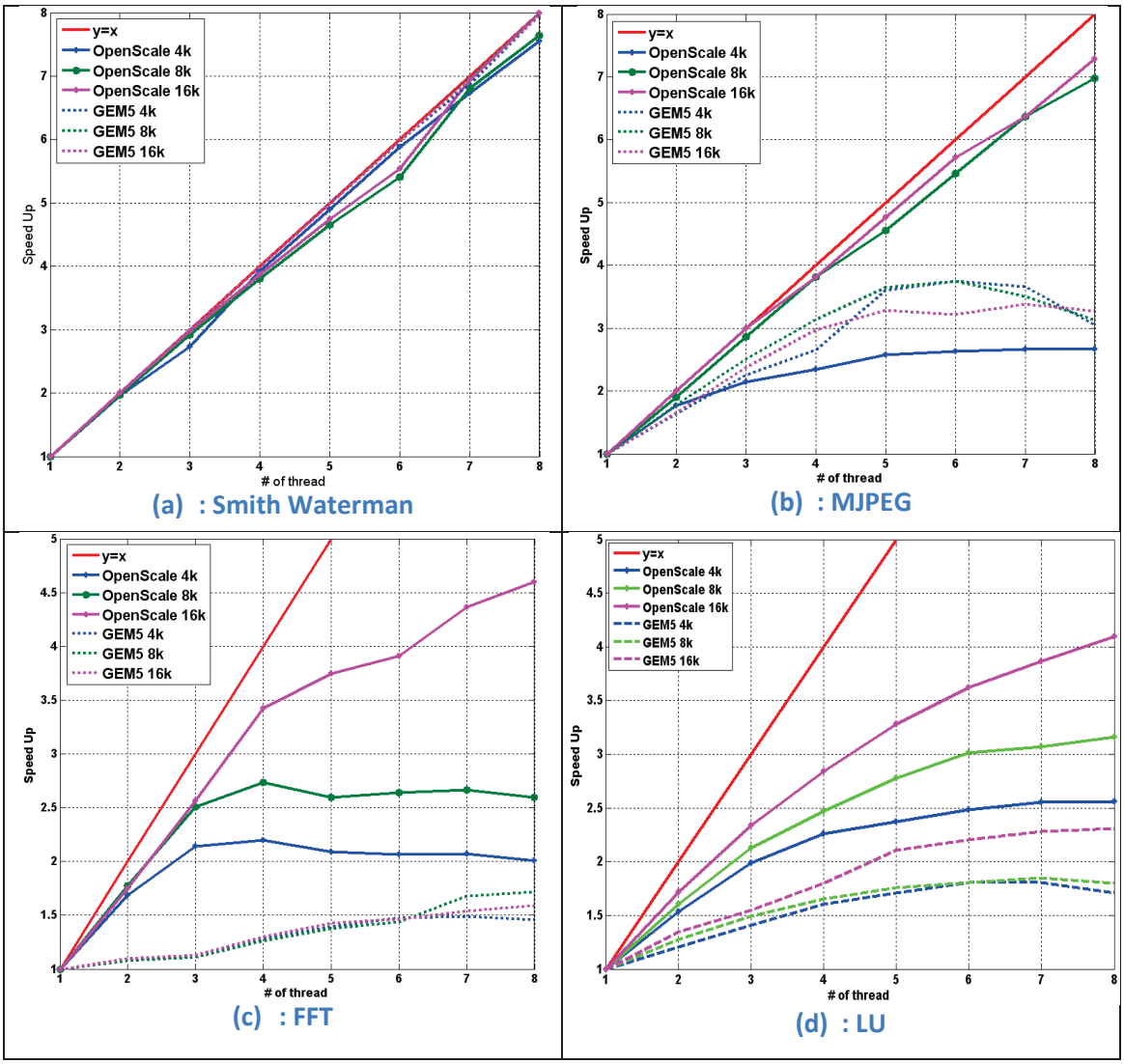


Figure 45 : Accélération du temps d'exécution en fonction du nombre de threads

Cette figure montre l'efficacité de notre implémentation face à un système basé sur un bus, en termes de mise à l'échelle. En effet, l'ensemble des scénarios réalisés sur OpenScale donne des résultats similaires ou meilleurs que ceux réalisés sur la plateforme multicœur.

4.3.2.1.1. Smith Waterman

Dans le cas de l'application Smith Waterman, la compacité du code et les capacités de parallélisme de l'application entraine une mise à l'échelle quasi linéaire des performances. Dans les deux configurations matérielles (OpenScale et la plateforme multicœur), les performances augmentent de manière linéaire en fonction du nombre de thread, ce qui suggère un taux de cache miss d'instructions comme de données particulièrement bas.

De plus, comme l'application ne nécessite aucune dépendance de données, le parallélisme de celle-ci est idéal.

4.3.2.1.2. MJPEG

Dans le cas de l'application MJPEG, la mise à l'échelle des performances est linéaire pour l'architecture OpenScale configuré avec 8 Ko et 16 Ko de cache : avec un nombre de thread doublé, l'application est deux fois plus rapide. Cependant, dans le cas d'un cache de 4 Ko, on voit apparaitre une saturation des performances pour un nombre de thread important. Ce plateau est dû à la saturation des communications vers la mémoire possédant les instructions et les données.

Dans le cas du système multicœur présenté Figure 44, les performances atteignent un maximum aux alentour de 6 threads, et ce, quelle que soit la taille du cache. Ce comportement laisse suggérer une forte saturation des communications vers la mémoire centrale. Comme la mémoire DDR souffre de latences importantes (de l'ordre du millier de cycles), la mise à l'échelle d'une application comme MJPEG, demandant de nombreux accès mémoire est faible.

Ces mesures permettent de mettre en évidence les limitations d'un système centralisé face à un système à mémoire distribué tel qu'OpenScale. En effet, bien que l'ensemble des instructions et des données partagées soient centralisés sur la mémoire du NPU 00, le système d'exploitation s'exécute quant à lui localement, contrairement au cas du système centralisé. De plus la structure de réseau du système de communication permet une bande passante plus importante que celle du bus, et donc une saturation moins rapide de cet élément.

4.3.2.1.3. FFT

L'utilisation de l'application FFT a permis d'observer le comportement de notre plateforme face à une application faiblement parallélisable. En effet, du fait de la forte dépendance

des données entre elles sur cette application, la mise à l'échelle des performances sur cette application est souvent difficile.

Cette information se confirme en observant les résultats en performance de cette application sur la Figure 45 : la plus grande accélération obtenue sur cette application ne dépasse pas les 4,6.

Cependant, bien que peu élevé, les performances obtenues avec OpenScale sont meilleures qu'avec le système basé sur bus, qui n'atteint qu'une accélération maximum de 1,7.

4.3.2.1.4. LU

L'application LU possède elle aussi des dépendances de données. Celles-ci se manifestent par l'implémentation de barrières de synchronisation au milieu et à la fin de chaque itération de la boucle d'exécution principale.

Ces synchronisations vont là encore impacter la mise à l'échelle des performances de l'application, qui ne permet une accélération de l'application que d'un facteur maximum de 4,1. Cette faible scalabilité est due à deux paramètres distincts : premièrement, les dépendances de données, qui entrainent une augmentation des invalidations et des vidages de cache, mais aussi la gestion de la synchronisation. Ainsi, si la plateforme OpenScale, du fait de son réseau sur puce, est capable de mieux appréhender les nombreux défauts de caches par rapport à un système basé sur bus, la synchronisation de l'exécution est plus difficile, du fait de la nécessité d'envoi de messages à de multiples destinateurs (broadcast).

4.3.2.2. Evaluation des transactions mémoires

Les performances en termes de parallélisme de ce système sont liées aux transactions mémoires : plus un programme va faire de transactions vers la mémoire partagée (à cause d'un défaut de cache ou d'une synchronisation), moins la parallélisation sera efficace.

Pour évaluer le comportement des transactions mémoires, le nombre d'occurrence de ces transactions a été mesuré en fonction du temps. En prenant le cas de l'application MJPEG avec 8 threads, la Figure 46 montre le nombre d'accès à la mémoire partagée demandés respectivement par les caches d'instructions et les caches de données, par unité de temps de 500 cycles. L'ensemble des demandes des 8 threads ont été additionnée.

Cette figure permet de constater la saturation du nombre d'accès à la mémoire partagée par le cache d'instruction. En effet, les threads parallèles de l'application MJPEG ont un nombre d'instructions élevé (voir Tableau 4), ce qui oblige le cache d'instruction à réaliser de nombreuses demandes d'accès lorsque celui-ci est faible. A contrario, ces nombreuses demandes d'accès en instruction ralentissent considérablement le temps d'exécution, et

donc le nombre de demandes d'accès par le cache de données. On a donc des occurrences bien plus faibles de nombre d'accès à la mémoire partagée par le cache de données avec un cache de taille de 4 Ko, qu'avec un cache de taille 8 Ko ou 16 Ko.

