N° D'ORDRE : 9924

THÈSE DE DOCTORAT

SPECIALITE : PHYSIQUE

Ecole Doctorale « Sciences et Technologies de l'Information des Télécommunications et des Systèmes »

Présentée par :

Alexandre GUERRE

Sujet :

Approche hiérarchique pour la gestion dynamique des tâches et des communications dans les architectures massivement parallèles programmables

Soutenue le …24 septembre 2010 …..devant les membres du jury :

M.Lavenier Dominique, Rapporteur, DR CNRS, IRISA

M.Torres Lionel, Rapporteur, Professeur, Université de Montpellier 2

M.Auguin Michel, Examinateur, DR CNRS, Université de Nice-Sophia Antipolis

M.Gogniat Guy, Examinateur, Université de Bretagne Sud

M.Mérigot Alain, Directeur, Professeur, Université Paris Sud

M.David Raphaël, Co-directeur, Ingénieur Chercheur au CEA LIST

M.Ventroux Nicolas, Co-directeur, Ingénieur Chercheur au CEA LIST

Computers are useless. They can only give you answers.
par Pablo Picasso (1881 - 1973)

Remerciements

J'ai réalisé mes travaux de thèse au sein du Laboratoire Calcul Embarqué (LCE) du Laboratoire d'Intégration des Systèmes et des Technologies (LIST) au Commissariat à l'Energie Atomique et aux Energies Alternatives (CEA) de Saclay. C'est pourquoi je voudrais remercier Laurent Letellier et Raphaël David de m'avoir accueilli au sein du Laboratoire et de m'avoir permis de mener à son terme ma thèse dans les meilleures conditions.

Je remercie Alain Mérigot, professeur à l'Institut d'Electronique Fondamental (IEF) de l'université Paris Sud, de m'avoir fait l'honneur d'encadrer ma thèse et pour ses conseils avisés durant ma thèse.

J'adresse mes remerciements sincères à Nicolas Ventroux, ingénieur chercheur, et Raphaël David, chef du Laboratoire, pour m'avoir encadré et conseillé tout au long de ces trois années. Je tiens à les remercier plus particulièrement pour leur patience et leur soutien durant ma thèse et enfin pour m'avoir offert l'opportunité de continuer ma démarche de recherche au CEA.

Je remercie Michel Auguin, directeur de recherche CNRS à l'université de Nice-Sophia Antipolis, d'avoir accepté de présider mon jury de thèse.

Je remercie également Dominique Lavenier, directeur de recherche CNRS à l'IRISA, et Lionel Torres professeur à l'université de Montpellier 2, d'avoir accepté de juger mon travail en tant que rapporteurs.

Je tiens à remercier Guy Gogniat, professeur à l'université de Bretagne Sud pour sa participation à mon jury de thèse.

Je tiens également à remercier Tanguy Sassolas, Maroun Ojail, Jean-Marc Philippe et Pierre Guironnet de Massas pour le temps qu'ils ont consacré à relire ma thèse ainsi que pour leurs remarques pertientes.

Je remercie l'ensemble des membres du Laboratoire LCE pour leur aide durant ma thèse mais aussi pour la bonne humeur régnant dans le laboratoire. J'aimerais aussi remercier l'ensemble des participants de nos pauses thé pour nos nombreux fous rires.

Je remercie Julien, Pablo et le laboratoire LFSE pour m'avoir si bien accueilli lors de mon passage au bâtiment 451.

Je remercie également Tof et Marina et tous mes amis d'école (Bérichon, Joker, JJ, Ben et Camille) qui m'ont soutenu et encouragé durant ma thèse ainsi que pour les bons moments passés avec eux.

Je tiens à remercier les membres de l'association MAIOT pour avoir supporté humour et de m'avoir permis de passer de très bons moments tout au long de ma thèse. Je tiens à remercier plus particulièrement Véronique, Agnès, Julien, Benoit, Clémence, Sophie, Laura, Sarah, Aurélie et tous les autres avec qui j'ai passées de bonnes soirées.

Je voudrais terminer ces remerciements par un grand merci à mes parents et à mon frère pour leur aide et leurs encouragements dans les moments durs.

Table des matières

Introduction **1**
Contexte de l'étude . . . 1
Plan du manuscrit . . . 4

1 Etat de l'art **7**
1.1 Les architectures multiprocesseurs existantes . . . 8
1.1.1 Les solutions statiques . . . 9
1.1.2 Les solutions dynamiques . . . 10
1.1.3 Conclusion sur les architectures multiprocesseurs . . . 11
1.2 Choix d'un type d'interconnexion . . . 12
1.2.1 Connexion point à point . . . 12
1.2.2 Connexion par bus . . . 13
1.2.3 Réseaux sur puce : NoC (Network on Chip) . . . 14
1.2.4 Synthèse sur les modes de communication . . . 16
1.3 Les caractéristiques des réseaux sur puce . . . 17
1.3.1 L'étude théorique des réseaux sur puce . . . 17
1.3.2 Les éléments de base des NoC . . . 18
1.3.3 Les mécanismes de transport . . . 19
1.3.4 La topologie . . . 21
1.3.5 Les algorithmes de routage . . . 26
1.3.6 Le contrôle de flux . . . 27
1.3.7 Qualité de service (Quality of Service : QoS) . . . 29
1.4 Conclusions de l'état de l'art . . . 30

2 Mise en place d'un environnement de simulation pour l'exploration des réseaux sur puce **33**
2.1 Choix du niveau de simulation . . . 34
2.2 Etat de l'art des simulateurs de réseaux sur puce . . . 36
2.3 Timed TLM contre Approximate-Timed TLM . . . 38
2.4 Description de notre structure de simulation . . . 40
2.4.1 Implantation des réseaux sur puce . . . 40
2.4.2 Un environnement de simulation pour la comparaison de topologies . . . 46
2.5 Caractérisation de la rapidité et de la précision de notre simulateur . . . 48
2.5.1 Caractérisation de la rapidité . . . 49
2.5.2 Caractérisation de la précision . . . 50
2.6 Conclusion sur la nouvelle approche proposée . . . 51

3 Proposition d'une topologie hiérarchique indépendante de l'applicatif et efficace d'un point de vue silicium **53**
3.1 Description de réseaux représentatifs de l'espace de conception et leur implémentation dans le simulateur . . . 54
3.1.1 Les réseaux à plat . . . 55
3.1.2 Les réseaux hiérarchisés . . . 56
3.2 Comparaison des performances des différentes topologies . . . 59
3.2.1 Comparaison des performances sous trafic uniforme . . . 59
3.2.2 Comparaison des performances sous trafic localisé . . . 62
3.3 Description de la méthode de synthèse et résultats de synthèse ASIC 67
3.3.1 Méthodologie de synthèse . . . 67
3.3.2 Résultat de synthèse . . . 67
3.4 Comparaison des efficacités silicium des différentes topologies . . . 69
3.4.1 Comparaison de l'efficacité surfacique sous trafic uniforme . . 70
3.4.2 Comparaison de l'efficacité surfacique sous trafic localisé . . . 73
3.5 Conclusion sur le choix de la topologie . . . 74

4 Modèle d'exécution pour structures many-core hiérarchiques embarquées **77**
4.1 Présentation des familles d'algorithmes d'ordonnancement existants dans le domaine du calcul haute performance . . . 78
4.1.1 Les algorithmes par liste . . . 79
4.1.2 Les algorithmes par groupe (Clusterisation) . . . 81
4.1.3 Exemple de réalisation pour une architecture many-core et conclusion sur l'existant . . . 83
4.2 Description des algorithmes de faible complexité sélectionnés et de l'environnement de test . . . 84
4.2.1 L'environnement de test . . . 85
4.2.2 Les algorithmes d'ordonnancement par liste existants . . . 86
4.2.3 Les nouveaux algorithmes d'ordonnancement par liste . . . 86
4.2.4 Les nouveaux algorithmes d'ordonnancement par groupe statique . . . 88
4.2.5 Le nouvel algorithme d'ordonnancement dynamique par groupe 89
4.3 Mise en place de métriques et de moyens de mesure . . . 90
4.4 Comparaison entre les différentes propositions d'algorithmes . . . 92
4.4.1 Résultats avec un graphe hautement dépendant . . . 93
4.4.2 Résultats avec un graphe parallèle . . . 96
4.4.3 Synthèse de l'étude sur les algorithmes d'ordonnancement de faible complexité . . . 99
4.5 Proposition d'un modèle d'exécution pour une architecture many-core hiérarchique . . . 99
4.5.1 Gestion des tâches . . . 100
4.5.2 Gestion des données . . . 105
4.5.3 Les critères de placement des groupes de tâches . . . 106

4.6 Vérification des performances . . . 106
4.7 Conclusion . . . 108

5 Mise en place d'une architecture et validation de l'architecture et du modèle d'exécution 109
5.1 Description de notre architecture massivement parallèle . . . 110
5.1.1 Vue d'ensemble du fonctionnement . . . 111
5.1.2 L'architecture globale . . . 113
5.1.3 Le cluster matériel . . . 116
5.2 Une application de test . . . 122
5.3 Implémentation de notre architecture dans SESAM . . . 125
5.3.1 SESAM : Simulation Environment for Scalable Asymmetric Multiprocessors . . . 125
5.3.2 Implémentation de notre architecture dans SESAM . . . 127
5.4 Caractérisation de notre architecture hiérarchique . . . 128
5.4.1 Dynamique versus Statique . . . 129
5.4.2 Caractérisation de la hiérarchisation du contrôle . . . 131
5.5 Impact de la hiérarchisation sur la surface . . . 133
5.6 Conclusion . . . 134

Conclusions et Perspectives 135
Synthèse des travaux . . . 135
Perspectives . . . 137
A court terme . . . 137
A long terme . . . 138

Bibliographie 141

Publications personnelles 149

Table des figures

1 Besoin en calcul des applications embarquées. 2
2 Temps d'exécution en cycles d'un pipeline de rendu 3D en fonction de chaque image créée. 3

1.1 Liste non exhaustive des architectures MPSoC (MultiProcessor System on Chip) en fonction du nombre de processeurs. 8
1.2 Les interconnexions point à point. 13
1.3 Les interconnexions par bus. 14
1.4 Les réseaux sur puce. 16
1.5 Routeur élémentaire. 18
1.6 Représentation des topologies régulières. 23
1.7 Représentation des topologies irrégulières. 24
1.8 Classification des réseaux. 31

2.1 Vitesse de simulation des niveaux de modélisation en fonction de leurs précisions. 34
2.2 Timed TLM vs Approximated Time TLM. 39
2.3 Différence entre temps explicite et implicite dans le TLM. 41
2.4 Représentation du fonctionnement du simulateur de réseaux sur puce. . . 43
2.5 Représentation d'un chemin dans le simulateur. 46
2.6 Représentation de la structure de comparaisons des différentes topologies. (TG : générateur de trafic) 47
2.7 Temps de simulation en fonction du nombre de mémoires dans les cas du TTLM et ATTLM. (TG : générateur de trafic) 49
2.8 Comparaison entre Noxim et notre approche. 50

3.1 Représentation d'un cluster matériel. 57
3.2 Représentation de la topologie Multi-bus + Anneau et Multi-bus + Tore. 58
3.3 Représentation de la topologie Multi-bus + Anneau amélioré. 58
3.4 Performance des réseaux pour un trafic uniforme. 60
3.5 Performance des réseaux hiérarchiques pour un trafic pseudo-localisé. 63
3.6 Représentation de la probabilité d'envoi pour un trafic localisé sur un anneau. 64
3.7 Performance des réseaux pour un trafic localisé. 66
3.8 Courbe de surface des réseaux à plat. 68
3.9 Surface de tous les réseaux proposés pour 256 connexions. 70
3.10 Efficacité surfacique des réseaux proposés pour un trafic uniforme. . 72
3.11 Efficacité surfacique des réseaux proposés pour un trafic localisé. . . 75

4.1 Représentation du fonctionnement d'un algorithme par liste. 80

4.2 Etapes de la clusterisation linéaire. 82
4.3 Représentation de l'environnement synthétique de tests. 85
4.4 Représentation de la découpe non-linéaire et dynamique en cluster de l'algorithme Dynamic-Clustering-Mapping. 90
4.5 Exemples de graphes générés avec TGFF. 94
4.6 Résultat des différents algorithmes pour un graphe hautement dépendant. 95
4.7 Résultat des différents algorithmes pour un graph parallèle. 98
4.8 Modification de la clusterisation linéaire. 102
4.9 Représentation des informations nécessaire à l'ordonnancement. . . . 103
4.10 Résultat de performances de notre solution. 107

5.1 Représentation de l'architecture globale. 110
5.2 Représentation du cluster de base. 111
5.3 Représentation des différentes étapes de lancement d'une application. 112
5.4 Représentation du fonctionnement du TLB. 114
5.5 Représentation de la boucle d'ordonnancement du contrôle central. . 115
5.6 Représentation de la boucle d'ordonnancement du contrôle cluster. . 118
5.7 Pipeline de l'application de surveillance de carrefour. Pour chaque tâche, il est mentionné si celle-ci est dynamique (ou non) et selon quels critères. . . 123
5.8 Représentation du graphe de tâches de l'application d'étiquetage. . . 125
5.9 Représentation d'une branche du graphe de tâches de l'application d'étiquetage. 126
5.10 Images d'entrée et de sortie de l'application d'étiquetage 126
5.11 Représentation de l'architecture contenant un contrôle hiérarchique. . 127
5.12 Image à labelliser contenant une fausse silhouette de piéton. 129

Liste des tableaux

1.1 Résumé des caractéristiques des différentes solutions. 17

3.1 Estimation de la surface des interconnexions en mm^2. 68
3.2 Estimation de surface des réseaux hiérarchiques. 69

5.1 Temps d'exécution de l'application de test pour un placement statique ou dynamique. 130
5.2 Temps d'exécution de l'application de test en fonction du type de placement. 131
5.3 Temps moyen d'ordonnancement durant l'application de test en fonction du type de contrôle. 132
5.4 Temps d'exécution de l'application de test en fonction du type de contrôle. 132
5.5 Estimation de la surface des architectures en mm^2. 133

Introduction

Sommaire

Contexte de l'étude 1
Plan du manuscrit 4

L'un des premiers systèmes embarqués fut l'Apollo Guidance Computer. Conçu par la société MIT Intrumentation Laboratory sous la direction de Charles Stark Draper, ce dispositif équipa les navettes et modules lunaires des missions Apollo dans les années 60-70 comme ordinateur de navigation et de pilotage. Il était composé d'environ 4500 portes NOR, 32 Kbits de mémoire vive et 576 Kbits de mémoire morte. L'ensemble du dispositif fonctionnait à 2 MHz et traitait des mots de 16 bits. Depuis cette révolution, les systèmes embarqués se sont répandus dans notre quotidien jusqu'à devenir quasiment indispensables. Poussés en permanence par la demande croissante des consommateurs, les systèmes embarqués n'ont cessé d'évoluer en termes de performance, de puissance de calcul et de capacité mémoire.

Contexte de l'étude

La conséquence majeure de cette demande des consommateurs est que les besoins en puissance de calcul sont devenus de plus en plus importants au fur et à mesure qu'évoluent les systèmes. La figure 1 fait apparaître les puissances de calcul nécessaires en fonction des applications pour différents domaines applicatifs. Ainsi, par exemple, dans le domaine des télécommunications, à mesure que les standards évoluent les besoins en puissance de calcul augmentent jusqu'à atteindre environ 50 Giga Opération Par Seconde (GOPS) pour le standard 3 GPP-LTE. On retrouve la même tendance dans le multimédia où, du fait de l'augmentation de la taille des images et de la complexité croissante des algorithmes de compression, de plus en plus de puissance de calcul est nécessaire. Par exemple, la décompression en H264 du flux d'une vidéo HD nécessite plus de 100 GOPS. Cette même évolution de la puissance de calcul touche tous les domaines de l'embarqué que ce soit pour les applications de vision ou encore les applications militaires.

De plus, aujourd'hui un téléphone, par exemple, ne doit pas seulement téléphoner mais également permettre de gérer ses mails, surfer sur internet, regarder des films en haute définition ou encore jouer à des jeux vidéo. Cette demande impose aux systèmes embarqués d'être capables d'exécuter un large panel d'applications provenant de domaines différents. Cette multiplication des domaines applicatifs pose un problème d'efficacité des dispositifs de calcul intégrés. En effet, dans le cas de calculs intensifs provenant de deux domaines différents, les calculs ne vont pas être réalisés sur les mêmes types de données. Ainsi, dans le domaine des télécommunications,

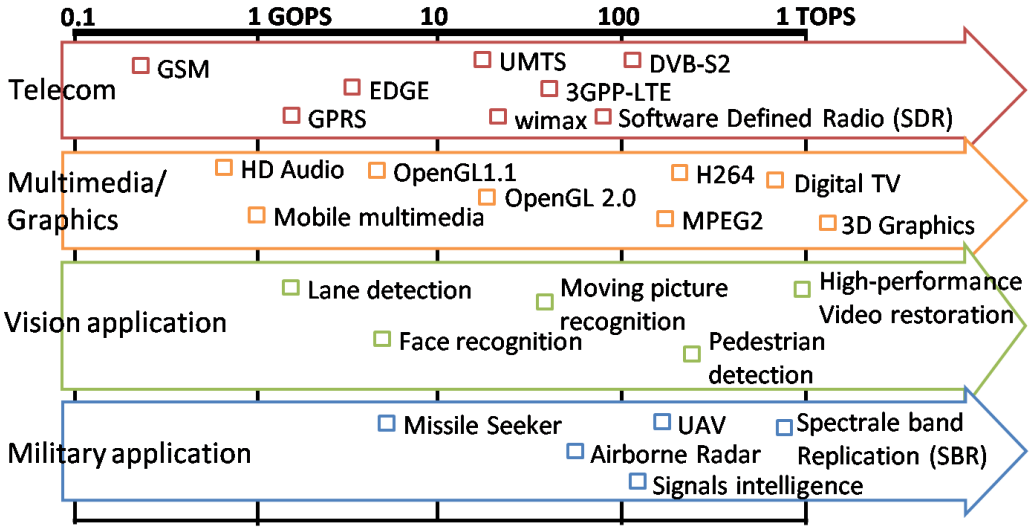


FIG. 1 – Le graphique représente les besoins en calcul de différentes applications regroupées par thème. (GOPS : Giga Opération Par Seconde ; TOPS : Tera Opération Par Seconde)

les calculs sont souvent réalisés sur des données codées en virgule fixe sur 16 bits alors que dans le domaine graphique, les données manipulées sont plutôt codées en virgule flottante sur 32 bits. Il sera donc compliqué d'optimiser les processeurs pour supporter ces deux types de données.

Enfin, en conséquence de cette évolution, les applications deviennent également de plus en plus dynamiques. En effet le temps d'exécution de ces applications dépend de paramètres non prédictibles au moment de l'écriture du programme. Ceci se traduit par un accroissement des séquences de contrôle dans le code du programme. On peut prendre comme exemple une application de reconnaissance de caractères. La séquence de reconnaissance de cette application va dépendre des caractères précédents ainsi que de la donnée courante. Et il n'est donc pas possible de prédire à l'avance le temps exact que va durer le traitement. En revanche, il est en général possible d'estimer une borne maximale au temps de traitement. Ceci se retrouve de plus en plus dans le domaine graphique, où les algorithmes s'adaptent aux données à traiter. Par exemple, dans [1], les auteurs présentent un pipeline de rendu 3D. Ils découpent le pipeline en trois phases (setup, géométrie et rendu) et pour chaque phase, ils mesurent le temps de calcul en fonction des images à créer. La figure 2 présente leurs résultats pour 201 images. Dans leur cas, la partie géométrie varie de plus d'un facteur 4 sur le temps d'exécution en fonction de l'image à créer. Ce dynamisme des applications implique qu'aucune solution optimale d'ordonnancement ne peut être réalisée hors-ligne et donc que le contrôle du système doit être lui-même dynamique afin de permettre une optimisation en ligne.

L'évolution des systèmes embarqués pose donc un problème au niveau de leur

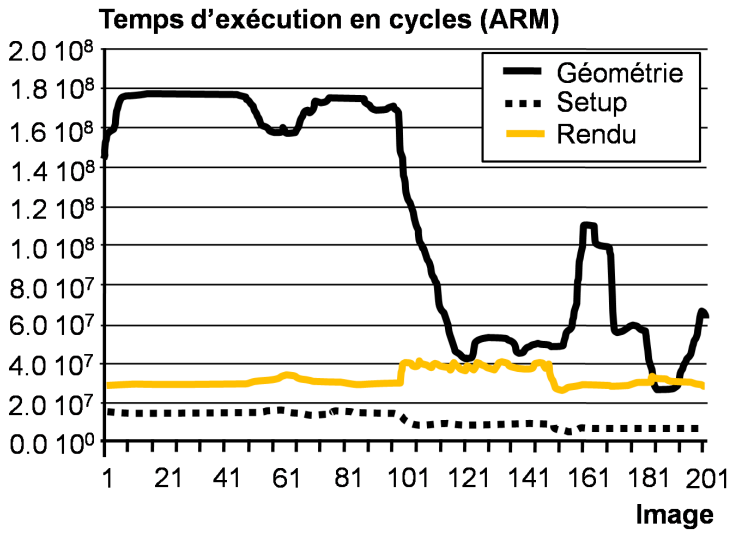


FIG. 2 – Temps d'exécution en cycles d'un pipeline de rendu 3D en fonction de chaque image créée. Chacune des trois phases qui composent le pipeline sont détaillées. Les valeurs sont issues de [1].

conception car ces systèmes doivent trouver un compromis entre leurs capacités (puissance de calcul, dynamisme) et les contraintes des systèmes embarqués (surface silicium, consommation). Le problème de la puissance de calcul est celui qui a été résolu en premier même si augmenter la puissance de calcul tout en respectant les contraintes de surface et de consommation du domaine de l'embarqué n'est pas un problème aisé. En effet, afin d'augmenter la puissance de calcul d'un monoprocesseur, il est possible d'augmenter la taille des caches ainsi que la complexité des mécanismes de prédiction afin de limiter les cycles d'inactivité dans le processeur. Ces techniques augmentent, par la même occasion, la surface silicium et la consommation énergétique du processeur ce qui, en général, diminue l'efficacité surfacique et énergétique. Or ces deux paramètres sont des paramètres importants pour le domaine de l'embarqué. La solution envisagée pour adresser ce problème est de passer à des systèmes multiprocesseurs. D'après [2], il est plus efficace d'intégrer plusieurs petits processeurs, spécialisés ou non, dont les efficacités énergétique et silicium sont meilleures, que d'augmenter les performances d'un seul processeur. Ainsi, au total, on obtient une puissance de calcul crête équivalente à un monoprocesseur pour une consommation en surface et en consommation plus faible. Suivant cette tendance, l'ITRS [3] prévoit une croissance annuelle de plus de 32% du nombre de processeurs, pour les années à venir, jusqu'à atteindre plus de 800 processeurs en 2015.

En résumé, aujourd'hui les utilisateurs demandent d'avoir des systèmes embarqués performants capables d'offrir de grandes puissances de calcul pour un large spectre d'applications en respectant les contraintes du monde de l'embarqué. De plus, ces applications ont de plus en plus un comportement et un temps d'exécution non prédictible au moment de la compilation. Il est donc nécessaire de proposer

des architectures multiprocesseurs offrant une grande puissance de calcul pour plusieurs domaines d'application. Ces architectures doivent être gérées dynamiquement afin d'offrir les meilleurs performances. Le tout doit être fourni pour une empreinte silicium et énergétique les plus faibles possibles.

Les différentes architectures multiprocesseurs disponibles actuellement répondent en partie aux besoins exprimés mais jamais complètement. On retrouve deux catégories d'après l'étude menée par les auteurs de [4]. Soit ces architectures sont massivement parallèles mais ne gèrent que statiquement les applications, soit elles proposent une gestion dynamique des applications mais ne proposent pas une puissance de calcul suffisante.

Plan du manuscrit

L'objectif de cette thèse est de mettre en place un modèle d'exécution adapté à la gestion de centaines de processeurs ainsi qu'une architecture multiprocesseur massivement parallèle programmable et homogène. Pour cela les aspects de communication puis de gestion dynamique d'application vont être explorés avant de proposer une solution au problème énoncé.

Le premier chapitre de ce document propose une étude des architectures multiprocesseurs représentatives des solutions contenant plus de 50 processeurs. Celle-ci montre que les infrastructures de communication ainsi que la gestion dynamique des applications sont deux problèmes importants dans les architectures massivement parallèles. C'est pourquoi, dans un premier temps, un état de l'art est réalisé sur les réseaux sur puce afin de choisir une interconnexion capable de connecter des centaines de processeurs et mémoires tout en gardant de bonnes performances. Les contextes de chaque étude étant différents, cet état de l'art met en évidence qu'il est très difficile de comparer tous les réseaux proposés. De plus, il en ressort également que la topologie ne peut pas être choisie facilement sans une phase d'exploration. Cette étude va tout de même permettre de choisir un certain nombre de topologies représentatives.

Le deuxième chapitre met en place une structure de simulation des réseaux sur puce qui va permettre de comparer les différentes topologies sélectionnées dans le premier chapitre. Avant de mettre en place ce simulateur, un niveau de simulation doit être défini afin de permettre le meilleur compromis entre précision et rapidité. Puis un état de l'art des simulateurs conclut qu'aucun simulateur ne propose de solutions permettant la comparaison de topologie au niveau de modélisation transactionnelle. Enfin le détail de notre simulateur de réseaux sur puce est donné dans ce chapitre.

Le troisième chapitre propose de présenter et commenter les résultats de la simulation des différentes topologies. Afin d'obtenir une comparaison en surface de ces réseaux, leurs synthèses ASIC ont été réalisées. Nous proposons une nouvelle façon d'analyser l'efficacité surfacique des réseaux en faisant apparaître la latence réseau en fonction de la charge réseau normalisée en surface. A l'issue de cette comparai-

son, une topologie Multibus+Anneau amélioré sera sélectionnée pour son efficacité silicium.

Une fois la structure de communication mise en place, le quatrième chapitre propose une analyse de l'enjeu de la gestion des tâches qui est le deuxième problème des architectures massivement parallèles énoncé dans le chapitre 1. Pour cela un état de l'art des algorithmes en ligne provenant du calcul haute performance et utilisable dans l'embarqué est réalisé. Ensuite une comparaison des différents algorithmes est effectuées. Cette étude va faire apparaître les meilleurs algorithmes d'ordonnancement en fonction des types d'application. De ces résultats, nous proposons un modèle d'exécution hiérarchique et dynamique qui combine les résultats précédents afin d'offrir les meilleures performances quel que soit le type d'application.

Le cinquième chapitre propose une implémentation et une validation de l'architecture ainsi que des services du modèle d'exécution. L'architecture est implémentée dans un environnement de simulation. Afin de permettre une évalusation, notre architecture est comparée à une seconde utilisant un contrôle centralisé. Enfin différents tests vont être réalisés permettant d'évaluer le gain d'une gestion dynamique, de la hiérarchisation du contrôle ainsi que le surcoût de la migration.

Finalement, le dernier chapitre présente un résumé des contributions de ces travaux de recherche. Il propose une discussion sur la suite de ces travaux ainsi que des perspectives à plus long terme.

Chapitre 1

Etat de l'art

Sommaire

1.1 Les architectures multiprocesseurs existantes 8
1.1.1 Les solutions statiques 9
1.1.2 Les solutions dynamiques 10
1.1.3 Conclusion sur les architectures multiprocesseurs 11
1.2 Choix d'un type d'interconnexion 12
1.2.1 Connexion point à point 12
1.2.2 Connexion par bus 13
1.2.3 Réseaux sur puce : NoC (Network on Chip) 14
1.2.4 Synthèse sur les modes de communication 16
1.3 Les caractéristiques des réseaux sur puce 17
1.3.1 L'étude théorique des réseaux sur puce 17
1.3.2 Les éléments de base des NoC 18
1.3.3 Les mécanismes de transport 19
1.3.4 La topologie 21
1.3.5 Les algorithmes de routage 26
1.3.6 Le contrôle de flux 27
1.3.7 Qualité de service (Quality of Service : QoS) 29
1.4 Conclusions de l'état de l'art 30

La course pour l'augmentation de la puissance de calcul des processeurs embarqués est régie par les contraintes de surface et de consommation électrique. Or l'augmentation de la puissance de calcul d'un monoprocesseur impose une augmentation de sa complexité et donc réduit son efficacité énergétique et surfacique. C'est pourquoi, l'on s'oriente maintenant vers les structures multiprocesseurs en grande partie car il est moins coûteux de mettre sur une puce plusieurs processeurs, plus petits et plus efficaces d'un point de vue silicium et consommation, qu'un seul processeur pour des performances théoriques équivalentes.

Afin de mieux comprendre l'état de l'art multiprocesseur, ce chapitre détaille cinq architectures qui répondent le plus aux besoins de puissances de calcul et de dynamisme. Cependant aucune ne répond complètement au problème. Pour les architectures offrant le plus de dynamisme, il apparaît que c'est l'interconnexion et le modèle d'exécution qui freinent l'augmentation du nombre de processeurs. Nous allons d'abord nous pencher sur le problème de l'interconnexion. C'est pourquoi dans un deuxième temps, ce chapitre présente un état de l'art des interconnexions sur puce et détaille les différentes caractéristiques des réseaux sur puce.

1.1 Les architectures multiprocesseurs existantes

La figure 1.1 superpose l'estimation ITRS (International Technology Roadmap for Semiconductors) concernant l'évolution des architectures multiprocesseurs avec la réalité des puces réalisées et détaillées dans [4]. Il apparaît que la tendance suivie par les industriels et les académiques suit la vision de l'ITRS ce qui tend à confirmer les estimations. Parmi tous les multiprocesseurs, la volonté principale des concepteurs est de fournir une puissance de calcul maximale. Pour cela, ils adoptent différents modèles d'exécution (axé flot de données, axé contrôle), et différentes structures matérielles (mer de processeurs, approche symétrique ou asymétrique).

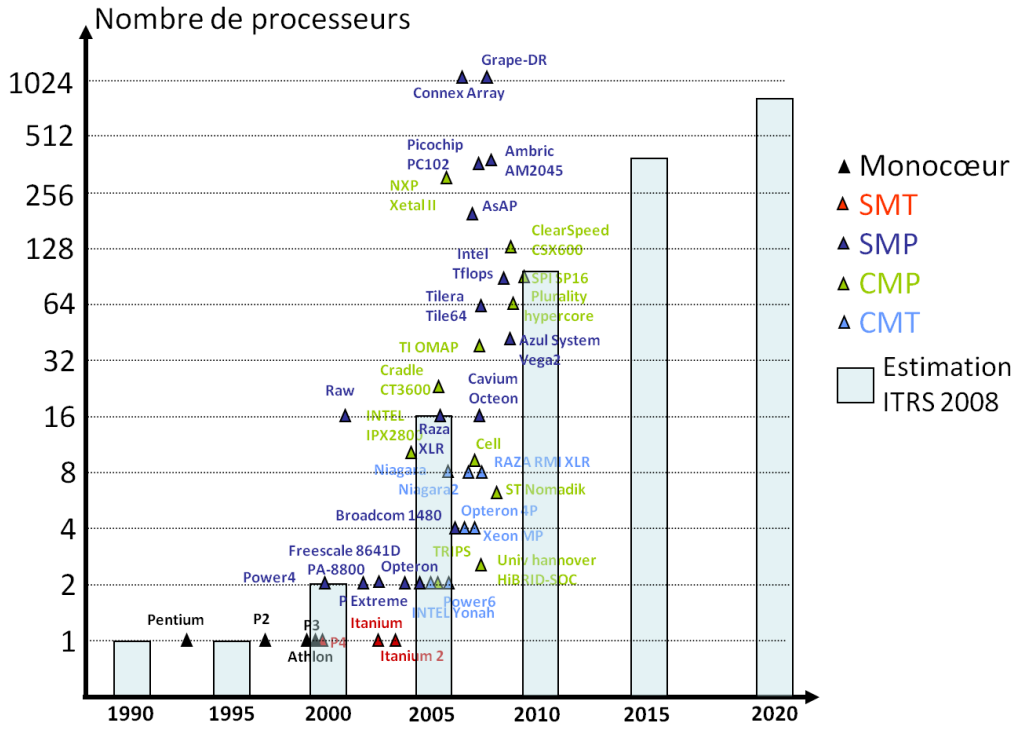


FIG. 1.1 – Le graphique représente une liste non exhaustive d'architectures mono et multi processeurs en fonction de leur nombre de processeurs et de leur année de fabrication. L'estimation de l'ITRS se superpose à la réalité. Les processeurs sont référencés en fonction de leur structure. SMT signifie *Simultaneous MultiThreading*, SMP : *Symetric MultiProcessor*, CMP : *Chip MultiProcessors* et CMT : *Chip MultiThreading*.

Comme nous l'avons dit plutôt, l'augmentation de la puissance de calcul passe par l'accroissement du nombre de cœurs. Ces cœurs peuvent être soit homogènes, soit hétérogènes. Les solutions hétérogènes cherchent à répondre aux besoins de puissance de calcul en plaçant sur la puce des modules dédiés à chaque domaine d'application. Les solutions homogènes cherchent à être flexibles en proposant seulement un type de cœur. Les solutions proposant les plus grands nombres de cœurs sont

toutes des solutions homogènes. Parmi ces architectures, on trouve deux stratégies de contrôle. Soit l'architecture propose une solution avec un placement statique des applications sur l'architecture, soit les applications sont déployées dynamiquement sur l'architecture. Selon les solutions, le problème de dynamisme des applications est pris ou non en considération. Cinq architectures représentatives des solutions existantes ont été choisies pour être détaillées.

1.1.1 Les solutions statiques

Les solutions de PICOCHIP [5], ASAP [6] et AMBRIC [7] sont basées un modèle de programmation statique. Les solutions de PICOCHIP et d'AMBRIC se focalisent sur des domaines d'application précis qui sont respectivement les télécommunications et le traitement d'images.

PICOCHIP propose ainsi un multiprocesseur composé de plus de 300 processeurs élémentaires. Les processeurs sont répartis par groupes de quatre. Ces groupements sont connectés à deux bus. Ces deux bus relient deux routeurs entre eux. Ceux-ci forment un réseau sur puce en grille à travers toute la puce. Afin d'optimiser le fonctionnement, toutes les applications sont placées statiquement sur la puce. Cette opération est aidée par la suite logicielle mise à la disposition de l'utilisateur. L'ensemble des communications à travers le système est donc déterminé au moment de la compilation de l'application. Ceci permet donc de s'affranchir d'arbitrage entre les communications pendant l'exécution. Cette technique de communication simplifie grandement le réseau d'interconnexion. Cependant il est nécessaire de déterminer, hors-ligne à l'aide des outils, le fonctionnement de l'application et donc de prendre en compte les pires cas d'exécution afin d'assurer un bon fonctionnement du système. Cette solution offre de très bonnes performances, mais le placement statique ne permet pas d'optimiser au mieux l'utilisation des ressources de calcul pour les applications dynamiques.

AMBRIC propose, dans son cas, un multiprocesseur dédié aux applications de traitement d'images. Ce multiprocesseur contient plus de 300 DSP (Digital Signal Processor). Les processeurs et les mémoires sont placés sur la puce selon un damier. Les cases sont connectées avec leurs proches voisines. Chacune de ces connexions est réalisée par un lien dédié. Le non partage des liens permet d'assurer qu'aucune contention ne viendra diminuer les performances. Du fait de la structure de l'architecture, les applications sont découpées en objets. L'ensemble de l'application cascade un certain nombre de ces objets. Chaque objet est placé soit sur un processeur, soit sur plusieurs processeurs voisins. L'ensemble du placement est défini statiquement. Comme pour le multiprocesseur de PICOCHIP, des outils sont fournis afin d'aider le placement et la programmation de la puce. Mais dans le cas d'AMBRIC, la problématique de la gestion des communications est grandement simplifiée car il n'y a pas de partage des liens de communication. Le modèle de programmation ne permet que de réaliser du flot de données. Cette solution obtient, comme dans le cas du PICOCHIP, de bonnes performances, en contrepartie les applications sont placées statiquement ce qui ne permet pas d'optimiser l'utilisation des ressources de

calcul dans le cas d'applications dynamiques.

L'architecture « Asynchronous Array of Simple Processors » (AsAP) propose pour sa part d'élargir le spectre des applications visées. Cette architecture comporte 164 processeurs génériques plus 4 modules de calcul dédiés à certain domaine (Viterbi, FFT...). Chaque processeur est associé à une mémoire d'instructions et de données ainsi qu'à un routeur afin de former une tuile. Chaque tuile possède sa propre horloge afin d'adapter sa fréquence aux besoins de calcul. Toutes les tuiles sont connectées à travers un réseau sur puce disposé en grille. Chaque routeur est contrôlé par le processeur de la tuile. C'est le processeur qui configure les connexions dans le routeur. Afin de simplifier les communications, le signal d'horloge de la source est envoyé en même temps que la donnée. Du fait des horloges séparées, des FIFO à double horloge sont installés entre les tuiles et le routeur. Les applications exécutées sur la puce doivent être décrites dans un modèle flot de données. Ensuite chaque tâche de l'application est placée statiquement sur une tuile afin de limiter les distances de communication. Cette architecture permet de reconfigurer chaque routeur de la puce et ainsi de faciliter les communications dynamiques entre les tuiles. Cependant les différentes tâches sont placées statiquement sur l'architecture. Cette architecture propose un compromis entre une solution complètement statique et une solution complètement dynamique.

Les solutions de ce type intègrent des centaines de processeurs homogènes. La programmation de ces structures passe par une phase d'optimisation statique afin d'obtenir les performances attendues. Du fait qu'aucune optimisation en ligne n'est réalisée, le dimensionnement doit se faire au pire cas afin de garantir l'exécution et les performances. Ceci limite donc la charge effective de ces architectures.

1.1.2 Les solutions dynamiques

La deuxième philosophie est de prendre le problème en entier et de proposer des multiprocesseurs multi-domaine afin d'exécuter n'importe quelle application dynamiquement. Les solutions proposées par PLURALITY [8] ou TILERA [9] sont de ce type.

La société TILERA propose un multiprocesseur multi-domaine. Celui-ci est composé de 64 processeurs élémentaires avec leur cache. Chaque processeur est connecté avec son cache sur un routeur d'un réseau-sur-puce. Chaque ensemble processeur+mémoire cache est nommé tuile. La particularité est que chaque tuile peut exécuter un noyau linux seul. De plus, il est possible de définir statiquement des zones constituées de plusieurs tuiles où un noyau Linux SMP peut s'exécuter. Les tuiles sont organisées selon une grille. Afin de permettre la cohérence de données à travers toute la puce, un système de cohérence distribué sur tous les caches est mis en place. La gestion mémoire permet l'utilisation de la mémoire aussi bien dans un mode à mémoire partagée que dans un mode de passage de messages. Le plus de cette architecture est de simplifier le portage des applications sur cette architecture grâce à l'utilisation d'un noyau Linux. Toutefois, cette structure en grille limite la possibilité d'ajouter de nouveaux éléments sans diminuer les performances. En-

fin cette architecture offre une gestion dynamique des applications mais seulement sur un nombre statique de processeurs ce qui ne permet pas d'optimiser le taux d'utilisation de la puce.

La société PLURALITY propose une architecture à mémoire partagée. Elle peut être constituée de 16 à 256 processeurs élémentaires tous connectés à un contrôleur central. Une interconnexion Crossbar découpée en étages permet de relier tous les processeurs à tous les bancs mémoires avec un accès uniforme. Ceci permet d'éviter les problèmes d'engorgement comme dans le cas de TILERA. Cette architecture adopte un modèle de programmation par tâche. Chacune de ces tâches sera dupliquée et exécutée en parallèle en fonction du taux de parallélisme souhaité. Le contrôleur centralisé permet de placer dynamiquement les tâches sur les différents processeurs. Une particularité de l'architecture est de permettre un accès uniforme à la mémoire sans se soucier du placement des tâches et ainsi de réaliser simplement un équilibrage de charges à travers toute l'architecture. Cette structure permet donc d'appréhender le côté dynamique des applications ainsi que la nécessité d'exécuter des applications provenant de plusieurs domaines. Cependant la structure de l'architecture contraint la taille de la structure et limite ses performances. En effet afin de maintenir un accès uniforme entre les processeurs et les mémoires, l'interconnexion risque d'augmenter la surface de manière non linéaire par rapport au nombre de processeurs. Ceci aura pour impact de diminuer l'efficacité surfacique. Le problème majeur de cette architecture réside dans son modèle de programmation qui limite les possibilités de programmation. Ainsi ce modèle ne permet de créer que du parallélisme régulier. De plus, le streaming n'est pas supporté par l'architecture.

Ces deux architectures visent un large spectre de domaines applicatifs avec un contrôle dynamique. Toutes les deux proposent une solution homogène dynamique qui offre un début de flexibilité. Cependant les deux proposent des structures de communication qui limitent l'augmentation du nombre de processeurs. De plus les deux architectures utilisent des modèles d'exécution qui ne permettent pas soit d'utiliser dynamiquement l'ensemble des processeurs (TILERA), soit qui limitent l'expression du parallélisme (PLURALITY).

1.1.3 Conclusion sur les architectures multiprocesseurs

Toutes les architectures présentées sont constituées de processeurs homogènes. Ce choix permet de faciliter la programmation de l'architecture ainsi que de simplifier le contrôle par rapport à des systèmes hétérogènes. De plus, l'homogénéité des processeurs permet une conception plus simple de la puce. C'est pour les mêmes raisons que nous avons également choisi de nous placer dans le cas de processeurs homogènes.

Les architectures multiprocesseurs embarquées sont capables soit d'une grande puissance de calcul en offrant plus de 300 processeurs sur une même puce comme le propose PICOCHIP ; soit d'une gestion dynamique d'application sur un petit nombre de processeurs comme le propose TILERA. Cependant aucune architecture ne permet la gestion dynamique d'un grand nombre de processeurs. Actuellement

les architectures sont limitées par leur infrastructure de communication mais surtout leur modèle d'exécution borne le dynamisme afin de simplifier la gestion des processeurs.

Nous venons donc de voir qu'augmenter le nombre de processeurs sur une puce ajoute de nouveaux problèmes à la conception de celle-ci. Nous allons traiter, dans un premier temps, le problème de l'infrastructure de communications. Celle-ci doit donc d'après nos conclusions permettre l'augmentation du nombre de processeurs sur la puce.

De plus comme nous l'avons mentionné dans l'introduction, les applications embarquées qui demandent toute cette puissance de calcul deviennent de plus en plus dynamiques ce qui limite l'efficacité des ordonnancements statiques. L'utilisation d'ordonnancement dynamique, afin d'optimiser l'exécution des applications dynamiques, rend pour sa part difficile la prédiction des communications entre les éléments du multiprocesseur.

C'est pourquoi les sections suivantes vont explorer le domaine de conception des interconnexions sur puce afin de déterminer quelle est l'interconnexion la plus efficace dans un contexte non prédictible où un grand nombre de modules doit être connecté.

1.2 Choix d'un type d'interconnexion

Les réseaux sur puce sont devenus une des solutions incontournables. Cette première partie va permettre de comprendre pourquoi il existe un tel engouement autour de ces réseaux sur puce.

Pour répondre au besoin en puissance de calcul dans les systèmes embarqués, de plus en plus d'éléments de calcul sont placés sur une même puce. Face à ces besoins en communication, trois grandes familles d'interconnexion sur puce existent et vont être détaillées.

1.2.1 Connexion point à point

La connexion point à point consiste à créer un lien de communication spécialisé entre chaque élément à interconnecter (Figure 1.2(a)). Ainsi lorsqu'une communication doit être réalisée, l'émetteur place le message sur le lien dédié et le récepteur renvoie une confirmation de réception. Ce type de connexion a l'avantage d'être extrêmement performant puisque qu'aucun partage de support de communication n'est réalisé. Il n'y a donc aucun arbitre ou autre stratégie de partage de ressources à mettre en place. Cependant du fait qu'un lien est créé entre chaque module qui communique, il est nécessaire de connaître à l'avance qui communiquera avec qui ou de connecter l'ensemble des modules entre eux dans un cas non prédictible. Dans notre contexte de contrôle dynamique, tous les modules sont a priori susceptibles de communiquer entre eux et donc toutes les possibilités d'interconnexion doivent être envisagées. Ceci conduit à un nombre important de liens et donc à une consommation importante de surface.

Cependant afin de réduire la surface de ce genre de connexion, il est possible de limiter le nombre d'entrée/sortie. Ainsi le Crossbar (Figure 1.2(b)) propose une solution pleinement connectée dont chaque élément possède une seule entrée/sortie. Dans ce cas, un arbitrage devient nécessaire pour n'avoir qu'un seul message à la fois à chaque entrée. IBM propose le Cyclops64 Crossbar étudié dans [10] qui permet de connecter jusqu'à 95 éléments. La complexité d'un Crossbar augmente avec le carré du nombre d'éléments connectés. Il est donc évident que ce genre d'interconnexion limite le nombre d'éléments interconnectés si la surface est une contrainte forte. Les connexions point à point possèdent les meilleures performances quelque soit le nombre d'éléments mais en contrepartie leur coût en surface est élevé.

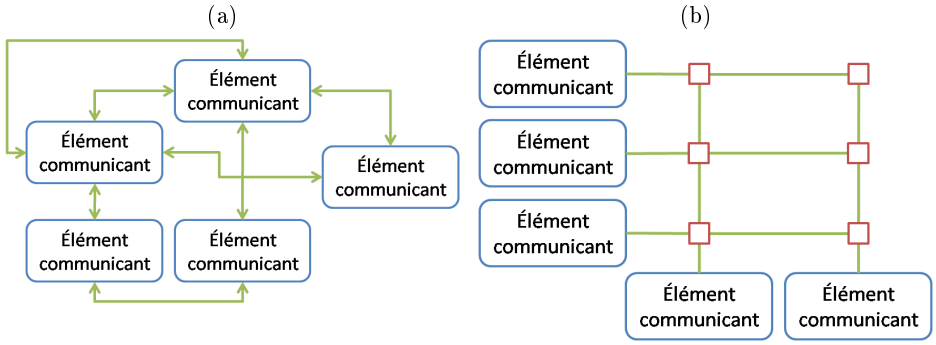


FIG. 1.2 – (a) Représentation d'une connexion point à point, (b) Représentation d'un Crossbar

1.2.2 Connexion par bus

A la différence des interconnexions point à point, le bus cherche à partager les ressources de communication. Il est donc constitué de deux éléments principaux : un arbitre et un support de communication. Ce support de communication relie tous les éléments communicants (Figure 1.3(a)). Ainsi lorsqu'un élément veut envoyer un message sur le support de communication, il consulte tout d'abord l'arbitre afin d'ajouter sa requête sur une liste d'attente. Une fois qu'il a reçu l'autorisation, il envoie le message sur le support. Tous les éléments connectés reçoivent le message et seuls les destinataires du message le mémorisent. L'identification du ou des destinataires est réalisée à l'aide d'une adresse envoyée avant ou en même temps que le message. A tout moment il n'y a qu'un seul message sur le support de communication. Ce mode de fonctionnement partage les ressources de communication entre tous les éléments. Il existe une multitude de styles d'arbitrage qui dépendent des contraintes données. Ainsi, on trouve des arbitrages très simples comme le First In First Out (FIFO) qui est un arbitrage centralisé ou le Time Division Multiple Access (TDMA) qui est distribué à chaque interface. D'autres arbitrages plus compliqués existent également prenant en compte des priorités sur les émetteurs des messages par exemple.

Le bus est l'interconnexion la plus répandue du fait de sa simplicité de fonctionnement. De plus, le fait de partager un seul lien entre tous les modules lui permet d'être pleinement connecté avec un minimum de coût en surface. Cependant l'unicité du support de communication et donc le partage de la bande passante impose une limite au nombre d'éléments connectés. Dans le cas où tous les éléments connectés se partagent le temps d'accès au bus, au-delà d'une dizaine d'éléments maîtres (qui émettent des requêtes) sur le bus, on peut estimer que l'attente avant l'envoi du message devient non négligeable. Afin d'augmenter le nombre d'éléments sans trop sacrifier les performances, il est possible de connecter entre eux plusieurs bus à l'aide de ponts afin de cloisonner les communications. Ainsi le nombre d'émetteurs sur chaque bus diminue, ce qui augmente la part de bande passante pour chaque émetteur. On retrouve différents industriels proposant chacun leur bus. Ainsi ARM propose le Bus AMBA [11], ST Microelectronics le STBus [12] ou encore IBM le CoreConnect [13].

D'un autre côté, les communications entre éléments de deux bus différents sont complexifiées par le passage des différents ponts. Afin d'augmenter les performances, il est également possible de multiplier le nombre de bus disponibles afin de limiter les effets d'engorgement dus au partage des liens. On nommera ce genre d'interconnexions multi-bus (Figure 1.3(b)). Ceux-ci peuvent être vus comme un Crossbar incomplet. Augmenter le nombre de bus a pour effet de multiplier la bande passante et donc les performances mais étend par la même occasion la surface. En résumé, le bus est une solution simple et peu coûteuse d'un point vue surface. Les performances d'un bus sont contraintes par le nombre d'écrivains potentiels sur le bus ce qui pose problème pour de gros systèmes. Cette interconnexion a pour avantage de proposer une solution qui supporte naturellement les communications non prédictibles.

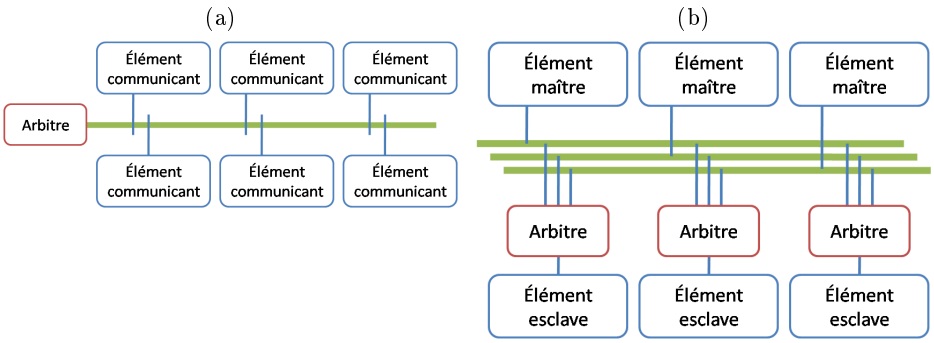


FIG. 1.3 – (a) Représentation d'un bus, (b) Représentation d'un multi-bus

1.2.3 Réseaux sur puce : NoC (Network on Chip)

Du fait des limitations des bus et des connexions point à point, en ce qui concerne respectivement les performances et la surface, il est devenu nécessaire de proposer une nouvelle solution modulaire. Ainsi dès le début des années 2000 [14], W.J.

Dally et B. Towles formalisent la notion de NoC afin de remplacer les connexions longues distances sur les puces. Les réseaux sur puce dérivent des réseaux entre ordinateurs. Ils possèdent des caractéristiques similaires même si les contraintes sont différentes. Ainsi l'idée de base est de faire circuler l'information dans le réseau sur puce sous forme de messages découpés en paquets. Comme les réseaux Ethernet, ils sont constitués d'un ensemble de liens et de routeurs permettant la communication entre les éléments communicants (Figure 1.4(a)). Une interface réseau est placée entre le routeur et l'élément afin de mettre en forme le message. Les réseaux sur puce permettent de créer toutes les combinaisons possibles de routeurs et de liens. Ils sont donc modulaires selon les contraintes imposées. Cette modularité permet l'ajout de nouveaux éléments communicants sans avoir un impact important sur les performances comme pour le bus. De plus, cet ajout d'éléments se fait avec un minimum de liens ce qui limite son coût à la différence des connexions point à point.

Une autre notion adaptée aux NoC est la notion de couches [15, 16]. On retrouve les sept couches réseaux du modèle OSI (physique, liaison, réseau, transport, session, présentation et application) telles qu'on les voit sur la Figure 1.4(b). On retrouve d'ailleurs sur cette figure la partie du NoC réalisant les différentes couches. Dans l'ordre de la couche physique à la couche application,

- La couche physique définit les moyens physiques mis en œuvre pour le transport de l'information ainsi que les caractéristiques à respecter. Par exemple, pour le transport électrique, les informations de tension, de temps, etc. sont définies.
- La couche liaison assure le transfert d'un noeud à l'autre ainsi que de la fiabilité des transferts. Cette couche s'occupe donc du partage des liens physiques en intégrant de l'arbitrage. Cet arbitrage peut se faire, par exemple, par du TDMA ou encore par jeton (« Token Ring »). Elle peut mettre en place aussi des fonctions de détection et de correction d'erreurs afin d'assurer la fiabilité.
- La couche réseau met en place les techniques de routage à l'intérieur du réseau. Ces routages peuvent être statiques et offrir un seul chemin possible, dynamique et offrir des chemins multiples en fonction de l'engorgement du réseau. Cette couche permet d'abstraire la topologie et les liens du réseau.
- La couche transport s'occupe des fonctions au niveau paquet. Ainsi ce niveau découpe ou assemble les messages. Cette couche s'occupe également du contrôle des paquets à l'envoi et à la réception (« end-to-end »). Elle abstrait toutes notions de paquets pour les couches supérieures.
- La couche session gère les notions de synchronisation entre les modules. Elle permet de vérifier la bonne réception des messages. Elle offre aussi la possibilité d'établir une connexion entre deux modules, de la maintenir et de la fermer.
- La couche présentation prend en charge la transformation des formats des différents messages afin de permettre la bonne communication. Un exemple simple est la transformation d'une requête provenant d'un processeur en un message géré par le réseau.
- La couche application permet d'ajouter une dernière couche d'abstraction afin de simplifier l'envoi de données tel qu'une trame vidéo. De cette façon, aucune

contrainte due au réseau sur puce n'est visible depuis le niveau utilisateur, ce qui facilite son utilisation.

De plus, l'utilisation des réseaux par rapport au bus ou au point à point permet d'ajouter de nouveaux services dans les interconnexions sur puce. On peut citer par exemple la détection et la correction d'erreurs au niveau du message. Ils offrent une solution pleinement connectée grâce aux mécanismes de routage avec de bonnes performances et qui autorisent le partage de ressources. Ceci répond à nos besoins de connectivité, de performance tout en prenant en compte les contraintes de l'embarqué. En contrepartie, l'ajout d'interfaces réseaux augmente la latence d'envoi et de réception d'un message et pénalise le réseau par un surcoût en surface qui n'existe pas dans le cas du bus et des connexions point à point. Nous allons détailler par la suite les paramètres des réseaux sur puce sans approfondir la couche physique des réseaux. Pour plus d'information sur cette dernière, [17] détaille les couches physique et liaison.

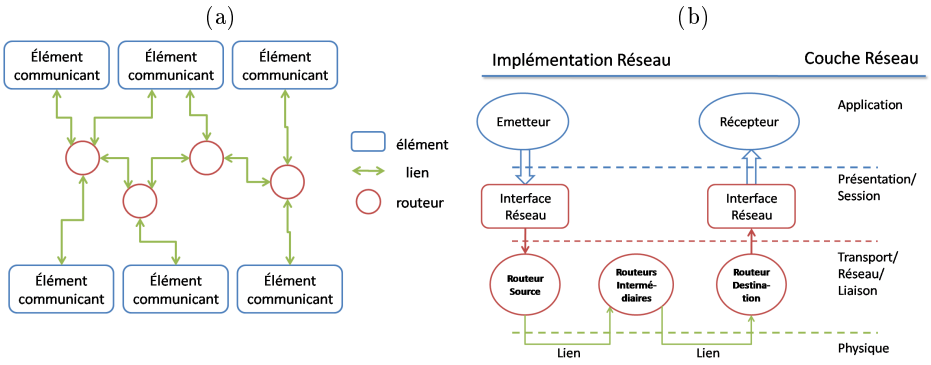


FIG. 1.4 – (a) Réseau sur puce constitué de liens et de routeurs, (b) Les différentes couches réseau et leur implémentation

1.2.4 Synthèse sur les modes de communication

Dans cette section, nous avons détaillé les différents types d'interconnexions répandues dans le monde des systèmes embarqués. Le tableau ci-dessous récapitule les caractéristiques de chacun. Un point important par rapport à notre contexte est la « scalabilité » de l'interconnexion choisie. On dit d'un système qu'il est « scalable » lorsque ce système est capable de garder ses caractéristiques lors de l'augmentation de sa taille.

Nous avons expliqué l'arrivée des réseaux sur puce dans le domaine de l'embarqué par une nécessité de connecter de plus en plus d'éléments sans sacrifier les performances et en limitant le surcoût en surface. Les réseaux sur puce permettent le partage de liens comme dans le bus tout en gardant de bonnes performances vis-à-vis des connexions point à point. Ils introduisent de nouvelles possibilités telles que la correction d'erreurs ce qui ajoute de la fiabilité. Ils nous permettent donc de

Type d'interconnexions	Bus		Point à point		Réseau sur Puce	
Nb d'éléments interconnectés	<=10	>10	<=10	>10	<=10	>10
Performance	+	- -	++	++	+	+
Surface	++	++	-	- -	-	-
« Scalabilité"	+		-		++	

TAB. 1.1 – Résumé des caractéristiques des différentes solutions. Les signes +/- permettent de caractériser les différents points caractéristiques recherchés. Le signe + indique que l'interconnexion obtient de bons résultats et inversement pour le signe -. Le nombre de signe pondère la caractérisation.

connecter des dizaines de modules dans des conditions optimales et en permettant le côté non prédictible des communications.

Les réseaux sur puce semblent répondre à nos attentes cependant l'espace de conception est très vaste. C'est pourquoi la section suivante décrit et commente les caractéristiques principales des réseaux sur puce à l'aide d'une étude bibliographique.

1.3 Les caractéristiques des réseaux sur puce

Cette section présente tout d'abord le point de vue théorique de l'étude des réseaux. Puis elle propose une description pratique des différentes caractéristiques des réseaux sur puce et analyse chacune d'entre elles. Cette étude a pour but de déterminer quelles sont les caractéristiques possibles dans le cas d'une architecture massivement parallèle sur puce. Pour cela, dans un premier temps, il est nécessaire de connaître les différents éléments d'un réseau sur puce. Et dans un deuxième temps, cette section présente les différentes possibilités pour chaque caractéristique.

1.3.1 L'étude théorique des réseaux sur puce

Les réseaux, en général, peuvent être vus d'un point vue théorique. On retrouve de nombres ouvrages, [18, 19] par exemple, qui traitent de cette problématique. Dans ce cas, le réseau est vu comme un graphe où chaque routeur est considéré comme un noeud du graphe et chaque lien comme un arc du graphe. Un réseau se résume, ensuite, à une suite de propriétés mathématiques et de paramètres tels que le diamètre du réseau qui correspond à la distance moyenne parcourue dans le réseau, ou encore le degré des routeurs qui correspond au nombre d'entrées et de sorties de celui-ci.

Plus récemment, le network calculus [20] est un environnement permettant l'analyse des performances et des garanties de bande passante dans le réseau. Cet environnement reprend les théories sur les réseaux. Il se base sur l'enveloppe du trafic dans le réseau, les contraintes des liens ou encore sur la gestion de la congestion afin de calculer la taille des mémoires tampon dans le système ou encore les délais d'un point à un autre.

Ces études théoriques sont adéquates dans le cas de grands réseaux où le nombre

de routeurs dépasse le millier et que le réseau devient difficilement analysable par simulation ou dans un cas de contraintes dures de fiabilité. Dans notre cas, le réseau comporte quelques centaines de routeurs ce qui reste raisonnable pour une étude pratique. C'est pourquoi la suite de ce chapitre traite les réseaux d'un point de vue pratique et non théorique.

1.3.2 Les éléments de base des NoC

Ce paragraphe détaille les différents constituants des réseaux sur puce afin d'introduire le vocabulaire utilisé dans le reste de la section. Dans un réseau, les différents modules envoient des paquets de données. Ces paquets sont découpés en « flits » qui correspondent aux plus petits éléments gérés par le réseau. Comme nous l'avons dit avant, les réseaux sur puce sont composés de routeurs, de liens et d'interfaces réseau.

Le routeur permet l'aiguillage des paquets dans le réseau. La Figure 1.5 montre un exemple élémentaire de routeur avec une mémorisation en entrée. Il est constitué au minimum d'une matrice d'interconnexion, d'un arbitre et d'éléments de mémorisation. Les éléments de mémorisation (buffer) permettent de mémoriser les flits qui arrivent ou qui partent d'un lien physique. Il existe différentes implémentations de routeurs avec les buffers placés à l'entrée, à la sortie ou bien aux deux extrémités [21]. Nous avons choisi de nous placer, pour le reste de l'étude, dans le cas où les buffers sont à l'entrée car c'est la solution la plus simple et la plus répandue. La matrice d'interconnexion connecte toutes les entrées à toutes les sorties. Elle est contrôlée par l'arbitre qui détermine les connexions entre les entrées et les sorties. Il prend en compte la destination de chaque message et ainsi décide des connexions entrée/sortie en fonction des contentions et des priorités éventuelles.

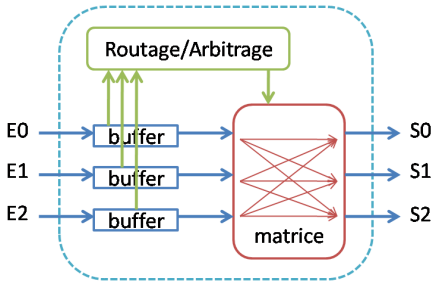


Fig. 1.5 – Routeur élémentaire.

Le lien physique est le moyen de communiquer entre deux routeurs. Il peut être synchrone ou non et ne possède aucun élément de mémorisation. Les liens physiques peuvent être mono ou bidirectionnels.

L'interface réseau est un élément important qui permet d'isoler l'élément communiquant du réseau. De ce fait, on peut facilement connecter un nouvel élément qui n'aurait pas le même protocole de communication étant donné que ce module permet d'adapter son interface. Pour ce faire, l'interface réseau peut être vue comme

la mise en cascade d'un adaptateur de protocole de communication et des fonctions usuelles des couches réseau « présentation » et « session ». Cette interface réseau peut apporter de nouvelles fonctionnalités comme la possibilité de réaliser une barrière de fréquence afin d'autoriser la multiplicité des domaines de fréquence sur une même puce.

Cette sous section a permis d'expliquer les notions. La sous section suivante va décrire les caractéristiques principales. Dans un premier temps, nous allons détailler les différents mécanismes de communication possibles.

1.3.3 Les mécanismes de transport

Les mécanismes de transport correspondent à la manière dont le chemin du message est créé à travers le réseau. Ce paramètre est le premier présenté car il agit sur une grosse partie des autres paramètres. Il existe deux techniques : la commutation par circuit et la commutation par paquet.

Avec la commutation par circuit, le chemin pris par un paquet, n'est pas déterminé par le paquet lui-même mais par un contrôle distribué ou centralisé. Avec ce type de transport, le paquet est constitué exclusivement de l'information à transporter. Les chemins peuvent être déterminés statiquement ou dynamiquement. Le PicoArray de PicoChip [22] ou l'ASoC (adaptive System-On-a-Chip) du MIT [23] proposent de définir hors-ligne des séquences de chemin à travers le réseau. Ainsi dans l'ASoC, pour chaque routeur, un séquenceur vient lire dans une mémoire la configuration de la matrice d'interconnexion. Le problème de cette technique est qu'elle impose beaucoup de mémoire dans chaque routeur. De plus, le nombre de séquences est forcément borné par la taille de la mémoire ce qui limite les possibilités de reconfiguration et donc l'utilisation de l'architecture. Dans le cas du PicoArray, le changement de connexions dans la matrice du routeur est conditionné par le temps. On parle de Time Division Multiple Access (TDMA). Ceci permet de ne pas avoir à mémoriser des séquences et donc de limiter la mémoire nécessaire. Le problème de ce mécanisme provient de la réservation du chemin pour un seul utilisateur ce qui risque de contribuer à l'engorgement du réseau.

Dans tous les cas, définir les chemins statiquement implique de connaître hors-ligne toutes les informations concernant les communications et donc de savoir à l'avance l'application précise ce qui n'est pas possible dans le cas d'applications dynamiques. Æthereal de Philips Research [24] implémente une commutation par circuit dynamique. Lorsqu'un élément veut émettre un message vers un autre élément, il doit dans un premier temps réserver des créneaux de temps dans chaque routeur sur son chemin. Afin de créer ce chemin, un message de réservation est envoyé et alloue les créneaux dans chaque routeur. Lorsqu'un routeur répond négativement au message, le message de réservation est effacé et un message de refus est envoyé à travers le chemin déjà réservé afin d'annuler les créneaux précédemment réservés. Ce mécanisme répond à la contrainte de dynamisme des applications mais chaque communication met un temps variable pour créer le chemin, ce qui pose un problème lorsque les applications ont des contraintes temporelles. De plus, les

auteurs montrent que ce mécanisme reste coûteux au niveau surface par rapport à la communication par paquet.

Dans la communication par paquet, le chemin est décidé dynamiquement en fonction des informations contenues dans l'en-tête du paquet lorsqu'il arrive dans un routeur. Il existe trois manières principales de passer le message entre les routeurs.

La première technique, le store-and-forward est une technique dérivée des réseaux entre ordinateurs. Les paquets sont mémorisés entièrement avant d'être envoyés au routeur suivant. Ce mécanisme permet en cas de conflit entre deux paquets de mémoriser tout le paquet en un seul routeur. Cependant il est coûteux au niveau surface car chaque routeur doit être capable de mémoriser un paquet en entier sur chaque entrée. Par conséquent, cette solution n'est pas retenue dans le domaine de l'embarqué. On peut citer, tout de même, INTEL 80 Cores [25] qui implémente cette technique.

La seconde technique, le wormhole, consiste en l'envoi (le plus rapidement possible) d'un paquet par morceau à travers le réseau. Ainsi, un paquet est envoyé flit après flit à travers le réseau dès que le chemin se libère. Contrairement à la technique précédente où le paquet est mémorisé en entier dans le routeur, à tout moment, un paquet est donc réparti à travers les différents routeurs qui jalonnent son chemin. En cas de contention, le message reste distribué sur l'ensemble des routeurs du chemin. Afin de ne pas mémoriser, la destination dans chaque flit, un flit d'en-tête réserve le chemin et le flit de queue le libère. Ce mécanisme permet de diminuer le temps d'arrivée de chaque paquet mais a pour effet de surcharger le réseau en cas de contention. Cette technique est utilisée dans la majorité des NoC [26] car elle nécessite un minimum de mémoire dans chaque routeur.

La troisième technique est le Virtual-Cut-Through. Elle consiste à utiliser le wormhole mais en cas de contention, le paquet est mémorisé dans le dernier routeur. Ce mécanisme permet d'avoir les avantages des deux précédentes techniques qui sont la rapidité d'envoi et la mémorisation dans le routeur en cas de contention. Cependant il a aussi les inconvénients qui sont l'étalement du message à travers le réseau et la nécessité d'avoir des buffers capables de mémoriser un paquet en entier. Il est peu courant dans la littérature, par exemple, [27] l'utilise dans le réseau Protéo.

Nous avons dit précédemment qu'afin de trouver son chemin dans le réseau, le paquet possède un en-tête qui contient soit le chemin jusqu'à la destination, soit l'adresse de la destination. MicroSpider [28], réalisé au sein du Lab-STICC, par exemple, offre la possibilité de placer dans l'en-tête du paquet les différentes sorties que le paquet doit prendre à chaque routeur. Cela permet de connaître l'émetteur du message et ainsi éviter les attaques par faux messages sur le réseau ou des erreurs. Les trafics sont alors sécurisés étant donné que l'on peut retracer un expéditeur mais cela impose à chaque émetteur de connaître la topologie du réseau afin de générer la séquence de sortie. La deuxième solution consiste à placer dans l'en-tête l'adresse de la destination. Avec cette deuxième technique, le routeur doit déterminer vers quelle sortie, il doit acheminer le paquet en fonction de cette adresse. Deux solutions sont possibles. Dans la première solution, le routeur peut s'aider d'une table de routage comme dans [29], mais cela implique une capacité de mémorisation proportionnelle

au nombre de destinations ce qui est en opposition avec nos contraintes embarquées. La deuxième solution consiste à implémenter un algorithme de routage dans chaque routeur. Cela permet de guider dynamiquement les paquets sans avoir à mémoriser des tables de routages. Cette solution est celle retenue par la majorité des NoC car elle consomme un minimum de surface. Le paragraphe 1.3.5 va détailler ces algorithmes.

En conclusion, dans notre cas de système embarqué avec un trafic dynamique, la commutation par paquet guidée par algorithmes de routage semble être le choix le plus pertinent. En effet, cette solution permet d'accepter un trafic dynamique sur les réseaux en limitant la surface nécessaire au bon fonctionnement. Une fois le mécanisme de transport choisi, la deuxième caractéristique influente pour la suite est la topologie.

1.3.4 La topologie

La topologie est la caractéristique qui détermine la manière dont le réseau s'organise. Il y a deux catégories de base : régulière et irrégulière qui correspondent respectivement à des topologies dont on peut extraire un motif répétitif ou non.

1.3.4.1 Les topologies régulières

Les topologies régulières sont caractérisées par leurs motifs répétitifs. A l'intérieur de cette catégorie, il y a deux sous-catégories : les topologies directes et les topologies indirectes.

Les topologies directes - Une topologie est dite directe lorsque tous les routeurs sont connectés à un composant, un processeur, une mémoire ou tout autre module autre que le réseau. On différencie ces topologies en fonction du nombre d'entrée/sortie que chaque routeur possède. On trouve alors différentes topologies directes régulières possibles.

La topologie anneau comporte deux entrées/sorties en plus du lien vers le composant connecté. On peut voir sur la Figure 1.6(a) qu'un paquet ne peut transiter que de proche en proche. Cette topologie a l'avantage d'avoir un routage très simple. En effet, en fonction de la destination, il choisit de partir vers l'un de ces deux voisins. Cette simplicité implique l'unicité des chemins entre deux routeurs et un nombre important de sauts[1] dès que le nombre d'éléments à connecter augmente. Les longues distances à parcourir entraînent des latences importantes dans le réseau. De plus l'unicité du chemin impose une intolérance aux pannes et de nombreuses possibilités de contentions. Une solution simple pour repousser ces deux problèmes est de dupliquer le nombre d'anneaux entre les routeurs sur la puce. On retrouve cette solution dans le Larrabee d'Intel [30] et le CELL d'IBM [31]. Par exemple, l'EIB (Element Interconnect Bus) du CELL propose 4 bus bidirectionnels d'une largeur de 128 bits chacun.

[1] distance entre deux routeurs

En ajoutant une liaison en plus à chaque routeur, on peut obtenir la topologie décrite Figure 1.6(b). On voit sur cette figure que chaque routeur possède trois entrées/sorties. Cet ajout permet de réduire les latences entre deux routeurs. On peut citer le réseau Octagon de STMicroelectronics [32] qui offre la possibilité à chaque émetteur d'atteindre un destinataire en seulement deux sauts sous réserve de rester en dessous de huit éléments. Cette structure est limitée en nombre d'éléments interconnectés donc, pour remédier au problème, [32] propose de combiner des réseaux Octagon. La combinaison des réseaux Octagon ensemble complexifie sensiblement le routage tout en augmentant le nombre de modules connectés. Dans [33], les auteurs ne se limitent pas à huit éléments et proposent Sipdergon.

Toujours en augmentant le nombre de liens, on obtient deux topologies régulières planes : la grille et le tore représentés sur les Figures 1.6(c) et 1.6(d). Le tore est une grille améliorée dont les extrémités ont été reliées entre elles ce qui réduit le nombre de sauts nécessaires pour accéder à n'importe quelle destination. Ces deux réseaux permettent une diversité de chemins entre deux routeurs et disposent de routages simples à mettre en place. Malgré de meilleures performances, le tore est très peu présent dans les topologies choisies. Cela s'explique par le fait que la grille est plus facilement implantable sur puce même s'il est possible de replier le tore sur lui-même afin de simplifier son implémentation. Les caractéristiques simples de la grille en font l'une des topologies les plus utilisées [26]. On peut citer Æthereal de PHILIPS Research [24], ANoC (Asynchronous Network-on-chip) du CEA LETI [34] ou encore Nostrum de Royal Institute of Technology (KTH) [35], Cliché de S. Kumar [36].

En augmentant encore le nombre de liens à chaque routeur, on arrive à des topologies en trois dimensions qui sont plus difficiles à implémenter sur un seul plan. Différentes topologies 3D commencent à voir le jour dans de futurs systèmes embarqués grâce notamment à la possibilité d'utiliser l'empilement 3D. Dans [37], Feero et Pande présentent une grille 3D. Les topologies 3D sont très dépendantes de la possibilité de les réaliser en matériel. Nous avons choisi de ne pas parler d'implémentation matérielle dans cette étude.

Toutes ces topologies directes privilégient les communications de voisin à voisin. Ainsi, selon le nombre de liens à chaque routeur, les topologies obtenues sont de plus en plus performantes mais aussi de plus en plus imposantes d'un point de vue surface silicium.

Les topologies indirectes - Les topologies indirectes, à la différence des topologies directes, utilisent des routeurs qui peuvent ne pas posséder de liens avec des modules. On distingue deux catégories régulières : l'arbre et le multi-étage.

La topologie arbre est constituée de différents étages de routeurs permettant la communication. La topologie peut être simple ou élargie en fonction du nombre de routeurs dans chaque étage. L'arbre simple comporte le minimum de routeurs nécessaire par étage afin de connecter tous les étages entre eux (Figure 1.7(a)). Dans l'arbre étendu ou élargi, le nombre de routeurs par étage est fixe mais supérieur au minimum de l'arbre simple (Figure 1.7(b)). Dans le cas d'un arbre simple,

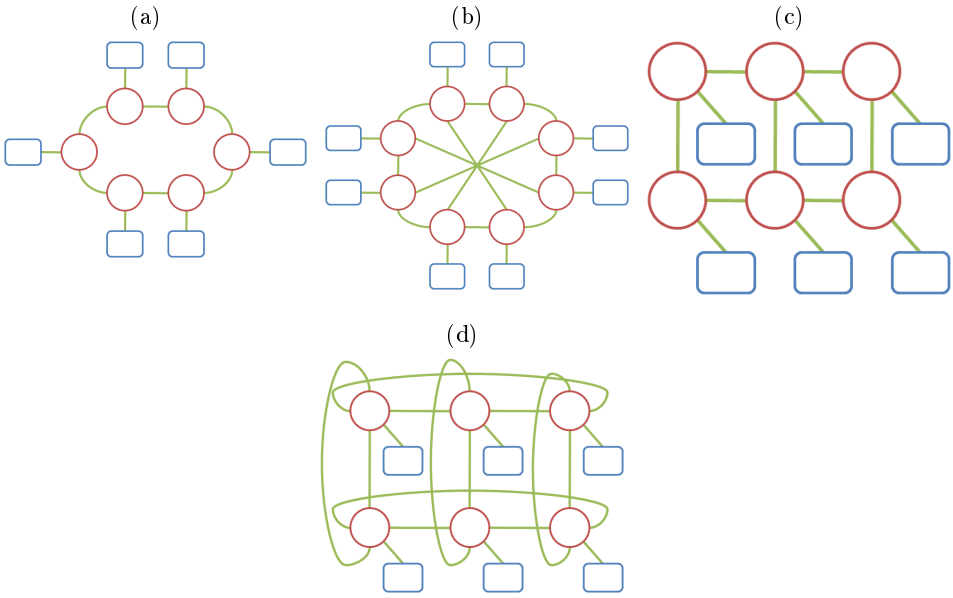


FIG. 1.6 – (a) topologie anneau, (b) topologie octagon, (c) topologie grille 2d, (d) topologie tore 2d. Les différents objets sont repérés par des couleurs. Le rouge représente les routeurs, le bleu les modules connectés et le vert les liens physiques

le paquet n'a qu'un seul chemin jusqu'à sa destination ce qui permet un routage simple. Dans le cas d'un arbre élargi, il y a une diversité des chemins qui impose d'avoir un routage pouvant gérer la multiplicité des chemins. Le nombre d'étages de l'arbre représente les différentes couches de routeurs que le message pourra traverser. Ce nombre d'étages est dépendant du nombre d'entrées/sorties des routeurs et du nombre d'éléments à interconnecter. Ainsi plus le nombre d'entrées/sorties des routeurs est important, plus le nombre d'étages sera réduit. Le réseau SPIN du laboratoire Lip6 [38] est un exemple de topologie arbre. Il implémente une topologie d'arbre élargi constituée de routeurs à quatre entrées/sorties. La topologie arbre est limitée par le nombre de liens entre les différents étages. Cette topologie favorise, comme les topologies directes, les communications locales, étant donné que plus la communication est lointaine et plus on doit traverser d'étages dans l'arbre.

La topologie multi-étage est une autre forme des réseaux indirects. Cette topologie réduit les limitations de l'arbre entre les étages en augmentant entre chaque étage le nombre de routeurs et en permettant au message d'aller dans le sens trigonométrique ou inversement. Ces routeurs possèdent, comme dans le cas de l'arbre, différentes entrées/sorties. Le nombre de ces entrées/sorties va influencer le nombre de routeurs par étages et le nombre d'étages. Il existe différentes manières de connecter ces étages. Butterfly multi-étage, Clos-Benes multi-étage sont les deux principales manières de connecter les routeurs [21]. La Figure 1.7(c) présente la topologie Butterfly. Dans cette topologie, le nombre de routeurs par étage est fixe et correspond au nombre d'éléments à connecter divisé par le nombre d'entrées du routeur. Le nombre

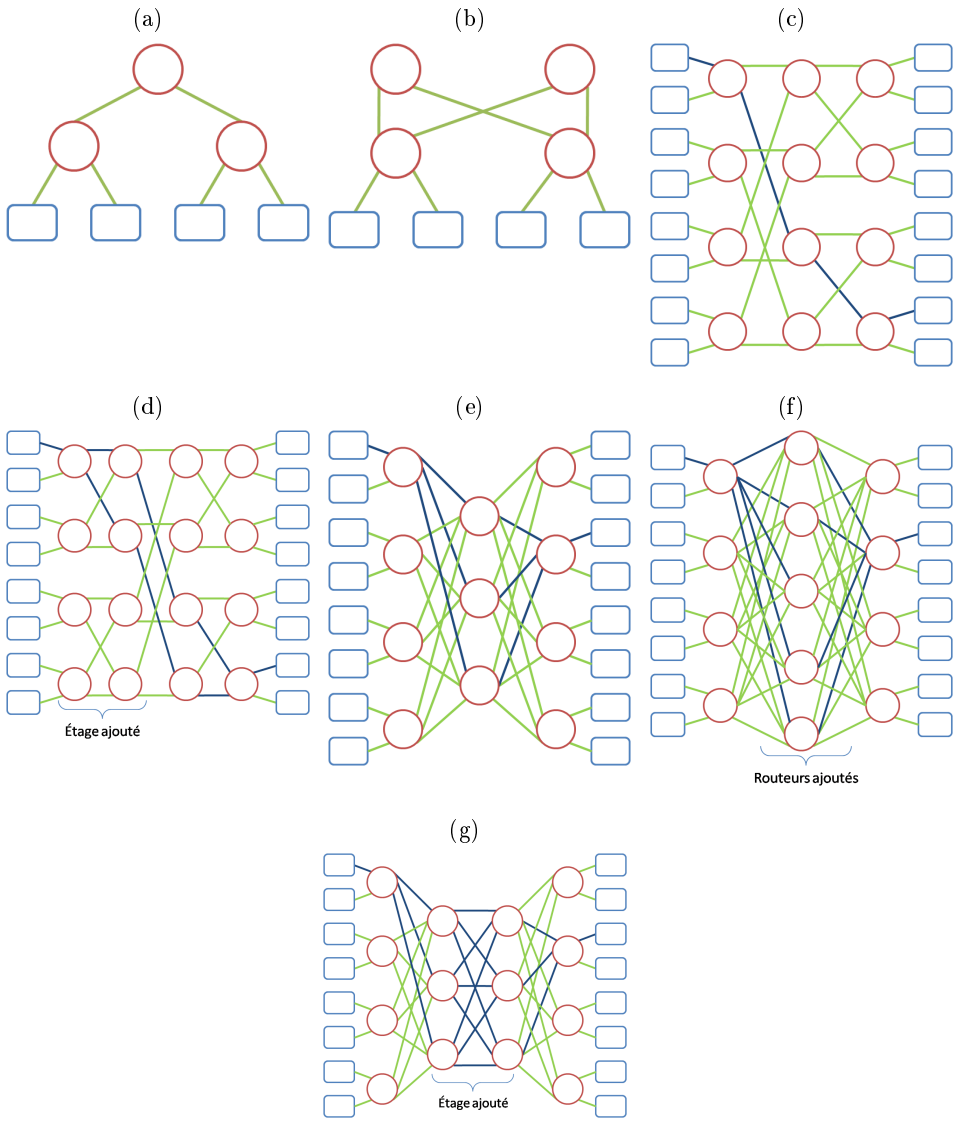


FIG. 1.7 – (a) topologie arbre, (b) topologie arbre élargie, (c) multi-étage butterfly, (d) multi-étage butterfly avec un étage supplémentaire, (e) multi-étage clos, (f) multi-étage clos avec deux routeurs centraux supplémentaires, (g) multi-étage clos avec un étage supplémentaire. Le code couleur reste le même que dans la figure 1.6. Les liens physiques en bleu foncé permettent de visualiser la multiplicité des chemins possibles.

d'étages se calcule par le logarithme du nombre d'éléments dans la base du nombre d'entrées du routeur. Cette topologie propose un seul chemin possible entre deux éléments et donc un routage simple. Afin de proposer plusieurs chemins, il est possible de rajouter des étages dans la topologie. Ainsi, sur la Figure 1.7(d), le premier étage a été ajouté et permet de doubler le nombre de chemins possibles. La topologie Clos-Benes (Figure 1.7(e)) offre de base une plus grande diversité de chemins

entre tous les éléments. Cependant, on peut aisément augmenter la diversité de chemins en augmentant le nombre de routeurs dans les étages centraux (Figure 1.7(f)) ou encore en augmentant le nombre d'étages comme pour le Butterfly multi-étage (Figure 1.7(g)). Il est important de mentionner que le fait d'ajouter des chemins supplémentaires, quel que soit le type de multi-étage, est coûteux des liens entre ces routeurs sont ajoutés. De plus, les tailles de ces liens, qui différent selon les types de multi-étages, vont avoir un impact sur la surface finale du réseau. Les topologies multi-étages ont l'avantage d'avoir un accès uniforme quelle que soit la destination d'un message. Les réseaux sur puce ne se limitent pas aux topologies régulières. La suite décrit les topologies irrégulières.

1.3.4.2 Les topologies irrégulières

Les topologies irrégulières consistent en un assemblage de routeurs et de liens dont on ne peut pas extraire un motif régulier. Elles sont dans la majorité des cas, le résultat d'une observation précise des différents besoins en communication entre les différents éléments à connecter. Ainsi grâce à un profilage d'un ensemble d'applications à exécuter, une optimisation est réalisée sur le nombre de liens et les routeurs nécessaires au bon fonctionnement. [39] présente le réseau Xpipes qui propose un routeur optimisé que l'on utilise pour construire un réseau suivant une topologie quelconque. La société Arteris [40] propose également d'étudier précisément les besoins en communication et, avec l'aide de ses outils, accompagne l'utilisateur dans le choix des différents routeurs et liens constituant le réseau. Comme dans le cas de Xpipes, Arteris propose des routeurs modulaires permettant la réalisation de tous les réseaux.

Dans [41], les auteurs proposent d'utiliser les graphes de Bruijn pour connecter les routeurs. Ils utilisent ainsi une propriété connue de ce type de graphe qui est la minimisation des chemins entre deux points du graphe. Ainsi grâce à cette propriété le réseau peut connecter deux points de façon minimale à travers la topologie.

Ces réseaux irréguliers cherchent à optimiser les communications entre les éléments, ce qui permet de limiter la surface du réseau global mais en contrepartie ils s'adaptent difficilement à différents domaines d'application.

1.3.4.3 Les compositions de topologie

Les topologies précédemment citées sont capables d'être étendues afin de connecter de nombreux modules. Cependant lorsqu'ils sont étendus soit leur performance diminue (tore, grille, anneau) soit leur surface n'augmente pas proportionnellement avec le nombre de modules (Multi-étage). Afin de garder les performances et la surface des réseaux tout en augmentant leur connectivité, il est possible de combiner des topologies entre elles comme dans le cas de l'interconnexion Bus.

Une manière simple de les combiner est de construire des topologies hiérarchiques. On retrouve de la même façon que pour les réseaux à plat, des topologies hiérarchiques régulières et irrégulières. CHNoC s'inspire du réseau internet et pro-

pose une solution irrégulière [42]. Ce réseau peut intégrer tous les types d'interconnexion de toutes les topologies possibles. La topologie finale de ce réseau dépend des besoins de communication entre les différents modules du système. Dans [22], le PicoArray rassemble par quatre les processeurs entre chaque routeur d'un réseau grille 2D. Ces groupements permettent de réduire les latences dans le réseau étant donné qu'il y a moins de routeurs à traverser. Pour la même raison, cela permet aussi de diminuer la surface sans diminuer le nombre de processeurs dans l'architecture. Dans [43], l'auteur ajoute un anneau au dessus d'une grille 2D afin de réduire les temps d'accès à une tuile centrale contenant les I/O. Ainsi un message qui sort de la puce doit dans un premier temps trouver son chemin jusqu'à un des points d'accès de l'anneau avant de pouvoir trouver l'I/O concerné. Cette stratégie a pour but de distribuer les accès aux I/O sur différents points d'entrées et donc de diminuer engorgement qu'il y aurait s'il n'existait qu'un seul point d'accès.

La hiérarchisation permet principalement de réduire les latences pour des communications longues distantes en diminuant le nombre de routeurs à traverser. Grâce à cette caractéristique, ces topologies sont souvent plus « scalable » que les autres topologies. Toutefois, les passages d'une hiérarchie à une autre sont des goulots d'étranglement si leur dimensionnement est mal réalisé. Elles favorisent donc les communications locales à la hiérarchie la plus basse.

1.3.4.4 Synthèse sur la topologie

La topologie est une caractéristique qu'il est difficile de déterminer par une étude papier. Une étude papier ne peut pas permettre de choisir la topologie adaptée, étant donné qu'il est nécessaire de les comparer dans le cadre de nos besoins de « many-processeurs » avec un contrôle dynamique. De plus d'après nos recherches, aucune étude ne propose de comparer un large panel de topologies d'un point vue performance et surface. En général, deux ou trois topologies sont présentées comme dans le cas de [44], où seulement une grille 2D et un butterfly multi-étage bidirectionnel sont comparés en termes de performance et de surface. Il montre que le butterfly multi-étage offre 30% de performance de plus que la grille et surtout il apparaît comme permettant une plus grande scalabilité. On cherche donc la topologie la plus adaptée dans un cas donné. Une vision opposée consiste à créer la topologie en fonction des besoins. Par exemple, Murali et De Micheli [45] proposent un outil qui génère une topologie en fonction de l'application. Ceci se fait à partir d'un ensemble d'applications qui sont profilées afin de déduire des séquences de communication. A partir de ces séquences, l'outil crée la topologie. Une fois la topologie choisie, il faut déterminer la manière dont les paquets se déplacent à l'intérieur.

1.3.5 Les algorithmes de routage

Les algorithmes de routage, implémentés dans chaque routeur, permettent le choix de la sortie à prendre pour chaque paquet. Ceci implique que les algorithmes de routage sont dépendants de la topologie. Toutefois il existe deux familles d'algo-

rithmes de routage : les déterministes et les adaptatifs.

Les algorithmes déterministes sont ceux pour lesquels on peut prévoir le chemin qui sera choisi par un paquet en fonction de sa destination. Le routage « dimension-order » ou XY [46] est, par exemple, un des routages déterministes les plus simples. Il ne peut fonctionner que dans une grille 2D ou un tore 2D car il fait déplacer le paquet uniquement horizontalement avant de le déplacer verticalement lorsqu'il se trouve dans la colonne de sa destination. Cet algorithme est souvent utilisé car il est extrêmement simple à implémenter. En revanche sa simplicité implique qu'il n'y a pas de décision prise en fonction de l'état du réseau. Pour les topologies régulières, un algorithme simple et déterministe existe en général. L'algorithme « destination-tag-routing » permet, par exemple de router des paquets à travers un multi-étage butterfly. Dans ce cas, l'adresse est la transcription de la destination dans la base 2 (si il y a deux sorties par routeur). L'adresse devient, du fait de la topologie butterfly, la suite des sorties à prendre à chaque routeur pour traverser le réseau.

Les routages adaptatifs sont utilisés afin de modifier, à tout moment, le chemin pris par le paquet en fonction de l'état du réseau. Ces algorithmes peuvent être minimaux s'ils fournissent un chemin qui est de longueur minimale. Ces algorithmes peuvent avoir différents buts comme limiter au maximum la latence d'un paquet, ou encore permettre un routage même avec des liens défectueux. Le problème majeur des routages adaptatifs est qu'ils peuvent facilement créer des « livelock » entre les paquets étant donné que les paquets peuvent se déplacer librement dans le réseau. Ce phénomène apparaît lorsqu'un paquet n'arrive pas à sa destination et tourne en rond dans le réseau. Glass et Ni définissent dans [47] le « turn model » qui permet d'aider au développement de routage adaptatif. Ils montrent que les « livelock » sont dus à la possibilité qu'un paquet réalise un tour complet dans le réseau. Ainsi, si l'algorithme ne permet pas au paquet de faire un tour complet dans le réseau alors le routage est libre de « livelock ». Les algorithmes West-first [47] ou Odd-Even [48] sont des routages adaptatifs basés sur le « turn model » qui sont dits libres de « livelock ».