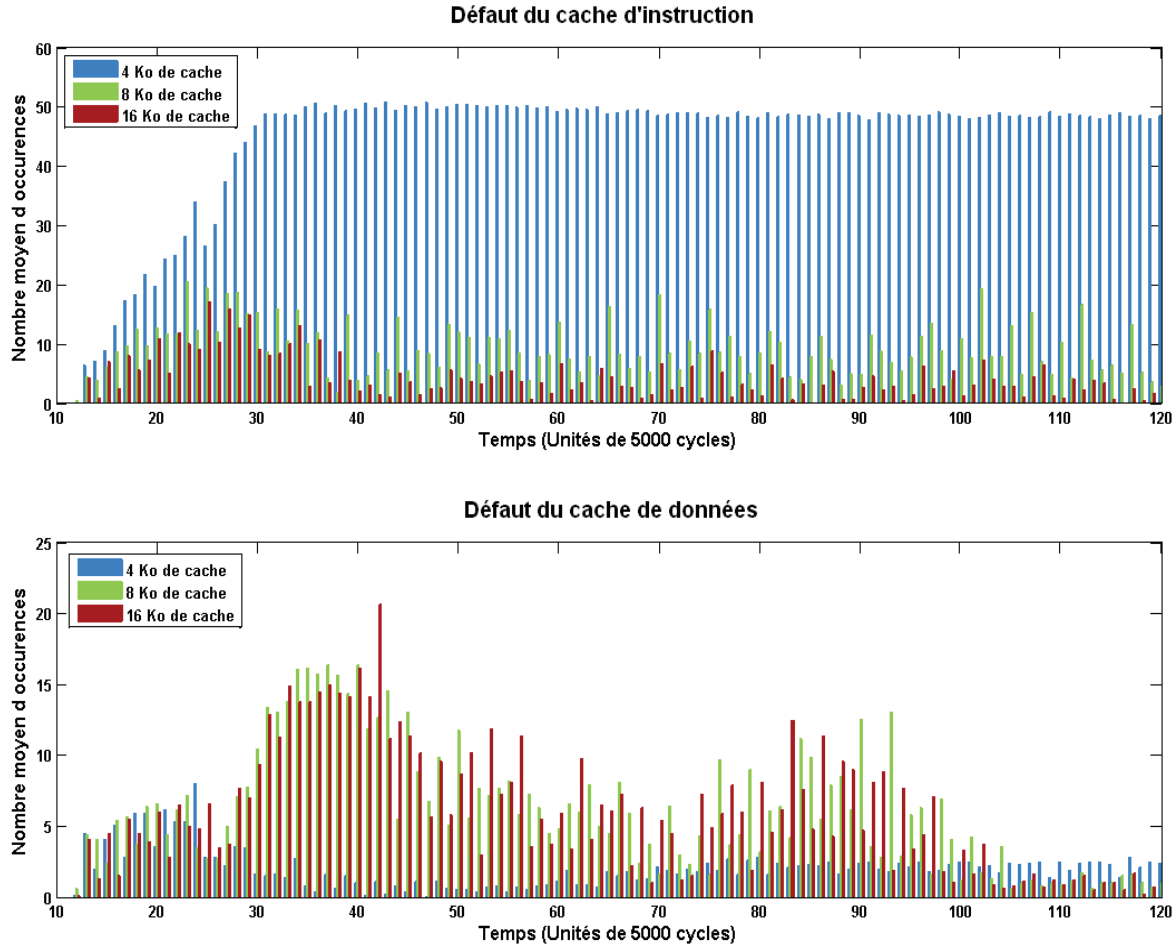


Figure 46 : Nombre moyen de demandes d'accès à la mémoire partagée par intervalle de 5000 cycles pour l'application MJPEG

La saturation s'effectue aux alentour de 50 demandes d'accès à la mémoire partagée sur une période de 5000 cycles, ce qui correspond à une demande en moyenne toute les 100 cycles. Bien que la latence d'un accès mémoire soit de l'ordre de 200 cycles environ (voir Tableau 3), ceux-ci peuvent être effectués en parallèle, grâce à la structure du réseau : si par exemple l'une des requêtes est envoyée sur un des liens, au même instant la réponse d'une autre peut être envoyée, et d'autres requêtes venant de liens physiques différents peuvent également être lancées.

La Figure 47, détaillant les différentes étapes d'un accès distant et des latences associées, permet d'illustrer ce parallélisme : en dehors de la période où le RMA satisfait la requête demandée, l'ensemble des autres étapes peuvent se faire en parallèle d'autres requêtes. Ainsi, seuls les 64 cycles de l'opération du RMA ne sont pas parallélisables, ce qui implique un taux de défaut de cache maximum d'environ 74 pour 5000 cycles.

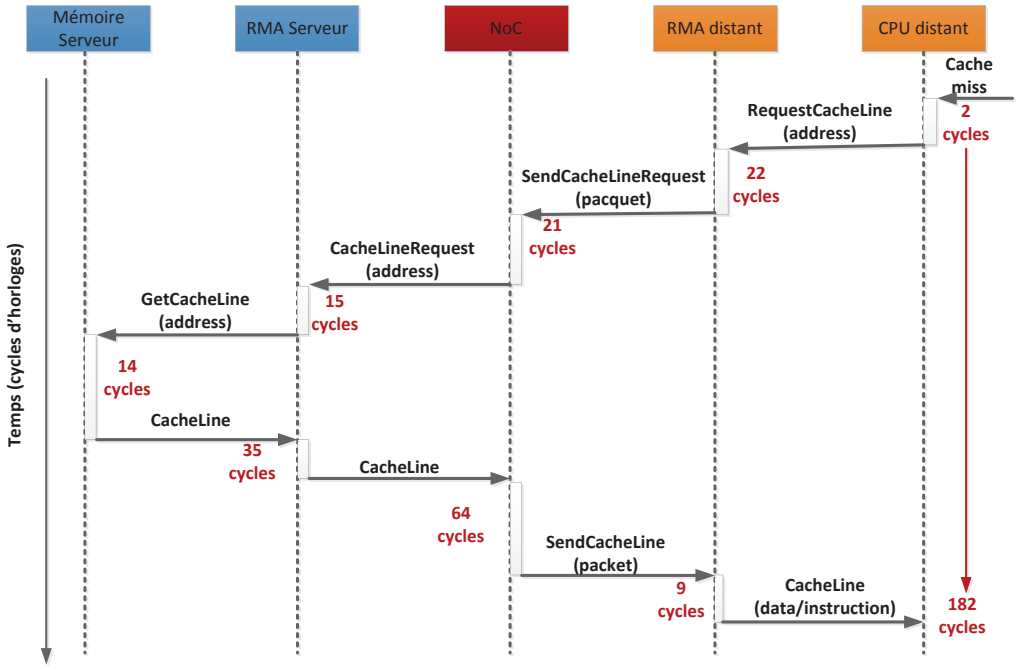


Figure 47: Détails d'un accès distant via le RMA

4.3.2.3. Evolution de la latence des accès

L'évaluation de la saturation des accès peut également se retrouver par la mesure de la latence moyenne d'une requête vers une donnée partagée, en fonction du nombre de threads exécuté en parallèle. Ces mesures sont illustrées sur la Figure 48, pour l'ensemble des quatre applications de notre étude.

La première remarque que l'on peut effectuer et que pour l'ensemble des applications, la latence moyenne augmente en fonction du nombre de threads. Cette évolution se justifie par le fait que l'augmentation du nombre de threads va augmenter la probabilité de conflits entre requêtes (deux requêtes arrivant sur le RMA serveur en même temps), et donc la latence de celles-ci.

La saturation des accès distants peut s'observer par rapport à l'intensité de cette augmentation : dans le cas d'une saturation (MJPEG avec 4Ko, FFT et LU), l'augmentation de la latence des instructions et des données dépasse les 50% lors de l'ajout d'un thread. Dans le cas d'une non-saturation, cette augmentation est beaucoup plus limitée (moins de 10%).

Pour le cas particulier de l'application Smith Waterman, on observe une augmentation significative de la latence des instructions, bien qu'aucune saturation des performances ne soit visible Figure 45. Cette situation s'explique par le fait que l'ensemble des instructions de chacun des threads de cette application est chargé au démarrage de l'exécution. En effet, la compacité de cette application (voir Tableau 4), fait que l'ensemble des instructions d'un thread tient dans le cache d'instruction. La latence moyenne est ainsi

calculée uniquement en fonction des défauts de cache dus au démarrage de l'application. Le fait que la latence des données, chargées tout au long de l'exécution, n'augmente pas de manière significative permet donc de confirmer que la gestion des accès distant ne sature pas.

Ainsi, contrairement à l'observation faite sur la Figure 46, la saturation du nombre d'accès distant (scénarios MJPEG avec 4Ko, FFT et LU) entraine une augmentation de la latence aussi bien sur le cache d'instructions que sur le cache de données : en effet, les requêtes passant par le même lien physique, la saturation affecte autant la latence de l'un que de l'autre.