Les algorithmes de routage sont dépendants de la topologie choisie et des contraintes applicatives. Par exemple, si l'application nécessite des contraintes dures sur les temps de communication, les algorithmes déterministes permettront de garantir une latence dans le cas où il n'y a pas de contentions dans le réseau. Le routage permet de trouver le chemin entre les routeurs mais il est aussi important de maîtriser les communications entre routeurs.

1.3.6 Le contrôle de flux

Le contrôle de flux regroupe tous les mécanismes qui permettent le transfert des paquets entre les différents routeurs. On y retrouve la gestion de la bufferisation ainsi que les canaux virtuels.

Dans la communication entre deux routeurs, il est nécessaire de mettre en place un mécanisme permettant de connaître l'état des buffers d'entrée voisins afin de ne pas effacer ou perdre un flit. Il existe trois principales techniques. Etant donné que

les routeurs sont reliés par des connexions point à point, il est facile pour un routeur de comptabiliser le nombre de flits envoyés et ainsi connaître la place disponible dans chaque routeur voisin. Ce système est le contrôle par crédit utilisé dans SPIN [38] par exemple. Une solution plus simple consiste à mettre en place un signal full/not_full. Dans ce cas, un signal est placé entre chaque routeur afin d'informer son voisin s'il reste de la place dans son buffer. On peut aussi choisir un système plus robuste qui consiste en l'envoi d'une requête afin de connaître la place qui reste dans le buffer d'arrivée avant l'envoi de la donnée. Ces techniques ont peu d'impact sur la taille du routeur mais plutôt sur le nombre de signaux de contrôle nécessaires entre chaque routeur ou le nombre de messages à travers le réseau. La technique simple du signal full/not_full, reste la moins coûteuse et la plus rapide à mettre en place.

Les canaux virtuels ont pour but d'augmenter le taux d'utilisation des liens entre les routeurs en permettant à plusieurs canaux virtuels de transmettre sur un même canal physique. Pour cela, un certain nombre de canaux virtuels, implémentés par des buffers différents, sont connectés à une même entrée. Au moment de la décision du routage, il faut également choisir quel canal virtuel sera autorisé à utiliser le canal physique. Les canaux virtuels permettent donc également de réduire les interblocages entre paquets. En effet, si on place deux canaux virtuels sur un même canal physique, cela permet d'éviter que deux paquets se bloquent mutuellement. L'implémentation de canaux virtuels est coûteuse car autant de buffers que de canaux virtuels sont nécessaires pour chaque entrée.

Dans [49], l'auteur mesure les performances en fonction du nombre de canaux virtuels sur le réseau Hermès [50]. Cette étude montre que les canaux virtuels peuvent avoir un gain important suivant les cas de figure sur les performances surtout sur les grands réseaux. Cependant, le coût en surface augmente linéairement avec le nombre de canaux virtuels. Implémenter des canaux virtuels sur un système embarqué doit être un choix justifié car on double la taille de chaque routeur en ajoutant le premier canal virtuel.

Comme nous venons de le voir le contrôle de flux gère l'envoi des paquets entre les routeurs. Nous allons voir ensuite quelle taille de buffers choisir en fonction de l'utilisation.

1.3.6.1 La taille des buffers d'entrée

Afin de mémoriser les différents flits quand ils arrivent dans un routeur, les entrées des routeurs comportent des buffers. Ces buffers possèdent une largeur et une profondeur. Leur largeur correspond toujours à la largeur des liens entre les routeurs afin de simplifier la gestion. Cependant la profondeur de ceux-ci peut nécessiter des tailles variables en fonction du comportement voulu et de la taille des paquets qui transitent sur le réseau. En effet, si les paquets sont de grande taille, il sera préférable de choisir des buffers de grande profondeur afin de limiter le nombre de liens réservés lors d'un transfert utilisant le wormhole par exemple. Inversement, de petits messages supporteront de petits buffers car les paquets seront découpés en moins de flits. Dans [51], les auteurs proposent d'adapter la taille de chaque buffer

aux différentes entrées du routeur en fonction des besoins applicatifs. Le choix de la profondeur de chacun des buffers se fait par le profilage des applications qui seront utilisées. En revanche dans le cas d'un réseau généraliste, si on ne peut pas prédire les communications afin de choisir la profondeur des routeurs, il est plus simple de fixer une taille maximale pour les paquets et ainsi choisir la taille des buffers en fonction de celle ci. De plus, il est important de choisir au mieux la taille de ces buffers afin de limiter la surface et la consommation dans un contexte embarqué. Tout au long de cette sous-section, un certain nombre de paramètres des réseaux sur puce ont été détaillés. Ces paramètres vont permettre d'assurer un certain nombre de services dans le réseau. La sous-section suivante traite de la qualité de service qui regroupe tous ces services.

1.3.7 Qualité de service (Quality of Service : QoS)

La qualité de service correspond aux garanties offertes par le réseau aux transferts de paquets. Ainsi, deux grandes catégories de garantie peuvent être extraites : celles relatives aux performances et celles relatives à l'intégrité des paquets.

QoS relative aux performances - La première catégorie est censée assurer à l'utilisateur que le réseau puisse offrir le minimum de performances nécessaire à la bonne utilisation du réseau. Dans cette catégorie, la latence des paquets ainsi que la bande passante du réseau sont les deux critères qui évaluent les performances. [52] traite des différentes méthodes qui garantissent latence et bande passante à chaque module. Ils extraient trois principales méthodes qui consistent en une garantie soit par dimensionnement du réseau, soit par l'utilisation de la commutation par circuit ou soit par l'utilisation de priorités.

La première méthode consiste à garantir les performances en dimensionnant le réseau en fonction des communications prévues ce qui nécessite de connaître parfaitement l'applicatif du système. La deuxième méthode utilise la commutation par circuit. Dans ce cas, ce n'est pas le paquet qui maîtrise sa traversée mais un contrôleur. Ainsi le contrôle va permettre de maîtriser les paquets dans le réseau. Cette méthode garde l'inconvénient de la commutation par circuit favorisant l'engorgement du fait de la réservation des chemins. La dernière méthode propose de donner une priorité à chaque message et donc lorsque le routeur doit choisir entre deux paquets il favorise celui qui a la plus haute priorité. L'inconvénient de cette méthode est que seulement les messages de plus hautes priorités ont une garantie de performance. Par exemple, le réseau Æthereal NoC de Philips Research [24] permet l'envoi d'un paquet soit en « best-effort », soit en « guaranteed service ». Dans le premier cas, le paquet est envoyé dans le réseau grâce à un mécanisme de commutation par paquet et donc aucune garantie n'est donnée sur le temps de traversée. Dans le cas du « guaranteed service », la commutation par circuit est utilisée. Dans un premier temps un système de messages réserve des créneaux de temps dans chaque routeur entre l'émetteur et le destinataire, puis le paquet est envoyé dans ce canal réservé ce qui garantit une bande passante minimale pour le paquet. Ce type de mécanisme

permet, par exemple, de respecter des contraintes temps réel. Dans le domaine de l'embarqué, gérer la réorganisation des paquets et les demandes de réémissions sont coûteuses en surface et donc sont à éviter.

QoS relative à l'intégrité des paquets - La deuxième catégorie de services gère l'intégrité des paquets. Les différents services sont répartis à différents niveaux dans les couches réseau. Au plus bas niveau, le réseau peut corriger des erreurs de transmission grâce, notamment, à des codes correcteurs d'erreur. Au niveau routage, les routages adaptatifs permettent, par exemple, d'éviter les liens défectueux. Les routages déterministes garantissent que les paquets arriveront ordonnés. Dans les dernières couches réseaux, des mécanismes de retransmission de messages peuvent être mis en place.

Toutes les garanties fournies par les réseaux sur puce dépendent des choix des différents paramètres décrits dans cette section. Il est donc nécessaire de bien cibler les besoins afin de choisir aux mieux les paramètres. Nous allons donc récapituler nos besoins et voir quelles caractéristiques restent indéterminées.

1.4 Conclusions de l'état de l'art

L'augmentation de la demande en puissance de calcul a conduit à l'augmentation du nombre d'éléments de calcul sur la puce. Multiplier le nombre d'éléments de calcul sur la puce entraîne une nouvelle demande vis à vis des interconnexions dans les systèmes embarqués. De plus, la gestion dynamique nécessaire à l'utilisation des nouvelles applications embarquées rend difficile la prédiction des communications dans le système. Nous venons de voir que les réseaux sur puce sont des solutions d'interconnexion viables pour supporter ces contraintes. Toutefois, leur conception est difficile du fait du nombre important de caractéristiques.

La Figure 1.8 répertorie et classifie quelques réseaux sur puce afin de voir comment ils s'organisent selon les deux caractéristiques principales : la topologie et le type de routage. Pour avoir une liste plus exhaustive, [26] répertorie soixante réseaux sur puce. La majorité des NoC de la littérature se situe dans le cas de la commutation par paquet sur une topologie grille, cela s'explique par le fait que c'est la solution la plus simple et la plus polyvalente à implémenter.

Le reste de l'espace n'est pas forcément exploré ce qui s'explique facilement car certaines combinaisons n'ont pas d'intérêt pour le monde des réseaux sur puce.

Le choix du réseau d'interconnexion est d'autant plus délicat que les réseaux de la littérature sont rarement présentés dans le même contexte. Cependant un certain nombre d'études permettent tout de même de choisir une partie des paramètres. Les contraintes de l'embarqué favorisent, d'après notre étude, les réseaux implémentant une commutation par paquet, utilisant du wormhole avec un algorithme de routage déterministe minimal. Nous n'avons pas choisi de mettre de canaux virtuels pour des raisons de surface. Le problème de la taille des buffers d'entrée des routeurs devra être résolu lors du raffinement du réseau car elle ne va pas influencer une

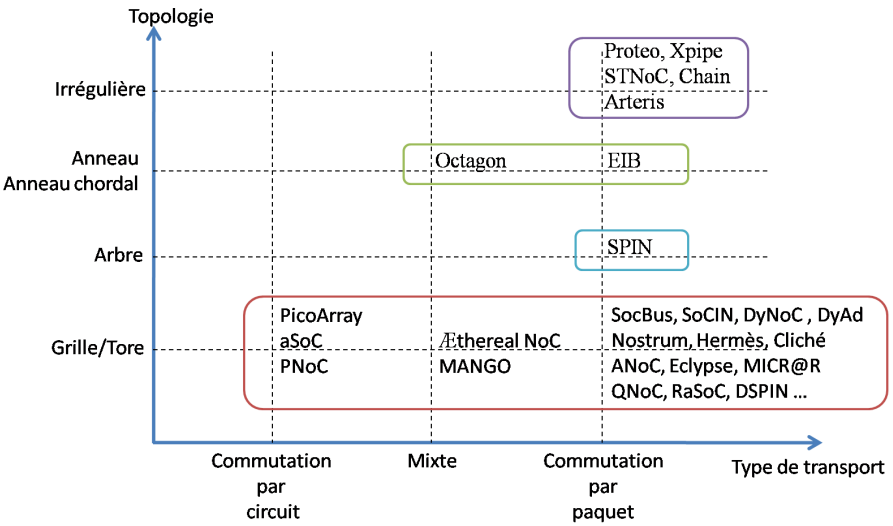


FIG. 1.8 – Classification des réseaux.

comparaison si ces paramètres restent constants. Un dernier paramètre, la topologie, reste difficilement déterminable à cause du grand nombre de possibilités.

Cependant nous avons comme contrainte d'être capable d'accepter sur le réseau des communications non prédictibles. Etant donné cette contrainte, il n'est pas possible d'envisager une optimisation de la topologie en fonction de l'applicatif ce qui pousse le choix vers des topologies régulières qu'elles soient à plat (grille, multi-étage...) ou hiérarchisées. Malgré cette restriction, le choix parmi les topologies régulières est vaste et une étude comparative devra être menée afin de retenir la meilleure topologie.

D'après l'étude de ce chapitre, nous devons sélectionner un panel représentatif de l'espace de conception. Nous allons donc retenir le multi-bus comme topologie de référence et la grille 2D, le tore 2D, le multi-anneau, le butterfly multi-étage pour les topologies plates directes et indirectes. Pour les topologies hiérarchiques, nous avons vu que pour de petits nombres de connexions le Multi-bus semble être une bonne solution par conséquent il sera toujours considéré comme interconnexion dans le cluster. Ensuite, l'anneau et le tore 2D seront les topologies retenues entre les clusters car ce sont respectivement la topologie la moins et la plus performante. Cependant, il est nécessaire d'avoir un environnement de comparaison afin de choisir la topologie la plus adaptée.

Chapitre 2

Mise en place d'un environnement de simulation pour l'exploration des réseaux sur puce

Sommaire

2.1 Choix du niveau de simulation 34
2.2 Etat de l'art des simulateurs de réseaux sur puce 36
2.3 Timed TLM contre Approximate-Timed TLM 38
2.4 Description de notre structure de simulation 40
2.4.1 Implantation des réseaux sur puce 40
2.4.2 Un environnement de simulation pour la comparaison de topologies 46
2.5 Caractérisation de la rapidité et de la précision de notre simulateur 48
2.5.1 Caractérisation de la rapidité 49
2.5.2 Caractérisation de la précision 50
2.6 Conclusion sur la nouvelle approche proposée 51

Les réseaux sur puce sont devenus incontournables dans la conception de multiprocesseurs. Toutefois leur composition est ardue étant donné que l'espace de conception est très vaste. Il n'est donc pas aisé de déterminer tous les paramètres à partir de l'état de l'art car chaque contexte d'utilisation implique des paramètres différents. Une exploration de cet espace de conception est donc indispensable pour obtenir les paramètres manquants du réseau. D'après le chapitre 1, la topologie des réseaux sur puce est, dans notre cas, le point important à explorer.

Le chapitre 2 propose une technique de simulation de réseaux sur puce. Celle-ci doit permettre une simulation rapide et fournir la précision nécessaire afin de comparer les différentes topologies choisies dans le chapitre 1. L'étude sera réalisée dans le chapitre 3. Avant de proposer le simulateur, dans un premier temps, il est nécessaire de choisir un niveau de modélisation permettant une simulation rapide et offrant une précision suffisante afin de discriminer les différentes topologies. Dans un deuxième temps, une étude des simulateurs existants montre le besoin de proposer une nouvelle technique de modélisation. Enfin ce chapitre va présenter la description de la solution retenue et sa caractérisation.

2.1 Choix du niveau de simulation

La simulation d'un système MPSoC complet est une tâche ardue. Il existe différents niveaux de simulation utilisant différents langages. La difficulté est de réaliser le bon compromis entre la précision et la vitesse de la simulation [53]. En effet, la précision et la rapidité sont deux caractéristiques opposées. Une simulation rapide ne pourra pas avoir le meilleur niveau de précision étant donné que plus on est précis et plus il y a de choses à prendre en compte et donc à simuler. Cette section décrit les différents niveaux de modélisation et argumente le choix effectué. La figure 2.1 récapitule les différents niveaux de simulation et leurs vitesses de simulation estimées. Ces estimations sont extraites de [54] et sont exprimées en Million d'Instructions Simulées Par Seconde (MISPS) ou en Kilo d'Instructions Simulées Par Seconde (KISPS). Cette figure met en évidence quatre niveaux de simulation.

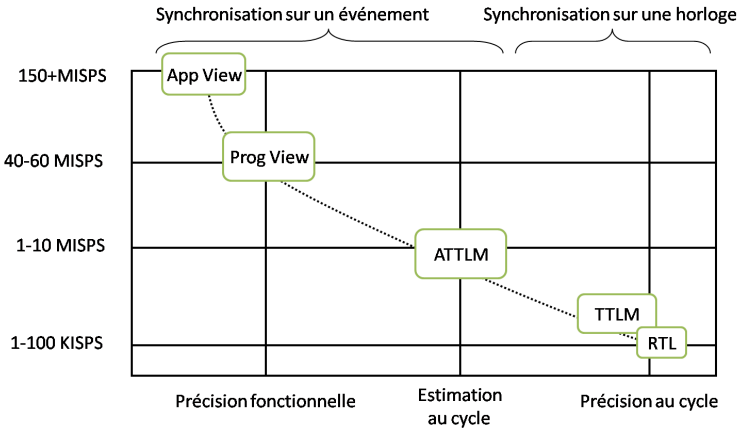


FIG. 2.1 – Vitesse de simulation des niveaux de modélisation en fonction de leurs précisions. (App View ; Application View, Prog View : Program View, ATTLM : Aprroximate-Timed Transaction Level Modeling, TTLM : Timed Transaction Level Modeling, RTL : Register Transfer Level

Le niveau de simulation le plus abstrait est « application view ». Celui-ci est un niveau fonctionnel. Son but est de permettre un développement rapide d'applications embarquées. A ce niveau, les aspects matériels et logiciels sont confondus et aucune considération n'est prise sur la structure de la plate-forme MPSoC. Ce niveau de simulation a pour but, dans un premier temps, de valider les choix réalisés au niveau de l'applicatif, tel que la parallélisation de l'applicatif envisagé. Dans un second temps, ce niveau aide au raffinement des services proposés par le contrôle de l'architecture tel que l'allocation mémoire ou encore l'algorithme d'ordonnancement. Le niveau de simulation « application view » permet une précision fonctionnelle et donc offre les simulations les plus rapides, plus de 150 MISPS.

Le niveau suivant « program view » est un raffinement du niveau précédent. En effet, dans ce niveau, une première spécification de l'architecture est réalisée. L'architecture hôte et le logiciel sont donc décrits séparément. Cette séparation

permet de réaliser un profil de l'application sur l'architecture. Ce profilage met, par exemple, en évidence les points sensibles de l'architecture et donc permet de choisir si une fonctionnalité de l'architecture devra être réalisée en matériel ou en logiciel. A ce niveau de modélisation, les communications entre les modules sont réalisées à l'aide de transactions [55]. Ces transactions sont implémentées par des appels de fonction et donc aucune supposition n'est réalisée sur l'implémentation [56]. La durée d'une transaction est ignorée à ce niveau. L'ajout de la spécification matérielle ralentit la simulation autour des 50 MISPS mais augmente la précision du simulateur.

Le niveau « Approximated time TLM » (Transaction Level Modeling) permet l'exploration de la partie matérielle de l'architecture et donc le dimensionnement de celle-ci. Pour cela, ce niveau intègre des informations de temps plus précises à l'intérieur des composants de l'architecture ainsi qu'entre ces composants. Ces informations de temps vont mettre en évidence, par exemple, les goulots d'étranglement ou encore les problèmes de synchronisation. A ce niveau, la technique principalement utilisée dans la littérature consiste à ajouter des informations de temps à la modélisation par transaction (TLM) [57]. Il existe deux niveaux de précision : « cycle approximate » et « cycle accurate » qui offrent deux vitesses de simulation respectivement de 5 MISPS et 150 KISPS. Dans le premier cas, les temps sont estimés en prenant en compte les caractéristiques matérielles sans pour autant avoir besoin d'une horloge pour synchroniser les modules. On dit alors que la simulation est synchronisée par ses évènements (event driven). Dans le deuxième cas, la précision « cycle accurate » requiert que les modules soient synchronisés sur une horloge (clock driven), ce qui explique la perte de vitesse de simulation.

Le dernier niveau de simulation est le niveau avant l'implémentation matérielle : RTL (Register Transaction Level). A ce stade, tous les composants de l'architecture matérielle sont complètement définis à l'aide d'un langage de description matérielle (HDL : Hardware Description Language). Tous les protocoles de communication sont définis et les communications sont modélisées précisement. L'interface entre le côté matériel et logiciel est de la même manière définie de façon précise. Ce niveau de modélisation permet d'être précis au cycle près (cycle accurate) et même au bit près (bit accurate). Ce niveau permet une vérification précise avant la prochaine étape de conception. En contrepartie, il est le plus lent en termes de temps de simulation, et offre des simulations autour de 50 KISPS.

Tous ces niveaux de simulation ont leur utilité selon où l'on se place dans le flot de conception. Il est donc nécessaire de choisir le bon niveau en fonction de ses besoins. Le but de notre structure de simulation est de permettre une exploration architecturale rapide avec une précision suffisante pour choisir une topologie pour le réseau sur puce. Afin de discriminer les différentes topologies, il est nécessaire de s'approcher de la précision au niveau cycle. En effet, si l'on veut pouvoir discriminer deux topologies différentes, il est nécessaire de prendre en compte les conflits à l'intérieur du réseau, ce qui requiert d'avoir une idée approximative du temps de traversée de chaque message en fonction de l'activité dans le réseau. Etant donné que nous cherchons le niveau le plus rapide possible comportant la précision nécessaire, « Approximated time TLM » est le niveau de modélisation le plus adapté. De ce

fait, la section suivante présente les différents simulateurs haut-niveau qui peuvent proposer l'exploration des différentes topologies.

2.2 Etat de l'art des simulateurs de réseaux sur puce

Cette section a pour but d'analyser les différentes solutions existantes afin de savoir si une d'entre elles convient à nos contraintes de rapidité et de précision. Elle va permettre d'évaluer la nécessité de développer une nouvelle structure de simulation de réseau sur puce qui sera capable de s'intégrer ensuite dans un environnement complet. Il est possible de classifier les simulateurs existants en trois catégories.

La première catégorie propose des plateformes de simulation MPSoC mais elle se focalise sur un point particulier de l'architecture (processeur, tolérance aux fautes...) et ne permet pas d'explorer tous les paramètres de la plate-forme. En général, ces simulateurs implémentent une solution unique et simple du système d'interconnexions ce qui restreint l'exploration.

Les simulateurs Simics [58], SimOS [59] proposent une modélisation simplifiée de l'architecture afin d'être assez rapide pour simuler des OS sur différentes architectures. Dans les deux cas, leur vocation n'est pas de permettre l'exploration architecturale mais de permettre une mise en situation d'OS.

RSIM [60], MicroLib [61], SESC [62], UNISIM [63], CASSE [64], ROSES [65], CAAM [66] ou encore GEMS [67] sont des simulateurs qui permettent l'exécution d'applications sur une structure MPSoC mais pas l'exploration de l'ensemble de la structure.

Par exemple, RSIM est un environnement de simulation multiprocesseur à mémoire partagée monolithique. Ce simulateur est destiné à explorer les processeurs offrant un parallélisme d'instructions (ILP). Ainsi, il offre la possibilité de modifier de nombreux paramètres au niveau du processeur. Une plate-forme multiprocesseur est également disponible. Elle propose une structure multiprocesseur homogène contenant un cache cohérent qui s'organise autour d'un réseau grille 2D. Cette plate-forme n'utilise pas d'interface standard ce qui ne permet pas une grande modularité. De plus, seule une topologie, dont le temps de traversée est fixe, est disponible.

UNISIM remédie au problème d'interface. En effet, il est basé sur la librairie SystemC [68] et utilise l'extension TLM associée [69]. Il est donc facilement modulable du fait de l'utilisation de la librairie standard TLM. En fait, étant donné que tous les modules utilisent la même interface, il devient simple de rajouter, retirer et combiner des modules dans le simulateur. Mais comme pour RSIM, une seule interconnexion est proposée, il s'agit d'un PCI bus.

D'autres simulateurs introduisent de nouvelles fonctionnalités, par exemple, ReSP utilise la même base de librairie que le simulateur UNISIM mais il propose en plus d'encapsuler le noyau de simulation SystemC dans un environnement de contrôle écrit en Python. Ceci a pour but d'augmenter les services en permettant un contrôle non intrusif. Par exemple, ReSP [70] permet d'arrêter, de mettre en pause et de redémarrer une simulation à tout moment, ce qui n'est pas directement possible

avec le noyau SystemC. De plus à la compilation, les modules SystemC que l'on veut contrôler sont encapsulés dans une couche d'adaptation C++/Python. Cette surcouche permet, par exemple, d'introduire des fautes dans les communications et ainsi de tester la tolérance aux fautes d'un module sans devoir modifier le code SystemC. D'après [70], l'impact n'est pas significatif comparé à une simulation seule. Cette surcouche n'est pas nécessaire pour l'exploration et surtout elle aura sûrement un impact si l'on simule un système beaucoup plus gros. Cette catégorie de simulateur permet principalement l'étude d'un processeur élémentaire dans un environnement multiprocesseur, la vérification du portage de l'application et le début du dimensionnement architectural (nombre de processeurs, hiérarchie mémoire). Mais le manque de précision dans le modèle d'interconnexions ne permet pas l'exploration de celui-ci.

La deuxième catégorie de simulateur autorise l'exploration du réseau d'interconnexions. En effet, ces simulateurs sont, par nature, conçus pour explorer les réseaux selon un maximum de paramètres. On peut citer Noxim [71], Sicosys [72], gpNoCSIM [73], Garnet [74], NNSE [75]. Noxim, par exemple, est un simulateur SystemC synchronisé sur une horloge qui décrit précisément le routeur d'une grille 2D. Il suffit donc de relier ces routeurs entre eux afin de former une grille de taille variable. L'ensemble des communications entre les routeurs se fait par une série de fils connectés entre les modules. Ce simulateur permet d'obtenir des résultats précis au cycle près. De plus, il est possible de modifier un certain nombre de paramètres dans ces routeurs afin d'explorer les différentes possibilités de réseaux. Ainsi, l'algorithme de routage, la profondeur des FIFO et la politique d'arbitrage dans les routeurs peuvent être aisément remplacés. L'émission et la réception de messages dans le réseau sont effectuées par des générateurs de trafic paramétrables. Tous ces simulateurs sont synchronisés par une horloge afin de fournir une précision au cycle. Dans tous les cas, il est nécessaire de définir le routeur de base et de connecter correctement l'ensemble de ces routeurs selon la topologie choisie. Mais, malheureusement, leur défaut principal est de ne pas proposer un modèle de processeur précis se connectant sur le réseau, ce qui ne permet pas de réaliser une exploration complète.

La troisième catégorie de simulateur permet une exploration complète d'une architecture multiprocesseur, du réseau d'interconnexions aux processeurs. Dans cette catégorie, on trouve deux approches. Afin de permettre une simulation totale, la première approche consiste à coupler un simulateur de la première et de la deuxième catégorie. Ainsi on obtient un simulateur complet qui permet une simulation de toutes les parties du multiprocesseur. On peut citer l'association Sicosys+RSIM [76] ou encore GEMS+GARNET [74]. Dans les deux cas, le point délicat consiste à fournir une interface adaptant les protocoles de communication des deux simulateurs. En effet, dans le cas de Sicosys+RSIM, les deux simulateurs n'utilisent pas la même façon de se synchroniser. Sicosys est synchronisé à une horloge alors que RSIM se synchronise sur des évènements afin d'être plus rapide. Du fait de cette adaptation, la vitesse de simulation chute d'un facteur 3 [76]. On retrouve également une perte de performance dans le cas de GEMS+GARNET. Cette perte de performance peut en partie s'expliquer par l'ajout de communications entre deux processus différents

sur le processeur hôte. En effet, les communications entre les processus sont plus consommatrices en temps que des lectures dans la pile du même processus. Le problème des assemblages comme ceux-ci réside dans le fait que les simulateurs ne sont pas conçus dès le départ comme un seul simulateur et que donc leurs interfaces ne sont pas adaptées. La deuxième approche rectifie cette faiblesse en proposant des simulateurs complets destinés à l'exploration architecturale. Mentor Graphics [77] et Coware [78] proposent, chacun, une plate-forme complète mais payante permettant l'exploration architecturale. ASIM [79], MC-SIM [80], MPARM [81] et Soclib [82] sont des simulateurs MPSoC complets. Toutefois, ASIM, MC-SIM et MPARM ne proposent qu'une seule interconnexion chacun, ce qui limite les possibilités d'exploration.

Pour sa part, SoClib, proposé par le Lip6, est un simulateur MPSoC complet. Ce simulateur basé sur la librairie SystemC est à l'origine au niveau CABA (Cycle Accurate, Bit Accurate) et à ensuite proposé un niveau TLM. Il dispose d'un large choix de processeurs, d'interconnexions et d'IP diverses. Les connexions entre les modules se font à travers une interface VCI afin de normaliser les connexions. L'écriture de chaque module est contrainte par des règles d'écriture qui permettent l'uniformisation des modules. Au moment de choisir un simulateur SoClib n'avait pas encore complètement défini son modèle TLM, c'est pourquoi nous ne l'avons pas choisi. Toutefois depuis ce choix SoClib fourni la possibilité de simuler en TLM en utilisant la norme TLM-DT mais ils ont fait le choix de ne pas prendre en compte les contentions à ce niveau.

Cette étude de l'existant nous a permis de voir les différentes approches pour simuler une plate-forme MPSoC. Il apparait qu'au moment du choix aucune solution ne correspondait exactement à nos besoins de simulations rapides, précises et permettant l'exploration. Cette étude met en évidence que dans la majorité des cas, la modélisation du réseau d'interconnexions est réalisée en décrivant chaque élément de cette interconnexion. Ceci peut poser un problème si on envisage de simuler un réseau de grande taille. Par exemple en SystemC, il en résulte un grand nombre de threads SystemC ce qui risque de ralentir la simulation. Afin d'éclaircir ces propos, la section suivante se focalise sur les types d'approches possibles du TLM.

2.3 Timed TLM contre Approximate-Timed TLM

Réaliser les simulations au niveau transaction laisse le choix entre deux façons d'appréhender le TLM : soit timée, soit approximée. La différence entre les deux approches réside dans la manière dont les modules du simulateur sont synchronisés.

Dans le cas du Timed TLM (TTLM), les modules et donc les threads SystemC qui les composent sont synchronisés par une horloge. Du fait de cette synchronisation, à chaque tic d'horloge, le noyau de simulations SystemC traite l'ensemble des threads même si aucun traitement ne doit être réalisé. On obtient alors des communications entre les modules comme décrites sur la figure 2.2(a). Le noyau SystemC impose un coût élevé lors des changements de contexte qui se produisent lorsque

que le noyau passe d'un thread à un autre. Ce surcoût devient donc de plus en plus important à mesure que le nombre de threads à gérer augmente [83]. Le TTLM est donc un mécanisme précis mais il est coûteux en temps de simulation.

Au contraire l'Approximate-Timed TLM (ATTLM) n'utilise pas de synchronisation par le temps. L'ensemble des transactions sont réalisées indépendamment les uns des autres et chaque module SystemC traversé ajoute une pénalité de temps. Ainsi le temps est incrémenté à la fin d'une transaction. Par exemple, s'il est nécessaire de réveiller un thread afin de réaliser un calcul, ce réveil se fait à l'aide d'un évènement et pas sur un tic d'horloge comme dans le premier cas. La figure 2.2(b) représente ce mécanisme. Ce mécanisme obtient de meilleures performances car il fait intervenir le minimum de threads à chaque pas de simulation du noyau. En contrepartie, la précision peut être altérée en fonction de l'effort de description mis sur la fonction d'évaluation du temps. En effet, le temps dans un module peut être calculé en fonction des différentes actions effectuées ou simplement estimé en fonction du profilage. Etant donné que nous recherchons le meilleur compromis entre la rapidité et la précision, l'ATTLM semble être la solution la plus adéquate du fait que les vitesses de simulation sont supérieures et que l'on peut obtenir la précision voulue.

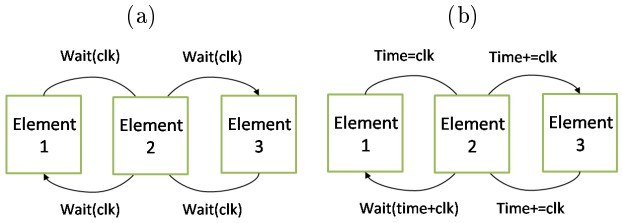


FIG. 2.2 – (a) Description d'une communication utilisant le Timed TLM. Les messages passent d'un module à un autre seulement sur un front d'horloge. Ce mécanisme permet une précision au cycle près. (b) Description d'une communication utilisant le Approximated Time TLM. Une transaction, qui passe de 1 vers 2 puis de 2 vers 3 et revient, ne doit pas attendre dans chaque élément une synchronisation par rapport à une horloge. Elle passe à travers 2 et 3 sans pénalité mais une variable « temps » est incrémentée celle-ci représentera le coût total en temps de la transaction. A la fin seulement le temps est incrémenté.

Il existe, dans la librairie TLM, deux manières d'implémenter l'ATTLM : « le temps implicite » ou « le temps explicite ». L'ATTLM avec un temps implicite est la dernière version disponible auprès de l'OSCI (Open SystemC Initiative) [68].

Le temps implicite consiste à laisser au noyau SystemC la responsabilité de l'incrémentation du temps au moment de l'appel de la fonction d'envoi de la transaction (figure 2.3(a)). Ceci correspond à la solution la plus rapide car intégrée au système. Cependant, du fait que le système réalise lui même l'incrémentation du temps, l'utilisateur n'a pas le moyen de connaître ce que fait la transaction avant son arrivée dans le bloc suivant. Cela n'a pas forcément d'importance sauf quand un calcul dynamique du temps d'attente est réalisé. En effet, lorsque par exemple (figure 2.3(c)),

deux transactions arrivent à une même mémoire, la première reçoit une pénalité de temps proportionnelle aux temps de lecture ou d'écriture du message. La seconde transaction reçoit pour sa part la même pénalité plus une pénalité par rapport à son temps d'attente dans la mémoire. Pour calculer ce temps d'attente, il est nécessaire de savoir à quel moment la première transaction quitte la mémoire.

Le temps explicite permet de connaître cette information. Pour cela, la pénalité de temps n'est pas incrémentée lors de la transaction mais dans le module qui produit cette pénalité (figure 2.3(b)). Ce mécanisme permet de connaître plus en détail l'emplacement à un moment donné d'une transaction et donc d'utiliser l'ATTLM d'une manière plus précise (figure 2.3(d)). La section suivante présente notre structure de simulation qui utilise l'ATTLM avec le temps explicite et propose une façon innovante de simuler les réseaux sur puce.

2.4 Description de notre structure de simulation

Cette section présente notre structure de simulation. Nous l'avons basée sur SystemC car ce langage de simulation permet de réaliser des simulations à tous les niveaux. De plus, il donne accès à une librairie TLM qui va simplifier la mise en place de notre structure. Il est nécessaire de mentionner que seuls les threads au sens SystemC sont capables d'utiliser la librairie TLM ce qui est une contrainte d'implémentation. Nous allons dans cette section décrire la technique de simulation réseau et ensuite la structure de comparaison des réseaux sur puce va être détaillée.

2.4.1 Implantation des réseaux sur puce

La technique la plus évidente pour simuler un réseau est de représenter chaque élément de ce réseau. Cependant, cette technique peut amener à décrire un grand nombre d'éléments et donc ralentir considérablement la simulation. Par exemple, si l'on veut connecter 256 processeurs selon une topologie grille 2D, il est nécessaire de simuler 256 routeurs ce qui risque de ralentir le système étant donné que les nombres de modules et de threads dans la simulation sont grands. En revanche, si l'on est capable de simuler l'ensemble du réseau dans un seul thread il n'y aura pas de pénalité à cause du nombre de threads. Cependant ce thread unique peut devenir compliqué car il doit estimer les temps pour l'ensemble des transactions. La suite de cette sous section présente une nouvelle approche pour simuler de façon précise un réseau sans modéliser l'ensemble des routeurs de l'interconnexion. Dans un premier temps, les transactions TLM vont être détaillées afin d'améliorer la compréhension puis dans un second temps l'ensemble des réseaux sont décrits.

Une transaction TLM est un appel de fonction entre deux modules. Cette fonction prend en paramètre une structure de données nommée « requête » et rend une structure « réponse ». Par abus de langage, on utilisera de la même façon les mots « transaction » et « requête » pour désigner l'envoi d'une requête par transaction TLM. La librairie TLM permet d'adapter ces deux structures au besoin de l'utilisateur. Dans notre cas, nous avons besoin d'envoyer des requêtes de lecture et

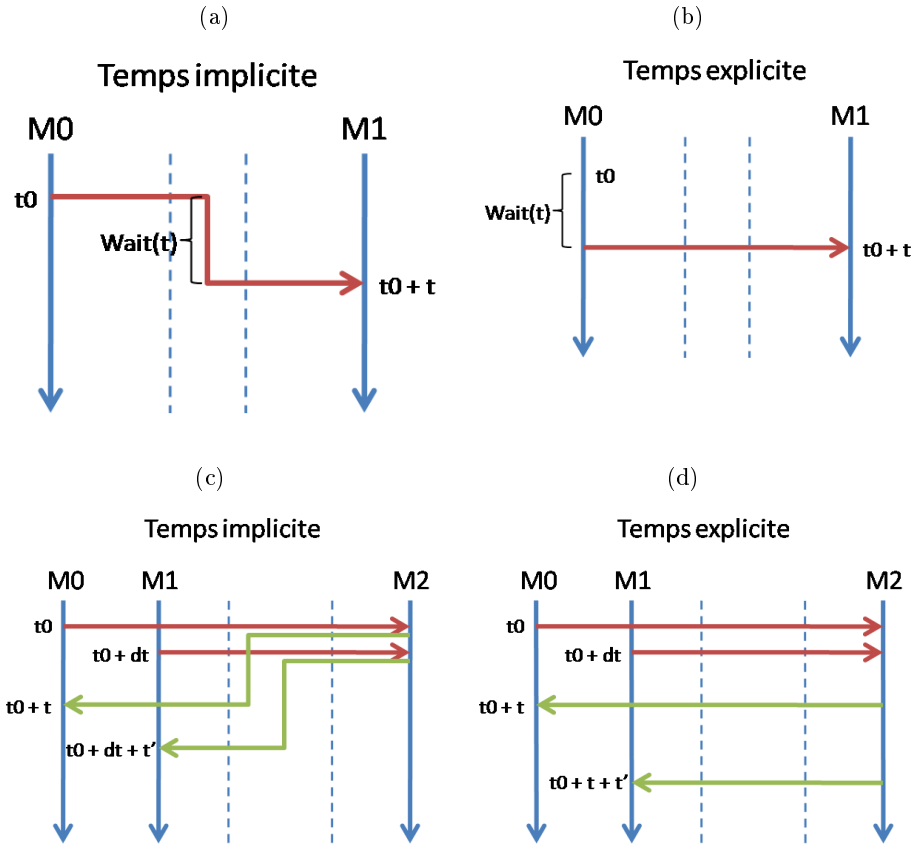


FIG. 2.3 – (a) Principe de fonctionnement du temps implicite. Le temps est incrémenté au moment de la transaction. (b) Principe de fonctionnement du temps explicite. Le temps est incrémenté à l'intérieur du module avant la transaction. (c) Exemple de deux transactions consécutives en temps implicite. Les modules 0 et 1 envoient quasiment simultanément chacun une transaction vers le module 2. En temps implicite, les deux réponses sont renvoyées sans avoir de notion de temps dans le module 2. Si la première transaction occupe le module 2 pendant un temps t et la deuxième pendant un temps t', la deuxième transaction devrait arriver à t0 + t + t' et non pas à t0 + dt + t'. Au contraire, (d) On retrouve la même situation que dans (c) mais cette fois ci, avec le temps explicite, le module 2 connaît la date de retour de la première transaction et l'ajoute au temps d'attente de la deuxième transaction.

d'écriture à un destinataire précis. Les requêtes peuvent comporter une donnée ou une structure de données.

Nous avons donc modifié ces structures afin de répondre à nos besoins. La structure requête comprend un identifiant et une adresse qui permettent de savoir la provenance de cette requête et sa destination, ainsi qu'un champ type pour identifier la nature de la demande (typiquement lecture/écriture). Elle contient également un entier nommé « donnée » qui peut contenir une valeur ou un pointeur. Dans le cas d'un pointeur, il est nécessaire de renseigner également, le champ de taille afin

que le destinataire connaisse la taille de la donnée à manipuler. Cette information servira également au réseau pour calculer la taille du paquet. On retrouve également à l'intérieur de la structure requête un pointeur sur une structure qui sert aux statistiques du réseau.

Pour sa part, la structure de réponse contient un statut pour savoir si la transaction a réussi ou non. Elle contient également un entier nommé « donnée » permettant de rapatrier une valeur ou un pointeur. Un point important est que cette structure de réponse possède une variable permettant de mémoriser le cumul du temps attendu par cette transaction. Etant donné que nous utilisons le temps explicite cette variable sert à informer de l'attente globale cumulée.

Parmi les réseaux développés, il est nécessaire de différencier le Multi-bus des autres réseaux sur puce. En effet, le Multi-bus est une interconnexion qui ne contient pas de routeur. Son implémentation et celle des réseaux sur puce vont être détaillées.

Multi-bus - Le réseau Multi-bus est constitué d'un thread principal qui sert de contrôleur central et d'un certain nombre de fonctions qui permettent la répartition des transactions en fonction des caractéristiques de l'interconnexion. Ainsi, il est possible de choisir la largeur des bus, le nombre de bus géré et le nombre de nanosecondes de traversée du Multi-bus ce qui permet de définir dans le même temps la fréquence de fonctionnement ainsi que le nombre de cycles de traversée du Multi-bus. Il est également possible de choisir le type d'arbitrage pour l'ensemble des bus.

Il existe trois arbitrages implémentés : Round-Robin, FIFO et « no-delay ». L'arbitrage Round-Robin correspond à choisir de façon systématique la transaction émise par l'entrée suivante non vide. L'arbitrage FIFO prend les transactions dans l'ordre de leur émission. L'arbitrage « no-delay » est particulier, il ne correspond pas à une implémentation physique mais permet de rediriger la transaction vers son destinataire sans pénalité. Cela a pour but de pouvoir étudier l'environnement du réseau sans asubir ses effets. Cet arbitrage permet donc principalement le déverminage de la plate-forme et de l'application. Les deux premiers arbitrages ont été choisis car le premier permet d'obtenir en moyenne qu'aucune entrée ne soit favorisée (Round-Robin) et le second est la politique la plus simple à implémenter (FIFO).

La largeur du bus permet de connaître le nombre de messages qui serait réellement envoyé et donc le temps que prendrait la transaction. Pour sa part, le nombre de bus permet de calculer le nombre d'entrées sur chaque bus et donc de connaître les requêtes susceptibles de créer des contentions. Ces deux paramètres jouent un rôle important dans le calcul des pénalités des différentes transactions.

Afin de décrire au mieux l'implémentation, nous allons expliquer le déroulement d'une transaction à travers le réseau. La figure 2.4 permet de suivre l'exécution lors de l'arrivée d'un message dans le Multi-bus.

(1) L'acheminement d'une transaction commence à l'intérieur de l'interface réseau devant le réseau. Cette interface communique un identifiant à la transaction afin d'identifier l'entrée dans le réseau. Ceci est réalisé principalement car l'appel de

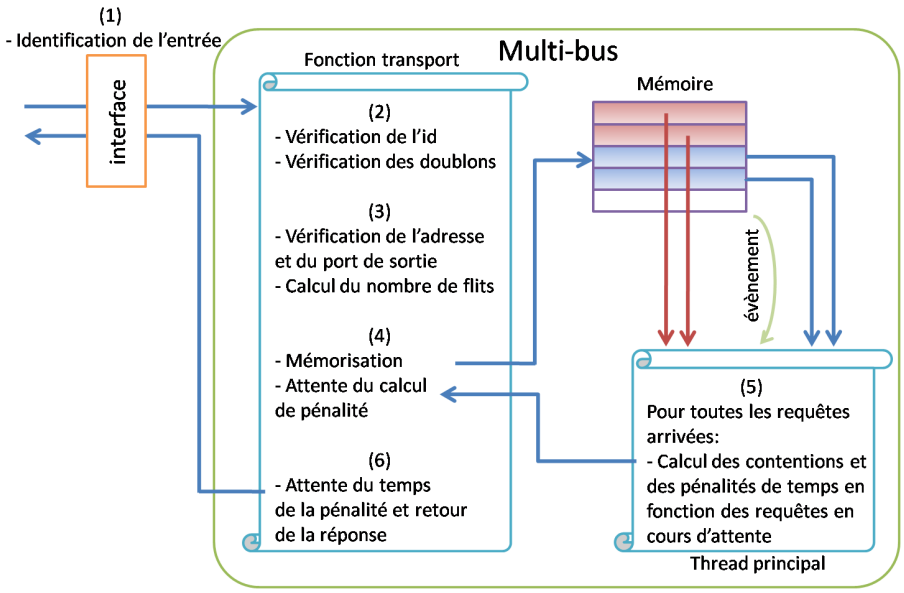


FIG. 2.4 – Représentation du fonctionnement du simulateur de réseaux sur puce.

fonction qui envoie la transaction TLM ne permet pas d'identifier le port par lequel passe la transaction.

(2) Une fois l'identification effectuée la transaction entre dans le réseau. Lors de la réception d'une nouvelle requête TLM, la fonction « transport » appelée retrouve dans la table de translations le numéro du port de sortie correspondant à l'adresse transmise par cette requête. L'identification précédemment donnée par l'interface réseau va permettre de mémoriser la requête dans un emplacement réservé. Durant la mémorisation, le réseau vérifie qu'aucune requête valide ne possède le même identifiant d'émetteur. Si c'est le cas, la transaction est refusée et une réponse avec un statut négatif est renvoyée. Cette identification permet également de mémoriser l'instance de la fonction « transport » afin d'être capable de répondre à la requête correspondante. En effet, à travers l'ensemble des différents ports, c'est toujours la même fonction « transport » qui est appelée.

(3) Ensuite une seconde fonction détermine le numéro du bus à travers lequel le message devrait être propagé et le nombre de messages nécessaires pour le transfert de la donnée.

(4) Enfin, la requête est mémorisée dans une liste de requêtes en attente de traitement et la fonction « transport » est bloquée sur l'attente d'un événement. La mémoire, qui contient les requêtes, envoie un événement au thread principal lorsqu'un élément y est ajouté.

(5) Une fois le thread principal activé, celui-ci traite de façon séquentielle les différents bus virtuels et ainsi traite l'ensemble des requêtes en attente. Ainsi dans un pas de simulation SystemC, l'ensemble des requêtes en attente sont traitées avec un seul réveil du thread. Ce thread définit l'ordre de passage de l'ensemble

des requêtes contenues dans sa mémoire et dans un second temps il détermine la pénalité de temps pour chaque requête en fonction de la taille de la requête, de la taille des requêtes précédentes et du temps de traversée du Multi-bus.

(6) Lorsqu'une requête a été traitée un événement est envoyé à la fonction « transport » correspondante afin qu'elle réalise l'attente de la pénalité. Une fois l'attente terminée, la requête est transmise à sa destination et se met en attente de la réponse avant de la transmettre à l'émetteur.

Réseaux sur Puce - Dans le cas des réseaux sur puce, les paramètres de configuration sont en partie différents. En effet, on communique comme dans le cas du Multi-bus, le temps de traversée ainsi que la largeur des liens entre les routeurs. Comme dans le cas du Multi-bus, le temps de traversée permet de définir la fréquence de fonctionnement et le nombre de cycles pour traverser le routeur et le lien suivant. Dans le cas des réseaux sur puce, un seul arbitrage a été mis en place arbitrage : Round-Robin. L'arbitrage Round-Robin a été choisi car en moyenne aucune entrée n'est privilégiée. Cependant il est possible d'utiliser l'arbitrage « no-delay » afin de court-circuiter le réseau.

Les paramètres principaux pour les réseaux sur puce sont la topologie, le nombre d'entrées et le nombre de sorties. Selon la topologie d'autres informations sont nécessaires comme le nombre de colonnes dans le cas de la grille et du tore 2D. Deux paramètres principaux sont calculés automatiquement à l'instanciation du réseau dans le simulateur en fonction de la topologie. On retrouve le nombre de lignes dans une grille si on connaît le nombre de colonnes ou le nombre d'étages d'un Multi-étages en fonction du nombre de points d'accès et du nombre d'entrées/sorties d'un routeur de base.

Pour chaque topologie, le routage le plus simple a été implémenté. Si à l'avenir, il est nécessaire de changer le routage, seule une fonction serait à remplacer dans l'implémentation. La profondeur des buffers des routeurs du réseau a été fixée à 1 et aucun canal virtuel n'est disponible car ces paramètres permettent d'obtenir la plus petite surface silicium étant donné les contraintes de l'embarqué.

Dans le cas des réseaux sur puce, l'interconnexion décrit dans le simulateur est virtuelle. Aucun routeur n'est réellement implémenté, il est donc nécessaire qu'une structure de renseignements soit associée à chaque requête. En effet, il est nécessaire de mémoriser un certain nombre d'informations calculées en fonction des topologies afin de déterminer les pénalités de temps. Typiquement, les points d'entrée et de sortie dans le réseau, le chemin et le sens de parcours dans le réseau, ou plus particulièrement le numéro du bus utilisé pour la topologie anneau sont des éléments indispensables.

L'implémentation des réseaux sur puce est similaire à celui du Multi-bus pour la majorité des points de fonctionnement. L'implémentation diverge lorsque l'on considère le calcul des pénalités de temps. De la même façon que pour le Multi-bus, nous allons décrire comment une transaction TLM traverse le réseau et donc comment les différentes pénalités de temps sont calculées.

(1) et (2) Avant de rentrer dans le module réseau, une transaction traverse l'interface réseau de la même manière que pour le Multi-bus, il remplit l'identifiant de la requête entrante. L'interface réseau fait ensuite appel à la fonction « transport » du module réseau.

(3) Dans un premier temps, l'identifiant de l'entrée est mémorisé. On vérifie qu'aucune autre requête ne possède le même identifiant afin de refuser ou non la transaction. Comme dans le cas du Multi-bus l'adresse contenue dans la requête est traduite en numéro de sortie grâce à une table de translations renseignée à chaque ajout de sortie au réseau.

(4) Une fois la requête mémorisée et acceptée par le module réseau, une série de fonctions spécifiques à chaque topologie est lancée afin de remplir la structure de renseignements nécessaires au calcul des pénalités. La première fonction permet de renseigner le positionnement de l'entrée et de la sortie du réseau. Ceci se calcule en fonction de l'identifiant de la requête (donnée par l'interface réseau), le numéro de sortie de la destination, la topologie et surtout des caractéristiques de la topologie (nombre de colonnes, nombre de bus, nombre d'étages...). Cette première fonction ne prend en compte que les caractéristiques topologiques du réseau. Elle est suivie par le calcul du nombre de flits nécessaires au transport de toute la donnée.

(5) Ensuite, une seconde fonction utilise l'algorithme de routage afin de déterminer le chemin pris à travers le réseau, elle renseigne, par la même occasion, les timings au niveau de chaque routeur en fonction du nombre de flits du message et du temps de traversée d'un routeur. Le chemin, pris par un paquet, est donc représenté par une liste dont chaque élément correspond au numéro du routeur et au lien de sortie pris par le paquet. Pour chaque élément, on mémorise aussi les timings d'entrée et de sortie du paquet.

La figure 2.5(a) donne un exemple dans le cas d'une grille 2D. Le paquet part de (S) au temps 0 et se dirige vers (D), il contient 3 flits et le temps de traversée (routeur + lien) est de 1. Le paquet passe par les routeurs 0, 1, 2, 3 et n+3. La liste routeur + lien de requête peut être représentée comme suit sur la liste 2.1. Cette liste se lit de la manière suivante : numéro du routeur, numéro du lien, timing d'entrée, timing de sortie.

$$(0,1,0,3) \rightarrow (1,1,3,6) \rightarrow (2,1,6,9) \rightarrow (3,2,9,12) \rightarrow (n+3,4,12,15) \qquad (2.1)$$

(6) Une fois le calcul du chemin terminé, la requête est mémorisée afin d'être traitée par le thread principal. Pendant ce temps de traitement, la fonction « transport » se met en attente.

(7) Comme dans le cas du Multi-bus, le thread principal n'est activé que lorsqu'il y a des communications. Il traite l'ensemble des requêtes mémorisées. Il est nécessaire de déterminer si il y a des contentions et de mettre à jour les pénalités de temps pour chaque requête. Pour cela, le thread principal compare le chemin de la nouvelle requête avec l'ensemble des autres requêtes encore présentes dans le réseau et, à l'aide des timings, détermine si oui ou non il y a contention entre deux requêtes. On considère qu'il y a contention entre deux requêtes lorsqu'elles utilisent le même

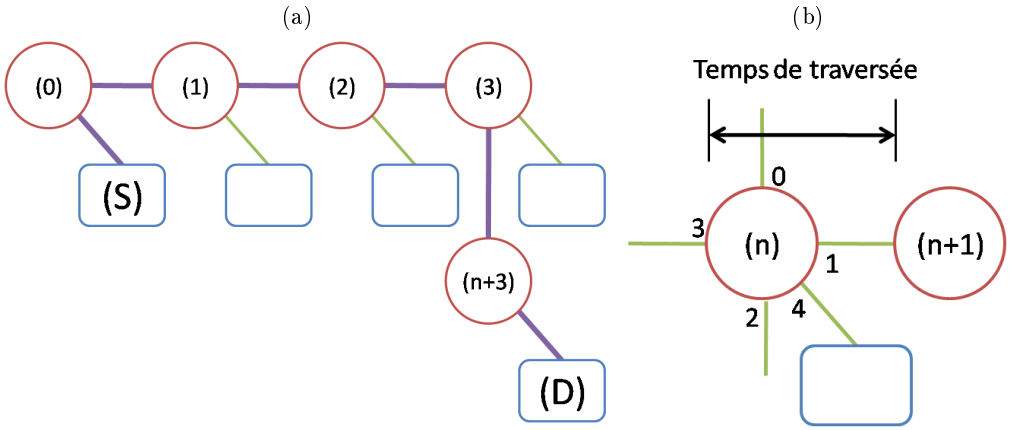


FIG. 2.5 – (a) Exemple de chemin possible dans une grille 2D. Le réseau est en partie représenté et les routeurs source (S) et destination (D) sont annotés. Tous les routeurs sont numérotés afin de les identifier. (b) visualisation du temps de traversée et de la numérotation des liens.

ensemble routeur+lien pendant le même laps de temps. C'est toujours la dernière requête ajoutée à la liste qui attend le passage de la précédente. Cette approximation simplifie le traitement mais crée une erreur. Cependant, le traitement des requêtes se fait à travers un arbitrage Round-Robin ce qui permet en moyenne de corriger notre erreur sur le traitement des contentions.

(8) Une fois une requête traitée, un événement est envoyé à la fonction « transport » correspondante ce qui lance l'attente du temps cumulé lors du calcul du parcours et des contentions. Puis la fonction « transport » transmet la requête à la bonne destination et attend la réponse de celle-ci.

Cette sous section a permis de détailler l'implémentation des différentes interconnexions proposées. Cette implémentation va explorer des différentes topologies, mais pour cela il est nécessaire d'avoir un environnement adéquat. C'est pourquoi, un environnement de comparaison de réseaux est décrit dans la sous section suivante.

2.4.2 Un environnement de simulation pour la comparaison de topologies

Cette sous section présente la structure de comparaison utilisée pour mettre en parallèle les topologies. La figure 2.6 représente cette nouvelle structure. Cette structure de comparaison comprend principalement des générateurs de trafic, le réseau à tester et des mémoires qui vont répondre aux requêtes des générateurs de trafic. Afin d'aider l'exploration, un fichier de paramètres permet de changer certains paramètres sans recompilation.

Nous avons choisi de modéliser, à l'aide des générateurs de trafic, le cas où les processeurs sont directement connectés aux mémoires sans mémoires caches. Ce choix suppose que les processeurs soient capables d'accéder aux différentes mémoires

rapidement. Il en résulte donc des accès 32 bits aléatoires aux différentes mémoires.

Le réseau a déjà été largement détaillé dans la sous section précédente. Les différents modules mémoires sont capables de recevoir de multiples requêtes, de les gérer une à une afin de garantir l'intégrité des données et enfin d'attribuer à chacune une pénalité correspondante à l'attente et au temps de traitement des demandes. Il est possible de modifier à travers le fichier de paramètres le temps de traitement d'une donnée dans la mémoire. Le générateur de trafic est un module capable d'envoyer des transactions TLM en fonction de différents paramètres. Il permet de représenter un type d'application sans être dépendant d'une application en particulier. Il est également utile afin de modifier facilement la charge réseau. Le générateur de trafic a aussi pour but de mesurer la latence entre l'envoi d'une transaction et la réception de la réponse. Nous avons choisi cette latence globale comme indicateur de la performance d'un réseau. En effet dans le cas de messages directs entre processeur et mémoire, la latence est un point important. Dans la pratique, une mémoire est toujours associée à un processeur. Ils ont le même identifiant ce qui permet de les placer au plus près l'un de l'autre dans le réseau lors de l'instanciation.

L'implémentation d'un générateur de trafic se résume à deux threads SystemC. Le premier thread crée des requêtes TLM à une fréquence déterminée. Ces requêtes sont stockées dans une FIFO surdimensionnée afin d'éviter tout débordement. Le second thread se charge d'envoyer les requêtes de la FIFO. L'envoi d'un thread est effectué une fois que la précédente réponse est revenue ce qui simule des appels bloquants de la part d'un processeur. Les deux principaux paramètres d'un générateur de trafic sont donc la période entre deux envois et le choix de l'adresse de destination.

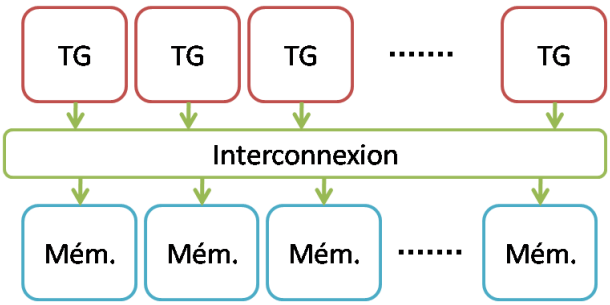


FIG. 2.6 – Représentation de la structure de comparaisons des différentes topologies. (TG : générateur de trafic)

La période entre la création de deux requêtes permet de générer différentes charges réseau. La variable, présente dans le fichier de paramètres, gère la période maximale de la création. Elle représente le temps maximum entre deux créations. Afin de ne pas créer toutes les requêtes en même temps, le temps attendu avant la prochaine création est un nombre pseudo-aléatoire compris entre 0 et le temps maximum défini dans le fichier de paramètres.

Le choix de l'adresse fournie à chaque requête est soumis à différentes lois suivant

le type de trafic que l'on veut générer. Il existe deux types de trafics implémentés.

Le trafic aléatoire - Le premier type suit une loi uniforme sur l'ensemble de l'espace mémoire. Le trafic alors généré est aléatoire sur l'ensemble de l'espace mémoire. La loi uniforme est dimensionnée en fonction de l'espace d'adressage. Elle a été choisie car elle permet d'exprimer le cas le plus général possible, celui où aucune communication ne peut être prédite.

Dans le cas de structures hiérarchisées, on considère que les générateurs de trafic et les mémoires sont répartis par groupe nommé « cluster ». Il est possible de choisir, à l'aide d'un paramètre, un pourcentage de communications. Celles-ci resteront alors à l'intérieur du cluster émetteur. Ainsi pour savoir si une communication sort ou non du cluster, un tirage aléatoire entre 0 et 100 est comparé au pourcentage voulu. Ensuite, si la requête reste dans le cluster, le choix de l'adresse se fait sur l'espace d'adressage de celui-ci suivant une loi uniforme. Sinon dans le cas où l'on quitte le cluster émetteur, dans un premier temps, le choix du cluster de destination se fait suivant la loi uniforme puis l'adresse est choisie dans le cluster de destination toujours suivant une loi uniforme. Cette version, pour les structures hiérarchisées, servira principalement à mesurer l'impact de la localité des communications sur les performances.

Le trafic localisé - Le deuxième type de trafic implémenté suit la loi normale. Ce deuxième trafic a été choisi car il permet d'exprimer une localité, qui n'est pas présente dans le premier type. Dans ce cas, il est indispensable de donner la variance comme paramètre de la loi. Celle-ci détermine l'aplatissement de loi normale, ce qui correspond dans notre cas à la distance moyenne parcourue par les messages. La mesure de distance dans le réseau, que nous avons retenu, est le nombre de sauts réalisés pour aller d'un point à un autre. Dans cette loi, la variance permet de définir la distance moyenne entre le générateur de trafic émetteur et la mémoire réceptrice.

Toute cette section a permis de présenter la structure de simulation et plus particulièrement l'implémentation des réseaux et du générateur de trafic qui vont servir à l'exploration. Il reste à comparer le simulateur à d'autres structures afin de caractériser sa précision et sa rapidité.

2.5 Caractérisation de la rapidité et de la précision de notre simulateur

Dans cette section, nous allons vérifier deux aspects de notre simulateur. Dans un premier temps, nous allons contrôler s'il y a un réel gain de rapidité entre le TTLM et l'ATTLM. Ensuite les résultats du simulateur seront comparés à un simulateur précis au cycle afin de voir sa précision. La vérification de la précision a pour but de savoir si les comparaisons permettront de discrétiser chaque topologie.

2.5.1 Caractérisation de la rapidité

Nous avons dit dans la première section de ce chapitre que l'ATTLM offre un gain en rapidité par rapport au TTLM. Les tests suivants ont été réalisés afin de vérifier et mesurer ce gain en rapidité. La comparaison a été effectuée sur le temps d'exécution d'un test simple dans le cas de deux plates-formes identiques : la première utilisant le niveau TTLM et la deuxième le niveau ATTLM.

Ces plates-formes sont composées de 32 ou 64 générateurs de trafic et d'un nombre variable de mémoires. Le test n'a pas besoin de toutes les mémoires instanciées pour s'exécuter mais celles-ci vont permettre d'augmenter facilement la taille de la plate-forme à étudier. Dans le cas du TTLM, chaque module de la plate-forme possède un thread qui est synchronisé par une horloge commune. Dans le cas de l'ATTLM, chaque module de la plate-forme possède également un thread mais ces threads sont synchronisés sur des événements. On a donc, dans les deux cas, un nombre de threads présents dans le simulateur égal au nombre de mémoires plus un nombre fixe d'autres modules.

La figure 2.7 présente le temps de simulation en secondes du même test dans le cas de 32 et 64 générateursde trafic pour la plate-forme au niveau TTLM et au niveau ATTLM en fonction du nombre de mémoires. Cette même figure fait également apparaître l'accélération entre les deux versions.

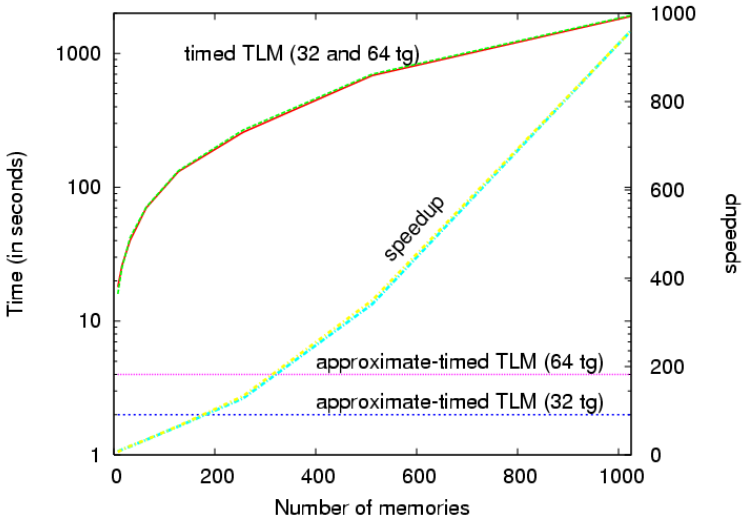


FIG. 2.7 – Temps de simulation en fonction du nombre de mémoires dans les cas du TTLM et ATTLM. (TG : générateur de trafic)

La figure 2.7 met en évidence que dans le cas d'une simulation TTLM le nombre de threads à un impact considérable sur le temps de simulation.

Cela s'explique principalement par le fait qu'il est nécessaire de réveiller l'ensemble des threads afin de vérifier si oui ou non quelque chose est à réaliser. Le réveil inutile d'un grand nombre de threads a pour effet de ralentir la simulation.

Dans le cas du ATTL, seuls les threads utilisés sont réveillés quel que soit le nombre de mémoires présentes dans le simulateur et donc quel que soit le nombre de threads au total. On voit sur la figure 2.7 qu'à partir de 200 mémoires l'accélération dépasse 100. Ce qui permet de diviser par 100, les temps de simulation de grands systèmes.

Cette sous section a mis en évidence que l'ATTLM possède deux caractéristiques. Il permet d'accélérer les simulations SystemC et de plus il n'est pas influencé par le nombre de threads non utilisés. Maintenant que la rapidité a été vérifiée, des tests doivent être réalisés afin de caractériser la précision du simulateur.

2.5.2 Caractérisation de la précision

L'implémentation du TLM approximé, détaillée précédemment, n'est pas précise au cycle près. Il est donc nécessaire de vérifier que la précision obtenue est suffisante pour comparer les différentes topologies. Afin de réaliser une comparaison, Noxim [71] a été choisi comme point de référence. Ce simulateur SystemC est précis au cycle près. Il décrit l'ensemble du réseau en spécifiant chaque routeur selon une grille ou un tore 2D. Les liens entre les routeurs sont des signaux SystemC. La comparaison se fera sur une grille 2D utilisant le « wormhole ». L'algorithme de routage utilisé est le routage XY. La figure 2.8 présente la latence moyenne en fonction de la charge réseau pour un trafic uniforme pour différentes tailles de réseau.

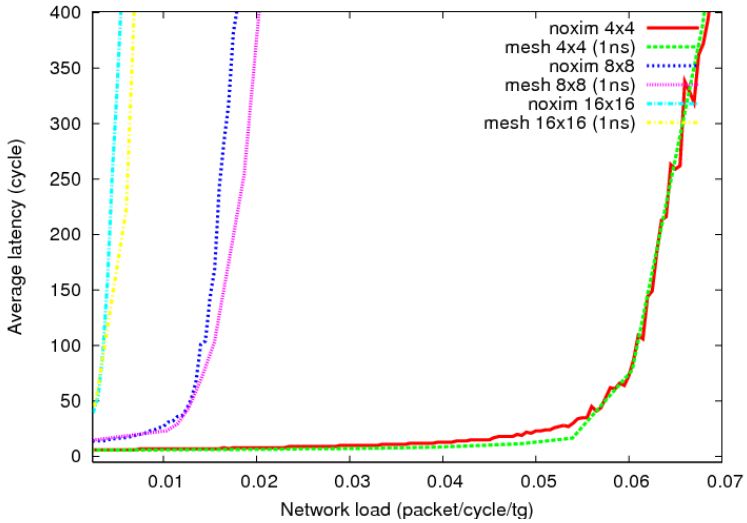


FIG. 2.8 – Comparaison entre Noxim et notre approche.

On voit que selon la taille, les différences restent inférieures à 10% avant la saturation et varient entre 0 et 20% d'erreur après saturation. Après saturation, l'augmentation étant exponentielle, une petite erreur d'estimation entraîne une grande variation. On trouve en moyenne moins de 10% d'erreur. Cette précision est largement suffisante pour discriminer correctement les performances de chaque topologie. Seule la topologie grille 2D a été vérifiée, faute de temps. Cette caractérisation de

la précision du simulateur permet de supposer que le modèle de simulation, qui consiste en la gestion de listes de routeur+lien, est assez précis. Et donc du fait que la distinction entre les différentes topologies réside uniquement dans la création de listes de routeur+lien, les résultats pour les autres topologies seront sensiblement identiques. Les deux caractérisations en performance et en précision étant terminées, la section suivante conclut ce chapitre.

2.6 Conclusion sur la nouvelle approche proposée

Le chapitre 1 a mis en évidence la nécessité d'avoir un environnement de comparaison afin d'être capable de choisir une topologie adaptée à nos besoins. Cet environnement doit permettre d'explorer rapidement les différentes solutions retenues.

Ce chapitre nous montre que les simulateurs actuels ne répondent pas à nos problèmes de rapidité et de précision et surtout ne proposent pas les topologies retenues pour la comparaison. C'est pourquoi, ce chapitre décrit une nouvelle approche TLM afin de simuler des réseaux sur puce. Cette approche propose d'estimer le temps de traversée ainsi que les contentions dans le réseau sans modéliser le réseau de façon structurelle. Cette approche ATTLM obtient des simulations cent fois plus rapides par comparaison avec un modèle TTLM. Cette vitesse de simulation est nécessaire afin de réduire les temps d'exploration. Cependant étant donné que les temps sont estimés, le simulateur est précis à 90% par rapport à un simulateur précis au cycle. La précision obtenue avec notre simulateur permet de discriminer correctement différentes topologies. Ce chapitre a également présenté un environnement d'exploration afin de comparer les topologies.

Le chapitre suivant présente une étude comparative d'un point de vue performance et surface des différentes topologies retenues dans le chapitre 1 utilisant l'environnement de simulation de ce chapitre. Cette étude a pour but de choisir une topologie adaptée à un environnement « many-core » exécutant des applications dynamiques.

Chapitre 3

Proposition d'une topologie hiérarchique indépendante de l'applicatif et efficace d'un point de vue silicium

Sommaire

3.1 Description de réseaux représentatifs de l'espace de conception et leur implémentation dans le simulateur 54
3.1.1 Les réseaux à plat 55
3.1.2 Les réseaux hiérarchisés 56
3.2 Comparaison des performances des différentes topologies . 59
3.2.1 Comparaison des performances sous trafic uniforme 59
3.2.2 Comparaison des performances sous trafic localisé 62
3.3 Description de la méthode de synthèse et résultats de synthèse ASIC 67
3.3.1 Méthodologie de synthèse 67
3.3.2 Résultat de synthèse 67
3.4 Comparaison des efficacités silicium des différentes topologies 69
3.4.1 Comparaison de l'efficacité surfacique sous trafic uniforme . . 70
3.4.2 Comparaison de l'efficacité surfacique sous trafic localisé . . . 73
3.5 Conclusion sur le choix de la topologie 74

Le chapitre 2 a mis en place une structure de comparaison de performance afin de permettre de choisir une topologie. Cette topologie devra être capable de connecter un grand nombre de composants. De plus, du fait des propriétés de dynamisme des applications, la topologie devra accepter des trafics non prédictibles et donc être indépendante de l'application. Enfin, l'implémentation de cette topologie devra répondre aux contraintes de surface et de consommation électrique des systèmes embarqués.

Le chapitre 3 présente la démarche suivie pour choisir une topologie indépendante du point de vue de l'applicatif, performante et de petite surface. A partir d'ici, la performance d'un réseau représente la capacité de celui-ci à accepter une charge réseau importante. Cette performance est évaluée par la charge réseau à partir de laquelle la latence moyenne dans le réseau sature à une vitesse exponentielle. On dira donc qu'un réseau est plus performant qu'un autre s'il accepte une charge réseau plus importante avant de saturer.

Le choix de la topologie se fait à partir des deux critères : performance et surface. Cependant, comme il n'est pas aisé d'interpréter ces deux critères en même temps, l'efficacité surfacique va être introduite et permettra d'effectuer le choix final. Afin de réaliser les comparaisons, nous allons dans un premier temps décrire l'ensemble des réseaux utilisés, leurs topologies ainsi que leurs autres paramètres. Ensuite, les différents réseaux seront comparés d'un point de vue performance. Afin de déterminer l'impact de la surface, une section de ce chapitre est dédiée à l'évaluation de la surface des différentes solutions. Cette évaluation de la surface va permettre de faire une comparaison des différentes topologies au niveau efficacité surfacique.

3.1 Description de réseaux représentatifs de l'espace de conception et leur implémentation dans le simulateur

Le chapitre 1 a mis en évidence une liste de topologies représentatives d'un échantillon de l'espace de conception. Afin de permettre cette comparaison, un réseau sur puce doit être défini pour chaque topologie. Cette section décrit les choix de paramètres pour chaque réseau et donc chaque topologie. Pour faciliter la comparaison, nous avons tenté de maximiser les invariants entre ces réseaux. De ce fait, un certain nombre de ces paramètres sont identiques pour tous les réseaux. Ainsi, afin de limiter la surface, tous les réseaux proposés utilisent le « wormhole » comme type de routage et n'implémentent aucun canal virtuel. On considère que tous les buffers des routeurs ont une profondeur de 1 étant donné notre trafic sur le réseau. En effet, nous supposons qu'il n'y a pas de mémoires caches entre les processeurs et les mémoires et donc que les accès vont être de 32 bits. Il n'y a donc aucun besoin de bufferisation dans un fonctionnement optimal d'où le choix de la profondeur des buffers. Tous les liens et bus ont une largeur de 32 bits et sont bidirectionnels. Puisque que le routage est différent pour chaque topologie, le routage le plus simple a été choisi. Nous avons choisi de réaliser les comparaisons sur un multiprocesseur à mémoire partagée. Ceci implique que l'architecture contient des initiateurs (générateur de trafic) et des cibles (mémoires). Ils se composent, pour toutes nos expériences, de 256 générateurs de trafic et 256 mémoires afin d'avoir plus d'une centaine d'initiateurs. La topologie butterfly Multi-étage ne permet pas n'importe quel nombre d'initiateurs à cause de son calcul d'étages. En effet étant donné que le nombre d'étage est le logarithme du nombre d'entrée dans le réseau, dans la base de l'ordre du nombre d'entrées sorties des routeurs. Le simulateur nécessite d'être renseigné au niveau du temps de traversée de chaque routeur. Pour cela, nous avons supposé que la fréquence de fonctionnement de ces réseaux était de 1 GHz et nous avons estimé le nombre de cycles de traversée en fonction de la longueur des fils et de la taille des routeurs. Cette estimation se base sur un placement/routage hypothétique de chaque réseau. Pour chaque réseau, il sera donné la valeur utilisée par le simulateur pour évaluer le nombre de cycles de traversée. Afin de différencier tous ces réseaux, ils seront identifiés par le nom de leur topologie. Nous allons dans un premier temps

décrire les réseaux à plat puis les structures hiérarchisées.

3.1.1 Les réseaux à plat

Cette sous section spécifie les caractéristiques des interconnexions à plat choisies pour la comparaison. On retrouve le Multi-bus, l'Anneau, le Butterfly Multi-étage, la Grille 2D et enfin le Tore 2D.

Multi-bus - Le Multi-bus implémente les initiateurs (générateurs de trafic) d'un côté du réseau et les cibles (mémoires) de l'autre comme sur la figure 1.3(b) du premier chapitre. On considère le Multi-bus dans sa version la plus connectée, ce qui signifie que chaque entrée possède un bus dédié pour écrire dessus. Par conséquent, il n'y aura pas de contention sur les bus car il n'y a qu'un seul écrivain par bus. Cependant à l'entrée des mémoires, l'arbitre qui se trouve l'entrée de chaque mémoire doit alors gérer les contentions possibles. Durant tous les tests, l'arbitre choisi est le Round-Robin afin qu'en moyenne aucune entrée ne soit privilégiée. Le Multi-bus reçoit une pénalité de quatre cycles lorsqu'il est la seule interconnexion de la puce et une pénalité d'un cycle lorsqu'il est dans un cluster. Ces temps de pénalité ont été déduits de la profondeur de l'arbre de multiplexage nécessaire.

Anneau - Pour l'Anneau, on considère qu'une mémoire et un générateur de trafic constituent une tuile. A chaque routeur est connecté une tuile, comme vu précédemment sur la figure 1.6(a). Dans le simulateur, il est nécessaire de choisir, pour ce réseau, le nombre d'Anneaux que l'on veut utiliser. Ce nombre d'Anneaux correspond au nombre de liens entre chaque routeur. Ce paramètre a un fort impact sur la surface et les performances. En effet, dans un souci de simplicité, le contrôle et la matrice d'interconnexion de chaque routeur sont répliqués pour chaque Anneau. Il est important de noter que les performances ne sont pas proportionnelles au nombre d'Anneaux. En effet, on arrive rapidement à une saturation des performances lorsqu'on augmente le nombre d'Anneaux. Cette saturation est due à la latence moyenne d'un message dans l'Anneau qui est directement liée à la taille de l'interconnexion. En effet, pour une distribution uniforme des communications, cette latence moyenne vaut le quart du nombre d'éléments sur l'Anneau. Un initiateur ne peut écrire que sur un seul Anneau en fonction de son identifiant. Au contraire une cible est capable de lire tous les Anneaux. Dans le cas de notre étude, nous avons choisi quatre Anneaux pour ne pas obtenir une surface trop importante comme nous le verrons plus tard. Le routage dans l'Anneau se fait en fonction de d'adresse de destination, le message part dans le sens qui produit le chemin le plus court. L'Anneau reçoit une pénalité d'un cycle pour la version à plat et de trois cycles s'il est entre plusieurs clusters. Cette différence de pénalité est due à la différence de longueur de fils entre le cas où ce sont les processeurs qui sont connectés et le cas où ce sont les clusters.

Butterfly Multi-étage - Le Multi-étage, comme le Multi-bus, place d'un côté du réseau l'ensemble des générateurs de trafic, et de l'autre les mémoires. Comme expli-

qué dans le chapitre 1, le nombre d'étages est directement lié au nombre d'entrées et de sorties. Le détail des connexions entre les étages sont visibles sur la figure 1.7(c) du premier chapitre. 256 entrées et 256 sorties imposent des routeurs de 2, 4 et 16 entrées/sorties. Ainsi pour un routeur 2 entrées/sorties, on obtient 8 étages de 128 routeurs. Cette configuration engendre un chemin de 8 sauts et un réseau de 1024 routeurs. Ceci implique un temps de traversée minimal élevé et un grand nombre de routeurs. Avec des routeurs 16 entrées/sorties, on obtient 2 étages de 16 routeurs. Cette nouvelle configuration permet de réduire le nombre de routeurs et le temps de traversée minimal mais on peut aisément soupçonner qu'il y aura beaucoup de contentions dans les routeurs. Dans le cas de SPIN [38], les auteurs ont ajouté deux buffers supplémentaires dans chaque routeur afin de mémoriser les messages en attente du fait des contentions. Afin de limiter le temps de traversée du réseau et le nombre de routeurs sans engendrer trop de contentions, nous avons choisi d'implémenter des routeurs de 4 entrées/sorties afin d'obtenir 4 étages de 64 routeurs. Dans le cas de cette topologie, un seul chemin est possible entre deux points. Le routage correspond au « Destination Tag Routing ». Cela signifie que l'adresse code l'ensemble des sorties successives que doit prendre le message à la sortie de chaque routeur. Pour l'estimation de la latence, le Multi-étage est particulier car il n'a pas une longueur de fils homogène entre tous les étages. Ainsi nous avons fait une moyenne sur tous les étages et nous avons estimé à trois cycles par étage la pénalité de temps.

Grille 2D / Tore 2D - Pour ces deux topologies, les paramètres sont les mêmes. A chaque routeur est connecté un ensemble processeur mémoire à travers le même port. Du fait qu'il y ait 256 générateurs de trafic et 256 mémoires, on considère une Grille de 16 x 16 routeurs. Le routage choisi est le « XY routing » car il permet d'assurer que le réseau soit sans « dead lock ». De plus, il limite le routage à de simples comparaisons dans chaque routeur ce qui restreint la surface du réseau. La Grille et le Tore reçoivent tous les deux une pénalité d'un cycle dans le cas de topologie à plat. Pour le Tore, nous avons supposé qu'il est replié afin d'égaliser la longueur des fils. Dans le cas où le Tore est entre différents clusters sa pénalité augmente à trois cycles à cause de l'augmentation significative de la longueur de ces liens entre les routeurs.

3.1.2 Les réseaux hiérarchisés

Les structures hiérarchisées suivantes sont basées sur un groupement de générateurs de trafic et de mémoires nommées « cluster ». On considère que chaque cluster contient N générateurs de trafic, N mémoires et un réseau Multi-bus comme le montre la figure 3.1(a). Nous avons choisi un Multi-bus comme interconnexion dans le cluster car pour un petit nombre de connexions, celle-ci est la meilleure interconnexion [NOCS10] car elle obtient les meilleures performances pour la taille la plus petite. Le passage d'un niveau à l'autre s'effectue par des liens entre le Multi-bus du cluster et le routeur du niveau supérieur. Pour cela le routeur du niveau supérieur se

connecte en tant qu'initiateur et cible sur le Multi-bus du cluster. Le nombre de liens entre les niveaux est paramétrable. On considère que, pour chaque lien sortant des clusters, on duplique le réseau du niveau supérieur. Sur la figure 3.1(b) par exemple, le cluster possède deux liens, le routeur 1 est dupliqué afin de connecter chaque lien sur un réseau différent. Entre les différents niveaux une interface est placée afin de mettre à jour les identifiants qui servent à naviguer dans le réseau ainsi qu'à contrôler le nombre de messages qui traversent chaque lien. La différence entre les réseaux hiérarchiques réside dans le choix du réseau supérieur et de la manière dont les clusters sont connectés.

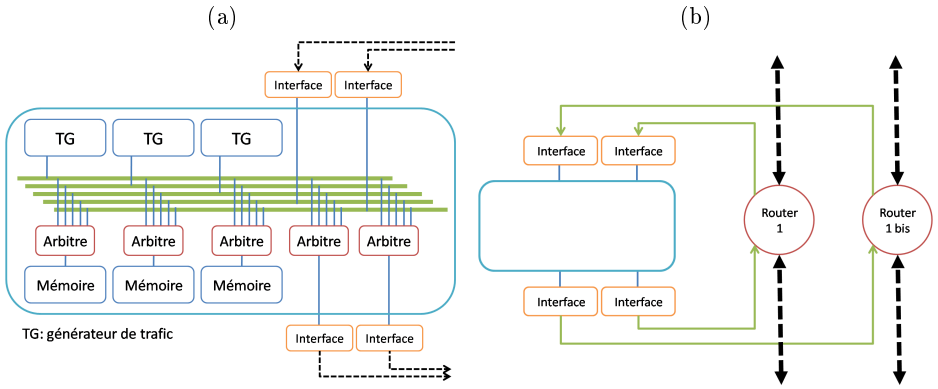


FIG. 3.1 – (a) Représentation d'un cluster contenant trois processeurs et trois mémoires. Ce cluster possède deux liens vers le niveau supérieur. (b) Représentation d'une connexion entre un cluster et deux routeurs.

Multi-bus + Anneau (réseau MA) et Multibus + Tore (réseau MT) - La figure 3.2(a) présente la structure Multi-bus + Anneau (réseau MA). Le réseau supérieur dans ce cas est un Anneau, alors que la figure 3.2(b) présente la structure Multi-bus + Tore (réseau MT) qui implémente un Tore 2D comme réseau de niveau supérieur. Dans les deux cas, les réseaux supérieurs sont des clones des réseaux à plat sauf que sur chaque routeur un cluster est connecté au lieu d'une tuile constituée d'un générateur de trafic et d'une mémoire.

Multi-bus + Anneau amélioré (réseau MX) - La figure 3.3 illustre la topologie Multi-bus + Anneau amélioré (réseau MX). Cette topologie se différencie des autres propositions car elle augmente le degré de connexion des routeurs. Ainsi sur un même routeur, on connecte quatre clusters au lieu d'un seul. Ce type de connexion permet d'utiliser la matrice d'interconnexion du routeur comme un niveau intermédiaire dans la hiérarchisation. Ce regroupement autour des routeurs permet de créer des liens privilégiés entre quatre clusters et ainsi optimise l'utilisation de la matrice d'interconnexion du routeur.

Pour les structures hiérarchiques, plusieurs configurations sont possibles si on

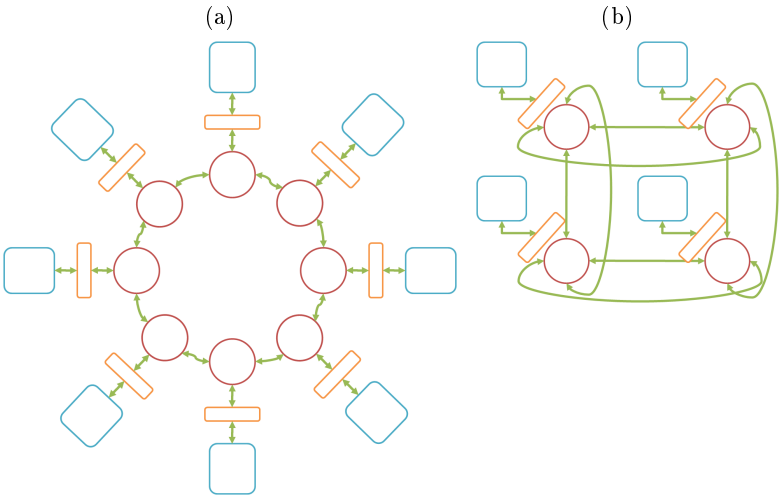


FIG. 3.2 – (a) Représentation d'un Multi-bus + Anneau (MA) connectant huit clusters avec un lien. (b) Représentation d'un Multi-bus + Tore (MT) connectant quatre clusters avec un lien.

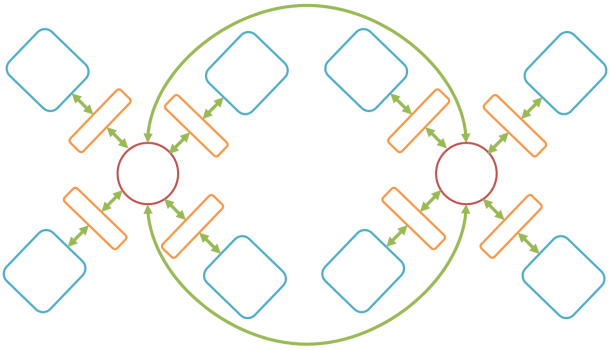


FIG. 3.3 – Représentation de la topologie Multi-bus + Anneau amélioré (MX) contenant huit clusters avec un lien.

considère des nombres de générateurs de trafic et de mémoires fixes. Ces différentes configurations sont paramétrées par le nombre de processeurs dans chaque cluster et par le nombre de liens. On considérera que les clusters peuvent contenir 4, 8 ou 16 générateurs de trafic et respectivement 4, 8 ou 16 mémoires ce qui répartit l'ensemble des générateurs de trafic et mémoires respectivement dans 64, 32 ou 16 clusters. Du fait que l'on duplique le réseau supérieur en fonction du nombre de liens, on se limite à 2 ou 4 liens entre les deux niveaux. Pour simplifier la suite du manuscrit, les noms des réseaux hiérarchiques vont être réduits à des acronymes. Ainsi l'association Multi-bus + Anneau sera nommée réseau MA, l'association Multi-bus + Tore : réseau MT et l'association Multi-bus et Anneau amélioré : réseau MX. La section suivante présente les résultats de performance obtenus pour tous les réseaux étudiés.

3.2 Comparaison des performances des différentes topologies

Cette section se focalise sur la comparaison des performances de toutes les solutions proposées. Afin de réaliser cette comparaison, nous avons choisi d'utiliser des générateurs de trafic et donc d'utiliser des trafics génériques au lieu d'applications réelles. Ce choix est motivé par le fait que les trafics génériques permettent de réaliser de plus larges séries de test en jouant sur la charge du réseau par exemple. De plus, dans notre contexte d'applications dynamiques, il est plus simple de considérer un trafic synthétique qu'un large panel d'applications réelles.

Nous avons choisi deux types de trafic qui seront générés dynamiquement pour nos comparaisons. Le premier type envoie des messages uniformément à travers toute la structure. Ce trafic uniforme représente un placement aléatoire des tâches sur les ressources de calcul. Le deuxième type crée un trafic localisé autour de la ressource émettrice à l'aide d'une loi normale. Ce type de trafic s'apparente à un placement intelligent des tâches au plus près des données consommées par celles-ci.

Toutes les courbes réalisées correspondent à des moyennes sur vingt simulations générant chacune plus de cinq mille requêtes par générateurs de trafic. Les sous sections suivantes présentent les résultats en fonction du trafic uniforme puis du trafic localisé.

3.2.1 Comparaison des performances sous trafic uniforme

Cette sous section a pour but de présenter les résultats obtenus avec un trafic uniforme et de comparer les performances des différentes solutions retenues. De plus l'impact des configurations des structures hiérarchiques sur les performances va être mis en évidence. Enfin l'étude des performances des structures hiérarchiques soumises à un trafic uniforme modifié sera présentée.

Les figures 3.4 présentent les latences moyennes en fonction de la charge réseau pour un trafic uniforme. La première figure 3.4(a) montre les courbes pour les réseaux à plat. Les figures 3.4(b), 3.4(c), 3.4(d) sont les courbes pour les différentes configurations respectivement des réseaux MA, MT et MX.

La figure 3.4(a) met en évidence que le Multi-bus est l'interconnexion la plus performante avec une saturation qui se produit à une charge réseau deux fois plus élevée que celle du Tore 2D. Ceci s'explique par le fait que le temps de traversée d'un Multi-bus est fixe et rapide. Comme prévu le Tore est plus performant (+43%) que la Grille du fait de ces liens supplémentaires. On voit aussi, sur cette figure, que le Multi-étage a des performances similaires à la Grille. Cependant, la différence est que le Multi-étage possède un temps de traversée fixe quelque soit l'émetteur et le destinataire du message. Enfin le réseau contenant quatre Anneaux a des résultats extrêmement faibles. On peut facilement comprendre que les performances maximales sont rapidement atteintes étant donné qu'en moyenne un message réalise 64 sauts à la différence d'une Grille qui a une moyenne de 16 sauts.