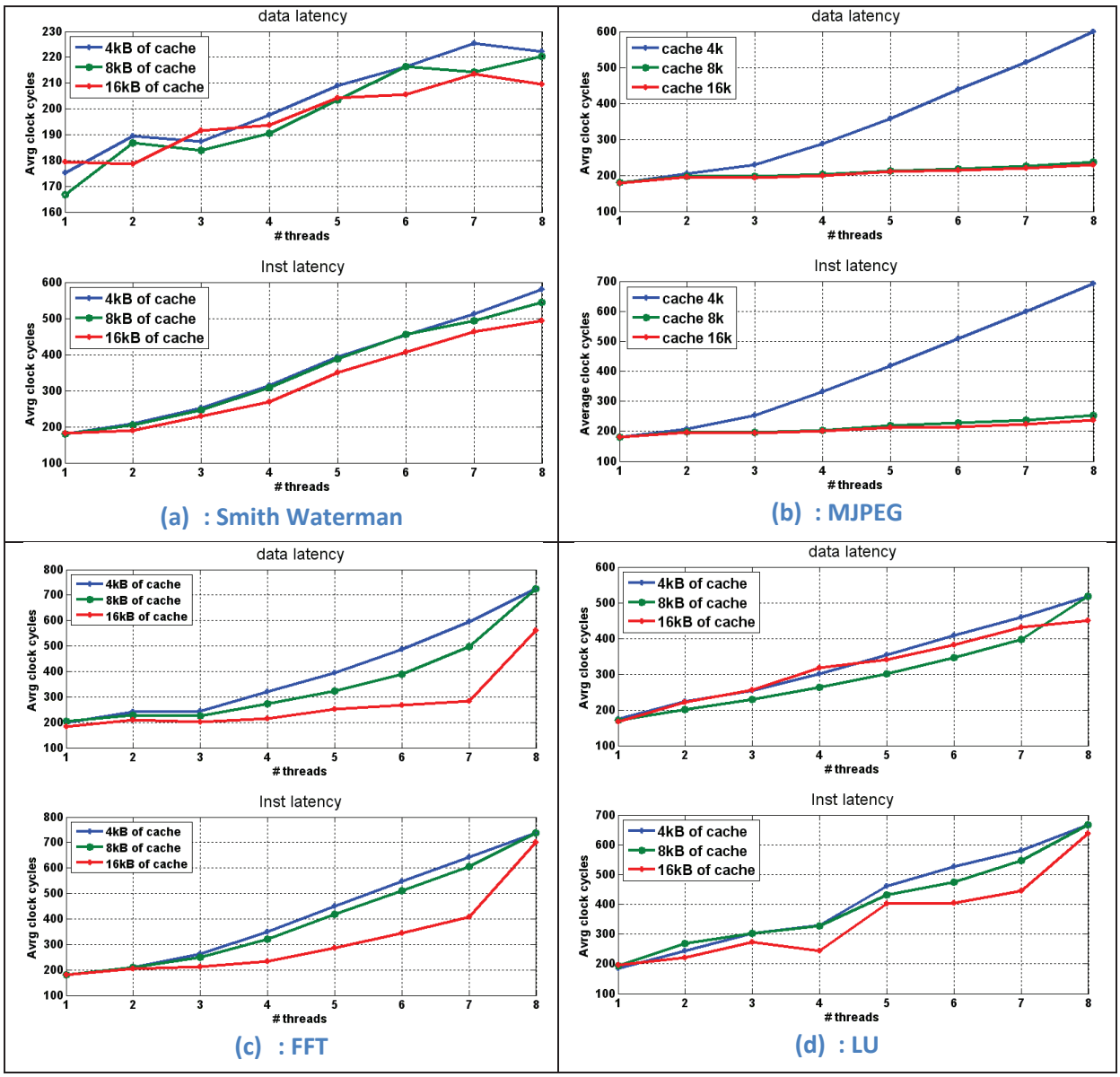


Figure 48 : Evolution de la latence moyenne des requêtes du cache d'instructions et du cache de données en fonction du nombre de thread exécutées en parallèle

Lors des scénarios avec saturation, la latence moyenne atteint des valeurs supérieures à 700 cycles dans le cas de 8 threads s'exécutant en parallèle. Sachant qu'une requête met au minimum environ 200 cycles pour être servie, mais que le service d'une autre requête en même temps augmente cette latence de 64 cycles environ, on en déduit qu'environ (700-200)/64 = 7,8 requêtes en moyenne naviguent en parallèle sur la plateforme à tout instant : on met ainsi en évidence la capacité d'un réseau sur puce face à un bus, capable de traiter uniquement une requête à chaque instant.

4.3.2.4. Bande passante du RMA

Le facteur provoquant la saturation des accès distants est celui de la centralisation des données partagées sur un unique nœud (ici le NPU 00). En effet, le réseau sur puce possédant 12 liens double sens, le nombre moyen de requêtes peut potentiellement dépasser les 7. Cette information peut être confirmée grâce à la Figure 49, affichant le débit de données provenant des requêtes sur les liens Est, Sud, et celui de RMA du NPU 00, pour l'application MJPEG.

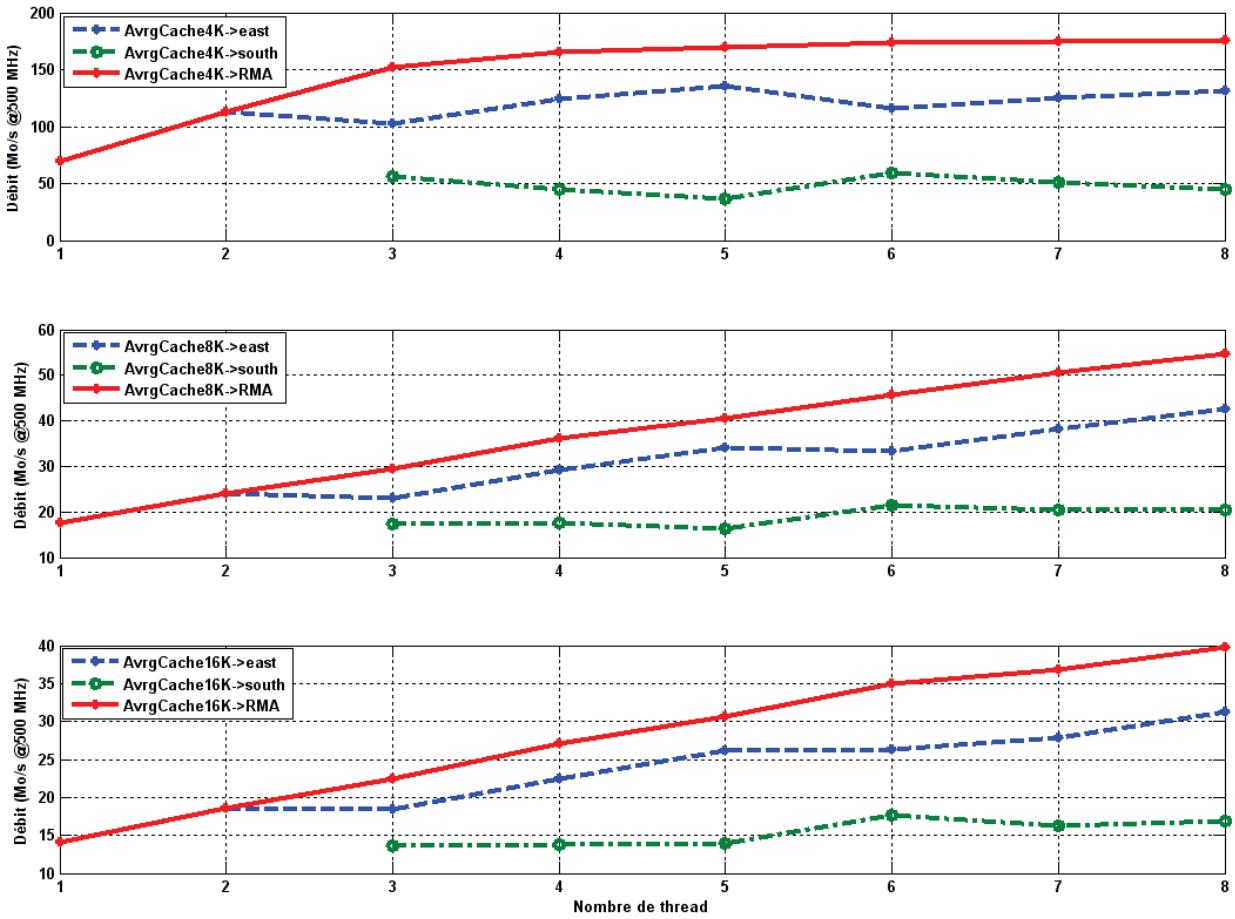


Figure 49 : Débit de données provenant des requêtes sur les liens Est, Sud, et RMA du NPU 00, pour l'application MJPEG

Cette figure permet d'observer la saturation du RMA : alors qu'avec un cache de 8 Ko ou 16 Ko, le débit du RMA augmente quasi linéairement, avec un cache de 4 Ko, le débit atteint un plateau aux alentours de 4 threads, d'une valeur de 175 Mo/s.

4.3.3. Evolution vers un adressage continu

L'implémentation de cette gestion de la cohérence de cache permet donc de tirer pleinement parti de la grande bande passante des architectures basées sur un réseau sur puce. De plus, elle permet d'envisager le partage de toute la mémoire pour l'ensemble des NPU à travers un adressage continu (c'est-à-dire un adressage où l'adresse haute du NPU_{n-1} est contigüe avec l'adresse basse du NPU_n), ce qui entraine deux principaux avantages :

- Tout d'abord, cela permet de réduire la quantité de mémoire nécessaire au fonctionnement de chaque NPU. En effet, dans le cas d'applications nécessitant beaucoup de mémoire (comme l'application MJPEG par exemple), le code instructions et données de l'application peut être réparti sur plusieurs NPU.
- Ensuite, cela permet de limiter la saturation des modules RMA : si le code d'une application est réparti sur plusieurs NPU, alors les accès mémoires venant des différents threads seront également répartis sur plusieurs RMA. On évite ainsi la saturation du RMA qui fut le goulot d'étranglement lors des expérimentations précédentes (voir section 4.3.2.4).

Afin de mesurer les performances d'une répartition de code sur plusieurs NPU, l'application MJPEG a été exécutée sur un système possédant 16 Ko de mémoire partagée, et 4 Ko de mémoire cache. Le code instruction de l'application est ainsi réparti sur 4 NPU. La Figure 50 donne l'accélération relative de temps d'exécution de cette application en fonction du nombre de threads, dans le cas du système multicœur présenté Figure 44, dans le cas de l'architecture OpenScale avec l'application situé sur un seul NPU, et enfin dans le cas de l'architecture OpenScale avec l'application situé sur 4 NPU.

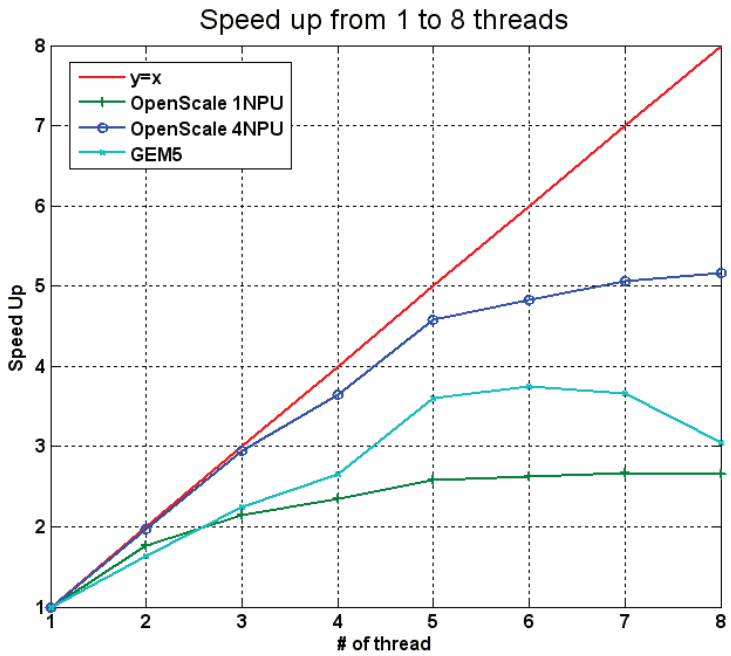


Figure 50 : Accélération de l'application MJPEG en fonction du nombre de threads

Cette figure montre l'avantage d'un code réparti sur plusieurs NPU : l'accélération obtenue est supérieure à celle où le code est situé sur un NPU unique, et dépasse ainsi l'accélération du système basé sur bus.