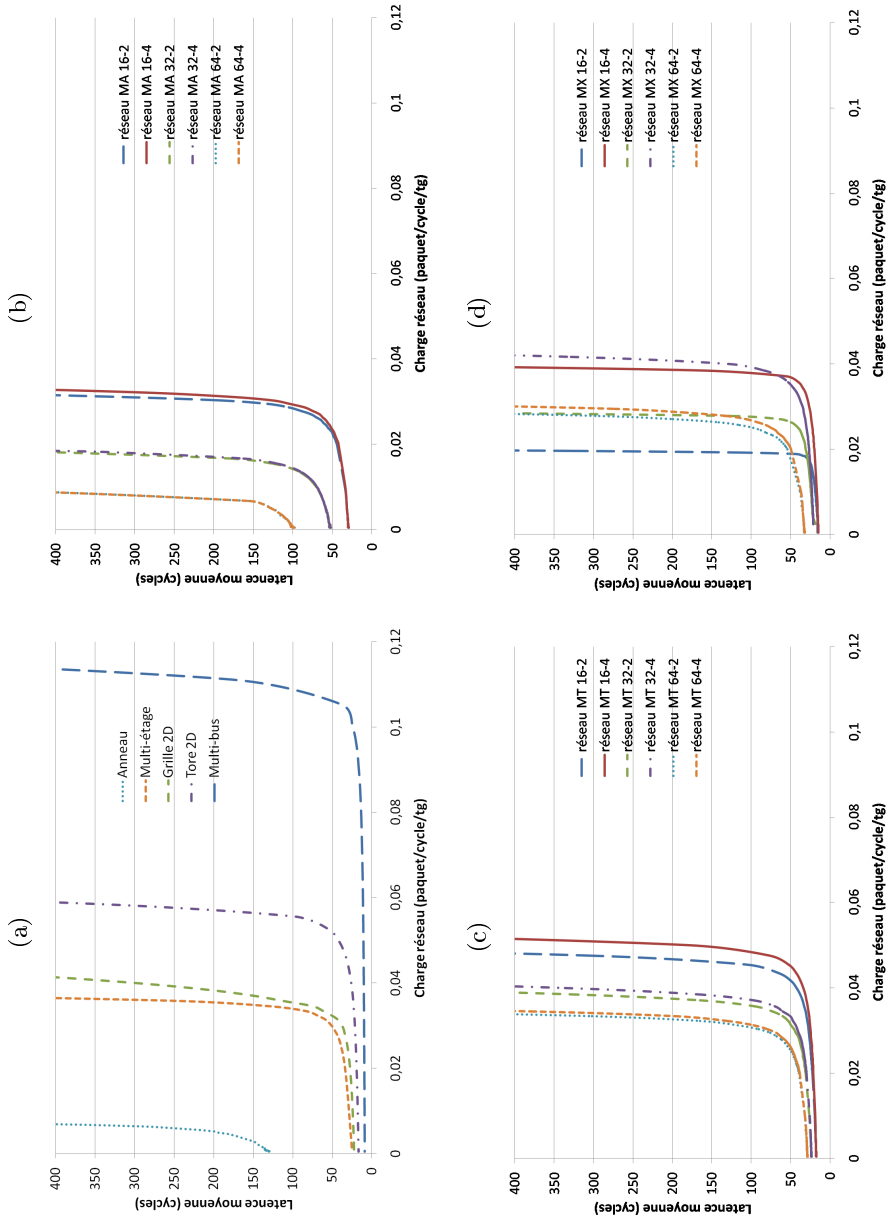


FIG. 3.4 – (a) Latence moyenne des réseaux à plat en fonction de la charge réseau pour un trafic uniforme. (b),(c),(d) représentent respectivement la latence moyenne des réseaux MA, MT et MX selon leur configuration et en fonction de la charge réseau pour un trafic uniforme. Les configurations des réseaux se lisent comme suit 16-2 ce qui signifie une découpe en 16 clusters avec 2 réseaux superposés entre les clusters. (MA : Multi-bus + Anneau ; MT : Multi-bus + Tore ; MX : Multi-bus + Anneau amélioré)

La figure 3.4(b) présente les résultats pour le réseau MA pour toutes ses configurations. On voit que moins il y a de clusters et plus les performances sont bonnes. Ceci s'explique par l'utilisation de l'Anneau comme réseau supérieur. En effet, moins il y a de routeurs sur l'Anneau et plus la distance moyenne diminue, donc plus les performances augmentent. Cette figure fait apparaître également que le nombre de liens entre les deux niveaux de hiérarchisation a peu d'influence sur les performances. En effet la latence moyenne est principalement influencée par la lenteur de l'Anneau. Cette tendance est confirmée par le fait que plus il y a de clusters et plus l'écart entre les courbes est faible voir nul. Il faut toutefois remarquer que la hiérarchisation de l'anneau a permis de doubler les performances par rapport au réseau Anneau à plat.

Etant donné que le Tore est beaucoup plus performant que l'Anneau, il est normal que le réseau MT (Figure 3.4(c)) obtienne de bien meilleures performances que le réseau MA. De la même manière le nombre de liens a peu d'influence sur les performances du réseau MA étant donné que c'est la distance moyenne qui pénalise le plus les performances de l'Anneau. Les résultats du réseau MT sont plus regroupés car le Tore est une topologie dont les performances décroissent moins vite avec l'augmentation de la taille du réseau que l'Anneau.

La dernière figure 3.4(d) présente les résultats du réseau MX. Ceux-ci mettent en évidence que l'augmentation du degré des routeurs (nombre d'entrées/sorties) permet un gain en performance par rapport au réseau MA. Ce gain provient du fait qu'il y a moins de routeurs dans le réseau Anneau supérieur. De plus des liens très rapides entre les clusters sont disponibles à travers le routeur. On peut remarquer que dans le cas du réseau MX, le nombre de liens a un impact important sur les performances. En effet, l'augmentation de ces liens fait croître de la même façon le nombre de liens privilégiés entre les groupements de quatre clusters.

On peut conclure de ces figures que, pour un trafic uniforme, les structures hiérarchisées ont des performances équivalentes aux structures à plat. Seul le Multi-bus parvient à surpasser tous les autres réseaux. Cependant, les réseaux hiérarchisés obtiennent une latence moyenne pour une charge nulle inférieure aux autres réseaux sur puce. Par construction, les réseaux hiérarchisés sont censés favoriser les communications dans le cluster du fait de leur topologie et ainsi limiter les contentions en dehors des clusters. Nous avons procédé à un test afin de vérifier cette hypothèse.

Premier test de localité - Les figures 3.5 suivantes présentent l'effet de la modification du trafic uniforme vers un trafic localisé à un certain pourcentage. Ainsi, on retrouve les différents réseaux hiérarchiques avec différents pourcentages de communication restant dans le cluster (50% et 75%). Au vu du nombre important de configurations, le test est réalisé avec la configuration la plus performante pour chaque réseau hiérarchique soumis à un trafic uniforme. Nous gardons également la courbe du Tore et du Multi-bus comme points de repère. Ces deux réseaux gardent des courbes identiques, du fait qu'ils ne sont pas hiérarchisés et donc pas affectés par cette modification du trafic.

D'après les figures 3.4, les réseaux hiérarchisés avaient tous des performances au mieux égales au Tore. Il apparait sur la figure 3.5(a) que les réseaux hiérarchisés MT et MX obtiennent de meilleures performances que le Tore avec 50% des communications qui se font dans le cluster. Il y a deux raisons principales à ce gain de performance. Dans un premier temps, les structures hiérarchiques vont répartir les contentions globales dans les différents clusters et donc par la même occasion limiter les latences dues à ces contentions. Enfin, étant donné que les Multi-bus sont les réseaux les plus performants, ceci permet de réduire la latence d'une communication si celle-ci reste dans le cluster.

Cette tendance s'accélère sur la figure 3.5(b). Le réseau MT dépasse en performance le Multi-bus alors que le réseau MX obtient les mêmes performances que le Multi-bus. Nous avons dit précédemment que le Multi-bus était l'interconnexion la plus performante cependant les Multi-bus dans les clusters sont plus petits et donc plus rapides ce qui explique que les réseaux hiérarchisés puissent atteindre les même performances que le Multi-bus. Avec l'ensemble des communications dans le cluster, la courbe de latence moyenne en fonction de la charge réseau se superpose avec celle d'un Multi-bus de cluster seul. Ces deux courbes font bien apparaître que les réseaux hiérarchisés offrent de meilleures performances lorsque les communications ne sortent pas de leur cluster. Donc, si on peut localiser le trafic dans un cluster, on obtient de meilleures performances que dans le cas le plus favorable d'une solution à plat.

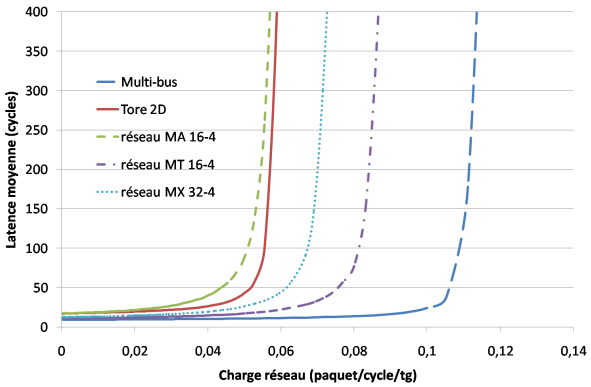
Au vu de ces résultats, nous pouvons affirmer que la localité a un très fort impact sur les performances des réseaux hiérarchisés. Cependant la solution de trafic retenue ne touche que les réseaux hiérarchisés. Nous avons donc proposé un autre trafic qui met en place un trafic localisé pour l'ensemble des réseaux. Ce type de trafic simule l'impact d'une répartition intelligente de tâches sur les ressources. La sous-section suivante teste les réseaux avec un trafic localisé.

3.2.2 Comparaison des performances sous trafic localisé

Cette sous-section a pour but d'étudier l'impact de la localité des messages sur les performances des réseaux. Pour cela, le trafic est changé et suit une loi normale centrée sur l'émetteur. Dans ce cas, la variance représente la distance maximale de la majorité des messages. La figure 3.6 montre la probabilité d'envoi des messages sur l'exemple d'un anneau.

En considérant un placement intelligent, nous avons supposé qu'une tâche nécessitant une donnée se place près de la tâche qui a produit la donnée. Ainsi nous avons considéré que la majorité des messages envoyés par un processeur atteignaient 10% des mémoires de la puce soit 20 mémoires dans son entourage le plus proche. Selon la topologie la variance de la loi normale à appliquer n'est donc pas la même. En effet, le nombre de sauts maximum pour atteindre 20 voisins n'est pas le même suivant les réseaux. Ainsi, un Anneau doit faire 10 sauts au maximum pour atteindre 20 voisins. Pour un Tore, on considère qu'il peut atteindre ces 20 voisins en 4 sauts. En revanche pour une Grille cela dépend de la position du processeur dans celle-ci,

(a)



(b)

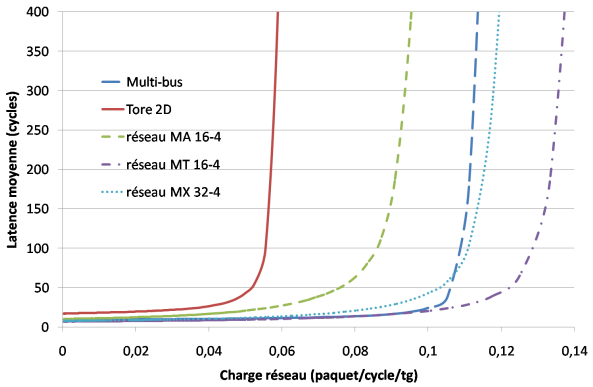


FIG. 3.5 – (a) Latence moyenne des réseaux Tore, Multi-bus, MA, MT et MX en fonction de la charge réseau pour un trafic uniforme avec 50% de communications dans le cluster. (b) latence moyenne des réseaux Tore, Multi-bus, MA, MT et MX en fonction de la charge réseau pour un trafic uniforme avec 75% de communications dans le cluster. Les configurations des réseaux se lisent comme suit 16-2 ce qui signifie une découpe en 16 clusters avec 2 réseaux superposés entre les clusters. (MA : Multi-bus + Anneau ; MT : Multi-bus + Tore ; MX : Multi-bus + Anneau amélioré)

ainsi la variance évolue entre 7 et 4 sauts. Pour simplifier l'implémentation dans le générateur de trafic, nous avons pris une moyenne de 6 sauts. Dans le cas des structures hiérarchisées, le nombre de sauts dépend de la taille du cluster. Ainsi, plus le cluster est petit et plus il est nécessaire d'accéder à des clusters éloignés. Typiquement, avec un cluster de 16 processeurs et 16 mémoires, il est seulement nécessaire d'accéder au cluster voisin pour atteindre 20 mémoires.

Les paramètres étant choisis, les figures 3.7 présentent les courbes de latence moyenne en fonction de la charge réseau pour un trafic localisé pour les différents réseaux étudiés. La première figure 3.7(a) présente les courbes des réseaux à plat.

Le premier point, à remarquer sur cette figure, est que pour la Grille, le Tore et l'Anneau, la localité des transferts a pour effet d'améliorer leurs performances. Il

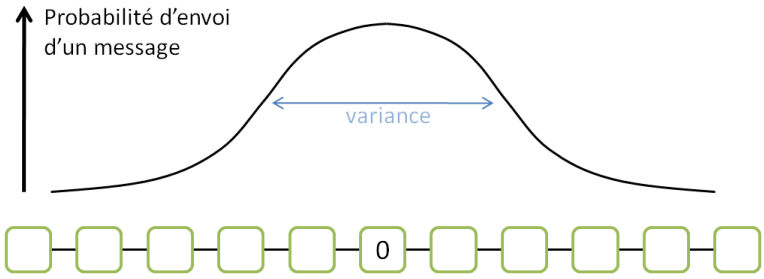


FIG. 3.6 – Représentation de la probabilité d'envoi pour un trafic localisé sur un anneau. La variance permet de modifier la largeur de la loi normale.

y deux raisons à cette augmentation des performances. Tout d'abord, en moyenne les distances parcourues sont grandement diminuées du fait de la localité et donc les latences par la même occasion. Ensuite, vu que les messages se déplacent moins loin, les contentions diminuent puisque deux messages ont des probabilités de se rencontrer plus faibles.

Le deuxième point à remarquer est que le Multi-bus et le Multi-étage sont insensibles au type de trafic étant donné que leurs performances restent identiques malgré le changement de trafic. Ceci s'explique par le fait que leur temps de traversée est identique quelque soit la destination. Le trafic n'a donc aucune incidence sur leurs performances.

Dans le cas des réseaux MA et MT, respectivement figure 3.7(b) et figure 3.7(c), on retrouve les mêmes conclusions globales sur la structure des réseaux que pour le trafic uniforme. Ainsi, plus le réseau supérieur comporte de routeurs et plus les performances se dégradent, étant donné que la latence du réseau supérieur devient beaucoup plus importante que le temps de sortie du cluster. Ceci se confirme en regardant la différence de performance entre des réseaux de même taille pour un nombre de liens différent. Plus le nombre de clusters augmente et moins le nombre de liens a d'impact sur les performances.

Le changement de trafic a toutefois une forte influence sur les performances des réseaux ce qui peut changer le classement des réseaux. Pour le réseau MA les performances ont triplé et il devient donc meilleur que le réseau MT. Ceci s'explique du fait que l'Anneau est bon que accéder à ces voisins. Pour le réseau MT, les performances ont progressé de 40%.

La figure 3.7(d) montre les performances du réseau MX sous différentes configurations avec un trafic localisé. On remarquera que les différences entre les possibles configurations ont diminué car la localité concentre les communications autour des routeurs. La localité du trafic permet de doubler les performances du réseau MX par rapport à un trafic uniforme. L'utilisation de l'Anneau dans le réseau MX permet d'avoir de bonnes performances dans tous les deux cas de trafic.

Contrairement à un trafic uniforme, le trafic localisé permet aux structures hiérarchiques d'obtenir des performances proches de celles du Multi-bus. De plus avec

ce trafic localisé, elles deviennent bien plus performantes que les autres réseaux à plat. On remarquera que pour ce type de trafic, c'est la structure réseau MA qui obtient les meilleures performances grâce à l'Anneau. A la vu du gain de performance obtenu par le réseau MA, on peut se demander pourquoi le réseau MX n'obtient pas de meilleures performances que le réseau MA comme c'était le cas dans le cas du trafic uniforme. Cela s'explique par le fait que les performances du réseau MX suturent. En effet la configuration 16 clusters - 4 liens se superpose avec la configuration 32 clusters - 4 liens. Cette saturation des performances est due au nombre de liens entre les niveaux de hiérarchie. En effet vu que l'on utilise le routeur comme niveau intermédiaire, on arrive à une saturation du routeur à cause de la localité du trafic. Si l'on rajoute des liens, les performances du réseau MX vont vraisemblablement augmenter alors que pour le réseau MA, la latence de l'Anneau risque de faire stagner les performances.

Les résultats de performance montrent que le Multi-bus reste quel que soit le type de trafic, la topologie offrant les meilleurs performances. Parmi toutes les autres solutions, le réseau MX est la plus polyvalente quel que soit le trafic.

Cependant la performance brute n'est pas le seul critère à prendre en compte lorsque l'on considère un système embarqué. En effet, la surface est un autre des critères importants de ces systèmes. C'est pourquoi, la section suivante traite de l'estimation de la surface de tous les réseaux proposés et pour les comparer d'un point de vue surfacique.

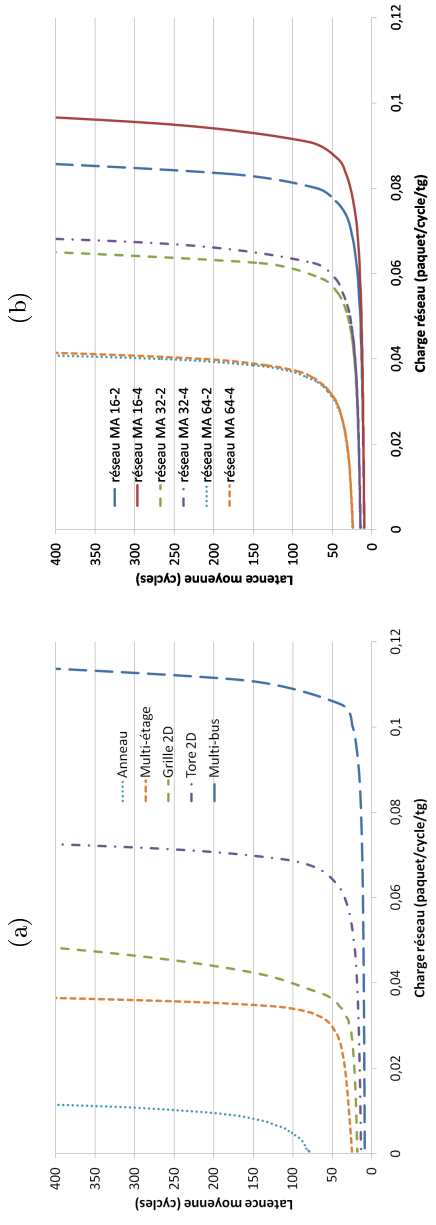


FIG. 3.7 – (a) Latence moyenne des réseaux à plat en fonction de la charge réseau pour un trafic localisé. (b), (c),(d) représentent respectivement la latence moyenne des réseaux MA, MT et MX selon leur configuration et en fonction de la charge réseau pour un trafic localisé. Les configurations des réseaux se lisent comme suit 16-2 ce qui signifie une découpe en 16 clusters avec 2 réseaux superposés entre les clusters. (MA : Multi-bus + Anneau ; MT : Multi-bus + Tore ; MX : Multi-bus + Anneau amélioré)

3.3 Description de la méthode de synthèse et résultats de synthèse ASIC

Cette section présente la méthodologie et les outils utilisées pour réaliser des synthèses ASIC. Elle présente également les résultats des synthèses réalisées sur les réseaux à plat et les estimations de surface des réseaux hiérarchisés.

3.3.1 Méthodologie de synthèse

Cette sous-section montre la démarche utilisée pour obtenir les informations de surface des différents réseaux. Seuls les réseaux à plat ont été décrits complètement au niveau RTL. Nous avons utilisé les codes sources de NoCEm comme base de développement. NoCEm [84] est un projet open-source d'émulateur de réseaux sur puce sur FPGA. Seuls les routeurs pour la topologie Grille et Tore sont disponibles dans le projet. Les topologies Anneau et Multi-étage ont été implémentées à partir du routeur de base disponible. Nous avons utilisé ensuite l'outil Design Compiler de SYNOPSYS et la libraire TSMC 40 nm low power pour réaliser les synthèses.

Les résultats de surface des réseaux à plat sont obtenus à partir de la synthèse du code VHDL avec une contrainte de fonctionnement à 500 Mhz. Il est important de préciser que nous n'avons pas utilisé d'outils de placement routage afin de raffiner ces chiffres de surface. Dans le cas des réseaux hiérarchisés, la surface totale est calculée en additionnant tous les réseaux de base dont ils sont constitués. La sous-section suivante présente l'ensemble des résultats.

3.3.2 Résultat de synthèse

Les résultats de synthèse obtenus sont présentés dans cette sous-section. Dans un premier temps, la figure 3.8 et le tableau 3.1 montrent les résultats de synthèse des différents réseaux à plat pour différentes tailles. Le symbole « X » du tableau signifie que le nombre d'entrées/sorties demandé n'est pas réalisable avec un Butterfly Multi-étage si les routeurs possèdent 4 entrées/sorties. En effet pour ces cas précis, le nombre d'étages nécessaire n'est pas un entier. Les valeurs en bleu, dans le tableau, sont des estimations et non des résultats de synthèse. L'axe des ordonnées de la figure 3.8 est en échelle *log* ce qui permet de mettre en évidence les tendances d'évolution de chaque réseau.

Les résultats confirment que l'Anneau, la Grille et le Tore ont une surface proportionnelle au nombre de noeuds processeur+mémoire à connecter ce qui s'explique par le fait qu'à chaque ajout d'un noeud, un routeur est nécessaire. Les différences entre ces trois réseaux résident dans le fait que le nombre d'entrées/sorties sur chaque routeur est différent. La surface du Multi-étage suit une loi en $N * log(N)$, où N est le nombre d'entrées/sorties du réseau, ce qui rend le Multi-étage rapidement plus gros que la Grille ou le Tore. La surface du Multi-bus suit une loi en N^2. Celui-ci commence en proposant une surface faible pour 8 entrées/sorties ; environ 5,5 fois plus faible que pour la topologie Grille à nombre d'entrées égales. Il conserve une

surface plus faible jusqu'à égaler la Grille pour 64 entrées/sorties. Sa faible surface fait du Multi-bus une interconnexion extrêmement intéressante pour de petits nombres d'entrées/sorties. Ceci confirme notre choix de prendre un Multi-bus dans chaque cluster.

↓ réseaux/Surfaces →	8	16	32	64	128	256
Multi-bus	0,0193	0,0706	0,247	0,905	3,62	14,48
Anneau	0,105	0,196	0,38	0,698	1,52	3,04
Anneau (4)	0,381	0.698	1.39	2.79	5.58	11.17
Grille	0,11	0,235	0,46	0,913	1,82	3,65
Tore	0,139	0,258	0,524	1,017	2.03	4,07
Multi-étage	X	0,3	X	1.21	X	6.85

TAB. 3.1 – Estimation de la surface des interconnexions en mm^2 basée sur une librairie TSMC 40nm. L'anneau (4) signifie qu'il y a quatre liens entre chaque routeur. (X signifie que cette taille ne peut pas être réalisée par le Multi-étage)

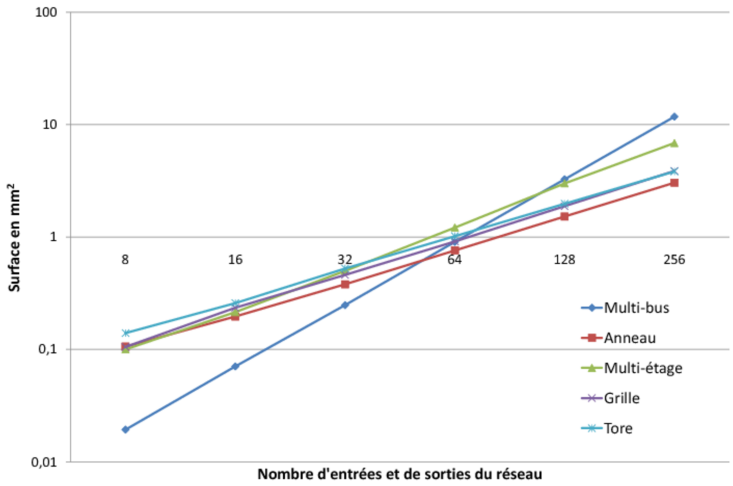


FIG. 3.8 – Evolution de la surface en mm^2 des réseaux à plat en fonction du nombre d'entrées/sorties du réseau.

Le tableau 3.2 présente le résultat de nos estimations pour les réseaux hiérarchisés. Pour chaque topologie hiérarchisée, la surface estimée est la somme de la surface des Multi-bus de chaque cluster plus la surface du réseau supérieur multipliée par le nombre de liens. La surface du réseau supérieur est multipliée par le nombre de liens étant donné que nous avons décidé de dupliquer le réseau supérieur pour gagner en bande passante en limitant le nombre de contentions.

La figure 3.3.2 récapitule la surface estimée ou synthétisée pour chaque réseau comparé pour 256 entrées et 256 sorties au total. on peut remarquer que la surface de l'Anneau à 4 liens entre chaque routeur est quasiment 4 fois plus élevée que

nombre de clusters	16		32		64	
nombre de liens	2	4	2	4	2	4
réseau MA	2.53	3.92	3.41	6.2	5.89	11.48
réseau MT	1.64	2.16	1.66	2.71	2.34	4.38
réseau MX	1.46	1.8	1.29	1.96	1.65	2.99

TAB. 3.2 – Estimation de la surface des interconnexions hiérarchiques en mm^2 en fonction des différentes configurations pour 256 entrées/sorties.

l'Anneau simple. On peut en déduire que l'implémentation que nous avons choisie risque de ne pas répondre à la contrainte de surface du monde de l'embarqué.

On peut remarquer également que les réseaux hiérarchisés permettent d'obtenir de plus petites surfaces en général par rapport aux réseaux à plat. Cela s'explique facilement par la diminution du nombre de routeurs et l'utilisation de Multi-bus de faible taille. La taille d'un routeur de tore seul n'apparait pas dans le tableau 3.1 mais elle est de 0.017 mm^2 soit environ la taille d'un Multi-bus 8 entrées/sorties. On peut facilement voir un routeur comme un Crossbar 5 entrées ce qui explique que les surfaces soient équivalentes. Ainsi le réseau MT 16 clusters - 2 liens vers le niveau supérieur comporte en tout 16 réseaux Multi-bus et 32 routeurs à 5 entrées/sorties contre 256 routeurs à 5 entrées/sorties dans le cas d'un Tore à plat. Ceci permet de diviser la surface par 2.5 par rapport à un Tore à plat et par plus de 7 par rapport à un Multi-bus seul.

Tous les résultats sur la surface des réseaux vont être utilisés dans la section suivante afin de comparer les performances normalisées des réseaux. Cette comparaison va permettre de conclure sur le choix de la topologie la plus adaptée à nos contraintes de l'embarqué et de non prédictibilité.

3.4 Comparaison des efficacités silicium des différentes topologies

Cette section réalise une comparaison des réseaux d'un point de vue efficacité surfacique afin de permettre une analyse globale. Pour cela elle met en place une comparaison entre les réseaux qui permet d'introduire la notion de surface dans l'étude des performances. Ceci est réalisé en divisant, pour chaque réseau, la charge réseau par sa surface, cela permet de faire apparaître les courbes de performance normalisée. Il résulte de ce changement, les mêmes courbes que pour les performances seules mais translatées en fonction de la surface. De ces nouvelles courbes, on peut en déduire la courbe qui a la meilleure performance normalisée et donc la meilleure efficacité surfacique. Comme dans la section 3.2, les trafics uniforme et localisé vont être pris en compte.

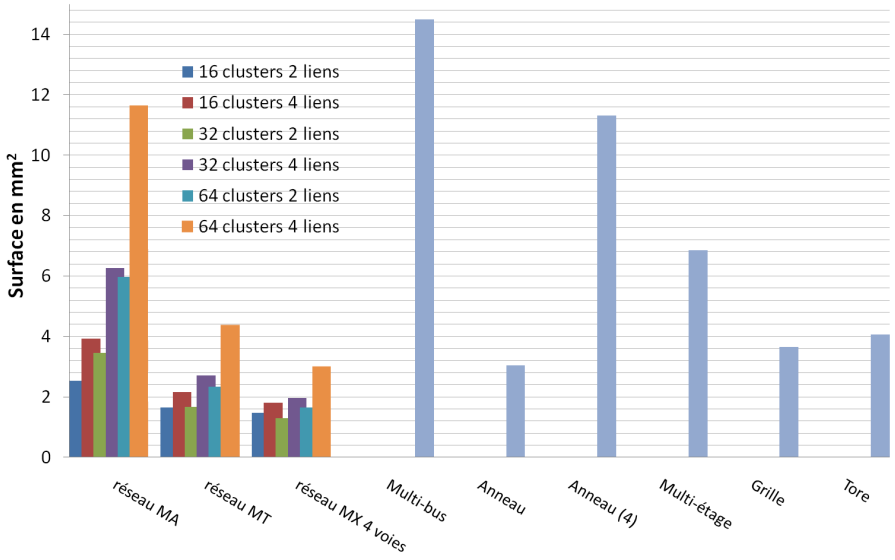


FIG. 3.9 – Présentation des surfaces de tous les réseaux proposés pour 256 entrées/sorties. (MA : Multi-bus + Anneau ; MT : Multi-bus + Tore ; MX : Multi-bus + Anneau amélioré)

3.4.1 Comparaison de l'efficacité surfacique sous trafic uniforme

Dans un premier temps, nous allons évaluer le réseau le plus performant d'un point de vue surface avec un trafic uniforme. Les surfaces utilisées sont celles des tableaux de la section précédente. Les figures 3.10 présentent donc la latence moyenne en fonction de la charge réseau normalisée pour un trafic uniforme. La normalisation de chaque courbe se fait par rapport à la surface du réseau correspondant. Ceci permet de visualiser pour chaque interconnexion, la charge réseau potentiellement acceptable par mm^2 et donc son efficacité surfacique. La figure 3.10(a) présente les courbes des réseaux à plat pour un trafic uniforme. Une fois normalisées, les performances de la Grille et du Tore deviennent meilleures que celles du Multi-bus. Ceci s'explique car le Multi-bus est 1,25 fois plus performant mais est aussi 3,5 fois plus gros que le Tore. C'est le Tore qui est le plus efficace d'un point de vue surface pour les réseaux à plat.

En ce qui concerne les structures hiérarchiques, le réseau MA obtient une efficacité surfacique relativement faible par rapport aux autres solutions hiérarchiques. En effet il rivalise à peine avec les réseaux à plat d'après la figure 3.10(b), ceci majoritairement à cause de la topologie Anneau qui a des performances limitées pour un trafic uniforme et surtout une surface plus importante que les autres topologies hiérarchisées. Le réseau MT dans sa configuration 16 clusters - 2 liens obtient la meilleure efficacité surfacique (figure 3.10(c)) avec 25% de charge réseau normalisée en plus que pour le réseau MX. La figure 3.10(d) fait apparaître que les configurations 16 clusters - 4 liens, 32 clusters - 2 liens et 32 clusters - 4 liens ont des

efficacités surfaciques similaires. Toutefois la configuration 16 clusters 4 liens du réseau MX permet d'avoir une latence avant saturation plus faible et donc des temps de communication sans contention plus faible.

Pour un trafic uniforme, le réseau MT dans la configuration 16 clusters - 2 liens est le réseau qui obtient la meilleure efficacité surfacique. La section suivante réalise cette même analyse de performance pour un trafic localisé.

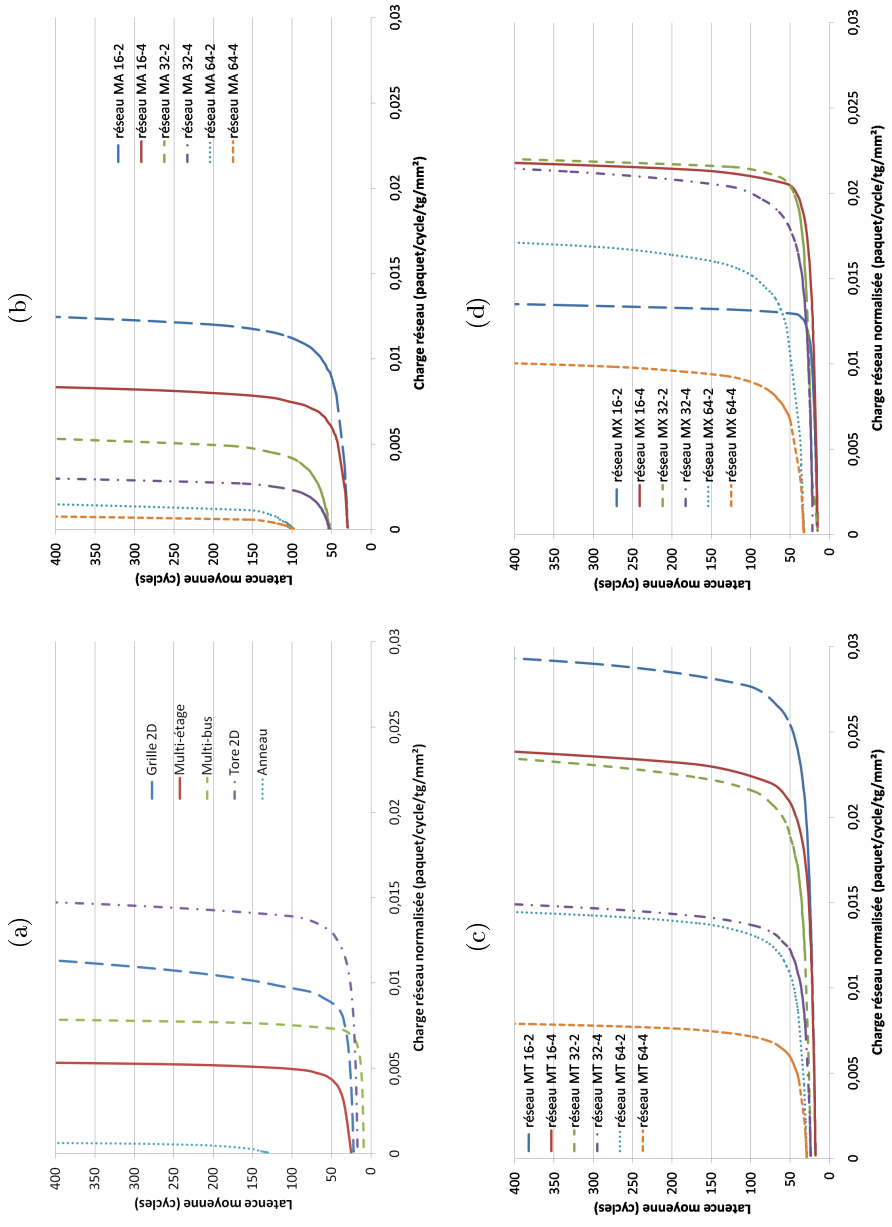


FIG. 3.10 – (a) Latence moyenne des réseaux à plat en fonction de la charge réseau normalisée pour un trafic uniforme. (b), (c),(d) représentent respectivement la latence moyenne des réseaux MA, MT et MX selon leur configuration et en fonction de la charge réseau normalisée pour un trafic uniforme. Les configurations des réseaux se lisent comme suit 16-2 ce qui signifie une découpe en 16 clusters avec 2 réseaux superposés entre les clusters. (MA : Multi-bus + Anneau ; MT : Multi-bus + Tore ; MX : Multi-bus + Anneau amélioré)

3.4.2 Comparaison de l'efficacité surfacique sous trafic localisé

Cette sous section traite de l'efficacité surfacique des réseaux proposés pour un trafic localisé. Les résultats de performance sont les mêmes que dans la sous-section 3.2.2. Ces résultats ont été normalisés, comme expliqué dans la sous section précédente. Cette dernière comparaison va permettre de conclure sur la topologie la plus adaptée pour un système « many-core » devant exécuter des applications dynamiques et impliquant un contrôle de tâche intelligent. Les figures 3.11 regroupent les courbes de latence moyenne en fonction de la charge normalisée des réseaux proposés.

La figure 3.11(a) se focalise sur les réseaux à plat. Elle fait apparaître que l'efficacité silicium est meilleure pour la Grille et le Tore par rapport à un trafic uniforme. Ceci tout simplement parce que le réseau garde la même surface quelque soit le trafic mais le trafic localisé permet de meilleures performances d'où une meilleure efficacité. En modifiant le trafic, on obtient environ 20% de gain en efficacité silicium pour la grille 2D et le Tore 2D. Pour le Multi-bus et le Multi-étage, il n'y a aucune modification de l'efficacité silicium car le trafic n'a pas d'influence sur les performances.

La figure 3.11(b) fait apparaître un gain de 200% entre le trafic uniforme et le trafic localisé pour le réseau MA en configuration 16 clusters - 2 liens. Ce gain provient de l'amélioration des performances lors du passage d'un type de trafic à un autre. L'ordre des courbes montre que l'efficacité décroît avec l'augmentation du nombre de clusters et la multiplication des liens. Ceci se produit principalement car le réseau en Anneau perd en performance lorsque sa taille augmente et que le doublement du réseau supérieur ne permet pas le doublement des performances. Pour le réseau MT c'est également la configuration 16 clusters - 2 liens qui obtient la meilleure efficacité d'après la figure 3.11(c). Pour le réseau MT, on observe un comportement similaire en fonction des configurations que pour le réseau MA. Plus le réseau supérieur connecte un nombre important de clusters et plus l'efficacité diminue pour les mêmes raisons que pour le réseau MA. Le cas du réseau MX est différent et c'est sa configuration 32 clusters - 2 liens qui obtient la meilleure efficacité surfacique d'après la figure 3.11(d). C'est d'ailleurs le réseau qui obtient la meilleure efficacité avec 18% de plus que pour le réseau MT dans sa meilleure configuration, 194% plus efficace que le Tore et 614% plus efficace que le Multi-bus. Il est important de remarquer qu'il y a une grande différence entre l'efficacité et les performances seules. Il apparaît que le réseau MX, qui a des performances moyennes pour un trafic uniforme, est l'un des meilleurs et le plus efficace dans le cas d'un trafic localisé.

Il est important de remarquer que les efficacités surfaciques ont beaucoup augmenté par rapport aux résultats pour un trafic uniforme. Ceci montre que le trafic a un fort impact sur les performances et donc sur l'efficacité du réseau.

D'après nos exigences et au vu des résultats obtenus, le réseau MX dans sa configuration 32 clusters - 2 liens est la topologie, qui a défaut d'être la plus performante estla plus efficace pour un trafic localisé avec une surface minimale. La section

suivante récapitule l'ensemble des observations et des conclusions de ce chapitre.

3.5 Conclusion sur le choix de la topologie

Ce chapitre a utilisé l'environnement de simulation présenté dans le chapitre précédent afin de comparer différentes topologies et de sélectionner la plus adaptée à nos besoins. Nos contraintes se résument en la nécessité de connecter 256 processeurs et 256 mémoires dans le système et d'être capable de supporter un trafic inconnu avant l'exécution. Du fait de cette dernière contrainte, nous avons choisi d'effectuer tous les tests sur des trafics synthétiques. Deux types de trafic ont donc été utilisés l'un uniforme et l'autre localisé. Ces deux types de trafic résultent de deux comportements de répartition des tâches dans le système. Le trafic uniforme est le résultat d'un placement aléatoire dans le système et illustre le cas où le contrôle n'est pas efficace. Le second trafic, localisé, correspond à un placement intelligent des tâches, au plus proche des données consommées, tout en laissant une part d'incertitude sur la réussite du placement. Ce trafic permet donc d'avoir une représentation d'un contrôle intelligent sans faire de suppositions sur ce contrôle.

Les différentes comparaisons qui ont été réalisées nous ont confirmé que le Multi-bus est l'interconnexion la plus performante et que le Tore permet d'obtenir également de bonne performance. Le Multi-étage dans une connexion Butterfly obtient des performances limitées principalement à cause de l'unicité du chemin entre deux points. L'Anneau, pour sa part, est inadéquat du fait que sa latence minimale de traversée est trop grande. On observe également que les structures hiérarchisées ont des performances moyennes dans le cas d'un trafic uniforme mais qu'elles deviennent de plus en plus performantes à mesure que le trafic se localise dans les clusters. Les comparaisons entre les différentes configurations ont permis de confirmer la nécessité de dupliquer le réseau supérieur afin d'obtenir plus de bande passante en limitant les contentions. Enfin, ces courbes de performance ont pu faire apparaître que l'augmentation du nombre de clusters autour d'un routeur permet de faire croître les performances étant donné que cela réduit la taille du réseau supérieur.

Le contexte de l'étude est le monde de l'embarqué, ce qui implique que le choix ne peut être seulement réalisé en terme de performance mais il est au minimum nécessaire d'introduire la notion de surface dans le choix. Pour cela, nous avons mis en place la normalisation de la charge réseau afin de faire apparaître l'efficacité silicium de ces interconnexions. Et donc à partir de la surface et de la courbe de latence moyenne en fonction de la charge réseau pour chaque topologie, une comparaison a été réalisée sur les performances normalisées. Cette étude a mis en évidence que le Multi-bus possède une mauvaise efficacité silicium pour de grandes structures alors que les structures hiérarchisées (association de petit Multi-bus et de réseaux supérieurs) permettent d'obtenir de meilleures efficacités. Il apparaît que le réseau MT en configuration 16 clusters - 2 liens obtient la meilleure efficacité pour un trafic uniforme alors que le réseau MX dans sa configuration 32 clusters - 2 liens possède la meilleure efficacité pour un trafic localisé.

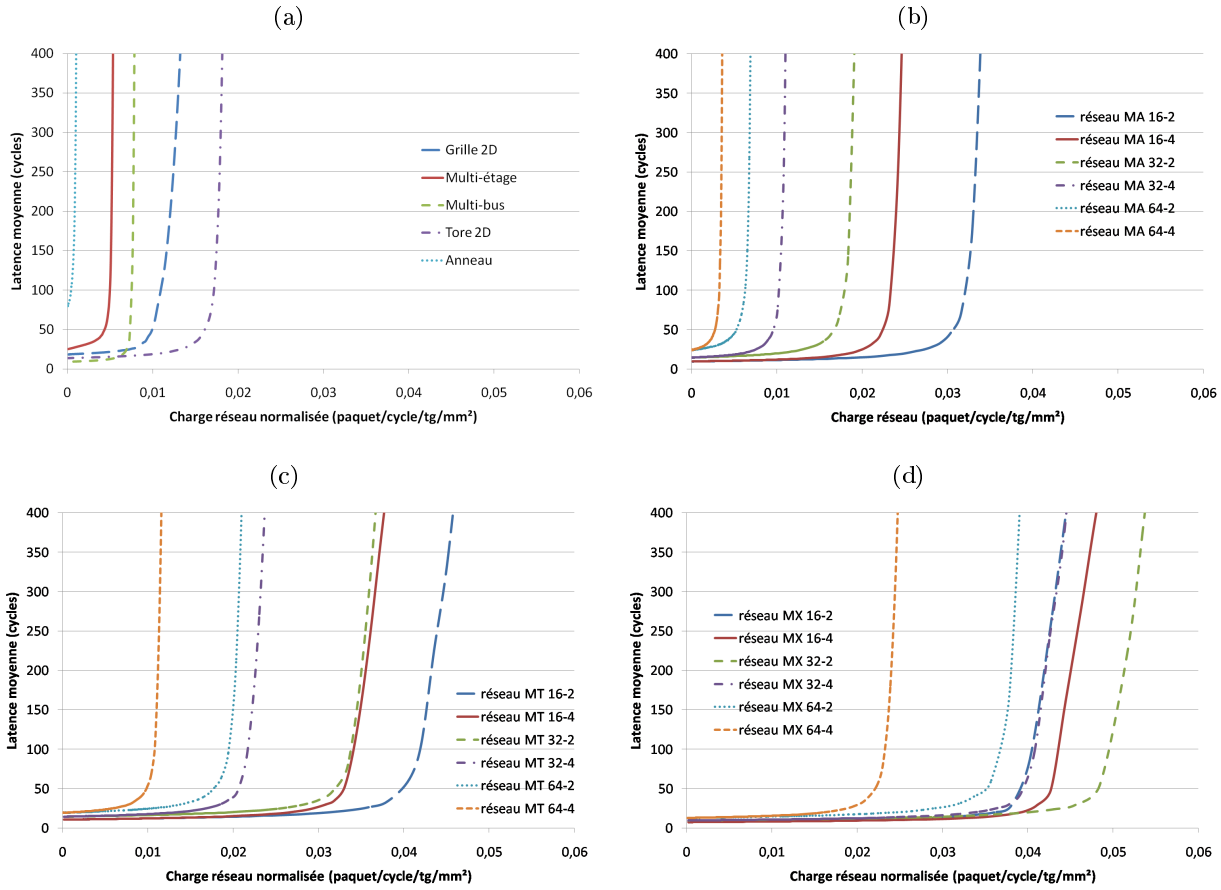


FIG. 3.11 – (a) Latence moyenne des réseaux à plat en fonction de la charge réseau normalisé pour un trafic localisé. (b),(c),(d) représentent respectivement la latence moyenne des réseaux MA, MT et MX selon leur configuration et en fonction de la charge réseau normalisé pour un trafic localisé. Les configurations des réseaux se lisent comme suit 16-2 ce qui signifie une découpe en 16 clusters avec 2 réseaux superposés entre les clusters. (MA : Multi-bus + Anneau ; MT : Multi-bus + Tore ; MX : Multi-bus + Anneau amélioré)

Les résultats ont montré qu'avec un trafic localisé (« contrôle intelligent »), les réseaux obtiennent de bien meilleures performances et une meilleure efficacité surfacique. Nous avons donc choisi de nous placer dans le cas d'un contrôle intelligent. Nous avons donc retenu la topologie réseau MX dans la configuration 32 clusters - 2 liens du fait de son efficacité silicium pour un trafic localisé, de ses bonnes performances pour les deux types de trafic et de sa faible empreinte silicium. Cette topologie sera donc utilisée dans la suite de cette étude.