Afin de visualiser le comportement des différents modules RMA, la bande passante de ceux-ci a été mesurée. La Figure 51 donne le débit de données sur les RMA des NPU 22, 21, 20 et 12, possédant le code de l'application MJPEG.

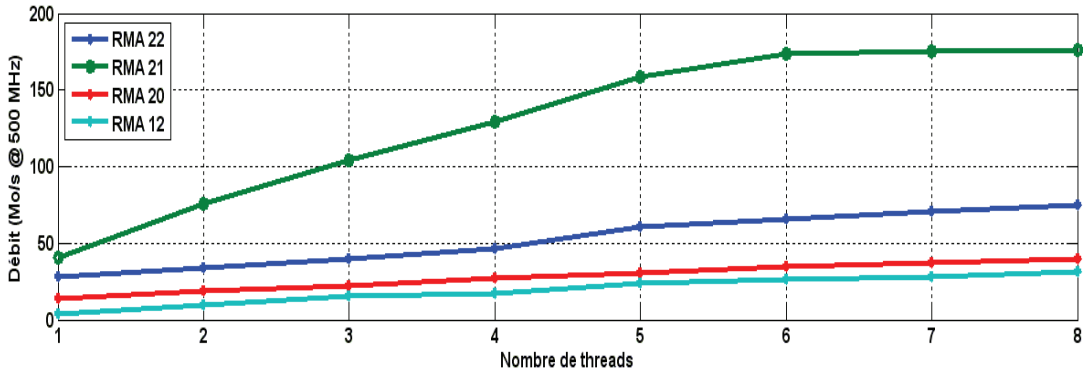


Figure 51: Débit de données provenant des requêtes sur les RMA des NPU 22, 21, 20 et 12, pour l'application MJPEG

On remarque que la saturation au niveau des RMA n'apparait plus que sur le RMA 21, et à partir de 6 threads. On a donc bien augmenté les performances d'exécution en répartissant l'accès à la mémoire partagée. Cependant, cette répartition n'est pas uniforme. Ceci est dû au fait que les requêtes des différents threads ne se répartissent pas uniformément dans l'espace d'adresse de la mémoire partagée.

4.3.4. Conclusion

L'implémentation de la gestion d'une mémoire partagée au sein d'une architecture distribuée comme OpenScale nécessite de concevoir cette gestion comme décentralisée. Les notions de diffusion multiple (*broadcast*) ou d'espionnage de transaction sont à éviter afin d'obtenir des performances convenables, permettant d'exploiter les potentialités du réseau sur puce.

Il est donc nécessaire d'avoir au niveau système une vue précise des données partagées, ainsi que des threads partageant ces données. Dans l'implémentation proposée ici, ces précisions sont données par l'utilisateur, mais des algorithmes automatiques permettent également de réaliser ce travail avec précision (comme celui proposé dans [112]).

Grâce à de telles précisions, cette méthode permet d'obtenir des performances en termes de parallélisation meilleures qu'avec un système centralisé autour d'un bus. On obtient ainsi une architecture de type « mémoire partagée-distribuée », qui permet de cumuler de nombreux avantages :

- Une programmation de type mémoire partagée, ce qui permet d'augmenter la programmabilité de l'architecture
- Une exploitation efficace de la bande passante du réseau sur puce qui permet de fortement diminuer la saturation des accès mémoires
- Une vision globale et uniforme de la mémoire qui permet de répartir le code sur plusieurs nœuds, et ainsi de diminuer la taille mémoire nécessaire au système.

Conclusion et perspectives

Du fait d'une concurrence rude et de barrières technologiques fréquentes, le développement de microprocesseurs fait aujourd'hui face à un nombre important de défis. Ceux-ci se retrouvent à tout les niveaux de la conception : depuis le développement technologique jusqu'au modèle de programmation, en passant par l'architecture matérielle. La parallélisation de l'exécution est devenue l'unique moyen pour atteindre le niveau toujours plus élevé de performances demandé à un processeur. Cependant, la duplication du nombre d'éléments de calcul ne suffit pas pour atteindre des performances plus élevées : la gestion de la mémoire, des communications entre les éléments de calcul, et de la répartition de l'exécution sont parmi d'autres des facteurs nécessitant également un investissement lors de la création d'un nouveau processeur.

Afin de réduire le coût de ce développement, la réutilisation des développements précédents est obligatoire. Plusieurs propriétés du système permettent d'augmenter l'efficacité de cette réutilisation :

- La scalabilité : qui concerne la capacité du système à voir ses performances augmenter proportionnellement au nombre d'éléments de calcul
- L'adaptabilité : qui concerne la capacité du système à optimiser ses performances en temps réel, en fonction des besoins demandés

- La généricité du modèle de programmation : qui concerne la popularité et la simplicité de programmation du système, quel que soit son utilisation.

Ces propriétés sont aujourd'hui peu présentes sur les systèmes multiprocesseurs actuels. Dans le domaine des multicœurs tout d'abord, le modèle de programmation historique, basé sur une mémoire partagée, donne lieu au développement d'architectures à mémoire centralisées, non scalable. Dans le domaine des MPSoCs, le besoin en performances sur des algorithmes spécifiques oblige la création d'architectures hétérogènes, de grande efficacité sur des algorithmes précis, mais peu adaptable et possédant un modèle de programmation personnalisé et souvent complexe. Enfin, dans le domaine de accélérateurs graphiques, ces dernières années ont vu l'apparition de langages de programmation permettant d'utiliser ces architectures dans le cadre d'une exécution d'un programme classique. La popularité de ces langages, tel qu'OpenCL, a permis l'essor de la programmation GP-GPU. Cependant, ces architectures graphiques possèdent également une mémoire centralisée, et une capacité d'adaptation très limitée.

Dans le cadre de cette thèse, ces trois thématiques de scalabilité, adaptabilité et modèle de programmation ont été abordées au sein du développement de la plateforme OpenScale. Historiquement réalisée pour faire face aux besoins de scalabilité et d'adaptabilité, cette plateforme a évoluée vers un système plus modulaire, avec des possibilités d'adaptation et de modèle de programmation plus variées. Ce système est basé sur une architecture homogène, où chaque processeur se situe sur un nœud de calcul différent, relié à travers un réseau sur puce. Chacun de ces nœuds possède sa propre mémoire, et fait tourner un micro système d'exploitation de manière indépendante.

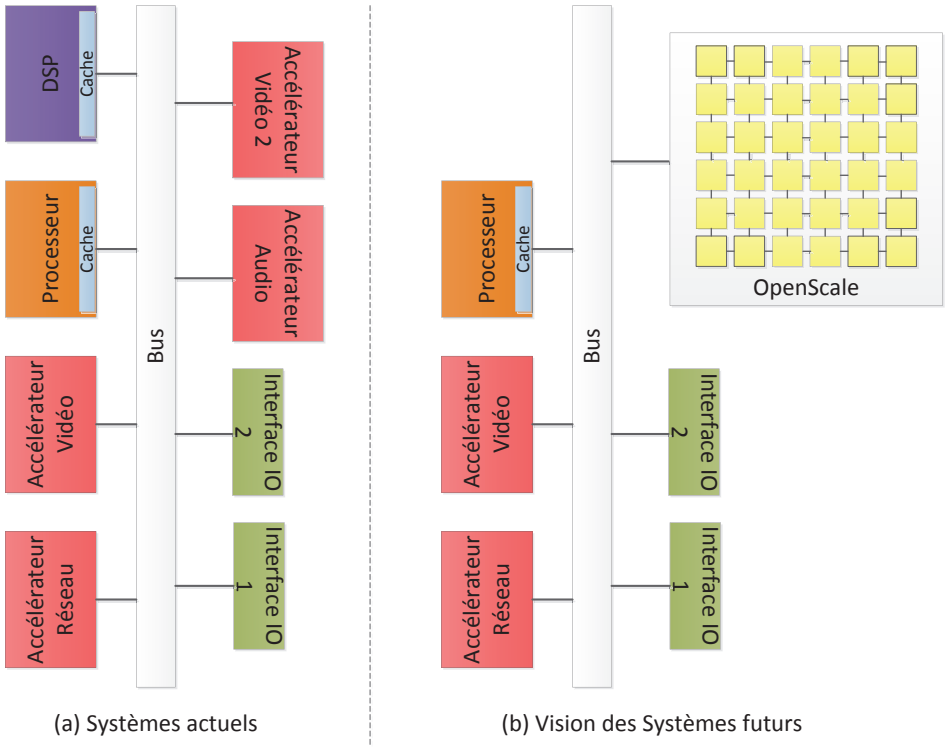
Contribution

On peut noter trois contributions principales dans le cadre de cette thèse :

- Le développement d'OpenScale : une architecture scalable, à mémoire distribuée. L'évolution principale de cette architecture concerne sa modularité : la taille des liens entre les nœuds, les caractéristiques de chaque processeurs ou encore la taille de la mémoire de chaque nœud peuvent être configurées. De plus, le cœur des nœuds de calcul possède maintenant une structure et des performances de l'ordre de celles d'un processeur industriel. Ces éléments ont été exposés dans le **chapitre 2**.
- L'amélioration des possibilités d'adaptation du système, grâce à une gestion des communications et de la mémoire plus évoluée. On peut citer dans ce cadre le développement d'une migration de tâche adaptée aux applications de flux, ou encore une répartition de charge basée sur l'exécution d'instructions situées sur un autre nœud (nommée d'exécution distante). Ces mécanismes ont été étudiés dans le **chapitre 3**.

- L'évolution du modèle de programmation : avec la mise en place d'un protocole de gestion de la cohérence mémoire conservant la scalabilité du système. Celui-ci a permis le développement de la librairie PThreads au sein de l'architecture, permettant une programmation de type mémoire partagée, tout en conservant les avantages de cette architecture. Cette évolution a été décrite dans le **chapitre 4**.