Il est évident que la répartition des différentes tâches à travers une architecture « many-core » ne peut pas se faire aléatoirement dans le système sans craindre une dégradation sévère des performances. De plus, afin d'utiliser au mieux les performances de notre réseau MX, il est important de choisir un modèle d'exécution qui permet une répartition localisée des tâches.

C'est pourquoi le chapitre suivant propose un modèle d'exécution optimisé pour notre système hiérarchique « many-core ». Pour cela une étude des algorithmes d'ordonnancement existants va être réalisée et une nouvelle approche va être proposée. De cette étude découlera notre proposition de modèle d'exécution.

Chapitre 4

Modèle d'exécution pour structures many-core hiérarchiques embarquées

Sommaire

4.1 Présentation des familles d'algorithmes d'ordonnancement existants dans le domaine du calcul haute performance . . 78
4.1.1 Les algorithmes par liste 79
4.1.2 Les algorithmes par groupe (Clusterisation) 81
4.1.3 Exemple de réalisation pour une architecture many-core et conclusion sur l'existant 83
4.2 Description des algorithmes de faible complexité sélectionnés et de l'environnement de test 84
4.2.1 L'environnement de test 85
4.2.2 Les algorithmes d'ordonnancement par liste existants 86
4.2.3 Les nouveaux algorithmes d'ordonnancement par liste 86
4.2.4 Les nouveaux algorithmes d'ordonnancement par groupe statique 88
4.2.5 Le nouvel algorithme d'ordonnancement dynamique par groupe 89
4.3 Mise en place de métriques et de moyens de mesure 90
4.4 Comparaison entre les différentes propositions d'algorithmes 92
4.4.1 Résultats avec un graphe hautement dépendant 93
4.4.2 Résultats avec un graphe parallèle 96
4.4.3 Synthèse de l'étude sur les algorithmes d'ordonnancement de faible complexité 99
4.5 Proposition d'un modèle d'exécution pour une architecture many-core hiérarchique 99
4.5.1 Gestion des tâches 100
4.5.2 Gestion des données 105
4.5.3 Les critères de placement des groupes de tâches 106
4.6 Vérification des performances 106
4.7 Conclusion 108

Les chapitres précédents ont mis en évidence qu'un système de communication hiérarchisé, dans le cas d'un système contenant plusieurs centaines de processeurs et de mémoires, est la solution qui permet d'obtenir les meilleures performances ainsi que la plus petite empreinte silicium. Nous avons conclu de cette étude que le réseau MX est le réseau sur puce le plus adapté à nos besoins et nous avons donc retenu

ce réseau de communication pour les données dans la suite de ce mémoire. Nous n'avons pour l'instant aucun a priori sur les communications de contrôle.

Une fois l'infrastructure de communication dédiée aux données décidée, la suite de l'étude traite de la gestion des tâches et des données dans le système. Nous avons vu en introduction que cette gestion doit être dynamique afin d'être capable d'optimiser en ligne le placement des tâches. Cette nécessité est due principalement aux temps d'exécution variables des tâches gérées et au dynamisme du graphe de contrôle.

L'architecture SCMP [85], par exemple, possède déjà un modèle d'exécution dynamique capable de gérer ces contraintes. Cependant ce modèle adresse des machines avec un accès uniforme de la mémoire (Uniform Memory Access : UMA) et est donc inutilisable dans l'état pour notre structure qui est une machine avec un accès non uniforme de la mémoire (Non Uniform Memory Access : NUMA) du fait de la hiérarchisation. Le modèle d'exécution de SCMP fait l'hypothèse que toutes les applications peuvent être décrites par un graphe de tâches où celles-ci sont liées entre elles par des dépendances de données ou de contrôle. Nous avons choisi de partir de cette même hypothèse.

Le chapitre 4 va permettre de choisir un modèle d'exécution pour une structure many-core hiérarchique embarquée s'inspirant du modèle d'exécution de l'architecture SCMP. Mais avant de proposer un modèle d'exécution, une présentation de l'état de l'art sur les algorithmes d'ordonnancement provenant du calcul haute performance et applicable à l'embarqué est proposée. Un certain nombre d'algorithmes, répondant à nos besoins de dynamisme dans des architectures massivement parallèles, vont être sélectionnés et d'autres vont être proposés afin de les comparer. Cette étude comparative sur les algorithmes d'ordonnancement a pour but de définir une façon optimale de gérer un grand nombre de tâches sur une structure de communication hiérarchique. Pour cela une description des algorithmes sélectionnés sera effectuée. Afin de les comparer, un environnement a été mis en place et sera présenté. Une fois cet environnement décrit, une évaluation sera réalisée entre toutes les propositions afin de déterminer l'ordonnancement le plus adapté à notre problème. De cette comparaison d'algorithmes, une proposition de modèle d'exécution pour un système many-core hiérarchique sera présentée. Ce modèle d'exécution va être décomposé en gestion de tâches et gestion de données afin de simplifier la compréhension.

4.1 Présentation des familles d'algorithmes d'ordonnancement existants dans le domaine du calcul haute performance

Le problème d'ordonnancement sur un multiprocesseur consiste à répartir de la manière la plus optimale les tâches disponibles sur les processeurs. Ce problème prend en général en considération les problèmes de répartition de charge, de temps réel. C'est un problème connu qui possède de nombreuses solutions dans un contexte

de machine UMA.

Cependant le passage à un ordonnancement multiprocesseur sur des machines NUMA fait appaître d'autres problèmes. Ainsi le temps de communication et donc le placement des tâches sur les processeurs devient une contrainte forte qui était transparente dans le cas de machines UMA. De plus, il devient nécessaire de considérer que le nombre de tâches dépassent le millier pour des centaines de processeurs.

Les systèmes embarqués n'ont que récemment passé le cap de la centaine de processeurs. C'est pourquoi, il n'existe que peu de travaux sur l'ordonnancement de milliers de tâches dans ce domaine. Cependant depuis des dizaines d'années, le monde du calcul haute performance (HPC) est confronté à ce problème. Deux approches principales ressortent de l'état de l'art [86].

La première approche place statiquement les tâches sur les processeurs et les données dans les mémoires. Celle-ci permet d'avoir un fonctionnement prédéterminé qui est optimisé hors-ligne. Cette approche est adaptée à un contexte prédictible où l'on connaît les temps d'exécution de chaque tâche et la séquence de traitement. Lorsque les temps d'exécution des tâches changent dynamiquement, seul le pire temps d'exécution de chaque tâche doit être envisagé pour cette approche. De ce fait, un placement statique des tâches ne répond pas à nos attentes de gestion optimisée.

La deuxième approche est celle envisagée dans cette étude. Elle consiste à répartir à la volée les différentes tâches et les données dans les ressources libres au moment de leur création. A l'aide de [87], qui propose un état de l'art des solutions en ligne, nous avons isolé deux sous techniques d'ordonnancement en ligne applicables au domaine de l'embarqué : par liste ou par groupe qui sont applicable au monde de l'embarqué. Ces deux techniques sont détaillées dans les sous-sections 4.1.1 et 4.1.2.

4.1.1 Les algorithmes par liste

Les algorithmes par liste utilisent une technique similaire à celle employée en ligne pour résoudre le problème de « bin packing » [88] qui correspond à ranger de façon la plus optimale possible des articles dans des boîtes. Ces algorithmes se composent de deux phases.

La première phase consiste à ordonner l'ensemble des tâches selon un critère choisi. Ce critère peut porter sur différents paramètres comme le moment de création de la tâche, le temps d'exécution estimé ou encore la quantité de données consommées.

La deuxième phase est le placement. Cela consiste à prendre les tâches dans l'ordre de la liste et à les assigner aux processeurs libres. Ainsi, nous obtenons deux techniques pour ce genre d'algorithmes : le rangement de la liste et le placement sur les processeurs.

La figure 4.1 ainsi que le pseudo-code 4.1 représentent un fonctionnement d'un algorithme par liste dans lequel l'ordre de la liste est l'ordre d'arrivée des tâches prêtes et le placement est aléatoire sur les processeurs libre.

Listing 4.1 – Pseudo code d'un algorithme par liste. Dans ce cas le tri est réalisé par l'ordre d'arriver et le placement se fait de manièère aléatoire

```
/* pas de tri realise */

/* placement des taches */
/* on determine combien de processeur sont en attente */
for( p=0; p<nb_proc; p++){
  if(proc[p].status == FREE){
    nb_ready_proc++;
  }
}

/* pour chaque tache il y a un tirage aleatoire d'un processeur libre
    */
for( p=0; p<nb_ready_proc; p++){
  task = ready_task_fifo.pop();
  val = rand(0,nb_ready_proc-p);
  /* val represente le nieme processeur libre */
  select_free_proc(val);
  assignation(p, ready_group[0]);
}
```

Ce type d'algorithme est très souvent utilisé pour les systèmes embarqués car il est simple à mettre en place. De plus, la phase de placement est transparente dans le cas d'une machine UMA car l'accès aux mémoires est uniforme quelque soit le processeur. Et donc les performances sont indépendantes du positionnement des processeurs. Les auteurs de [89] compare différents algorithmes par listes et les étudie.

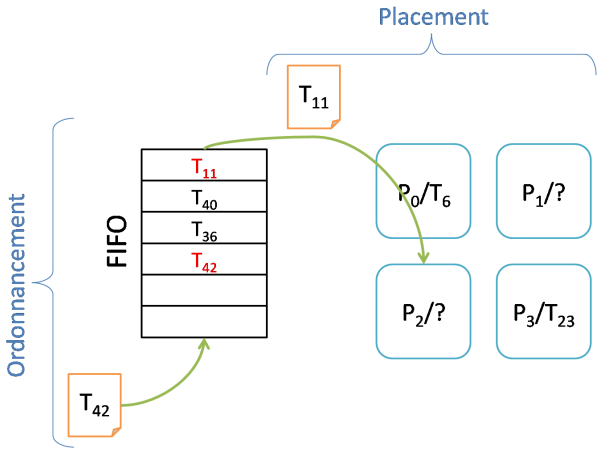


FIG. 4.1 – Représentation du fonctionnement d'un algorithme par liste. L'ordre de la liste est celui d'arrivée des tâches et le placement est aléatoire. La tâche T_{42} est ajoutée à la liste alors que la tâche T_{11} est retirée de la liste afin d'être exécutée sur le processeur P_2.

4.1.2 Les algorithmes par groupe (Clusterisation)

Les algorithmes par groupe sont dits par « clusterisation » dans la littérature. Ils se décomposent en trois phases. La première consiste en la création de groupes de tâches regroupant un certain nombre de tâches selon des critères choisis. Le nombre ainsi que la taille des groupes de tâches qui en résulte dépendent donc de la technique et surtout des critères de clusterisation choisis.

Les figures 4.2 détaillent les étapes de la clusterisation d'un graphe simple selon une technique linéaire. La clusterisation linéaire découpe le graphe en groupe de tâches ne contenant que des tâches successives. Ensuite les différents groupes de tâches sont attribués à chaque processeur. Ceci est, comme dans le cas des algorithmes par liste, la phase de placement qui est décrit par le pseudo code 4.2. Il reste tout de même une dernière phase qui consiste à ordonnancer les tâches des groupes sur le processeur hôte. Le pseudo code 4.3 donne un exemple pour cette phase. Nous retombons dans ce cas sur un problème d'ordonnancement monoprocesseur.

Listing 4.2 – Pseudo code du placement de groupes sur les clusters

```
/* tri des groupes prets */
/* l'ordre se fait sur la plus petite priorite */
for( i=0; i<nb_ready_group-1, i++){
  max_indice = i;
  max_value = ready_group[i].priority;
  for( j= i+1; j<nb_ready_group; j++){
    if(ready_group[j].priority < max_value){
      max_indice = j;
      max_value = ready_group[j].priority;
    }
  }
  /* echange les donnees entre les cases i et max_indice */
  switch(i,max_indice,ready_group);
}

/* placement des groupes */
/* le premier groupe de la liste est le premier place*/
for( p=0; p<nb_cluster; p++){
  if(cluster[p].status < nb_group_max){
    assignation(p, ready_group[0]);
    /* retire la premiere case du tableau */
    update_ready_group();
  }
}
```

Listing 4.3 – Pseudo code de l'ordonnancement des tâches sur les processeurs

```
/* tri des taches pretes */
/* l'ordre se fait sur la plus petite priorite */
for( i=0; i<nb_ready_task-1, i++){
  max_indice = i;
  max_value = ready_task[i].priority;
  for( j= i+1; j<nb_nb_ready_task; j++){
    if(ready_task[j].priority < max_value){
```

```
            max_indice = j;
            max_value = ready_task[j].priority;
          }
        }
        /* echange les donnees entre les cases i et max_indice */
        switch(i,max_indice,ready_task);
      }
      /* placement des groupes */
      /* placement de la premiere tache */
    for( p=0; p<nb_proc_cluster; p++){
      if(proc[p].status == FREE){
          assignation(p, ready_task[0]);
          /* retire la premiere case du tableau */
          update_ready_task();
        }
      }
```

Il faut remarquer que toutes ces phases peuvent être dynamiques ou statiques. Par contre, toutes les combinaisons ne sont pas possibles. Il n'est pas possible d'avoir la phase d'ordonnancement statique si le placement est dynamique. En effet, un ordonnancement statique ne peut être calculé que si le placement est statique également.

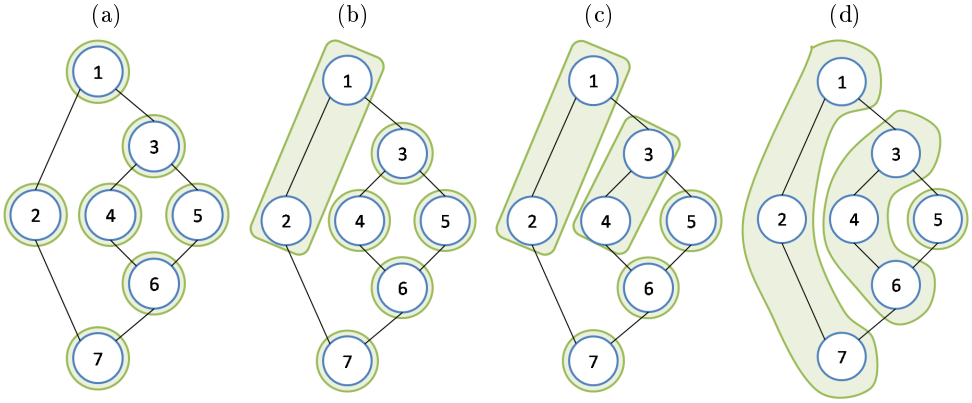


FIG. 4.2 – Etapes de la clusterisation linéaire. La clusterisation linéaire consiste à créer des groupes de tâches successives. La première étape de toute clusterisation consiste à placer chaque tâche dans un groupe comme le montre le figure 4.2(a). Ensuite en fonction des critères choisis tels que le poids des communications entre les tâches, les groupes de tâches sont considérés les uns après les autres afin de déterminer ceux à fusionner. Dans cet exemple, les groupes contenant les tâches 1 et 2 sont fusionnés dans un premier temps (la figure 4.2(b)). Ensuite le groupe contenant la tâche 3 ne peut pas être ajouté au groupe de 1 et 2 car cela ne créerait pas une suite linéaire de tâches. L'étape suivante (figure 4.2(c)) fusionne les groupes contenant les tâches 3 et 4 afin de créer une suite linéaire de tâches. La dernière figure 4.2(d) montre l'état final de la clusterisation.

La clusterisation n'est pas très utilisée dans le monde de l'embarqué. Cependant, dans [90], les auteurs proposent d'utiliser la clusterisation linéaire afin de réduire

la consommation électrique du système multiprocesseur. Du fait du placement des groupes sur un processeur unique, il est possible en fonction des applications et des techniques d'économiser entre 60% et 80% sur la consommation des communications d'un réseau sur puce disposé en Grille. La sous-section suivante présente deux implémentations d'ordonnancement pour des structures many-core.

4.1.3 Exemple de réalisation pour une architecture many-core et conclusion sur l'existant

On peut citer deux travaux représentatifs des solutions de gestion dynamique des tâches. Les premiers travaux sont ceux de Solaris et plus particulièrement d'un projet nommé « NUMA » [91]. Ce projet a été réalisé pour travailler sur un ensemble de machines. Il a pour but d'optimiser le placement des tâches et des données afin d'améliorer la localité des communications. Pour cela, une découpe de l'architecture NUMA est réalisée en *lgroups* (locality groups). Ces *lgroups* sont des rassemblements de processeurs et de mémoires dont les latences de communications sont bornées. Chaque nouveau thread créé est assigné à un *lgroup* afin de limiter les latences de communications. Le thread sera donc exécuté dans son *lgroup*. Le problème de cette solution est que les tâches sont ordonnancées une à une ce qui peut engendrer un surcoût de contrôle si il y a beaucoup de tâches à gérer.

Pour sa part, INTEL propose d'utiliser des RTID [92] (Related Thread ID) dans le cadre de multiprocesseurs pour éviter le problème précédent. Dans cette solution, un RTID est donné à chaque ensemble de threads. Les groupements sont réalisés hors-ligne au gré du programmeur en fonction des dépendances de données. Il est possible également d'associer un RTID à une donnée afin que les threads de cet ensemble soient placés sur les processeurs au plus près de la donnée. Une autre utilisation de ces groupes est de permettre l'équilibrage de la charge entre les processeurs sans se préoccuper du placement en fonction des données. Le choix de l'utilisation des groupes est fait par le programmeur statiquement avant l'utilisation. Cette solution est limitée car le programmeur doit créer l'ensemble des groupes et choisir lui-même statiquement leurs politiques de placement. Choisir le type de placement peut devenir rapidement difficile sur une architecture many-core étant donné qu'il est difficile de prédire l'état du système ce qui peut nuire à ses performances effectives.

Les deux techniques présentées ci-dessus peuvent être classées dans une des deux catégories d'ordonnancement citées précédemment. La solution de Solaris s'apparente à une solution par liste où des groupes de processeurs attendent sur une liste de threads. Alors que la deuxième solution proposée par INTEL est une solution par groupes de threads placés sur des processeurs.

Cependant il est important de préciser que ce problème d'ordonnancement est NP-difficile [93], il n'est pas possible d'obtenir en ligne une solution optimale. Plus le nombre de processeurs augmente et plus la complexité, pour trouver la solution, croit. C'est pourquoi, la répartition de la charge de travail sur les différents processeurs est difficilement homogène [94]. Le « load balancing » consiste à améliorer le

placement des tâches dans le système suite à des imperfections hors-ligne ou en ligne de l'ordonnancement et du placement. Cela consiste soit à placer les tâches sur les processeurs peu chargés, soit à réaliser des migrations de tâches entre processeurs. Le « load blancing » peut donc être réalisé au moment de l'ordonnancement ou du placement de la tâche, voir comme une phase supplémentaire.

Notons enfin que nous n'avons cité que les ordonnancements par liste et par groupe qui s'adaptent au mieux au monde de l'embarqué, mais il existe aussi des algorithmes génétiques [95] et des algorithmes par duplication [96]. Ils permettent d'obtenir de bons résultats mais ils ont de fortes contraintes. Dans le cas des algorithmes génétiques, le problème majeur est la complexité des algorithmes qui ne permet pas son implantation dans un environnement dynamique embarqué. Les algorithmes par duplication, dans leur cas, souffrent d'une surconsommation des ressources étant donné qu'ils allouent plusieurs fois une même tâche afin d'obtenir les données à différents endroits et donc limiter les coûts de communication. Cette nécessité de duplication ne correspond pas aux contraintes de l'embarqué.

La majorité des techniques décrites dans cette section n'ont pas été conçues dans un contexte embarqué, il est donc difficile d'extrapoler leur comportement. De plus le problème de l'ordonnancement reste un problème NP-difficile dans notre cas de figure massivement parallèle. C'est pourquoi dans la suite de ce chapitre, nous allons proposer un certains nombre d'algorithmes susceptibles d'ordonnancer un grand nombre de tâches sur une structure many-core hiérarchique embarquée. ces algorithmes devront être comparés afin de proposer un modèle d'exécution adapté aux structures massivement parallèles hiérarchiques.

4.2 Description des algorithmes de faible complexité sélectionnés et de l'environnement de test

Cette section a pour but de mettre en place une comparaison entre différents algorithmes d'ordonnancement. Ceux-ci sont en partie des algorithmes existants et des propositions. Les algorithmes proposés sont basés sur le fait que dans notre structure hiérarchique les communications les plus coûteuses sont celles qui sont entre les clusters. On peut donc supposer que les performances seront accrues si la fonction de placement choisit de mettre les tâches et leurs données dans le même cluster.

Afin de réaliser ces comparaisons, dans un premier temps, un environnement de simulation est décrit. Puis nous allons détailler les algorithmes comparés. Dans l'ordre, c'est d'abord les algorithmes par liste existants puis ceux par liste proposés qui seront décrits. Enfin, les algorithmes par groupe proposés seront détaillés en fonction des techniques d'ordonnancement et de placement.

4.2.1 L'environnement de test

La description de l'environnement commence par une définition d'un certain nombre de termes. On considère qu'une application peut être décrite par un graphe direct acyclique $G = (T, E)$. T représente un ensemble fini de tâches t. E est un ensemble fini d'arcs e. On considère que $e_{i,j}$ est l'arc reliant la tâche t_i à la tâche t_j. Dans un graphe direct, les arcs ont un sens, ce qui donne que $e_{i,j} \neq e_{j,i}$. Un arc $e_{i,j}$ possède un coût de communication $|e_{i,j}|$ qui corresponds à l'image du volume de données transférés $D_{i,j}$. Le pire temps d'exécution d'une tâche t_i est noté $|t_i|$. Comme le temps d'exécution est variable, le temps réel d'exécution est borné par le pire temps d'exécution. Une tâche t_i est exécutée seulement sur un seul processeur, sans possibilité de préemption de la tâche et/ou de migration. Ceci implique qu'une tâche ne peut pas être distribuée sur plusieurs processeurs afin de réduire son temps d'exécution. Les tâches sont donc dites non malléables.

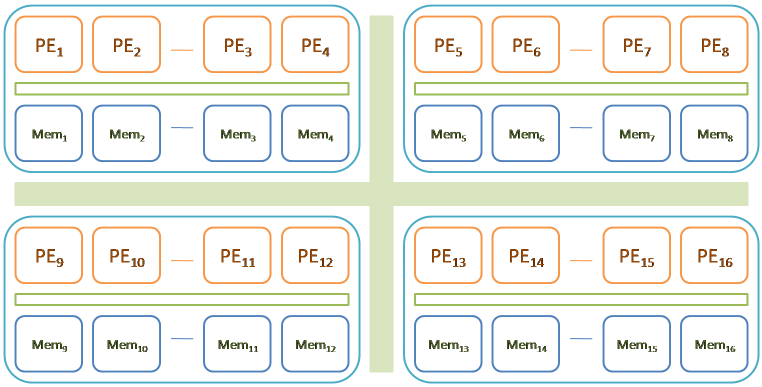


FIG. 4.3 – Représentation de l'environnement synthétique de tests. La topologie de l'interconnexion est pleinement connectée et seules les distances entre source et destination sont prises en compte pour les calculs de pénalité.

L'architecture de comparaison est composée d'un ensemble de processeurs P distribués à travers un ensemble de clusters C. La figure 4.3 montre l'architecture synthétique pour le test des algorithmes. Pour simplifier l'étude des algorithmes, on considère qu'un processeur p_i n'écrit que dans une mémoire qui lui est associée dans le même cluster physique. La taille de cette mémoire est illimitée pour les besoins des tests. Les communications dans le système seront donc vues comme des communications entre processeurs. Ces communications sont concurrentes dans cet environnement, ce qui signifie que chaque processeur p_i peut communiquer avec n'importe quel processeur p_j sans contentions entre les messages. $\forall p_i \in P$, il ne peut avoir qu'une seule tâche assignée à la fois à p_i. Une application A est donc exécutée sur P' où $P' \subset P$. Pour rappel, le système est une machine NUMA, donc toutes les communications n'ont pas les mêmes coûts. On prend comme hypothèse que les communications à l'intérieur du cluster sont moins coûteuses que celle à l'extérieur. On considère que $com_{i,j}$ représente la communication entre p_i et p_j et

$|com_{i,j}|$ son coût. Une description des différents algorithmes d'ordonnancement existants et proposés suit dans les paragraphes ci-après. Tous les nouveaux algorithmes d'ordonnancement tentent de réduire le nombre de communications entre clusters.

4.2.2 Les algorithmes d'ordonnancement par liste existants

Cinq algorithmes communément utilisés sont détaillés dans cette sous-section. Tous ces algorithmes n'ont aucune dépendance avec l'architecture mais uniquement avec les caractéristiques des tâches. Ils vont principalement servir de point de comparaison pour les algorithmes proposés.

L'algorithme *First-Come-First-Serve* (FCFS) ordonnance les tâches sans réaliser de réorganisation de liste de tâches en attente par rapport à leur ordre d'arrivée dans cette liste. Ainsi, la première tâche arrivée est la première attribuée à un processeur. Le placement se fait de la même manière, le premier processeur libre reçoit la première tâche de la liste.

L'algorithme *Randomized* (Rand) ordonnance et place les tâches de manière pseudo-aléatoire. Lorsqu'un processeur est libre, le contrôle choisit une tâche aléatoirement parmi la liste de tâches prêtes à être exécutées.

L'algorithme *Priority* se base sur des priorités statiques établies hors-ligne. $\forall t_i \in T, p(t_i)$ représente la priorité de t_i. Celle-ci est définie par l'utilisateur. La liste de tâches est réorganisée suivant la priorité de chaque tâche, de la plus importante à la moins importante. Ainsi, quand un processeur est libre d'exécuter une tâche, c'est celle de plus haute priorité qui est choisie.

Les algorithmes *Longest-Task-First* (LTF) et *Shortest-Task-First* (STF) prennent en compte le pire temps d'exécution afin d'organiser la liste. L'algorithme LTF privilégie les tâches dont le pire temps d'exécution est important. Alors que l'algorithme STF réalise le contraire en plaçant en haut de la liste les plus courtes tâches. Le placement dans les deux cas, prend le premier processeur libre pour exécuter la tâche en haut de la liste.

Les algorithmes par liste provenant de la littérature ont été décrits dans cette sous-section. Le suivant traite des algorithmes par liste proposées.

4.2.3 Les nouveaux algorithmes d'ordonnancement par liste

Nos nouveaux algorithmes proposent d'augmenter les performances en réduisant les communications entre les clusters et donc prennent en compte la structure de l'architecture. Ce paragraphe détaille 3 nouveaux algorithmes par liste nommés : *Most-Close-Affinity-Choice* (MCAC), *Most-Global-Affinity-Choice* (MGAC) and *Less-Communication-Cost* (LCC). Pour ces trois solutions proposées, les fonctions d'ordonnancement et de placement sont couplées en une seule. Ils prennent en compte les tâches précédentes de la tâche qui doit être ordonnancée. On obtient donc un tri de tâches différent en fonction du processeur libre étant donné que le tri dépend de la position du processeur dans l'architecture.

L'algorithme MCAC tente de minimiser le volume de données des communications entre les clusters. Pour cela, il place de préférence les tâches avec les plus gros volumes de données sur le même cluster. En pratique, si un processeur p_j est libre, deux coefficients sont calculés pour chaque tâche t_i. Le premier $w(t_i, p_j)$ représente la somme du volume de données qui resterait dans le cluster. Alors que le second $d(t_i, p_j)$ cumule les distances entre la tâche t_i et ces précédences si elle est exécuté sur p_j. On considère p_j comme un processeur libre de l'ensemble P se trouvant dans le cluster c_l,

$$w(t_i, p_j) = \sum_{t_k}^{previous\_task(t_i)} a(t_k, t_i)$$

avec

$$a(t_k, t_i) = \begin{cases} D_{k,i} & \text{si } t_k \text{ a été exécutée dans } c_l \\ 0 & \text{si } t_k \text{ n'a pas été exécutée dans } c_l \end{cases}$$

et

$$d(t_i, p_j) = \sum_{t_k}^{previous\_task(t_i)} distance(c_l, c_m)$$

avec c_m le cluster où t_k a été exécutée. La fonction $distance(c_l, c_m$ dépend du réseau est évalue le nombre de sauts entre routeur à traverser. Cette fonction vaut 1 dans le cas où c_l et c_m sont identique. Dans un premier temps l'algorithme choisit la tâche prête avec le plus gros coefficient $w(t_i, p_j)$ pour assigner t_i sur p_j. Si le coefficient est nul pour toutes les tâches et donc aucune tâche n'a de précédences dans le cluster, l'algorithme choisit la tâche avec le plus petit coefficient $d(t_i, p_j)$ et donc celle qui est la plus proche de ses précédentes.

L'algorithme MGAC propose d'utiliser une combinaison des deux variables précédents $D_{k,i}$ (le volume de données) et $distance(c_l, c_m$ (la distance du transfert nommée $W(t_i, p_j)$ afin de sélectionner la bonne tâche. Ce nouveau coefficient permet de choisir la tâche dont les données sont proches et de volumes élevés. Pour chaque processeur, cet algorithme recherche la tâche qui a le plus gros coeficient $W(t_i, p_j)$. Si on considère p_j un processeur libre de l'ensemble P se trouvant dans c_l, alors

$$W(t_i, p_j) = \sum_{t_k}^{previous\_task(t_i)} \frac{D_{k,i}}{distance(c_l, c_m)}$$

avec c_m le cluster dans lequel t_k a été exécutée.

L'algorithme LCC se base sur une fonction d'estimation du coût des communications afin de réduire le total de ces coûts. Cette fonction utilise une représentation de l'interconnexion pour prédire les coûts de communications. Dans nos tests cette fonction est la même que celle utilisée par le simulateur. Les contentions ne sont pas prises en compte. Cet algorithme choisit toujours un processeur pour une tâche qui minimise les coûts de communication.

Nous venons de décrire les algorithmes par liste proposés. Ceux-ci ont été créés afin de s'adapter au mieux à notre architecture hiérarchique. Le paragraphe suivant traite des algorithmes par groupe statique.

4.2.4 Les nouveaux algorithmes d'ordonnancement par groupe statique

Les trois algorithmes suivants sont basés sur la création de groupes de tâches avant l'exécution. Tous ces algorithmes démarrent par la même première étape de clusterisation. Dans un premier temps le graphe d'application est découpé à l'aide de l'algorithme de clusterisation linéaire afin de déterminer un certain nombre de groupes de tâches. Comme nous avons vu avant, la clusterisation linéaire utilise un critère afin de déterminer, lors d'une divergence, quelle tâche doit être choisie. Nous avons choisi d'utiliser $|e_{i,j}|$ qui est l'image du volume de données transférés entre les tâches comme critère. Ainsi, lorsque la clusterisation est réalisée, c'est toujours la tâche avec laquelle il y a eu le plus gros volume de données transférées qui est choisi, regroupant ainsi au sein du même groupe les tâches communiquant beaucoup. Ce choix permet de garder les plus gros transferts à l'intérieur d'un groupe afin de limiter les transferts coûteux à travers le système.

L'algorithme *Static-Clustering-Static-Mapping* (SCSM) propose de fusionner les groupes de tâches ensemble afin d'obtenir un nombre de groupes égal au nombre de clusters physiques dans le système. Cette fusion est réalisée hors ligne et ne prend pas en compte l'architecture du système. On a donc avant l'exécution un groupe de tâches assigné à un cluster matériel ce qui change de l'utilisation habituelle des groupes de tâches. En effet traditionnellement les groupes de tâches sont assignés à un processeur. On se retrouve donc dans chaque cluster avec un problème d'ordonnancement multiprocesseur sur machine UMA. D'après [89], l'algorithme LTF obtient de bons résultats et sera choisi pour réaliser l'ordonnancement dans un cluster. La limite de cet algorithme se trouve au niveau de l'allocation statique et de la fusion des groupes de tâches étant donné notre contexte dynamique.

Pour remédier au problème du précédent algorithme, l'algorithme *Static-Clustering-Static-Mapping-with-Migration* (SCSMM) propose d'ajouter au précédent, une solution d'équilibrage de charge qui va rattraper en partie les erreurs de choix statique. Le load balancing se traduit par une fonction qui s'exécute en parallèle de l'ordonnancement habituel. Lorsqu'un cluster physique est saturé de tâches en attente (arbitrairement plus de 3 fois le nombre de processeurs), la fonction de load balancing scrute la liste de tâches prêtes et sélectionne celle dont le coût total de communication sera le plus élevé. La fonction calcule ensuite le coût hypothétique si la tâche sélectionnée était placée sur les autres clusters non saturés. La fonction de load balancing sélectionne alors le cluster qui produit le plus faible coût de communication et fait migrer la tâche. Toute migration se fait avant que la tâche ait démarré pour limiter le coût de la migration. La migration n'est pas la seule solution pour améliorer les performances de SCSM.

L'algorithme *Static-Clustering-Dynamic-Mapping* (SCDM) ne réalise pas le placement statiquement comme dans les deux précédents algorithmes, mais au moment de l'exécution du groupe de tâches. Initialement, dans chaque cluster, l'algorithme alloue l groupes de tâches où l correspond au nombre de processeurs, dans un cluster. Durant l'exécution, de nouveaux groupes sont fusionnés aux groupes de tâches déjà

assignés au cluster physique lorsque le nombre de tâches prêtes dans le cluster est inférieur à deux fois le nombre de processeurs. Afin de trouver un groupe de tâches qui limite les coûts de communications, les groupes sont choisis en appliquant les critères de sélection de l'algorithme MCAC sur la première tâche de chaque groupe. Le choix se fait uniquement sur la première tâche car une fois dans le groupe de tâches, les coûts sont déjà minimisés du fait de la construction du groupe. L'ordonnancement dans le cluster se fait toujours à l'aide de l'algorithme LTF. L'ajout du dynamisme dans le placement des groupes de tâches remplace la fonction de load balancing.

Cette sous-section a décrit les algorithmes par groupe statique proposés pour coller avec une architecture hiérarchique. La sous-section suivante traite d'un algorithme par groupe où les trois étapes du clustering (groupement, placement et ordonnancement local) sont dynamiques. Celui-ci est le dernier algorithme proposé.

4.2.5 Le nouvel algorithme d'ordonnancement dynamique par groupe

Le dernier algorithme *Dynamic-Clustering-with-Migration* (DCM) consiste, à la différence des autres algorithmes par groupe proposés, à créer dynamiquement les groupes au moment de l'ordonnancement. Cette construction permet d'ajouter dynamiquement des tâches dans les groupes de tâches assignés aux processeurs. Lorsqu'un de ces groupes n'a plus assez de tâches prêtes à être exécutées, une fonction en ligne complète les groupes jusqu'à arriver à un total de tâches ajoutées égal à trois fois le nombre de processeurs pour chaque cluster. Cette limite est arbitrairement choisie mais devra être validée et ajustée si nécessaire.

La découpe est réalisée seulement selon les tâches qui suivantes de celles déjà présentes dans un cluster. La fonction détecte deux types de tâches. Sur les séquences linéaires de tâches, la fonction intègre automatiquement la tâche dans le groupe (tâches 6-7 de la figure 4.4). Sur une tâche divergente, la fonction répartit les tâches divergentes en groupes avec un parallélisme maximum égale au nombre de processeurs (tâches 2-3 de la figure 4.4). La découpe du graphe se fait donc de façon itérative en suivant l'ordre du graphe. Cette découpe est non linéaire, elle crée des groupes dont le parallélisme est égal au nombre de processeurs dans le cluster au maximum.

Cette solution n'est pas optimale étant donné qu'elle n'utilise que des optimisations locales. Pour permettre de meilleurs résultats, une fonction de load balancing est implémentée comme pour l'algorithme *Static-Clustering-Static-Mapping-with-Migration*. On rappelle que cette fonction de load blancing ne déplace que tâche par tâche. Tous les algorithmes étant présentés, il est nécessaire de choisir les métriques de comparaison ainsi que les moyens de les mesurer. Ceux-ci sont présentés dans la sous section suivante.

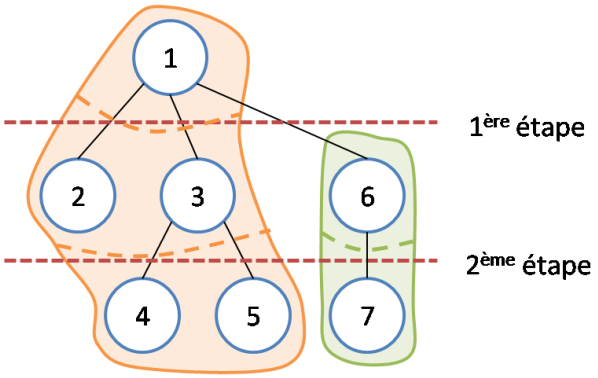


FIG. 4.4 – Représentation de la découpe non-linéaire et dynamique en cluster de l'algorithme Dynamic-Clustering-Mapping. On considère seulement 2 processeurs par cluster pour l'exemple. La figure fait apparaître les étapes du parcours du graphe.

4.3 Mise en place de métriques et de moyens de mesure

Cette sous-section présente les métriques utilisées pour comparer les algorithmes ainsi que le simulateur qui va permettre de mesurer ces métriques. Nous avons choisi de prendre trois métriques pour comparer tous ces algorithmes : l'accélération, le taux d'utilisation des processeurs et le nombre de communications en dehors du cluster.

La première métrique est la plus courante, nous allons mesurer l'accélération par rapport à un algorithme de référence, l'algorithme *First-Come-First-Serve* (FCFS). On considère $\lambda(H)$ comme l'accélération fournie par l'algorithme H par rapport à l'algorithme FCFS. Cette accélération se calcule comme suit : si $H(G)$ est le temps d'exécution du graphe G en utilisant l'algorithme d'ordonnancement H alors $\lambda(H)$ est égale à $\frac{FCFS(G)}{H(G)}$. Cette première métrique va permettre de terminer le gain de temps qu'apporte l'algorithme par rapport à une implémentation naïve.

La seconde métrique est le taux d'utilisation moyen des processeurs. Plus particulièrement, nous nous intéressons à la répartition du taux d'utilisation moyen entre le contrôle, les communications et le temps d'exécution des tâches. Cette seconde métrique a pour but de faire apparaître le ratio du temps de contrôle et de communication par rapport au temps consacré aux traitements de la tâche.

La troisième métrique est le décompte du nombre de communications entre clusters. Cette métrique va être couplée avec la distance moyenne à laquelle se déplace ces messages entre clusters. Cette métrique reflète le taux d'utilisation du réseau entre les clusters.

Afin de mesurer toutes ces métriques, un simulateur a été mis en place. Il permet de mesurer le temps d'exécution d'une application décrite à l'aide d'un graphe de dépendances entre tâches. Chaque tâche est caractérisée par son temps maximum d'exécution. Cette application doit être renseignée avec le temps d'exécution maximal et la priorité de chaque tâche ainsi que les poids des différents arcs entre les

tâches.

Ce simulateur est constitué d'un calculateur d'ordonnancement ainsi que de compteurs de temps pour représenter chaque processeur. Les compteurs permettent de calculer le temps écoulé sur chaque processeur. Le temps total d'exécution est le compteur qui affiche la valeur de temps la plus grande. Afin de faire avancer le temps sur chaque compteur, il y a deux types d'ajouts, soit un ajout de temps relatif à l'exécution fictive d'une tâche, soit un ajout de temps relatif à un coût de fonctionnement du système. On considère les coûts dû au contrôle et aux communications.

Le calculateur d'ordonnancement est une boucle sur l'ensemble des tâches du graphe qui à chaque tour traite les différentes étapes de l'ordonnancement choisi. Initialement la première tâche est exécutée sur le premier processeur, puis on rentre dans la boucle d'ordonnancement qui se termine une fois que toutes les tâches sont traitées.

Cette boucle commence par déterminer dans un premier temps le processeur qui a le plus petit compteur. Ce processeur est considéré comme le processeur libre qui a besoin d'une tâche. Puis, elle continue par mettre à jour les tâches prêtes en fonction de la tâche qui vient d'être finie sur le processeur libre.

En fonction des types d'ordonnancement, des fonctions de load balancing, de clusterisation dynamique et de placement de groupes de tâches sont réalisées. Toutes ces fonctions n'ont pas de coût de contrôle et de communications car on suppose qu'elles sont réalisées en parallèle de l'exécution des tâches.

A ce stade, la fonction d'ordonnancement à proprement dite est réalisée afin de déterminer la tâche qui sera exécutée sur le processeur considéré. La durée d'exécution de la tâche est modifiée aléatoirement en fonction d'une fourchette de temps. Ce temps est ajouté au compteur du processeur. Enfin les coûts de communication nécessaire à la tâche et le coût de contrôle sont additionnés au compteur du processeur.

Les fonctions de coûts utilisées dépendent de l'implémentation mais aussi de l'unité de temps considérée par les compteurs. La fonction de coût de communication est celle qui représente le réseau, elle va donc dépendre de l'implémentation du réseau. Le coût d'une communication est calculé comme suit $\forall com_{i,j}$, $p_i \in c_k$ *and* $p_j \in c_l$:

$$|com_{i,j}| = \begin{cases} const_m & \text{si } c_k = c_l \\ const_r * distance(c_k, c_l) * D_{i,j} & \text{si } c_k \neq c_l \end{cases}$$

où $const_m$ et $const_r$ sont des paramètres dépendant de l'implémentation du réseau. Le coût de contrôle est principalement dû au contrôleur d'interruptions entre le processeur et le contrôle centralisé ainsi qu'à l'ordonnancement. Il dépend donc du nombre de tâches gérées à l'instant t par le contrôle. Donc si cp_i est le coût d'ordonnancement, $cp_i = m * nb\_ready\_tasks$ où m dépend de l'implémentation et nb_ready_tasks le nombre de tâches ordonancées à ce moment précis. Nous avons choisi arbitrairement m égal à 2 cycles, $const_m$ égal à 1 cycle et $const_r$ égale à 10 cycles afin d'avoir une proportion raisonnable entre les constants.

Cette sous-section a permis de définir les métriques ainsi qu'un simulateur afin de les mesurer. La sous-section suivante va mettre en place des comparaisons des différentes métriques.

4.4 Comparaison entre les différentes propositions d'algorithmes

Cette sous-section compare les différentes propositions selon les métriques proposées : l'accélération par rapport à l'algorithme FCFS, la répartition du temps (contrôle, communication et calcul) et le nombre d'accès distants. La structure des tests suivants est fixée à 128 processeurs avec une répartition en cluster variable. Cette limitation de processeurs intervient afin de limiter les temps de simulation et de limiter la taille des applications à utiliser. En effet il est nécessaire d'utiliser des applications qui contiennent un grand nombre de tâches afin d'utiliser toutes les ressources de calcul de l'architecture. Ces graphes d'applications sont difficilement réalisables à la main et nécessitent donc un support logiciel. Nous avons utilisé, pour ces tests, le logiciel « Task Graph For Free » (TGFF) [97]. Ce logiciel permet, à l'aide d'un fichier de paramètres, de créer un graphe pseudo aléatoire qui respecte les paramètres donnés. Le listing 4.4 montre les paramètres utilisés pour générer un graphe parallèle.