Ma vision personnelle du positionnement de cette plateforme au sein d'un système, étayée par les différentes contributions de cette thèse, est représentée Figure 52. OpenScale est pour moi le développement d'un nouveau type d'accélérateur, dont le but est de prendre en charge la majorité des demandes en calcul. Aujourd'hui, un système actuel est composé d'une multitude d'accélérateurs permettant chacun le traitement d'un besoin particulier. De par sa flexibilité et ses capacités d'adaptation, cette plateforme est capable de gérer la majorité des applications demandées dans les systèmes actuels, et remplacerais ainsi la majorité de ces accélérateurs. Les seuls éléments cohabitant avec cette plateforme seraient le processeur principal ainsi que des accélérateurs très spécifiques pour les demandes les plus gourmandes en calcul. Un tel système permettrait d'économiser en énergie, grâce à la suppression des nombreux petits accélérateurs n'étant plus nécessaires, remplacée par une plateforme adaptable, et en temps de développement grâce à une architecture scalable disposant d'un modèle de programmation standard.



(a) Systèmes actuels (b) Vision des Systèmes futurs

Figure 52 : Vision personnelle de l'évolution des systèmes multiprocesseurs

Perspectives

Différentes perspectives peuvent être envisagées au regard des travaux effectués dans cette thèse.

Tout d'abord, au niveau adaptabilité du système, plusieurs mécanismes d'adaptation ont été démontrés, chacun ayant des caractéristiques propres les rendant plus performants dans certaines situations que dans d'autres. Une perspective à court terme serait la mise en place d'un système de gestion de l'ensemble de ces mécanismes afin de créer une adaptation optimisée à la situation. Plusieurs heuristiques, dépendant notamment des éléments mesurées, pourraient être proposées.

D'autres objectifs tels que la fiabilité pourrait également servir de but dans la création d'heuristiques utilisant les différents mécanismes d'adaptation en fonction du disfonctionnement observé (problème sur la mémoire, problème sur la partie calcul ou problème sur les communications).

Une vision accélérateur

Une des perspectives en cours d'exploration consiste à avoir une vision de la plateforme OpenScale en tant qu'accélérateur matériel. Ainsi, les avantages des langages de programmation GP-GPU pourraient être utilisés pour optimiser les performances de l'architecture.

Si l'on prend le cas du langage OpenCL par exemple, celui-ci possède notamment l'avantage, face aux langages de programmation parallèle classiques tel que PThreads, de fortement décomposer les différents types de données. En effet, lors de la création d'un thread, non seulement le code instruction est séparé de celui du programme appelant, mais les données partagées sont également explicitement définies (en taille comme en type de partage). Ces précisions permettent d'optimiser le placement des données et du code au sein de l'architecture.

Une des idées serait alors d'utiliser ces précisions pour placer au mieux les données au sein de l'architecture, afin d'équilibrer les accès distants entre les RMA. La Figure 53 donne un exemple de placement de données utilisant cette idée.

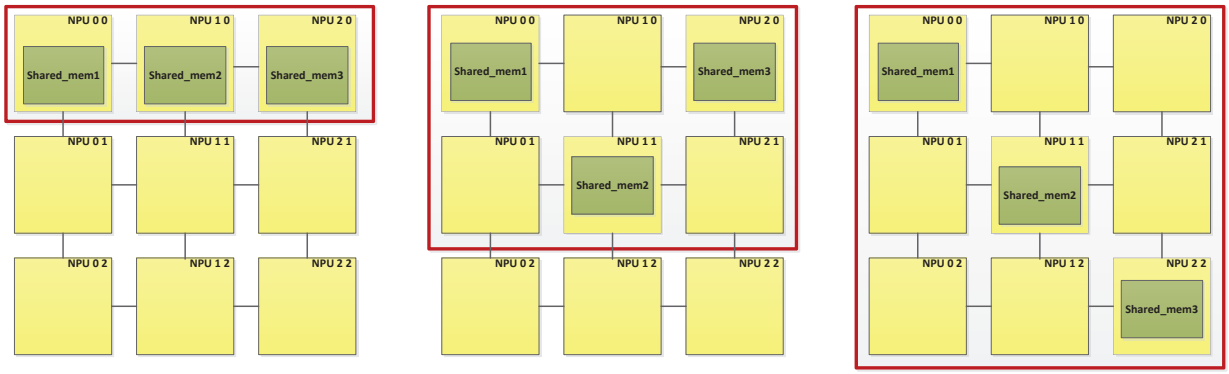


Figure 53 : Placement des données partagées en fonction du nombre de threads

De plus, l'organisation au niveau matériel de la connexion entre Openscale et un système basé sur bus pourrait être explorée, en considérant les avantages d'un réseau sur puce. Une connexion multiple point pourrait par exemple être envisagé, afin de répartir de manière plus efficace les données au sein de l'architecture (voir Figure 53).

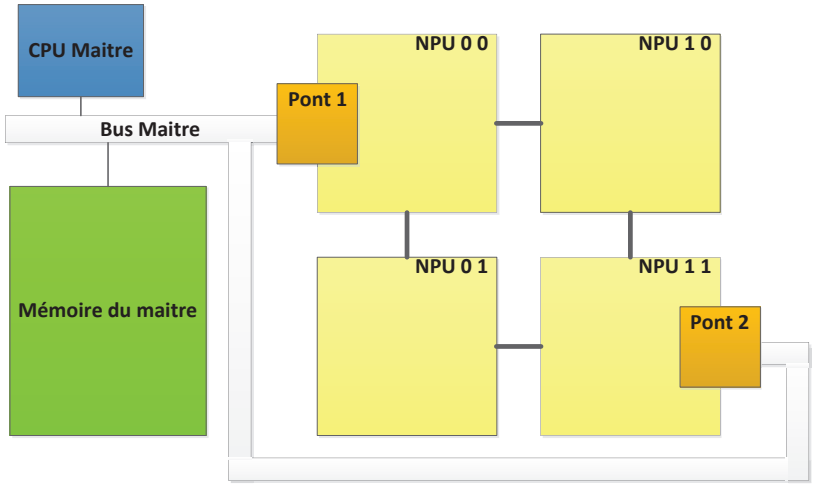


Figure 54 : Système de connexion par deux ponts

Cette vision de la plateforme OpenScale en tant qu'accélérateur matériel permettrait d'augmenter d'autant les performances, la programmabilité et la scalabilité du système. Plusieurs explorations sont d'ailleurs en cours de développement au sein de l'équipe ADAC sur cette thématique.

Perspectives long terme

De nombreuses autres possibilités peuvent encore être envisagées concernant l'utilisation d'OpenScale en tant qu'accélérateur. L'approfondissement des possibilités du langage OpenCL au sein de ce type d'architecture peut par exemple être effectué. Mais le véritable potentiel d'OpenCL serait dans le développement d'une plateforme de programmation unifiée pour systèmes hétérogènes. Comme OpenCL permet de décrire la compilation des

tâches, à partir d'un code écrit en OpenCL, on pourrait envisager son utilisation sur n'importe quel accélérateur. Le système ainsi envisagé est décrit Figure 55.

On pourrait ainsi envisager d'utiliser des mécanismes de répartition de charge ou encore de migration de tâche entre plusieurs accélérateurs, en fonctions des nécessités. L'utilisation de la virtualisation, c'est-à-dire l'utilisation d'un pseudo code précompilé pour décrire les tâches permettrait également leurs exécutions sur n'importe quel accélérateur libre de manière plus rapide (pas de recompilation).

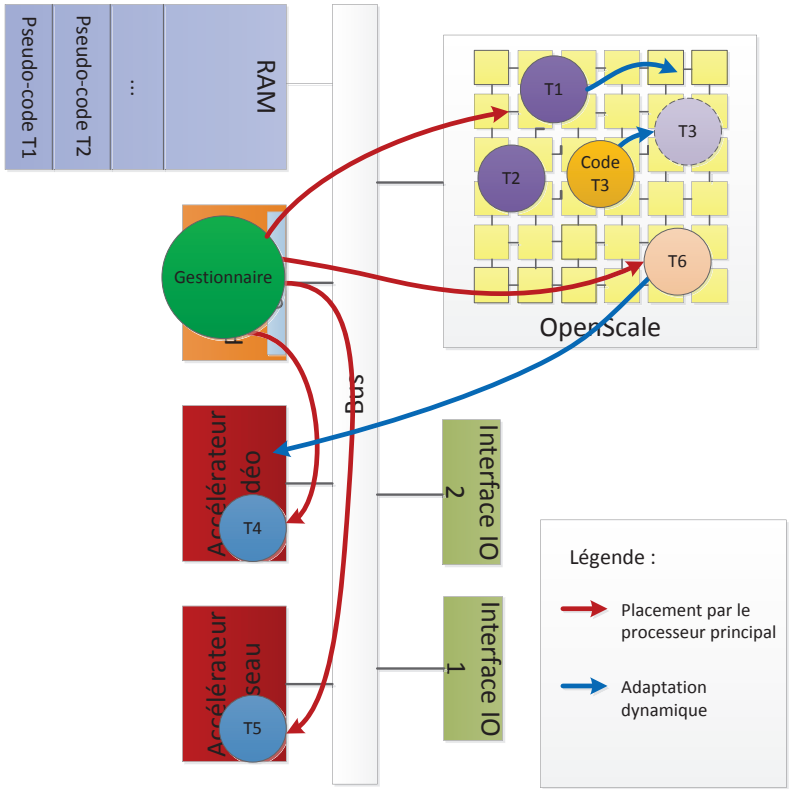


Figure 55 : gestion globale du placement des tâches par l'intermédiaire d'un langage unique

En conclusion, je pense que l'avenir des architectures multiprocesseurs est dans les systèmes modulaires. Devant la complexité grandissante et le temps de développement de plus en plus court, il est nécessaire d'envisager les systèmes au niveau macroscopique (niveau processeur), où l'ensemble de la conception jusqu'à la programmation, en passant par la gestion de la charge serait pris en charge à travers un modèle unifié.