Listing 4.4 – Fichier de paramétres pour TGFF afin d'obtenir un graph parallèle

```
#general parameters
tg_cnt  1      # nombre de graphes
task_cnt  10000 100 # nombre de taches et variation autorisee
task_degree 64  64  # degres maximum de convergences et divergences
task_type_cnt 1   # nombre de types de tache

#graph serie_parallel parameters
gen_series_parallel true  # mode parallele
series_must_rejoin  true  # oblige le graphe a avoir une seule tache
    de fin
series_len  10  3 # longueur des branches paralleles et variation
series_wid  256 20  # nombre de branches en parallele variation
series_local_xover 100    # connexions locales entre les branches
    paralleles
series_global_xover 2   # connexions globales entre les branches
    paralleles

#ouput options
tg_write      # sortie en format TGFF
eps_write     # sortie en format EPS
vcg_write     # sortie en format VCG
```

TGFF ne propose pas tout ce dont nous avions besoin. En effet, il n'alloue pas aléatoirement un temps d'exécution maximum pour chaque tâche, mais en revanche il alloue un poids à chaque arc. Des temps d'exécution maximum sont donc ajoutés à chaque tâche.

Ces temps sont des tirages aléatoires à l'intérieur d'une fourchette. Cette fourchette doit être calibrée en fonction des constantes du simulateur. En effet, afin d'avoir un ratio réaliste entre les communications et le temps de traitement de chaque tâche, quelques tests ont été réalisés. Ceux-ci ont été réalisés afin d'obtenir autour de 15% de communications par rapport au temps de traitement dans le cas de l'algorithme FCFS considéré comme référence. Ces tests ont permis également de choisir les constantes des fonctions de coûts du simulateur.

Deux types de graphe ont été créés pour les comparaisons d'algorithmes. Le premier type est un graphe parallèle dans lequel de nombreuses branches n'ont pas de dépendance entre elles. Ce type de graphe correspond à des applications qui ont peu de dépendances de données entre les tâches. Le deuxième type est un graphe extrêmement dépendant où les tâches ont des dépendances de données avec d'autres tâches à travers tout le graphe. Les graphes étant trop gros pour être affichés (environ 120 000 tâches) des versions miniatures sont visibles sur les figures 4.5. Etant donné que les temps de simulation sont longs à cause de la taille des graphes, les résultats sont les moyennes de 10 expérimentations pour des durées de tâches et des poids d'arc aléatoires et cela pour chaque graphe d'application.

Les résultats vont être présentés suivant le type de graphe et suivants les métriques. Les résultats pour un graphe hautement dépendant sont présentés en premier.

4.4.1 Résultats avec un graphe hautement dépendant

Ce paragraphe regroupe les résultats pour un graphe hautement dépendant. Dans ce cas, toutes les tâches ont de nombreuses dépendances de données avec d'autres tâches réparties dans tout le graphe. Les figures 4.6 présentent ces résultats.

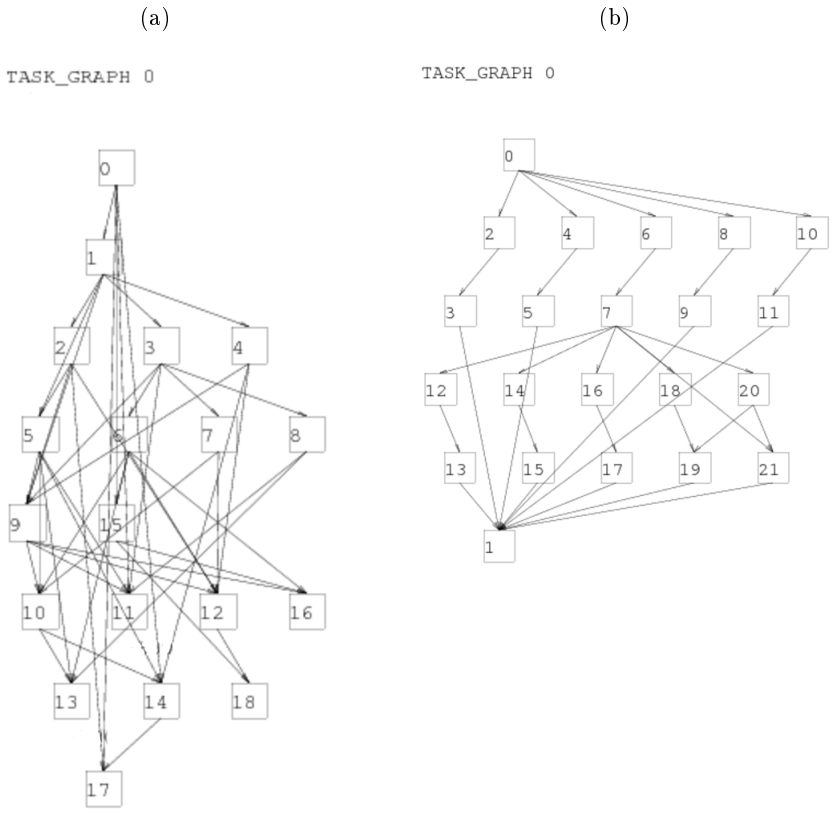


FIG. 4.5 – (a) présente un exemple de graphe hautement dépendant contenant 20 tâches. La seule contrainte de création est le nombre de tâches précédentes et suivantes maximum. (b) propose un exemple de graphe parallèle contenant 20 tâches, dans ce cas les mêmes contraintes que pour le premier graphe sont gardées auxquelles on rajoute les contraintes liées aux branches parallèles.

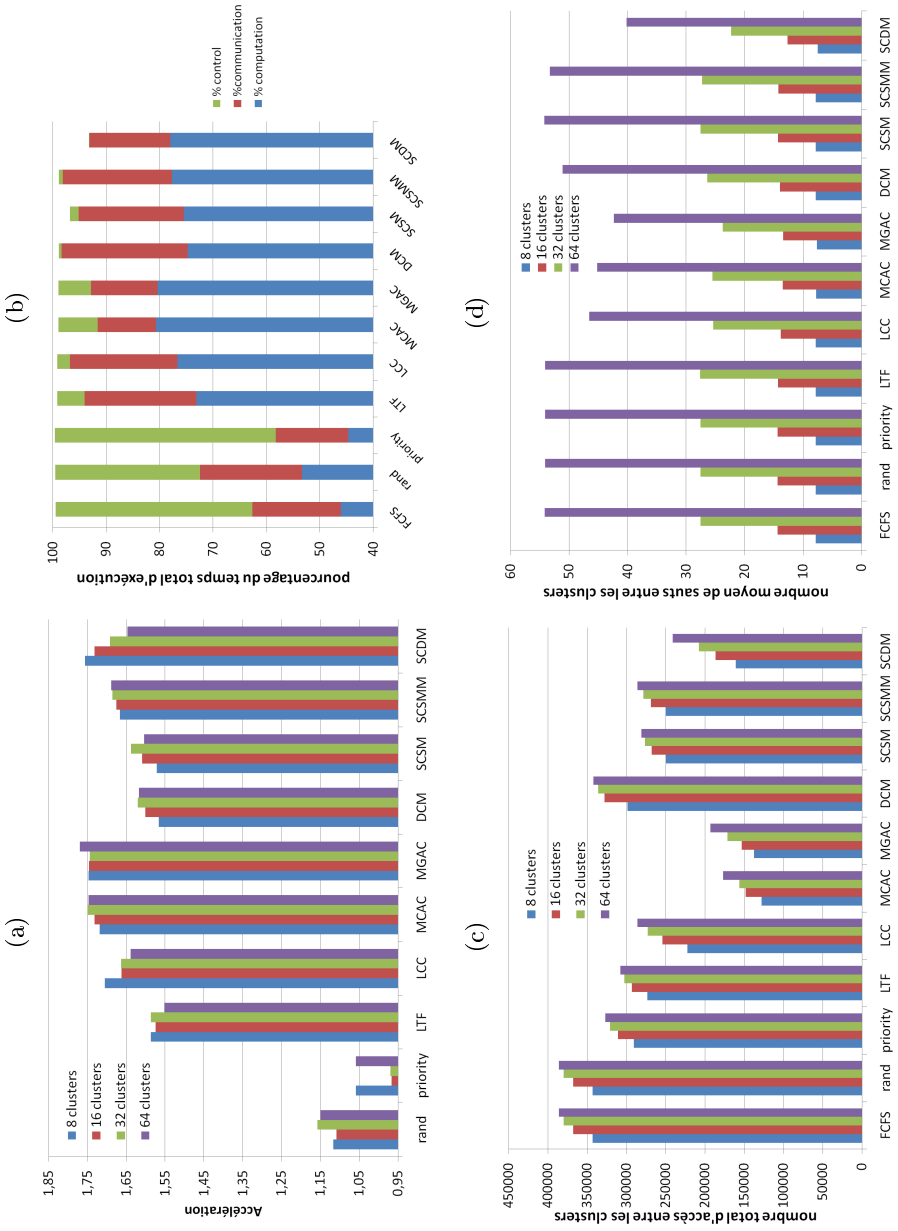


FIG. 4.6 – Pour un graphe hautement dépendant, (a) présente l'accélération de chaque algorithme par rapport à l'algorithme FCFS suivant différentes découpes en cluster. (b) présente la répartition des différents temps moyens d'exécution entre les phases de contrôle, de communication et de calcul dans le cas de 32 clusters. (c) et (d) montrent respectivement le nombre d'accès total à l'extérieur de tous les clusters et la distance moyenne parcourue en fonction du nombre de clusters.

Il apparaît que les algorithmes « Randomized » et « Priority » obtiennent des résultats similaires à l'algorithme FCFS, que ce soit d'un point de vue temps d'exécution, de nombre de communications entre cluster ou encore de distance moyenne parcourue. Pour sa part, l'algorithme LTF obtient de bonnes performances avec une accélération de 1.55 en moyenne malgré le fait que cet algorithme ne tienne pas compte de l'architecture. Cela est dû, comme la figure 4.6(b) le fait apparaître, à la diminution de la part consacrée au coût de contrôle.

Les algorithmes DCM et SCSM obtiennent des résultats similaires au LTF même s'ils tentent de prendre en compte la structure de l'interconnexion. La figure 4.6(b) montre que le coût de contrôle pour ces algorithmes sont faibles mais les coûts de communication sont plus élevés. Le nombre de communications entre clusters diminue de 28% alors que leurs moyennes des distances parcourues par un message reste les mêmes. L'ajout de la migration avec SCSMM permet d'augmenter les performances par rapport à SCSM et surtout sans impact sur les autres métriques. Cela s'explique par le fait que le placement de SCSM est statique et non optimal et que la migration permet d'améliorer les métriques mesurées. En effet en migrant des tâches en attente d'un cluster à un autre, cela permet leur exécution et ainsi diminue leur temps d'attente.

L'algorithme LCC, qui prend en compte le coût de communication uniquement, améliore les performances mais reste limité car il ne cherche que des optimisations locales et pas globales. En revanche, les algorithmes MCAC et MGAC proposent des optimisations plus globales et donc permettent d'obtenir les meilleures performances. De plus ils diminuent de 57% le nombre d'accès distant tout en réduisant les distances parcourues. Ces deux algorithmes sont parfaitement adaptés à cette situation. En effet, ils traitent l'application tâche par tâche en les plaçant sur le processeur le plus adapté pour fournir le plus petit coût de communication.

Le dernier algorithme, SCDM, obtient de bonnes performances pour un petit nombre de clusters mais ces performances diminuent avec l'augmentation du nombre de clusters. Cet algorithme est plus sensible à l'augmentation du nombre de clusters du fait du placement dynamique des groupes de tâches qui n'est pas adapté à ce type de graphe.

Cet algorithme SCDM comme l'ensemble des algorithmes par groupe réduit grandement le coût de contrôle car dans chaque cluster, seules les tâches courantes de chaque groupe est gérées dans le cluster et non pas toutes les tâches prêtes de toute l'architecture. Ainsi, le contrôle est parallélisé sur l'ensemble des clusters. La figure 4.6(d) montre également que l'algorithme SCDM, en particulier, obtient les plus faibles distances parcourues par les communications entre clusters en moyenne. Le prochain paragraphe présente les mêmes métriques pour un graphe parallèle.

4.4.2 Résultats avec un graphe parallèle

Ce paragraphe résume les résultats des métriques pour un graphe parallèle regroupés sur les figures 4.7. Le graphe parallèle est principalement constitué de branches de tâches séquentielles. La figure 4.7(a) détaille l'accélération de chaque

algorithme par rapport à l'algorithme FCFS. La figure 4.7(b) montre la répartition de l'occupation de l'architecture dans le cas d'un découpage en 32 clusters. Il est important de noter que, contrairement au graphe précédent, celui-ci ne permet pas de fournir une charge supérieure à 90%. Cela s'explique par le fait que le graphe est plus simple et donc le contrôle et les communications prennent moins de temps. Nous n'avons pas fait grossir le graphe car au delà, les temps de simulation étaient trop longs.

Contrairement au cas de graphe précédant, ce sont les algorithmes par groupe statique (SCxx) qui obtiennent les meilleurs résultats. Ceci est du au fait que les groupes de tâches vont se confondre aux branches du graphe. Ceci va permettre que lorsqu'un groupe est placé sur un cluster, il y a une minimisation du coût de communication. Nous arrivons à plus de 1.25 d'accélération pour l'algorithme SCDM par rapport à l'algorithme FCFS.

Nous observons que sur l'ensemble des figures que l'algorithme DCM obtient les plus faibles résultats. Ceci est dû à la manière de créer les groupes. En effet, cet algorithme cherche à créer des groupes au moment des divergences, or il y a peu de divergence dans ce genre de graphe et donc il n'arrive pas à placer correctement les tâches. Ceci implique une grosse pénalité de communication comme la figure 4.7(b) le montre. Il apparaît donc que notre proposition ne s'adapte pas à ce type de graphes.

L'algorithme LTF obtient de bons résultats avec une accélération équivalente à l'algorithme LCC qui pour sa part prend en compte la structure. Il est important de remarquer sur la figure 4.7(c) que le nombre d'accès pour les algorithmes par liste et par groupe statique proposés est divisé par plus de 10 par rapport à l'algorithme FCFS. Cette diminution entraîne pour tous ces algorithmes une diminution des temps de communication et un gain de performances. D'ailleurs, l'algorithme LCC permet bien de diminuer le nombre de communications entre cluster mais d'après la figure 4.7(d), il ne permet pas de diminuer la distance moyenne de ces communications.

L'algorithme SCDM offre les meilleurs résultats sur toutes les métriques choisies pour ce type de graphes. Nous voyons apparaître que l'allocation dynamique des groupes de tâches (SCDM) permet un gain de quasiment 50% sur la distance moyenne parcourue par les messages hors du cluster par rapport à une solution de placement statique (SCSM).

Le paragraphe suivant conclut sur l'ensemble de cette étude des algorithmes faible complexité pour une structure many-core hiérarchique.

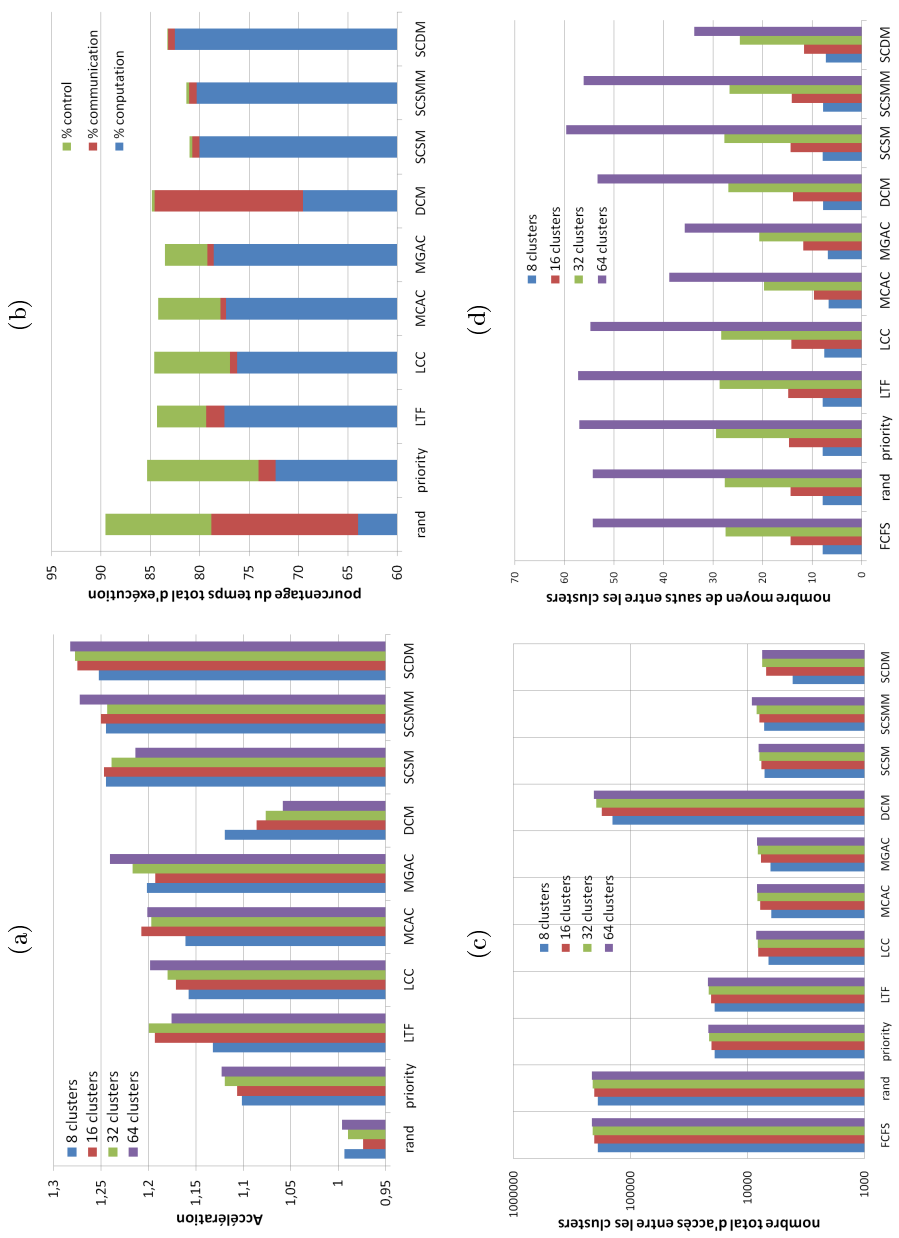


FIG. 4.7 – Pour un graph parallèle, (a) présente l'accélération de chaque algorithme par rapport à l'algorithme FCFS suivant différentes découpes en cluster. (b) présente la répartition des différents temps moyens pour le contrôle la communication et le calcul pour une découpe en 32 clusters. (c) et (d) montrent respectivement le nombre d'accès total à l'extérieur de tous les clusters et la distance moyenne parcourue en fonction du nombre de clusters.

4.4.3 Synthèse de l'étude sur les algorithmes d'ordonnancement de faible complexité

Ce paragraphe fait une synthèse de toutes les métriques suivant les deux types de graphe. On peut conclure de cette étude que les trois algorithmes de base FCFS, Randomized et Priority ne sont pas adaptés pour les structures hiérarchiques aux vues de leurs performances. En effet ils ne prennent pas en compte les problèmes de placement de tâches dans la structure.

L'algorithme LTF obtient une accélération moyenne dans les deux cas mais il ne permet pas de réduire les communications afin de garantir un maximum de performance. Cette étude confirme toutefois le choix que nous avons choisis d'employer ce type d'algorithme à l'intérieur des clusters dans le cas d'algorithme par groupe.

Pour sa part l'algorithme LCC n'est pas une bonne solution, principalement car il cherche seulement des optimisations locales à l'aide d'une fonction d'estimation des coûts de communication. De plus, les optimisations locales réalisées peuvent avoir un effet négatif d'un point de vue global ce qui explique les performances moyennes.

Le cas de l'algorithme DCM permet de conclure en partie sur la clusterisation. En effet les mauvaises performances de cet algorithme montrent qu'une découpe statique est plus performante qu'une découpe dynamique dans le cas précis de notre choix de clusterisation dynamique.

Les algorithmes proposés par liste ont les meilleures performances pour les graphes hautement dépendants ceci principalement par leur gestion tâche par tâche. Au contraire, les algorithmes par groupe linéaire gèrent mieux les graphes parallèles étant donné que les groupes de tâches se superposent sur les branches du graphe. Ils permettent aussi de diminuer le nombre de tâches à gérer et donc de diminuer le coût du contrôle.

Deux points importants sont à noter suite à cette étude. Le premier est que la migration tâche par tâche permet d'améliorer les performances sans détériorer les communications. Il est tout de même nécessaire que les calculs de migration se fassent en parallèle du traitement de l'application pour éviter d'ajouter un surcoût de contrôle qui pénaliserait les performances. Le deuxième point est que la clusterisation linéaire utilisé dans le cas des découpes statiques apporte de bonnes performances dans les deux cas d'applications, principalement grâce au pré découpage qui prend en compte les transferts les plus coûteux en termes de volume. Enfin, la hiérarchisation du contrôle permet une diminution du surcoût de temps de contrôle étant donné que le contrôle est parallélisé. Suivant ces conclusions, un modèle d'exécution va être proposé pour des architectures many-core hiérarchisées dans la section suivante.

4.5 Proposition d'un modèle d'exécution pour une architecture many-core hiérarchique

Cette section présente un modèle d'exécution dédié à une architecture hiérarchique ainsi que le support matériel associé. Ce nouveau modèle d'exécution s'appuie

sur des concepts du modèle d'exécution du processeur SCMP développé au sein du laboratoire durant la thèse de Nicolas Ventroux [85]. Le modèle d'exécution SCMP a été proposé pour une architecture asymétrique hétérogène permettant de considérer une séparation entre le contrôle et les calculs.

Dans ce modèle, le contrôle de l'application se fait tâche par tâche. Et pour chaque tâche, le contrôle est capable de gérer sa configuration, ses préemptions et ses migrations. Ce modèle d'exécution se base sur un modèle producteur/consommateur. Dans ce modèle, une donnée est écrite par un seul producteur à la fois. La donnée est ensuite disponible à la lecture lorsque la production est terminée, ce qui garantit la cohérence de la donnée dans un mode « zéro-copie » uniquement. Ce système nécessite une gestion dynamique des droits sur les données afin de permettre une synchronisation sur celles-ci.

Le modèle d'exécution supporte deux types d'applications de calculs intensifs : les applications utilisant un modèle d'exécution axé contrôle et les applications utilisant un modèle d'exécution axé flot de données. La gestion par tâche, dont l'ordonnancement proposé dans ce modèle, est adaptée à une architecture UMA. Or le chapitre 3 a montré qu'il est compliqué de garder une structure UMA pour une centaine de processeurs et donc qu'une structure NUMA, et plus particulièrement une structure hiérarchique est plus adaptée. Ce modèle va être adapté aux structures hiérarchiques contenant plusieurs centaines de processeurs. Afin de décrire ce modèle, la gestion des tâches puis la gestion de données vont être présentées. Enfin les critères de placement des groupes vont être spécifiés.

4.5.1 Gestion des tâches

Nous avons fait l'hypothèse que toute application peut être découpée en tâches et représentée par un graphe de tâches. Ce graphe est composé de tâches et d'arcs entre les tâches. Les arcs symbolisent les dépendances de données ou de contrôle. Une tâche est alors un ensemble d'instructions d'une taille minimale. Une tâche ne sollicite pas le contrôle de tâches durant son traitement. La taille d'une tâche doit être correctement choisie afin d'éviter une sollicitation trop importante du contrôle qui risquerait de dégrader les performances. Nous obtenons donc par construction du graphe, qu'à tout moment, les tâches, qui s'exécutent sur l'architecture, sont indépendantes.

Pour qu'une tâche soit prête à être exécutée, il est nécessaire que celle ci soit configurée. Cette configuration charge les instructions ainsi que le contexte de la tâche à exécuter. Elle se charge également d'allouer la pile de la tâche.

La gestion des tâches d'une application se fait à deux niveaux hiérarchiques séparés. Le premier niveau (contrôle externe cluster) considère l'application comme un ensemble de groupes qui contiennent un certain nombre de tâches. Ces groupes de tâches sont statiquement définis. Ils sont, de la même façon que les tâches, reliés entre eux selon un graphe de dépendances. Les groupes sont constitués de tâches successives. Deux tâches sont successives si la première n'a qu'une seule dépendance en sortie et que la seconde n'a qu'une seule dépendance en entrée. Le nombre de

tâches par groupe est au moins 1 et n'a pas de borne supérieure. Le rôle du contrôle de haut niveau est de répartir les groupes de tâches sur les clusters physiques.

Le deuxième niveau de contrôle (contrôle interne cluster) traite donc les groupes de tâches à l'intérieur de clusters physiques. Du fait que les groupes de tâches sont linéaires à tous moment ce niveau de contrôle doit ordonnancer seulement la tâche courante de chacun des groupes sur les processeurs du cluster. Le paragraphe suivant détaille la création de ces groupes.

La découpe statique de l'application - La découpe en groupes de tâches se fait statiquement selon une clusterisation linéaire modifiée. Dans notre version, les tâches de convergence et de divergence ne peuvent pas se placer au milieu d'un groupe de tâches. Les figures 4.8(a) et 4.8(b) montrent la différence entre les deux techniques.

Avec la clusterisation linéaire classique, la majorité des synchronisations sont reportées sur les données et pas sur le contrôle. En effet, par exemple, la figure 4.8(a) montre que le groupe de tâche 0 devra attendre une synchronisation sur les données de fin du groupe 1. La modification de l'algorithme est réalisée afin d'obtenir un graphe de groupes de tâches dont les groupes n'ont pas de dépendances de données pendant leurs exécutions. La figure 4.8(b), qui décrit notre solution, ne fait donc plus apparaître de synchronisation sur les données et ainsi limite le fait que les groupes se retrouvent tous alloués sur des clusters physiques en attente de données à traiter.

Il y a deux raisons à notre choix de modifier la clusterisation linéaire. Premièrement, ceci va permettre de placer au mieux les groupes de tâches en ne prenant en compte que les dépendances de la première tâche du groupe. Si la clusterisation linéaire normale, est considérée, les tâches 12 et 13 ne seront pas forcément placées sur le même cluster alors que c'est la plus forte dépendance. Dans notre cas, le contrôle placera le groupe C6 et donc la tâche 13 au mieux entre les groupes C1 et C5. Deuxièmement, c'est le contrôle qui va réaliser les synchronisations ce qui simplifie la gestion des données. Cependant le défaut de cette modification est de faire apparaître un plus grand nombre de groupes de tâches ce qui peut entrainer une sollicitation trop importante du contrôle qui s'occupe des groupes de tâches.

L'algorithme d'ordonnancement - D'après l'étude précédente, l'algorithme Static-Clustering-Dynamic-Mapping (SCDM) obtient les meilleures performances pour des graphes parallèles alors que c'est l'algorithme Most-Global-Affinity-Choice (MGAC) qui est le meilleur pour les cas de graphes extrêmement dépendants.

Nous avons donc choisi de modifier l'algorithme SCDM afin d'obtenir le meilleur compromis. En effet, à l'origine, l'algorithme SCDM utilise une clusterisation linéaire puis l'algorithme Most-Close-Affinity-Choice (MCAC) pour répartir les groupes de tâches mais d'après les résultats précédents, l'algorithme MGAC est meilleur que MCAC d'où le changement de technique. En pratique, il y aura donc une découpe statique décrite dans le paragraphe précédent, puis la gestion des groupes de tâches réalisée dynamiquement en utilisant l'algorithme MGAC.

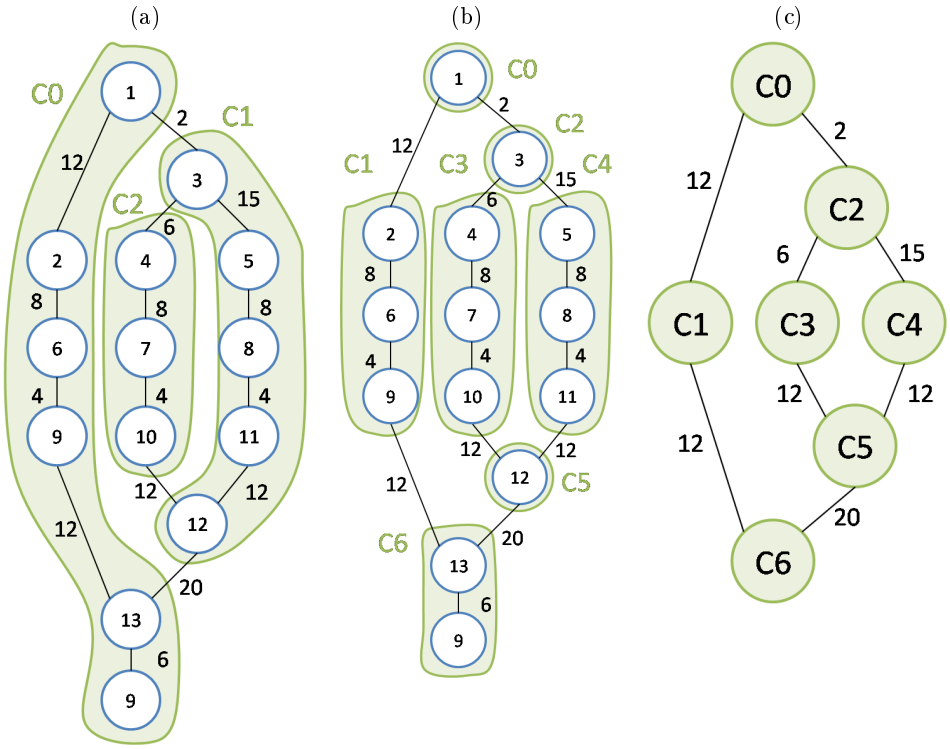


FIG. 4.8 – (a) Clusterisation linéaire suivant les liens de plus fort poids. (b) Clusterisation linéaire modifiée. (c) Représentation du graphe de groupes de tâches. Sur la première figure 4.8(a), l'algorithme de clusterisation linéaire est rappelé. Avec cette technique, 3 groupes de tâches sont créés. Dans notre cas, décrit sur la figure 4.8(b), 6 groupes sont créés dont le graphe final est représenté sur la figure 4.8(c).

Par rapport à l'algorithme SCDM, nous avons vu dans le paragraphe précédent que la découpe a été modifiée afin d'utiliser les avantages des deux algorithmes. En effet, nous avons modifié la clusterisation linéaire de façon à faire apparaître les groupes de tâches successives qui permettent d'obtenir de bonnes performances pour les graphes parallèles. Afin de tirer parti des bonnes performances de l'algorithme MGAC dans le cas de graphe dépendant, nous avons choisi d'arrêter la clusterisation au moment des convergences et divergences(figure 4.8(b)). En effet la nouvelle découpe permet dans le cas de graphes dépendants de ne pas fusionner beaucoup de groupes entre eux et donc de retrouver un graphe de groupes proche du graphe de tâches. Ceci permettra d'obtenir de bons résultats aussi dans le cas de graphes très dépendants.

Pour chaque groupe de tâches à ordonnancer, le contrôle réalise sont calcul de poids en fonction de la position des groupes de tâches précédents et du poids des communications comme le montre la figure 4.9. Il liste donc les groupes de tâches précédents, détermine où ils ont été exécutés et calcule le poids $W(vc_i, hc_j)$ pour

tous les clusters physiques disponibles :

$$W(vc_i, hc_j) = \sum_{vc_k}^{previous\_virtual\_cluster(vc_i)} \frac{D_{k,i}}{distance(hc_l, hc_j)}$$

où $D_{k,i}$ est le poids de l'arc entre les deux groupes de tâches, hc_l est le cluster matériel où s'est exécuté le groupe précédent et hc_j le cluster matériel disponible envisagé. Ensuite, parmi les clusters physiques envisagés, celui pour lequel $W(vc_i, hc_j)$ est le plus élevé, est sélectionné.

Pour le contrôle dans le cluster, le problème se résume à l'ordonnancement d'une liste de tâches sur une structure UMA. On choisira donc l'algorithme LTF (Longuest Task First) dont les bonnes performances dans ce cas ont été montrées.

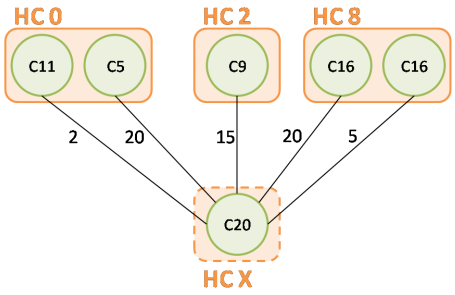


FIG. 4.9 – Représentation des informations nécessaires pour l'allocation du groupe C20 sur un cluster physique. Nous retrouvons le placement des groupes précédents ainsi que les volumes de données entre les groupes.

Les fonctionnalités complémentaires - Nous avons signalé dans le paragraphe 4.5.1, que notre découpe générait un grand nombre de groupes de tâches. Afin de diminuer le nombre de groupes pour le contrôle, il est possible de placer ces groupes de tâches dans des sur-groupes de tâches.

Concrètement, après la création statique des groupes de tâches, une phase de création de sur-groupes est réalisée. Suivant le choix du programmeur, les groupes de tâches sont regroupés en sur-groupe de N groupes de tâches parallèles. La valeur N peut être imposée par le programmeur ou peut prendre une valeur par défaut qui correspond au nombre de processeurs dans le cluster. Ces sur-groupes de tâches sont considérés par le contrôleur de groupes comme un groupe de tâches à part entière. Il est donc nécessaire d'ajouter à la description des groupes de tâches un paramètre qui contient le parallélisme de celui-ci afin d'allouer au mieux les groupes de tâches. En revanche, ces sur-groupes ne changent rien au fonctionnement du contrôle interne au cluster. En effet, lorsqu'un sur-groupe de tâches est alloué sur un cluster, le contrôle interne cluster voit l'allocation de plusieurs groupes de tâches au lieu d'un seul.

La gestion dynamique de l'ensemble de l'application peut engendrer des modes de fonctionnement qui ne conviennent pas aux programmeurs. Nous avons donc ajouté

une fonctionnalité qui permet aux programmeurs de contrôler les choix d'allocation du contrôle au niveau des groupes de tâches. Ceci lui est permis par la notion d'« affinité ». Cette notion a déjà été introduite par INTEL dans [98]. Dans ces travaux, les auteurs proposent à l'utilisateur de créer des groupes de threads sous un identifiant unique. Ainsi, les allocations de ces threads se font sur des processeurs proches.

Dans notre cas, nous proposons aux programmeurs de l'application de donner une affinité à des tâches ou à des groupes de tâches. Ceci principalement afin de permettre aux programmeurs de réaliser des optimisations de placement que le contrôle n'est pas en mesure de voir. Comme par exemple, pour prendre en compte, une dépendance de données non explicite dans le graphe de contrôle. Dans la pratique, l'affinité est héréditaire et donc lorsqu'une tâche avec une affinité est placée dans un groupe de tâches, celui-ci reçoit l'affinité de la tâche. En cas de conflit d'hérédité (si deux tâches avec des affinités différentes sont placées automatiquement dans le même groupe), c'est aux programmeurs de décider quelle sera l'affinité choisie par le groupe de tâches.

En pratique dans le contrôleur, cette affinité va obliger le contrôle à placer deux groupes de tâches avec la même affinité sur un même cluster physique ou au moins sur un cluster physique voisin. Cette technique va permettre d'influencer l'allocation afin d'augmenter les performances s'il n'est pas conscient de certaines propriétés de l'application. L'affinité est prioritaire sur le calcul du poids $W(vc_i, hc_j)$. Ainsi, si le contrôleur doit allouer un groupe de tâches avec une affinité, il y a deux possibilités. Soit ce groupe de tâches est le premier avec cette affinité et dans ce cas, il est placé selon les règles habituelles. Alors le cluster matériel choisi devient alors le référentiel de cette affinité. Soit l'affinité a déjà été détectée durant l'exécution et le contrôle va placer le groupe de tâches au plus près du cluster matériel référentiel pour cette affinité.

La migration - La migration correspond, dans notre cas, seulement au déplacement d'un groupe de tâches d'un cluster physique à un autre. Nous avons vu dans la première partie de ce chapitre que la migration permet d'équilibrer la charge entre les clusters physiques dans le cas où l'allocation dynamique n'y arriverait pas seule. Une fonction de migration est donc ajoutée dans notre modèle d'exécution afin d'améliorer les performances en rattrapant les erreurs d'allocation.

Cette migration doit être « intelligente » afin d'apporter un gain de performance. Celle-ci se fait en parallèle de l'exécution des tâches afin de ne pas ajouter de pénalité de contrôle. Une fonction est donc implémentée dans le contrôle externe aux clusters afin de ne pas gêner les allocations de tâches. Cette fonction fera partie de la boucle d'ordonnancement du contrôleur, elle sera alors lancée si un des clusters est surchargé de groupes.

Cette fonction de répartition permet donc de distribuer la charge sur un autre cluster. Afin que les calculs de transfert soient effectués, il est nécessaire qu'au moins un cluster physique soit en surcharge et qu'au moins un cluster physique soit

en sous-charge.

Le choix des groupes migrés se fait selon le coût des communications avant et après la migration. La fonction choisit parmi les groupes susceptibles de migrer, celui dont le coût de communications sans migration sera le plus élevé. En premier lieu, le choix se fait parmi les groupes n'aillant pas commencés leur exécution puis ceux dont aucune tâche n'est en cours d'exécution. Il choisi donc le groupe en fonction du poids utilisé pour l'algorithme d'ordonnancement de groupes (MGAC).

Ensuite, la fonction calcule ce nouveau coût de communication en fonction des clusters physiques disponibles et choisit le cluster qui produit le plus petit coût de communications. Ainsi, le cluster virtuel choisi est transféré sur son nouveau cluster physique. Etant donné qu'aucune tâche n'est interrompue, le coût de transfert est minimum.

4.5.2 Gestion des données

Ce paragraphe traite de la gestion des données dans le nouveau modèle d'exécution proposé. Ainsi, le modèle producteur/consommateur est gardé par rapport au modèle SCMP ainsi que la mémoire logiquement partagée et physiquement distribuée. Le modèle producteur/consommateur permet de simplifier le partage de données. La disposition de la mémoire logiquement partagée et physiquement distribuée, pour sa part, permet un grand nombre d'accès concurrents à la mémoire partagée et donc repousse la limite d'accès parallèle d'une mémoire partagée.

De la même façon, la gestion dynamique des droits sur les données est conservée. Cette gestion dynamique impose un unique écrivain mais ne limite pas le nombre de lecteurs autorisés. Il y a, à tout moment, qu'une seule copie de chaque donnée. Ces deux contraintes sur les données vont permettre de supprimer les problèmes de cohérence sur les données et ainsi éviter l'ajout de mécanismes pour assurer cette cohérence.

Cependant, ces contraintes de droits et de copies sur les données imposent un style de gestion des données particulier aux programmeurs. En effet, la gestion de la mémoire se fait donc explicitement dans le code de la tâche. Afin de gérer la mémoire, une API est disponible. Celle-ci contient les commandes de bases d'allocation et de libération, ainsi que des commandes particulières comme la gestion des droits. Ainsi, une tâche ne pourra accéder qu'aux données déterminées à la compilation.

L'allocation de ces données se fait d'une façon restreinte. Ce modèle d'exécution impose au système que les données créées par une tâche exécutée dans un cluster physique soient allouées dans le même cluster. Ceci pour simplifier et accélérer l'allocation mémoire. Afin de gérer au mieux les allocations mémoire, le programmeur devra renseigner la quantité de mémoire nécessaire pour chaque tâche. A la création des groupes de tâches, un cumul de l'ensemble de l'espace nécessaire au groupe est associé à celui-ci ce qui renseignera le contrôleur au moment de l'allocation.

Dans cette sous-section, la gestion de données a été décrite. Cette gestion est contrainte afin de permettre une cohérence des données sans ajout de mécanisme supplémentaire. La sous-section suivante présente les critères de déploiement d'un

groupe dans un cluster.

4.5.3 Les critères de placement des groupes de tâches

Il est important de préciser les critères qui font qu'un cluster matériel est capable de recevoir un groupe de tâches. Cette décision est fondée sur deux critères.

Le premier dépend de l'espace nécessaire au groupe pour s'exécuter. Ainsi, l'information sur la quantité de mémoire nécessaire à chaque groupe permet au contrôle de vérifier l'espace disponible dans le cluster avant d'autoriser l'allocation du cluster virtuel. Cette contrainte est du au mode d'allocation choisi. En effet, une donnée est toujours allouée dans le cluster qui a émis la requête. Cette certification permet de garantir le succès de l'allocation des données une fois le cluster virtuel placé étant donné que l'espace est virtuellement réservé. Le contrôleur doit donc connaître l'espace restant dans chaque cluster.

Le second critère dépend du nombre de groupes de tâches déjà alloués dans le cluster physique. Le nombre de groupes alloués dans un cluster est idéalement compris dans une fourchette dépendante du nombre de processeurs du cluster. Ainsi, si nous voulons que le taux d'utilisation du cluster ne soit pas trop faible, il est nécessaire d'avoir plus de groupes de tâches que de processeurs afin de ne pas avoir d'arrêt dans l'exécution. Au contraire un nombre trop important détériora les performances car de nombreux groupes seront en attente. Nous avons choisi cette fourchette de groupes entre 2 et 4 fois le nombre de processeurs. Ainsi, le contrôle qui s'occupe des groupes de tâches devra avoir un décompte des groupes pour chaque cluster physique afin de savoir, où une allocation est possible sans dépasser la limite supérieure.

La gestion des tâches a été décrite dans cette sous-section. Cette gestion est réalisée à deux niveaux séparés (groupe de tâches et tâche). Cette gestion prend appui sur l'étude réalisée précédemment sur les algorithmes d'ordonnancement. Ce modèle de gestion de tâches laisse une possibilité aux programmeurs d'influencer le choix dynamique réalisé par le contrôle haut niveau à l'aide de la notion d'affinité.

Cette sous-section a permis de terminer le détail du modèle d'exécution. Elle a aussi décrit la mise place un nouveau modèle d'exécution pour une structure hiérarchique capable d'exécuter dynamiquement les applications. Ce modèle repose sur la représentation sous forme d'un graphe hiérarchique de dépendances de l'application. Ainsi, en extrayant des groupes de tâches linéaires, le modèle d'exécution hiérarchique permet de gérer, dans un premier temps, un graphe de groupes de tâches plus simple que le graphe de tâches originel, et dans un second temps, un ensemble de groupes de tâches simple et parallèle. Nous avons maintenant le modèle d'exécution et une architecture massivement parallèle, la section suivante vérifie les performances de notre proposition.

4.6 Vérification des performances

Cette section a pour but de voir quelles sont les performances de la solution choisie. Afin d'obtenir ces résultats, la gestion de tâches décrit dans la section suivante

a été mise en place dans le simulateur de la section 4.4.

Pour les tests, les mêmes graphes que dans la section 4.4 ont été repris. Cependant, étant donné que les poids des communications ainsi que les temps de chaque tâche sont tirés aléatoirement, une nouvelle série de tests a été réalisée comparant notre solution aux algorithmes d'origine MGAC et SCDM.

Nous avons restreint l'étude à une découpe en 16 et 32 clusters qui sont les découpes les plus probables pour en ensemble de 128 processeurs. Les résultats sont visibles sur les figures 4.10 pour les deux types de graphes.

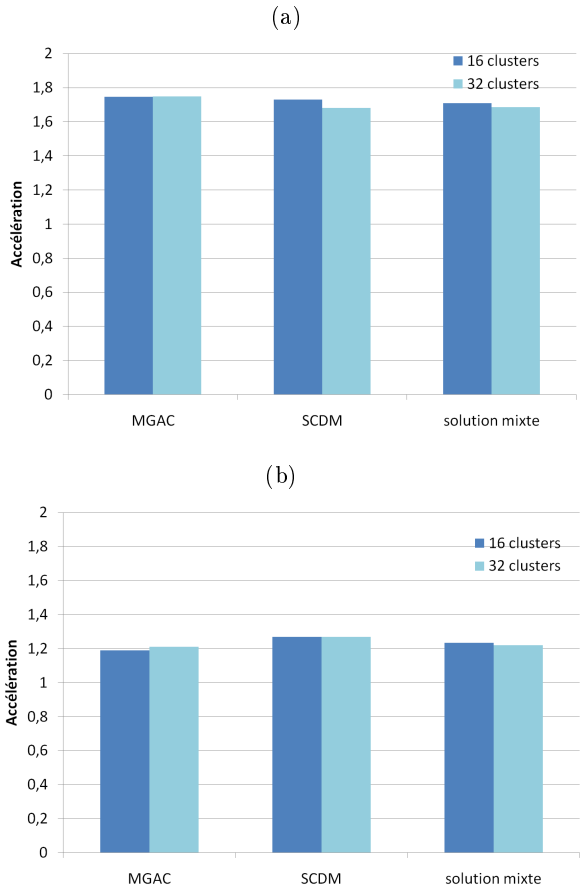


FIG. 4.10 – Résultat de performances de notre solution. (a) présente l'accélération pour un graphe hautement dépendant. (b) présente l'accélération pour un graphe parallèle.

Nous pouvons voir de ces deux graphiques que notre proposition se place entre les deux algorithmes comparées dans la majorité des cas. Etant donné que nous avons fusionné les deux algorithmes, il parait normal d'obtenir ce genre de résultats car les avantages et les défauts sont cumulés.