Acronymes

API : *Apllication Programming Interface* (Interface de Programmation)

DFS : *Dynamic Frequency Scaling* (Adaptation dynamique de fréquence)

DSP : *Digital Signal Processor* (Processeur de traitement du signal)

DVFS : *Dynamic Voltage and Frequency Scaling* (Adaptation dynamique de fréquence et tension)

DVS : *Dynamic Voltage Scaling* (Adaptation dynamique de tension)

FFT : *Fast Fourrier Transform* (Transformation de Fourrier Rapide)

FIFO : *First-In First-Out* (File)

FPGA : *Field Programmable Gate Array*

HPC : *High Performance Computing* (Programmation Haute Performance)

IDCT : *Inverse Discrete Cosine Transformation* (Transformation de Fourrier Discrète Inverse)

IQuant : *Inverse Quantization* (Quantification Inverse)

IVLC : *Inverse Variable Length Coding* (Codage à Longueur Variable Inverse)

KPN : *Kahn Process Networks*

MJPEG : *Motion Joint Photographic Experts Group*

MMU : *Memory Management Unit* (Unité de gestion mémoire)

MPI : *Message Passing Interface* (Interface à Passage de Message)

MPSoC : *Multi-Processor System on Chip* (Système Multiprocesseur sur Puce)

NoC : *Network on Chip* (Réseau sur Puce)

NPU : *Network Processing Unit* (Unité de traitement du réseau).

NUMA : *Non-Uniform Memory Access* (Accès mémoire Non-Uniforme)

OS : *Operating System* (Système d'Exploitation)

PE : *Processing Element* (Elément de Calcul)

POSIX : *Portable Operating System Interface for Unix*

PThread : *POSIX Thread*

PID : *Proportional-Integral-Derivative* (Proportionnel-Intégral-Dérivé)

RAM : *Random Access Memory* (Mémoire à Accès Aléatoire, Mémoire Vive)

RTOS : *Real-time Operating System* (Système d'Exploitation temps réel)

SMT : *Symmetric Multithreading* (programmation multithread symétrique)

SoC : *System-on-a-Chip* (Système sur Puce)

UART : *Universal Asynchronous Receiver Transmitter* (Liaison Série)

VoIP : *Voice over Internet Protocol* (Protocole de Communication Vocale par Internet)

Références Bibliographiques

[1] F. Clermidy et al., "MAGALI: A Network-on-Chip based multi-core system-on-chip for MIMO 4G SDR," in *International Conference on IC Design and Technology (ICICDT)*, 2010, pp. 74-77.

[2] G. Q. Zhang, M. Graef, and F. van Roosmalen, "The rationale and paradigm of 'more than Moore'," in *Proceedings of the 56th Electronic Components and Technology Conference*, 2006.

[3] B. Bentley, "Validating the Intel(R) Pentium(R) 4 microprocessor," in *Proceedings of the Design Automation Conference*, 2001, pp. 244 - 248.

[4] W. Wolf, "The future of multiprocessor system on chips," in *Proceedings of the 41st annual Conference on Design and Automation Conference (DAC)*, 2004, pp. 681-685.

[5] L. Gwennap, "MAJC gives VLIW a new twist," *Microprocessor Report*, vol. 22, pp. 12-15, 1999.

[6] J. M. Tendler, J. S. Dodson, J. S. J. Fields, H. Le, and B. Sinharoy, "POWER4 system microarchitecture," *IBM Journal of Research and Developpement*, vol. 46, no. 1, pp. 5-25, 2002.

[7] Advanced Micro Devices, "Multi-Core Processors," *Computer Power User*, vol. 4, no. 12, p. 44, 2004.

[8] M. Magee, "Intel's Roadmaps In Tatters," *Computer Power User*, vol. 11, no. 4, p. 102, 2004.

[9] Cavium Networks, "Cavium networks introduces octeon," 2004. .

[10] Intel Corporation, "TeraScale," 2007. [Online]. Available: http://www.intel.com/content/www/us/en/research/intel-labs-teraflops-research-chip.html.

[11] QUALCOM, "Msm7200a chipset solution," 2006. [Online]. Available: http://pdfserv.datasheetpro.com/.

[12] Texas Instrument, "Omap5432 - chip block diagram," 2011. .

[13] W. Wolf, A. A. Jerraya, and G. Martin, "Multiprocessor System-on-Chip (MPSoC) Technology," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 10, pp. 1701-1713, 2008.

[14] PicoChip, "Pc205 - high performance signal processor," 2008. [Online]. Available: http://www.picochip.com/.

[15] FreeScale, "Msc8156: Six core high performance dsp," 2008. .

[16] M. Henning, "The Rise and Fall of CORBA," *Queue*, vol. 4, no. 5, pp. 28-34, 2006.

[17] M. Saldana and P. Chow, "TDM-MPI: an MPI implementation for multiple processors across multiple FPGAs," in *Field Programmable Logic and Applications*, 2006.

[18] J. Kohout and A. D. George, "A high-performance communication service for parallel computing on distributed dsp systems," *Parallel Computing*, vol. 29, no. 7, pp. 851-878, 2003.

[19] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Transactions on Computer Systems (TOCS)*, vol. 2, no. 1, pp. 39-59, 1984.

[20] R. S. Engelschall, "GNU Pth - The GNU Portable Threads," 2006. [Online]. Available: http://www.gnu.org/software/pth/.

[21] 9945-1I., "The POSIX threads standard," 1996.

[22] OpenMP Architecture Review Board, "The OpenMP® API specification for parallel programming." [Online]. Available: http://openmp.org/wp/.

[23] L. J. Karam, I. AlKamal, A. Gatherer, G. A. Frantz, D. V. Anderson, and B. L. Evans, "Trends in Multi-core DSP Platforms," *IEEE Signal Processing Magazine, Special Issue on Signal Processing on Platforms with Multiple Cores*, vol. 26, no. 6, pp. 38-49, 2009.

[24] B. Chapman, L. Huang, E. Biscondi, E. Stotzer, A. Shrivastava, and A. Gatherer, "Implementing OpenMP on a high performance embedded multicore MPSoC," in *IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, 2009, pp. 23-29.

[25] AMD, "AMD FireSream." [Online]. Available: http://www.amd.com/us/press-releases/Pages/firestream-peak-performance-2010june23.aspx.

[26] Nvidia, "Parallel Programming and Computing Platform | CUDA." [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html.

[27] I. Buck et al., "Brook for GPUs: stream computing on graphics hardware," *ACM Trans. Graph.*, vol. 23, no. 3, pp. 777-786, 2004.

[28] C. Boyd, "DirectCompute: Capturing the Teraflop," *Microsoft Corporation*, 2009. [Online]. Available: http://ecn.channel9.msdn.com/o9/pdc09/ppt/CL03.pptx.

[29] T. Bray, "Introducing Renderscripts," *Android Developpers*, 2011. [Online]. Available: http://android-developers.blogspot.fr/2011/02/introducing-renderscript.html.

[30] J. Reinders, "Rapidmind + Intel - blogs," *Intel Corporation*, 2009. [Online]. Available: http://software.intel.com/en-us/blogs/2009/08/19/rapidmind-intel/.

[31] Kronos, "OpenCL - The open standard for parallel programming of heterogeneous systems." [Online]. Available: http://www.khronos.org/opencl/.

[32] ZiiLABS, "ZiiLABS OpenCL SDK Early Access Program." [Online]. Available: http://www.ziilabs.com/products/software/opencl.php.

[33] J. Leskela, J. Nikula, and M. Salmela, "OpenCL embedded profile prototype in mobile device," in *IEEE Workshop on Signal Processing Systems (SiPS)*, 2009, pp. 279-284.

[34] J. Lee et al., "An OpenCL framework for heterogeneous multicores with local memory," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, 2010, pp. 193-204.

[35] D. Brooks and M. Martonosi, "Dynamic Thermal Management For High-Performance Microprocessors," in *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, 2001.

[36] M. Gligor, N. Fournel, and F. Petrot, "Adaptive Dynamic Voltage and Frequency Scaling Algorithm for Symmetric Multiprocessor Architecture," in *12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools (DSD)*, 2009, pp. 613-616.

[37] J. Donald and M. Martonosi, "Techniques for Multicore Thermal Management: Classification and New Exploration," in *Proceedings of the 33rd annual international symposium on Computer Architecture*, 2006, pp. 78-88.

[38] F. Firouzi, A. Azarpeyvand, M. E. Salehi, and S. M. Fakhraie, "Adaptive fault-tolerant DVFS with dynamic online AVF prediction," *Microelectronics Reliability*, 2012.

[39] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi, "An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget," in *Annual IEEE/ACM International Symposium on Microarchitecture*, 2006, pp. 347-356.

[40] G. Marchesan Ameida, "Architectures Multi-Processeurs Adaptatives: Principes, Méthodes et Outils," Université Montpellier II, 2011.

[41] D. Tam, R. Azimi, and M. Stumm, "Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors," *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, vol. 41, no. 3, pp. 47-58, 2007.

[42] A. Schranzhofer, J.-J. Chen, L. Santinelli, and L. Thiele, "Dynamic and adaptive allocation of applications on MPSoC platforms," in *15th Asian and South Pacific Design Automation Conference (ASP-DAC)*, 2010, pp. 885-890.

[43] C. Yang and A. Orailoglu, "Towards no-cost adaptive MPSoC static schedules through exploitation of logical-to-physical core mapping latitude," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, 2009, pp. 63-68.

[44] H. Javaid, M. Shafique, S. Parameswaran†, and J. Henkel, "LowPower Adaptive Pipelined MPSoCs for Multimedia: An H.264 Video Encoder Case Study," in *16th Asian South Pacific Design and Automation Conference (ASP-DAC)*, 2011, pp. 1032-1037.

[45] A. K. Coskun, T. S. Rosing, and K. Whisnant, "Temperature aware task scheduling in MPSoCs," in *Proceedings of the conference on Design, automation and test in Europe (DATE)*, 2007, pp. 1659-1664.

[46] D. Cuesta, J. Ayala, J. Hidalgo, D. Atienza, A. Acquaviva, and E. Macii, "Adaptive Task Migration Policies for Thermal Control in MPSoCs," in *VLSI 2010 Annual Symposium, Lecture Notes*, 2011, pp. 83-115.

[47] L. Fiorin, G. Palermo, and C. Silvano, "MPSoCs run-time monitoring through Networks-on-Chip," in *Proceedings of the conference on Design, automation and test in Europe (DATE)*, 2009, pp. 558-561.

[48] S. Kumar et al., "A network on chip architecture and design methodology," in *IEEE Computer Society Annual Symposium on VLSI*, 2002, pp. 105-112.

[49] Tilera Corporation, "TILE64 Porcessor," 2007. [Online]. Available: http://www.tilera.com/products/processors/TILE64.

[50] F. Moraes, N. Calazans, A. Mello, L. Moller, and L. Ost, "Hermes: an infrastructure for low area overhead packet-switching networks on chip," *he VLSI Journal*, vol. 38, no. 1, pp. 69-93, 2004.

[51] X. Lin, P. K. McKinley, and L. M. Ni, "Deadlock-free multicast wormhole routing in 2-d mesh multicomputers," *IEEE Trans. Parallel Distrib. Syst.*, vol. 5, no. 8, pp. 793-804, 1994.

[52] L. Barthe, C. L. Vittorio, P. Benoit, and L. Torres, "The SecretBlaze: A Configurable and Cost-Effective Open-Source Soft-Core Processor," in *IEEE IPDPS/RAW*, 2011.

[53] S. Rhoads, "Plasma - most mips (tm)," *OpenCores*. [Online]. Available: http://www.opencores.org/project,plasma.

[54] G. Kahn, "The semantics of a simple language for parallel programming," in *Proceedings of the IFIP Congress*, 1974, pp. 471-475.

[55] O. O. R. Herveille, "Wishbone System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores," *OpenCores*, 2010. .

[56] Xilinx, "MicroBlaze Processor Reference Guide v11.1," *2010*. [Online]. Available: http//www.xilinx.com.

[57] J. Daemen and V. Rijmen, *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer, 2002, p. 238.

[58] N. Ahmed, T. Natarajan, and K. R. Rao, "Discrete Cosine Transform," *IEEE Trans. ofnComputers*, vol. 23, no. 1, pp. 90-93, 1974.

[59] C. Loeffler, A. Ligtenberg, and G. S. Moschytz, "Practical fast 1-D DCT algorithms with 11 multiplications," in *Acoustics, Speech, and Signal Processing, (ICASSP)*, 1989, pp. 988-991.

[60] R. P. Weicker, "Dhrystone: a synthetic systems programming benchmark," *Commun. ACM*, vol. 27, no. 10, pp. 1013-1030, 1984.

[61] L. T. SMIT, J. L. HURINK, and G. J. M. SMIT, "Run-time mapping of applications to a heterogeneous SoC," in *International Symposium on System-on-Chip (SoC)*, 2005, pp. 78-81.

[62] C.-L. CHOU and R. MARCULESCU, "Incremental Run-time Application Mapping for Homogeneous NoCs with Multiple Voltage Levels," in *International Conference on Hardware/software Codesign and System Synthesis (CODES+ISSS)*, 2007, pp. 161-166.

[63] A. NGOUANGA, G. SASSATELLI, L. TORRES, T. GIL, A. SOARES, and A. SUSIN, "A contextual resources use: a proof of concept through the APACHES platform," in *Design and Diagnostics of Electronic Circuits and systems (DDECS)*, 2006, pp. 42-47.

[64] A. K. SINGH, T. SRIKANTHAN, A. KUMAR, and W. JIGANG, "Communication-aware heuristics for run-time task mapping on NoC-based MPSoC platforms," *Systems Architecture*, vol. 56, no. 7, pp. 242-255, 2010.

[65] C.-L. CHOU and R. MARCULESCU, "Run-time task allocation considering user behavior in embedded multiprocessor networks-on-chip," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 1, pp. 78-91, 2010.

[66] A. Greiner, "Tsar: a scalable, shared memory, many-cores architecture with global cache coherence," in *9th International Forum on Embedded MPSoC and Multicore (MPSoC'09)*, 2009.

[67] J. Robinson, S. Russ, B. Heckel, and B. Flachs, "A task migration implementation of the message-passing interface," in *Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, 1996, pp. 61-68.

[68] M. A. R. Dantas and E. Zaluska, "Improving load balancing in an mpi environment with resource management," in *Proceedings of the International*

Conference and Exhibition on High-Performance Computing and Networking, 1996, pp. 959-960.

[69] L. Chen, C. Wang, and F. C. M. Lau, "A grid middleware for distributed java computing with mpi binding and process migration supports," *Journal of Computer Science and Technology*, vol. 18, no. 4, pp. 505-514, 2003.

[70] F. Mulas, D. Atienza, A. Acquaviva, S. Carta, L. Benini, and G. De Micheli, "Thermal balancing policy for multiprocessor stream computing platforms," *Transaction of Computer-Aided Design, Integration Circuit and System*, vol. 28, no. 12, pp. 1870-1882, 2009.

[71] M. Pittau, A. Alimonda, S. Carta, and A. Acquaviva, "Impact of task migration on streaming multimedia for embedded multiprocessors: A quantitative evaluation," in *Samarjit Chakraborty and Petru Eles*, ESTImedia., 2007, pp. 59-64.

[72] S. Bertozzi, A. Acquaviva, D. Bertozzi, and A. Poggiali, "Supporting task migration in multi-processor systems-on-chip: a feasibility study," in *Proceedings of the conference on Design, automation and test in Europe (DATE)*, 2006, pp. 15-20.

[73] G. Magklis, M. L. Scott, G. Semeraro, D. H. Albonesi, and S. Dropsho, "Profile-based dynamic voltage and frequency scaling for a multiple clock domain microprocessor," *Analysis*, vol. 31, no. 2, pp. 14-27, 2003.

[74] F. Xie, M. Martonosi, and S. Malik, "Compile-time dynamic voltage scaling settings: opportunities and limits," *SIGPLAN Conference on Programming Languages Design and Implementation*, vol. 38, no. 5, pp. 49-62, 2003.

[75] D. Puschini, F. Clermidy, P. Benoit, G. Sassatelli, and L. Torres, "A game-theoretic approach for run-time distributed optimization on mp-soc," *International Journal of Reconfigurable Computing*, vol. 2008, pp. 1-11, 2008.

[76] Q. Wu, P. Juang, M. Martonosi, and D. W. Clark, "Formal online methods for voltage/frequency control in multiple clock domain microprocessors," *SIGARCH Computer Architecture News*, vol. 32, no. 5, pp. 248-259, 2004.

[77] Y. Zhu and F. Mueller, "Feedback EDF Scheduling Exploiting Hardware-Assisted Asynchronous Dynamic Voltage Scaling," *ACM Sigplan Notices*, vol. 40, no. 7, p. 203, 2005.

[78] G. Marchesan Almeida et al., "Predictive Dynamic Frequency Scaling for Multi-Processor Systems-on-Chip," in *IEE International Symposium on Circuit and Systems (ISCAS)*, 2011, pp. 1500-1503.

[79] U. Y. Ogras and R. Marculescu, "Variation-adaptive feedback control for networks-on-chip with multiple clock domains," in *Proceedings of the 45th annual Design Automation Conference (DAC)*, 2008, pp. 614-619.

[80] A. Sharifi, H. Zhao, and M. Kandemir, "Feedback control for providing qos in noc based multicores," in *Proceedings of the conference on Design, automation and test in Europe (DATE)*, 2010, pp. 1384-1389.

[81] M. Ghasemazar, E. Pakbaznia, and M. Pedram, "Minimizing the power consumption of a chip multiprocessor under an average throughput constraint," in *International Symposium on Quality Electronic Design (ISQED)*, 2010.

[82] Diego Puschin, F. Clermidy, P. Benoit, G. Sassatelli, and L. Torres, "Dynamic and Distributed Frequency Assignment for Energy and Latency Constrained MP-SoC," in *Design Automation and Test in Europe (DATE)*, 2009, pp. 1564-1567.

[83] F. Clermidy, R. Lemaire, X. Popon, D. Ktenas, and Y. Thonnart, "An open and reconfigurable platform for 4g telecommunication: Concepts and application," in *Proceedings of the12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools (DSD)*, 2009, pp. 449-456.

[84] Valgrind, "An instrumentation framework for building dynamic analysis tools." [Online]. Available: http://www.valgrind.org.

[85] P. Guironnet de Massas and F. Pétrot, "Migration de données dans les MPSoC : une solution matérielle," in *3ème Symposium en Architecture de machines (SympA)*, 2009.

[86] V. Sarita, S. Vikram, D. Mark, and K. Mary, "Comparison of hardware and software cache coherence schemes," in *Proceedings of the 18th Annual International Symposium on Computer Architecture*, 1991, pp. 298-308.

[87] Nguyen, "Single-instruction-multiple-data processing in a multimedia signal processor," 2000.

[88] T. F. Smith and M. S. Waterman, "Identification of Commom Molecular Subsequences," *Molecular Biology*, vol. 147, no. 1, pp. 195-197, 1981.

[89] E. W. Dijkstra, "The structure of the multiprogramming system," in *ACM Symposium on Operating System Principles*, 1968, pp. 341-346.

[90] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming*. 1972.

[91] B. Dreier, M. Zahn, and T. Ungerer, "The Rthreads Distributed Shared Memory System," in *3rd International Conference on Massively Parallel Computing Systems*, 1998, pp. 42-53.

[92] W.-Y. Lyang, C.-T. King, and L. Feipei, "Adsmith: an Object Based Distributed Shared Memory System for Networks of Workstations," *Special Issue on Architecture, Algorithm and Network for Massively Parralel Computing*, 1997.

[93] M. Shalan and V. J. Mooney, "Hardware support for real-time embedded multiprocessor system-on-a-chip memory management," in *Proceedings of the*

10th International Workshop on Hardware/Software Codesign (CODES'02), 2002, pp. 79-84.

[94] V. De La Luz, M. Kandemir, and I. Kolcu, "Automatic data migration for reducing energy consumption in multi-bank memory systems," in *Proceedings of the 39th Conference on Design Automation (DAC'02)*, 2002, pp. 213-218.

[95] M. Monchiero, G. Palermo, C. Silvano, and O. Villa, "Exploration of distributed shared memory architectures for NoC-based multiprocessors," *Journal of Systems Architecture*, vol. 53, no. 10, pp. 719-732, 2007.

[96] L. Lamport, "How to make a multiprocessor computer that correctly executes multi-process programs," *IEEE Transaction Computer*, vol. 28, no. 9, pp. 690-691, 1979.

[97] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990, pp. 15-26.

[98] J. R. Goodman, "Cache Consistency and Sequential Consistency," in *Technical Report no. 61*, 1989.

[99] M. Dubois, C. Scheurich, and F. Briggs, "Memory access buffering in multiprocessors," *SIGARCH Computing Architecture*, vol. 14, no. 2, pp. 434-442, 1986.

[100] C. Holt, M. Heinrich, J. P. Singh, E. Rothberg, and J. Hennessy, "The effects of latency, occupancy, and bandwidth in distributed shared memory multiprocessors," in *Technical Report*, 1995.

[101] I. Tartalja and V. Milutinovic, "Classifying software-based cache coherence solutions," *IEEE Software*, vol. 14, no. 3, pp. 90-101, 1997.

[102] David Culler, J. P. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*. 1999.

[103] D. Lenoski et al., "The stanford DASH multiprocessor," *Journal of Computers*, vol. 25, no. 3, pp. 63-79, 1992.

[104] I. Tartalja and V. Milutinovic, "A survey of software solutions for maintenance of cache consistency in shared memory multiprocessors," in *Hawaii International Conference on System Sciences*, 1995.

[105] F. Petrot, A. Greiner, and P. Gomez, "On cache coherency and memory consistency issues in noc based shared memory multiprocessor SoC architectures," in *Digital System Design: Architectures, Methods and Tools*, 2006.

[106] J. W. Van den Brand and M. Bekooij, "Streaming consistency: a model for efficient mpsoc design," in *10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD)*, 2007, pp. 27-34.

[107] I. Tartalja and V. Milutinovic, "An approach to dynamic software cache consistency maintenance based on conditional invalidation," *System Sciences*, vol. 1, pp. 457-466, 1992.

[108] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: a tutorial," *Journal of Computers*, vol. 29, no. 12, pp. 66-76, 1996.

[109] F. Ophelders, M. J. G. Bekooij, and H. Corporaal, "A tuneable software cache coherence protocol for heterogeneous MPSoCs," in *Proceedings of the 7th IEEE/ACM International Conference on Hardware/software Codesign and System Synthesis*, 2009, pp. 383-392.

[110] H.-J. Boehm, "Threads cannot be implemented as a library," in *Proceedings of conference on Programming language design and implementation (PLDI)*, 2005, pp. 261-268.

[111] N. Binkert et al., "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1-7, 2011.

[112] A. Gupta and W.-D. Weber, "Cache invalidation patterns in shared-memory multiprocessors," *IEEE Transaction on Computers*, vol. 41, no. 7, pp. 794-810, 1992.

Publications

[1] **Busseuil, R.**; Marchesan Almeida, G.; Copello Ost, L.; Varyani, S.; Sassatelli, G.; Robert, M. : "Adaptation Strategies in Multiprocessors System on Chip", *VLSI-SoC: Forward-Looking Trends in IC and Systems Design,* pp. 233-257, 2011.
doi : 10.1007/978-3-642-28566-0

[2] Marchesan Almeida, G.; **Busseuil, R.**; Copello Ost, L.; Bruguier, F.; Sassatelli, G.; Benoit, P.; Robert, M. : "Enabling adaptability in multiprocessor system on chip by using PID controllers: an energy-aware distributed approach", *IEEE Embedded System Letters,* vol. 3, no. 3, pp. 77-80, 2011.
doi : 10.1109/LES.2011.2166373

[3] **Busseuil, R.**; Barthe, L.; Marchesan Almeida, G.; Copello Ost, L.; Bruguier, F.; Sassatelli, G.; Benoit, P.; Robert, M.; Torres, L. : " Open-scale: A scalable, open-source noc-based mpsoc for design space exploration", *International Conference on ReConFigurable Computing and FPGAs (ReConfig)*, 2011.

[4] Marchesan Almeida, G.; **Busseuil, R.**; Alceu Carara, E.; Hébert, N.; Varyani,S.; Sassatelli, G.; Benoit, P.; Torres, L.; Gehm Moraes, F. : Predictive dynamic frequency scaling for multi-processor systems-on-chip." *International Symposium on Circuits and Systems (ISCAS)*, 2011. IEEE Computer Society.

[5] Marchesan Almeida, G.; Varyani, S.; **Busseuil, R.**; Hébert, N.; Sassatelli, G.; Benoit, P.; Torres, L.; Robert, L. : "Providing better multi-processor systems-on-chip resources utilization by means of using a control-loop feedback mechanism." *International Conference on ReConFigurable Computing and FPGAs (Reconfig)*, pp. 382-387, 2010. IEEE Computer Society.

[6] **Busseuil, R.**; Marchesan Almeida, G.; Varyani, S.; Sassatelli, G.; Robert M. : "Exploration of task migration techniques for distributed memory multiprocessor systems on chips." *International Conference on VLSI and Systems-on-Chip (VLSI-SoC)*, 2010. IEEE Computer Society.

[7] Marchesan Almeida, G.; Varyani, S.; **Busseuil, R.**; Sassatelli, G.; Torres, L.; Alceu Carara, E.; Gehm Moraes, F. : "Evaluating the impact of task migration in multi-processor systems-on-chip." Symposium on Integrated Circuits and Systems Design (SBCCI), pp. 73-78, 2010. ACM Computer Society.

[8] **Busseuil, R.**; Marchesan Almeida, G.; Varyani, S.; Sassatelli, G.; Benoit, P. : "A self-adaptive communication protocol allowing fine tuning between flexibility and performance in homogeneous mpsoc systems." *Reconfigurable Communication-centric Systems on Chip (ReCoSoc)*, pp. 1-6, 2010.

[9] **Busseuil, R.**; Barthe, L.; Marchesan Almeida, G.; Sassatelli, G.; Benoit, P.; Robert, M.; Torres, L. : "Design of an open-source, noc-based mpsoc: Open-scale." *Colloque National du GDR SOC-SIP (GDR SOC-SIP)*, France, 2011.

[10] **Busseuil, R.**; Marchesan Almeida, G.; Varyani, S.; Robert, M. : "Exploration of task migration techniques for distributed memory multiprocessor systems on chips." *Colloque National du GDR SOC-SIP (GDR SOC-SIP)*, France, 2010.

Résumé

Bien que le développement actuel d'accélérateurs se concentre principalement sur la création de puces Multiprocesseurs (MPSoC) hétérogènes, c'est-à-dire composés de processeurs spécialisées, de nombreux acteurs de la microélectronique s'intéressent au développement d'un autre type de MPSoC, constitué d'une grille de processeurs identiques. Ces MPSoC homogènes, bien que composés de processeurs énergétiquement moins efficaces, possèdent une programmabilité et une flexibilité plus importante que les MPSoC hétérogènes, ce qui favorise notamment l'adaptation du système à la charge demandée, et offre un espace de solutions de configuration potentiellement plus vaste et plus simple à contrôler. C'est dans ce contexte que s'inscrit cette thèse, en exposant la création d'une architecture MPSoC homogène scalable (c'est-à-dire dont la mise à l'échelle des performances est linéaire), ainsi que le développement de différents systèmes d'adaptation et de programmation sur celle-ci.

Cette architecture, constituée d'une grille de processeurs de type MicroBlaze, possédant chacun sa propre mémoire, au sein d'un Réseau sur Puce 2D, a été développée conjointement avec un système d'exploitation temps réel (RTOS) spécialisé et modulaire. Grâce à la création d'une pile de communication complexe, plusieurs mécanismes d'adaptation ont été mis en œuvre : une migration de tâche « avec redirection de données », permettant de diminuer l'impact de cette migration avec des applications de type flux de données, ainsi qu'un mécanisme dit « d'exécution distante ». Ce dernier consiste non plus à migrer le code instruction d'une mémoire à une autre, mais de conserver le code dans sa mémoire initiale et de le faire exécuter par un processeur distinct. Les différentes expériences réalisées avec ce mécanisme ont permis de souligner la meilleure réactivité de celui-ci face à la migration de tâche, tout en possédant des performances d'adaptation plus faible.

Ce dernier mécanisme a conduit naturellement à la création d'un modèle de programmation de type « mémoire partagée » au sein de l'architecture. La mise en place de ce dernier nécessitait la création d'un mécanisme de cohérence mémoire, qui a été réalisé de façon matérielle/logicielle et scalable par l'intermédiaire du développement de la librairie PThread. Les performances ainsi obtenues mettent en évidence les avantages d'un MPSoC homogène tout en utilisant une programmation « classique » de type multiprocesseur.

Summary

Although the accelerators market is dominated by heterogeneous MultiProcessor Systems-on-Chip (MPSoC), i.e. with different specialized processors, a growing interest is put on another type of MPSoC, composed by an array of identical processors. Even if these processors achieved lower performance to power ratio, the better flexibility and programmability of these homogeneous MPSoC allow an easier adaptation to the load, and offer a wider space of configurations. In this context, this thesis exposes the development of a scalable homogeneous MPSoC – i.e. with linear performance scaling – and different kind of adaptive mechanisms and programming model on it.

This architecture is based on an array of MicroBlaze-like processors, each having its own memory, and connected through a 2D NoC. A modular RTOS was build on top of it. Thanks to a complex communication stack, different adaptive mechanisms were made: a "redirected data" task migration mechanism, reducing the impact of the migration mechanism for data-flow applications, and a "remote execution" mechanism. Instead of migrate the instruction code from a memory to another, this last consists in only migrate the execution, keeping the code in its initial memory. The different experiments shows faster reactivity but lower performance of this mechanism compared to migration.

This development naturally led to the creation of a shared memory programming model. To achieve this, a scalable hardware/software memory consistency and cache coherency mechanism has been made, through the PThread library development. Experiments show the advantage of using NoC based homogeneous MPSoC with a brand programming model.