Notre solution perd en performance par rapport à une clusterisation linéaire classique car elle produit plus de groupes à gérer. Cependant, nous avons l'avantage d'avoir une clusterisation qui n'est pas dépendante du poids des arcs et donc ce

qui offre toujours la même découpe en groupes quelque soit les poids des différents axes. Cette particularité de notre solution offre les mêmes performances quelque soit les variations du graphe alors que la découpe de la clusterisation linéaire varie en fonction du poids des arcs et donc les performances varient également.

Par rapport à l'algorithme par liste MGAC, notre solution garde le même défaut que l'algorithme par groupe ce qui réduit ces performances. Ce défaut est de placer en avance les différents groupes ce qui peut nuire dans le cas de migration entre les clusters. Cependant cette algorithme garde l'avantage d'avoir le contrôle sur deux niveaux donc réduit les pénalités du au contrôle, étant donné que ce contrôle est parallélisé.

En conclusion, on obtient des performances correctes mais inférieures à nos espérances. Il reste à améliorer le placement des groupes afin de gagner en performances. La section suivante conclut ce chapitre.

4.7 Conclusion

Ce chapitre a proposé un nouveau modèle d'exécution pour notre structure hiérarchisée massivement parallèle. Pour cela, ce chapitre s'est intéressé au problème d'ordonnancement et de placement dynamique de milliers de tâches sur une structure hiérarchique many-core avant de proposer un modèle d'exécution adapté aux structures massivement parallèles hiérarchiques.

Rappelons que le problème d'ordonnancement et de placement est, dans notre cas, un problème NP-complet. Ce chapitre a tout d'abord décrit les solutions présentes dans le monde du calcul haute performance afin de déterminer quelles seraient les solutions possibles à adapter pour le monde de l'embarqué. Nous avons retenu de cette étude, les algorithmes par liste et par groupe. Quatre algorithmes par liste existants ont été choisis afin de permettre une comparaison (FCFS, Rand, Priority et LTF). Nous avons pris comme hypothèse que si l'on réduit les communications entre clusters qui sont les plus coûteuses, les performances du système seront meilleures. Selon cette hypothèse, nous avons proposé trois nouveaux algorithmes par liste (MCAC, MCAG, LCC) ainsi que quatre algorithmes par groupe (SCSM, SCSMM, SCDM et DCM) qui tentent tous de diminuer les communications entre cluster. Nous les avons comparés pour deux types d'applications. Cette comparaison a mis en évidence que les algorithmes par liste sont plus performants dans le cas d'un graphe hautement dépendant alors que les algorithmes par groupes (SDxx) sont les plus performants pour les graphes plus parallèles.

De ces conclusions, un modèle d'exécution a été proposé, se basant sur la création de groupes linéaires de tâches tout en utilisant un algorithme par liste afin de gérer ces groupes. Le chapitre suivant va, pour sa part, s'attarder sur l'architecture associée à ce modèle de programmation. Il va également valider les concepts de placement dynamique, de hiérarchisation du contrôle et de migration à l'aide d'un environnement de simulation.

Chapitre 5

Mise en place d'une architecture et validation de l'architecture et du modèle d'exécution

Sommaire

5.1 Description de notre architecture massivement parallèle . . 110
5.1.1 Vue d'ensemble du fonctionnement 111
5.1.2 L'architecture globale 113
5.1.3 Le cluster matériel 116
5.2 Une application de test 122
5.3 Implémentation de notre architecture dans SESAM 125
5.3.1 SESAM : Simulation Environment for Scalable Asymmetric Multiprocessors 125
5.3.2 Implémentation de notre architecture dans SESAM 127
5.4 Caractérisation de notre architecture hiérarchique 128
5.4.1 Dynamique versus Statique 129
5.4.2 Caractérisation de la hiérarchisation du contrôle 131
5.5 Impact de la hiérarchisation sur la surface 133
5.6 Conclusion 134

Nous avons présenté dans le chapitre précédent, un modèle d'exécution permettant la gestion dynamique de tâches sur une structure « many-core » hiérarchique. Ce modèle d'exécution se base sur une découpe statique du graphe d'application afin de réaliser le placement dynamique des différents groupes choisis sur les différents clusters de l'architecture.

Ce chapitre a pour but de présenter une architecture massivement parallèle programmable qui va permettre l'utilisation du modèle d'exécution proposé. Afin d'évaluer cette architecture, il est nécessaire d'avoir une application test ainsi qu'un environnement de simulation. C'est pourquoi une application test est choisie et détaillée. Ensuite l'environnement de simulation du laboratoire sera présenté avant de décrire précisément l'implémentation de notre architecture à l'intérieur de cet environnement.

A l'aide de l'application d'exemple et du simulateur de l'architecture, ce chapitre valide l'utilisation d'un contrôle dynamique dans le cas d'une utilisation normale. Toutefois, nous allons voir dans ce cas de figure, que le dynamisme peut réduire les performances. Le fonctionnement hiérarchique du modèle d'exécution et de l'architecture seront vérifiés et les performances seront caractérisées. Nous allons nous

focaliser sur le gain en performance qu'apporte la hiérarchisation du contrôle dans ce genre de structure. Enfin l'architecture proposée impose un surcoût matériel, nous allons donc estimer l'impact de la hiérarchisation du contrôle sur la surface.

5.1 Description de notre architecture massivement parallèle

Le modèle d'exécution décrit dans le chapitre précédent a été conçu pour être exécuté dans une structure hiérarchisée. L'architecture proposée dans ce chapitre est une machine à mémoire logiquement partagée mais physiquement distribuée. Elle se compose d'un ensemble de clusters de calcul indépendants organisés autour d'un réseau sur puce. Ces différents clusters sont supervisés par un contrôleur central. La figure 5.1 présente la disposition de ces clusters ainsi que du contrôleur central autour des interconnexions de communication.

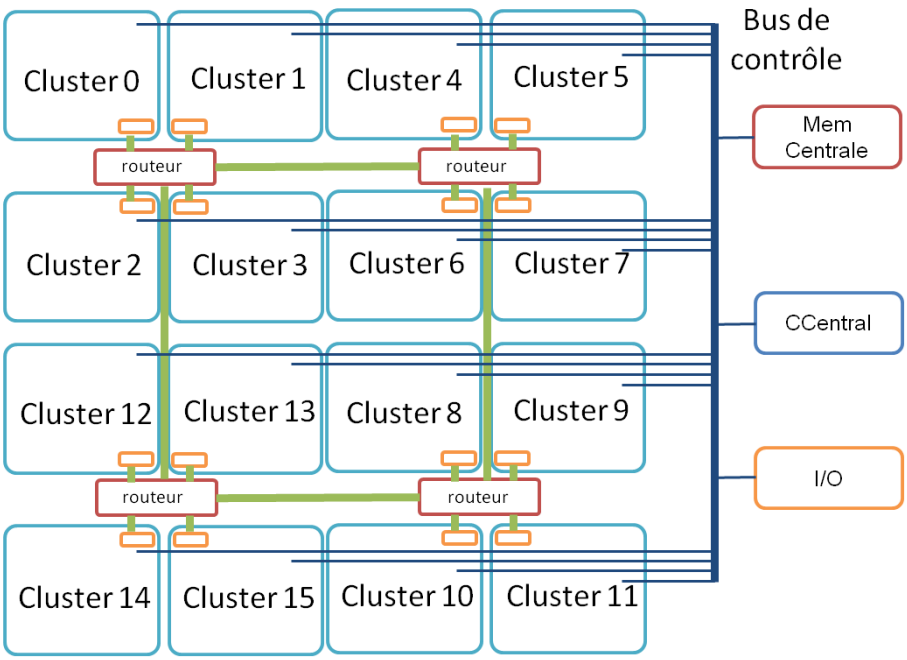


FIG. 5.1 – Représentation de l'architecture globale. Les clusters de calcul sont répartis par quatre autour de routeurs. Ces routeurs sont disposés en Anneau. Le contrôleur central ainsi que la mémoire centrale sont placés sur un réseau de contrôle séparé du réseau de données. Chaque cluster a un accès au réseau de contrôle.

Ce contrôleur central a pour but de répartir les différentes parties des applications exécutées sur les différents clusters de calcul. Ensuite, chaque cluster exécute les groupes de tâches assignés de façon autonome. Pour cela, un contrôleur cluster ordonnance les tâches sur les processeurs de calcul. La figure 5.2 détaille les différents composants du cluster.

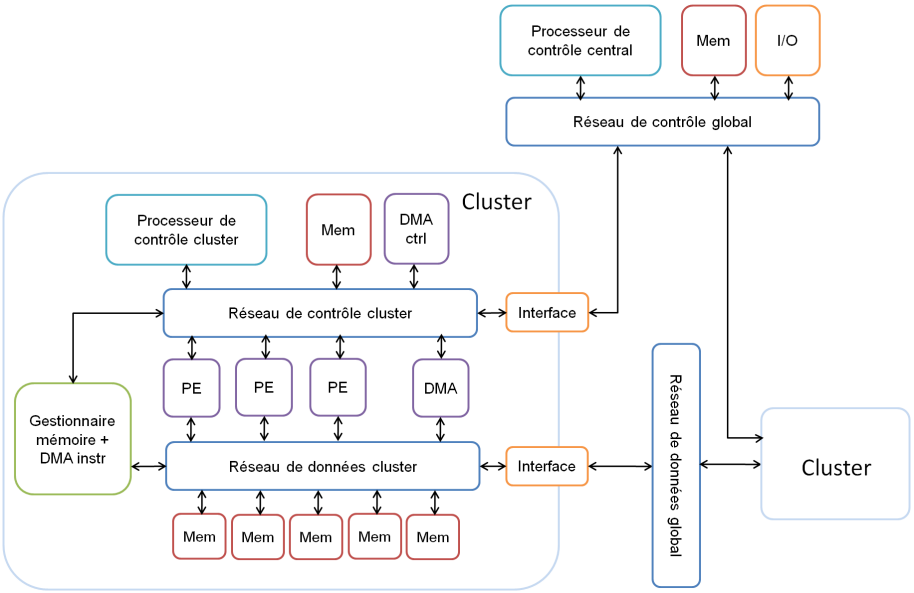


FIG. 5.2 – Représentation du cluster de base. Dans chaque cluster, on retrouve des processeurs de calcul, des bancs mémoire, un gestionnaire mémoire et un contrôleur cluster. L'ensemble des composants est connecté par deux interconnexions, une de contrôle et une de données.

Cette section détaille l'ensemble de l'architecture proposée. Pour cela, dans un premier temps, une vue d'ensemble du fonctionnement est présentée. Ensuite les composants globaux de l'architecture sont décrits en détail ainsi que les différents éléments du cluster.

5.1.1 Vue d'ensemble du fonctionnement

Cette sous-section a pour but de présenter, en entier, le fonctionnement de l'architecture en décrivant un exemple se basant sur l'exécution d'une application. La figure 5.3 présente les différentes étapes nécessaires entre le lancement de l'exécution d'une application et l'exécution de la première tâche.

Avant tout, nous considérons que l'ensemble des informations du graphe d'application et des groupes de tâches ainsi que le code des tâches sont mémorisés dans la mémoire centrale de la puce. Lorsqu'une application doit être exécutée sur l'architecture, le contrôleur central reçoit un ordre d'exécution de l'application (1). Celui-ci ajoute l'application dans la liste d'applications traitées. Afin de démarrer l'exécution, il alloue le premier groupe de tâches sur le premier cluster. L'allocation du groupe se fait par l'envoi d'un message sur le bus de contrôle contenant le numéro du groupe à exécuter (2). L'ensemble des contrôleurs cluster peut-être à tout moment interrompu par le contrôleur central.

Une fois ce message reçu, le contrôleur du cluster rapatrie les informations relatives au groupe de tâches afin de connaître les tâches à allouer sur les processeurs

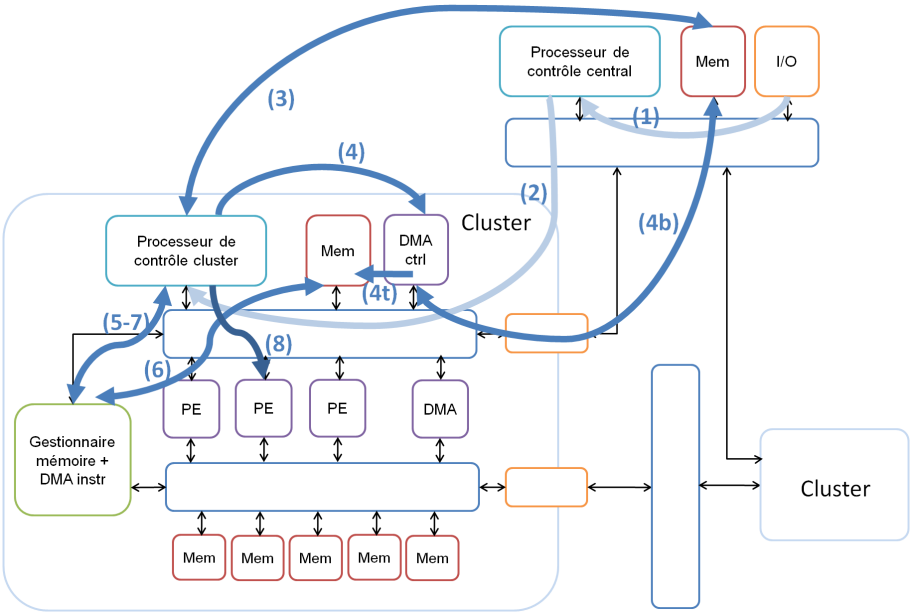


FIG. 5.3 – Représentation des différentes étapes de lancement d'une application. La demande d'exécution arrive au contrôleur central (1). Celui-ci envoie ensuite une demande d'exécution d'un groupe de tâches au contrôleur cluster (2). Le contrôleur cluster rapatrie les données concernant le groupe (3) et donne un ordre de copie des tâches au DMA de contrôle (4). Le gestionnaire mémoire configure ensuite la tâche (5-6-7). Enfin le contrôleur cluster envoie un ordre d'exécution de la tâche configurée sur l'un des processeurs (8).

(3). Une fois les informations du groupe disponible réceptionnées dans la mémoire de contrôle cluster, un ordre est envoyé au DMA contrôle chargé de rapatrier les codes des premières tâches du groupe dans la mémoire de contrôle du cluster (4). Une fois la première tâche copiée dans la mémoire, un ordre de configuration de cette tâche est envoyée au gestionnaire mémoire à travers le réseau de contrôle du cluster (5). Celui-ci doit réserver l'espace mémoire nécessaire, mémoriser la translation adresse physique en adresse virtuelle et copier le code de la tâche de la mémoire de contrôle dans le banc mémoire alloué (6). Lorsque la configuration est terminée, le gestionnaire mémoire envoie une confirmation de configuration de la tâche (7). Cette confirmation change l'état de la tâche qui peut être ordonnancée et placée par le contrôleur sur l'un des processeurs du cluster (8).

En parallèle de l'exécution de la première tâche, la tâche suivante est configurée afin de ne pas perdre de temps. A la fin de la première tâche, la suivante est ordonnancée et exécutée. Lorsque toutes les tâches d'un groupe sont terminées, un message de fin de groupe est envoyé au contrôleur central qui ordonnance les groupes suivants dépendants et prêts.

Durant l'exécution d'une tâche sur un processeur, celui-ci fonctionne indépendamment du contrôle. Il peut allouer, lire et écrire dans les bancs mémoire de l'ensemble de l'architecture. Cependant, les tâches manipulent les identifiants virtuels

de donnée, donc une translation est nécessaire afin de connaître l'adresse physique de la donnée.

Toutes les translations d'adresse d'un cluster sont contenues dans le gestionnaire mémoire. Cependant par souci de performance, un composant proche du processeur permet de réaliser ces translations. Ce composant est nommé TLB (Translation Lookaside Buffer). Il intercepte les demandes de lecture et d'écriture du processeur afin de changer l'identifiant virtuel en adresse physique.

A la première demande de lecture ou d'écriture d'une donnée, le TLB ne connaît pas encore la translation entre l'identifiant de donnée et l'adresse physique. Il est donc nécessaire que le TLB interroge le gestionnaire mémoire afin de connaître cette translation. Nous avons choisi, pour des considérations de surface, que le gestionnaire mémoire ne connaisse que les translations de son propre cluster. Dans le cas où celui-ci ne connaît pas la donnée, une requête est envoyée successivement à chaque autre gestionnaire mémoire afin de transmettre la translation au TLB.

Etant donné que le TLB intercepte les lectures et les écritures du processeur, il implémente également la traduction des différentes commandes envoyées du processeur. En effet, ces commandes se traduisent par des écritures et des lectures à des adresses particulières qui vont être reconnues dans le TLB.

Ceci décrit, de façon résumée, les fonctionnements de la hiérarchisation du contrôle et de la gestion mémoire. La suite détaille chaque composant évoqué.

5.1.2 L'architecture globale

L'architecture globale est composée, en plus de clusters, d'un contrôleur central, de deux réseaux d'interconnexion (un de contrôle et l'autre de données) et d'une mémoire centrale. Chaque partie va être détaillée.

Le contrôle central - Le premier élément de l'architecture est le processeur de contrôle central. Il permet de recevoir les demandes d'exécution d'application. Il a une vue uniquement au niveau groupe de tâches. Il se base sur un graphe de dépendance des groupes afin d'ordonnancer et d'allouer les groupes de tâches dans les clusters. Il envoie donc les numéros de groupes à chaque cluster et reçoit les événements de fin d'exécution de ces groupes. Ce contrôleur va également implémenter les fonctions de migration des groupes de tâches. Afin d'être capable de mettre en place la fonction de migration, les clusters signalent leur état au processeur de contrôle central soit sur demande de celui-ci, soit lorsqu'ils sont en sous ou surcharge. Cette sous ou surcharge est mesurée en fonction du nombre de groupes actifs et en attente dans le cluster.

Le contrôleur central est implémenté par un processeur généraliste associé à un TLB (Translation Lookaside Buffer) et un gestionnaire d'interruption. Nous avons choisi une version programmable avec l'utilisation d'un processeur afin de permettre l'introduction de la flexibilité dans le contrôle.

Le TLB mémorise les translations d'adresses virtuelles en adresses physiques et interprète la HAL (Hardware Abstraction Layer) dédiée au contrôle. Les translations

se font par comparaison des poids forts à une liste de poids forts connus qui sont remplacés ensuite par les poids forts de l'adresse physique. Utilisant le même système, le TLB interprète les écritures du processeur sur une plage d'adresses fixes comme étant les différentes commandes disponibles par la HAL. La figure 5.4 présente le mécanisme de translation du TLB.

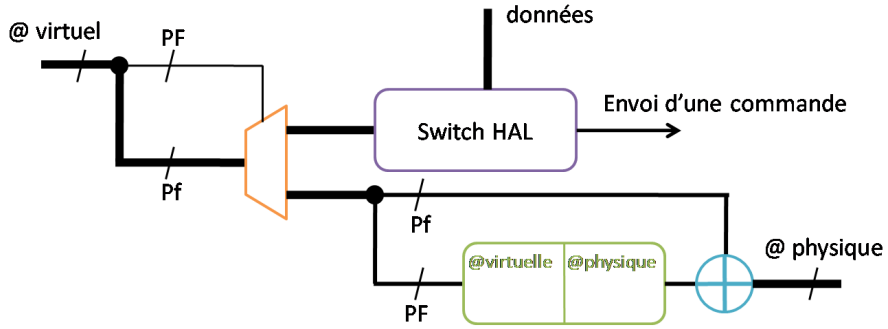


FIG. 5.4 – Représentation du fonctionnement du TLB. Un premier étage détermine si c'est une commande de la HAL ou une demande de lecture/écriture. Le deuxième étage permet soit d'envoyer une commande, soit de translater l'adresse.

Le TLB est capable d'interpréter un certain nombre de commandes afin de contrôler les groupes de tâches. Ces commandes sont listées ci-dessous.

- Demande d'exécution d'un groupe de tâches : Cette commande va permettre au contrôle central de lancer l'exécution d'un groupe sur un cluster. Les arguments à envoyer sont : le numéro du groupe, le numéro du cluster, le pointeur sur la première tâche à exécuter, la mémoire réservée par le groupe).
- Demande de préemption et suppression d'un groupe : Cette commande permet en un seul ordre d'arrêter et de supprimer le groupe du contrôle cluster. Il est nécessaire d'envoyer le numéro du groupe et le numéro du cluster.
- Demande de l'état du cluster : Cette commande demande au cluster de répondre par le nombre de groupes actifs et le nombre de groupes en attente dans le cluster. Il est juste nécessaire d'avoir le numéro du cluster destination.

Ces commandes vont permettre de placer et déplacer les groupes entre les clusters.

Le contrôleur d'interruption reçoit les demandes d'exécution d'application ainsi que les messages provenant des contrôleurs cluster. Il a pour mission de transformer les informations provenant des clusters en données manipulables par le processeur avant de l'informer de l'arrivée d'un message. En effet, les informations provenant des différents contrôleurs placés dans les clusters arrivent sous forme de messages qui ne sont pas directement interprétables par le processeur. Lorsqu'un message arrive au contrôleur, il est mémorisé dans une FIFO de profondeur égale au nombre de clusters afin de ne pas le perdre. Dans le cas où la FIFO est pleine, le message entrant n'est pas mémorisé et un message de demande de ré-envoi est expédié. Une fois un message mémorisé dans la FIFO, le contrôleur d'interruption signale la présence du message à l'aide d'une des lignes d'interruption du processeur. Celui-ci

utilise alors une commande de la HAL afin de récupérer le message par une lecture à une adresse spécifique fixe.

Le processeur du contrôle réalise les calculs d'ordonnancement, de placement et de migration des groupes de tâches. Pour cela, un cœur de boucles présenté sur la figure 5.5, qui en fonction de l'état des groupes de tâches en attente d'exécution et de l'état des clusters, place de nouveaux groupes de tâches sur les clusters lorsque cela est possible. Le placement est réalisé en fonction de l'affinité des groupes de tâches dans un premier temps puis en fonction du poids d'ordonnancement décrit dans le chapitre précédent.

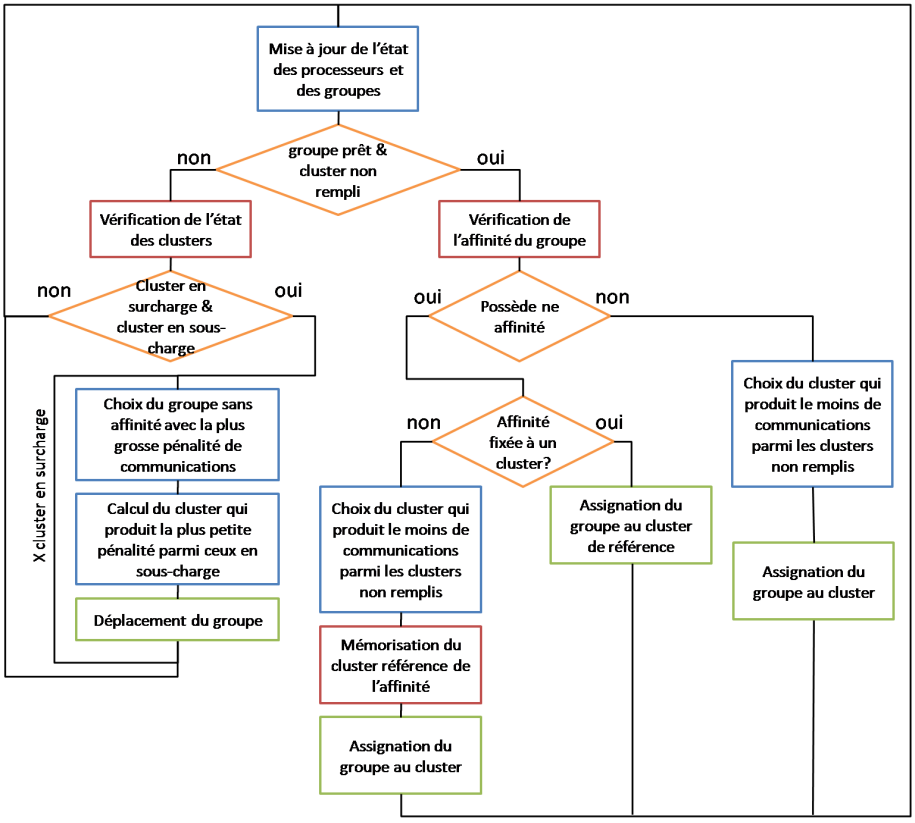


FIG. 5.5 – Représentation de la boucle d'ordonnancement du contrôle central.

Afin de réaliser le placement, le code de contrôle doit manipuler une structure qui contient l'état des groupes de tâches ainsi que leurs propriétés (parallélisme, taille mémoire consommée). Il doit également manipuler une structure contenant l'état des clusters (nombre de groupes parallèles alloués, taille mémoire restante). Lors de l'allocation d'un groupe, les différentes structures sont mises à jour. De plus, lorsqu'une interruption arrive au processeur, une routine d'interruption récupère l'ensemble des messages et traite chacun d'eux en mettant à jour l'état des groupes et des clusters.

Si aucun placement de groupes n'est possible la fonction de migration vérifie la

charge de chaque cluster et effectue des mouvements de groupes si nécessaire. Les migrations de groupes sont calculées sur la base d'un mouvement qui produira le moins de pénalités. Les pénalités sont calculées de la même manière que pour le placement des groupes. Une migration se compose de deux ordres : une suppression d'un groupe d'un cluster puis l'exécution de ce même groupe sur un autre cluster.

Le réseau de contrôle global - Nous n'avons réalisé aucune étude afin de dimensionner ce réseau d'interconnexions. On peut supposer que les messages de contrôle vont être courts et espacés dans le temps. Cependant sur ce réseau va également transiter les codes des différentes tâches et groupes de tâches. Il est donc nécessaire qu'il permette des transferts de données plus importants. Le nombre de points de connexion excède les 32 maîtres dans une configuration 32 clusters, la solution la plus simple, qui est le bus, risque d'obtenir des performances moyennes. Cependant la quantité de messages envoyés va rester faible, c'est pourquoi pour des soucis de contraintes de surface, nous avons donc décidé d'implémenter un bus comme interconnexion de contrôle global.

Le réseau de données global - L'étude menée dans le Chapitre 3 a montré, que la solution Multi-bus + Anneau amélioré propose de bonnes performances et une bonne efficacité énergétique. Le réseau de données global est donc défini par une topologie Anneau avec quatre clusters connectés sur chaque routeur de l'Anneau. Le transfert des messages se fait par paquets et chaque paquet est envoyé en whormhole à travers le réseau.

La mémoire centrale - La mémoire centrale contient l'ensemble des codes des tâches des applications, la liste des groupes de tâches et les tâches de chaque groupe. Enfin cette mémoire contient également les données externes de la puce. Dans une première version, cette mémoire est connectée à un des routeurs du réseau de contrôle.

La suite de la section décrit les différentes parties d'un cluster.

5.1.3 Le cluster matériel

La figure 5.2 fait apparaître les différentes parties du cluster. Ce cluster se compose de processeurs de calcul, de DMA (Direct Memory Access), de bancs mémoire, d'un réseau de données, d'un réseau de contrôle, d'un gestionnaire mémoire, du DMA de contrôle et d'un processeur de contrôle cluster. Chacune des différentes parties va être détaillée ainsi que leur fonctionnalité.

Le contrôle cluster - Le contrôle cluster permet de gérer les tâches dans le cluster. Celui-ci n'a qu'une vision au niveau tâche mais il est capable d'exploiter plusieurs types de tâches (calcul et DMA). Il reçoit, de la part du processeur de contrôle central, les groupes de tâches à exécuter. Lorsqu'il reçoit un nouveau groupe, il commence par configurer la première tâche de chaque groupe. Par la

suite, la configuration d'une tâche est réalisée avant la fin de la tâche précédente. Les configurations des différentes tâches doivent être terminées avant de pouvoir les ordonnancer et les allouer sur les processeurs ou les DMA.

Le contrôle cluster est constitué d'un TLB, d'un contrôleur d'interruption ainsi que d'un processeur. Comme pour le contrôle central, le TLB réalise une translation d'adresses virtuelles vers des adresses physiques. Il interprète également la HAL contenant les commandes spécifiques liées à la gestion de tâches. Il garde le même fonctionnement que celui du contrôleur central afin d'interpréter les lectures et les écritures. Les commandes de contrôle de tâches sont décrites ci-dessous.

- Demande de copies du code de tâches : Cette commande permet de rapatrier le code des tâches successives d'un groupe dans la mémoire de contrôle du cluster. Cet ordre est à destination du DMA de contrôle et comporte le numéro du groupe, le numéro de la première tâche et le nombre de tâches suivantes.
- Demande de configuration d'une tâche : Cette commande demande au gestionnaire mémoire de préparer l'exécution d'une tâche. Pour cela le gestionnaire mémoire a besoin du numéro de la tâche.
- Demande d'exécution d'une tâche : Cette commande est destinée à un processeur. Il est nécessaire d'envoyer le numéro de la tâche et le numéro du processeur.
- Demande de préemption d'une tâche : Cette commande envoyée au processeur arrête l'exécution de la tâche et demande une mémorisation du contexte. Les numéros de la tâche et du processeur accompagnent l'ordre.
- Demande de reprise d'une tâche : Cette commande permet de recharger le contexte d'une tâche sur n'importe quel processeur. Les numéros de la tâche et du processeur sont envoyés avec la commande.

Le gestionnaire d'interruptions reste le même que le gestionnaire d'interruptions du contrôleur central sauf pour la mémorisation des messages. Il doit, dans le cas du contrôleur cluster, recevoir l'ensemble des messages provenant des processeurs ainsi que du gestionnaire mémoire du cluster. Le gestionnaire reçoit donc les messages provenant du réseau de contrôle cluster, les mémorise dans une FIFO puis signale au processeur, par un signal d'interruption, la présence d'un message. Le gestionnaire d'interruptions contient deux FIFO. La première sert à mémoriser les messages provenant des processeurs. Elle a une profondeur égale au nombre de processeurs. La seconde FIFO mémorise les messages provenant du gestionnaire mémoire. Si cette FIFO est dimensionnée, dans le pire des cas, elle peut être extrêmement grande étant donné que le nombre de messages maximum dépend du nombre de tâches dans le cluster. C'est pourquoi il est nécessaire de mettre en place un mécanisme pour ne pas perdre de messages. Le gestionnaire mémoire connaît la profondeur de la FIFO qui lui est dédiée et n'envoie de messages que s'il y a de la place. Après chaque lecture de message provenant du gestionnaire mémoire, une confirmation de lecture est envoyée par le gestionnaire d'interruption.

Le processeur du contrôleur cluster exécute en permanence une boucle qui va permettre d'ordonnancer les tâches sur les processeurs. Cette boucle manipule deux structures. La première contient l'état de chaque processeur et la tâche en cours

d'exécution sur celui-ci. La seconde structure contient le numéro et l'état de la tâche courante de chaque groupe présent dans le cluster ainsi que la tâche suivante.

La figure 5.6 représente la boucle d'ordonnancement du contrôleur. Pour chaque type de tâches (calcul, DMA), il existe une liste distincte. La boucle réalise, dans un premier temps, le tri de la liste de tâches prêtes à l'exécution selon le temps d'exécution le plus long. Ensuite en fonction des disponibilités des processeurs, les premières tâches de la liste sont placées sur les processeurs libres. Afin de réaliser ce placement, un message est directement envoyé au processeur concerné contenant le numéro de la tâche à exécuter.

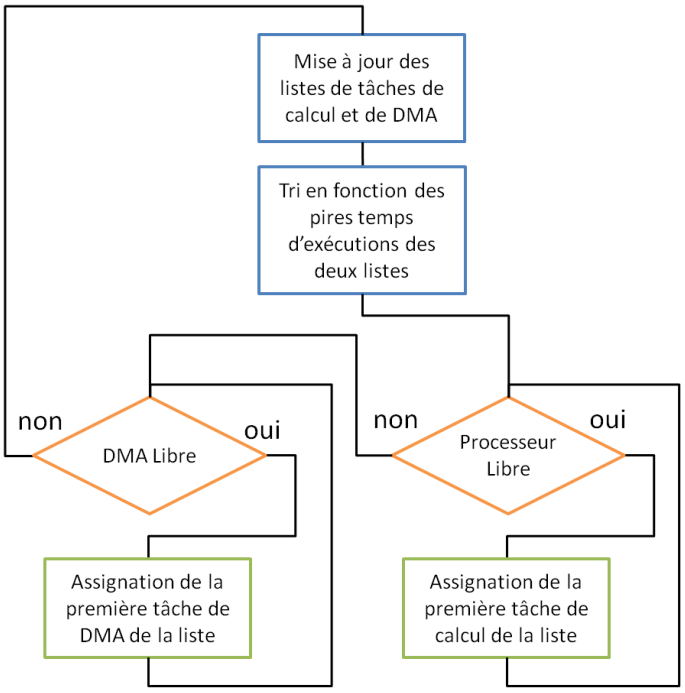


FIG. 5.6 – Représentation de la boucle d'ordonnancement du contrôle cluster.

Lors de la réception d'un message par le contrôleur cluster, une interruption vient arrêter l'exécution de la boucle d'ordonnancement, afin de traiter ce message. En fonction des situations, les différentes actions sont décrites ci-dessous.

- Lorsqu'une tâche se termine, le processeur envoie un message au contrôleur. Le code met à jour l'état de la tâche. Il lit le numéro de la tâche suivante et envoie la configuration de cette nouvelle tâche. Afin de libérer de la mémoire, une demande de suppression du code de la tâche terminée est demandée.
- Lorsqu'un nouveau groupe est ajouté au cluster, le processeur reçoit le numéro du groupe ainsi qu'un pointeur sur la première tâche du groupe. En fonction du pointeur dans le groupe, le processeur copie les informations du groupe et donne l'ordre au DMA de contrôle de rapatrier les codes des premières tâches dans la mémoire locale dédiée au contrôle du cluster. Ensuite, une mise à jour

des structures gérées est réalisée afin de faire apparaître le nouveau groupe. Enfin une demande de configuration est demandée pour la tâche pointée dans ce groupe.
- Lorsqu'un groupe de tâches se termine, les informations concernant le groupe dans la structure de contrôle sont retirées et une demande de suppression du code de la dernière tâche est envoyée au gestionnaire mémoire. Enfin un message de fin de groupe est envoyé au contrôleur central.
- Lorsqu'une demande de suppression d'un groupe de tâches est reçue par le processeur, celui-ci vérifie, dans un premier temps, que ce groupe n'a pas de tâches en cours d'exécution. Ce cas ne devrait pas arriver car seuls les groupes en attente sont autorisés à être déplacés. Cependant un message de non suppression est envoyé au contrôleur central. Dans un cas normal, le groupe est retiré de la gestion du cluster et le numéro de tâche courante est mémorisé afin de le renvoyer avec le message de confirmation. Une demande de suppression du code de la tâche courante du groupe est envoyée au gestionnaire mémoire.
- Lorsque le contrôle central demande l'état du cluster, afin de pouvoir répondre rapidement, le programme garde deux variables globales à jour contenant les nombres de groupes actifs et en attente. Un message est renvoyé contenant ces informations.

Le gestionnaire mémoire - Le gestionnaire mémoire permet de gérer l'ensemble des bancs mémoire ainsi que les droits associés à chaque donnée. Sa première tâche est d'allouer et de libérer la mémoire des différents bancs mémoire. Il va limiter l'accès à une donnée afin de permettre une gestion simple de la cohérence. Nous avons choisi de distribuer le gestionnaire mémoire dans chaque cluster de façon à limiter la taille des tables de translation. Le gestionnaire de chaque cluster ne s'occupe que de la mémoire de son cluster. Afin de simplifier l'accès aux données des autres clusters, il peut communiquer avec les autres gestionnaires de tâche afin de localiser une donnée qu'il ne connaît pas.

Ce composant est constitué d'une machine d'état, d'une mémoire adressable par le contenu (CAM) et d'un module DMA. La machine d'état va gérer les différents messages provenant des processeurs et du gestionnaire cluster. La CAM va stocker les translations d'adresse et les droits sur les données. Afin de réaliser des recherches rapides dans la mémoire, celles-ci sont réalisées sur les adresses virtuelles. Le module DMA permet de transférer les codes de tâche de la mémoire de contrôle cluster vers les bancs mémoire du cluster.

En fonction des demandes, différentes branches de la machine d'état sont empruntées.
- Lorsque le contrôleur cluster demande une configuration de tâche, dans un premier temps, le gestionnaire mémoire récupère les informations concernant la tâche dans la mémoire de contrôle du cluster (adresse du code dans cette même mémoire, taille du code, taille de la pile). Puis il alloue la mémoire nécessaire pour recevoir le code et la pile de la tâche. Enfin un ordre est donné

au module DMA afin de réaliser la copie en parallèle du traitement des autres demandes.
- Lorsqu'un processeur demande une translation d'adresse virtuelle en adresse physique, une recherche par rapport à l'adresse virtuelle est réalisée afin de pouvoir renvoyer l'adresse physique au processeur. Si la donnée ne se trouve pas dans le cluster, un message est envoyé à chaque gestionnaire mémoire des autres clusters afin de trouver la donnée. Le gestionnaire mémoire répond au processeur par la translation d'adresse ainsi que par le numéro du cluster contenant la donnée.
- Lorsqu'un processeur demande un accès en lecture ou en écriture à une donnée, une recherche est réalisée sur l'adresse afin de trouver les droits associés à cette donnée. En fonction du numéro de la tâche, le gestionnaire mémoire vérifie les droits et répond positivement ou non à la requête. Si la réponse est positive l'exécution de la tâche continue. Dans le cas contraire, la tâche reste bloquée en attente de la donnée et le gestionnaire mémoire mémorise la tâche en attente.
- Lorsqu'un processeur demande un changement de droit sur une donnée, s'il est le propriétaire de la donnée, les modifications sont apportées. Lors de modification de droits, le gestionnaire mémoire informe le contrôleur de tâche qui était en attente sur la donnée. Si la donnée ne lui appartient pas, un message d'erreur est renvoyé à la tâche.
- Lorsqu'un processeur demande un changement de propriétaire sur une donnée, s'il est le propriétaire actuel de la donnée, les modifications sont apportées. Si la donnée ne lui appartient pas, un message d'erreur est renvoyé à la tâche.
- Lorsqu'un processeur demande une allocation mémoire, un algorithme de placement de données choisit l'emplacement dans l'ensemble des bancs mémoire afin de remplir au mieux chaque banc mémoire. La translation d'adresse est enregistrée puis l'adresse physique est renvoyée au processeur.
- Lorsqu'un autre gestionnaire mémoire demande si une donnée est présente, une recherche est réalisée sur l'adresse virtuelle. Si la donnée est présente, le gestionnaire envoie son numéro de cluster, sinon il renvoie la valeur -1.

Le DMA - Les DMA permettent de transférer des données de la mémoire centrale vers les mémoires des clusters. Ils sont contrôlés à l'aide de tâches adaptées placées dans le graphe de l'application. Ils sont implémentés à l'aide d'une machine d'état qui envoie les différents messages de lecture et d'écriture et d'une mémoire interne qui sert de tampon entre les différents transferts. Cette mémoire conditionne la valeur maximale de chaque lecture. Le code d'une tâche DMA contient l'adresse mémoire où sont enregistrés les paramètres du transfert (adresse de la donnée, adresse de la destination, taille de la donnée, motif de la copie). Les paramètres du transfert peuvent être fixés à la compilation ou écrits de manière dynamique par une autre tâche.

Le processeur de calcul - Les processeurs de calcul exécutent les tâches allouées par le processeur de contrôle indépendamment du reste de l'architecture. Il renvoie au contrôleur l'information de fin de tâche. Il communique également avec le gestionnaire mémoire afin de connaître les adresses et les droits d'accès aux données.

Il est constitué d'un TLB, d'un processeur ainsi que d'un gestionnaire d'interruptions. Le TLB a la charge de mémoriser les translations d'adresse. Il réalise également les demandes au gestionnaire mémoire qui remplissent ces translations. Le TLB permet également d'interpréter la HAL afin d'envoyer les commandes vers le gestionnaire mémoire et le contrôleur cluster

- Demande d'allocation mémoire : Cette commande est envoyée au gestionnaire mémoire et permet de réserver de la mémoire partagée. Il est nécessaire de fournir les identifiants de la donnée et de la tâche ainsi que la taille de la donnée.
- Demande de suppression d'une donnée : Cette commande, envoyée au gestionnaire mémoire, permet de supprimer la translation. Elle s'accompagne des identifiants de la donnée et de la tâche.
- Demande de lecture ou d'écriture d'une donnée : Cette demande est envoyée au gestionnaire mémoire. Elle permet de connaître l'état de la donnée. Si la réponse est négative la tâche reste bloquée jusqu'à ce que le contrôleur la débloque. Il est nécessaire d'envoyer les identifiants de la donnée et de la tâche.
- Demande de changement de droits d'une donnée : Cette commande permet d'autoriser d'autres tâches en lecture ou en écriture sur une donnée. Les identifiants de la donnée et de la tâche sont envoyés avec la demande au gestionnaire mémoire.
- Demande de changement de propriétaire d'une donnée : Cette commande permet de céder l'appartenance d'une donnée à une autre tâche. Le gestionnaire mémoire a besoin des identifiants de la donnée et de la tâche.
- Fin de tâche : Cette commande est envoyée au contrôleur cluster avec le numéro de la tâche afin de continuer l'exécution de l'application.

Le TLB mémorise les numéros de clusters de chaque donnée ce qui permet lorsqu'une commande est envoyée de la diriger vers le gestionnaire mémoire du cluster contenant la donnée en question.

Le gestionnaire d'interruptions interprète les messages provenant de contrôleurs cluster afin de transformer le message pour le processeur. Seul le contrôleur cluster peut envoyer des messages au processeur. La FIFO de réception est de taille fixe connue par le contrôleur cluster afin de ne pas la saturer. Son implémentation reste la même que pour les autres gestionnaires d'interruption.

Le DMA de contrôle - Le DMA de contrôle est exclusivement utilisé par le contrôleur cluster afin de récupérer les codes des tâches d'un groupe à l'intérieur de la mémoire de contrôle du cluster. Il est constitué, comme pour les DMA conventionnels, d'une machine d'état qui commande les lectures et les écritures et d'une mémoire tampon afin de mémoriser la donnée entre une lecture et une écriture. A

la différence des autres DMA, il reçoit directement les informations du transfert par des messages provenant du contrôleur cluster. Il ne réalise qu'un seul type de transfert par bloc de données continues.

Le réseau de contrôle - Le réseau de contrôle dans le cluster doit permettre de transmettre les informations de contrôle le plus rapidement possible afin de limiter les latences de contrôle. Etant donné que les DMA utilisent également ce réseau, nous avons choisi un Multi-bus. Ce réseau offre une petite latence de communication et limite les contentions. Nous avons vu précédemment que ce réseau pouvait avoir une surface démesurée. Cependant dans notre cas de figure le nombre de connexions est assez faible ce qui permet de limiter la surface.

Le réseau de donnée - L'étude menée dans le Chapitre 3 a montré, que la solution Multi-bus + Anneau amélioré propose de bonnes performances et une bonne efficacité énergétique. Le réseau de données dans le cluster est donc, comme nous l'avons choisi précédemment, un Multi-bus.

Ceci décrit les différentes parties de l'architecture. Dans toute cette section, une architecture associée au modèle d'exécution a été présentée. Afin de la tester, il est nécessaire de choisir une application.

5.2 Une application de test

Comme cité dans l'introduction, les applications sont de plus en plus dynamiques et leur temps d'exécution est variable en fonction de différents paramètres non prédictibles hors-ligne. Dans cette section, nous allons présenter notre application d'exemple et détailler la partie implémentée sur le simulateur.

L'application de vision de surveillance de carrefour a été choisie car elle nécessite une puissance de calcul répartie sur l'ensemble de l'application. De plus, comme nous allons le détailler ensuite, cette application comporte des parties statiques et d'autres dynamiques ce qui ne permet pas de placer au mieux statiquement toutes les parties de l'application.

Cette application a pour but d'envoyer une alerte à un central de surveillance si un accident est détecté au carrefour surveillé. La détection est réalisée sur un flux vidéo obtenu par une caméra équipée d'un objectif « fisheye » afin d'avoir le plus grand angle de vue possible. L'application va détecter tous les objets en mouvement dans la vidéo et pister chacun de ces objets. Ensuite en comparant les trajectoires et les vitesses de chaque objet, l'application va être capable de détecter leurs collisions éventuelles. Cette application se présente sous la forme d'un pipeline logiciel qui intègre différents traitements à différents niveaux dans une première boucle, la compression et le stockage de la vidéo dans la deuxième. La figure 5.7 montre ce pipeline logiciel. Elle détaille les différentes tâches de l'application et fait apparaître le niveau de dynamisme de chaque tâche de la partie de la branche réalisant la détection.

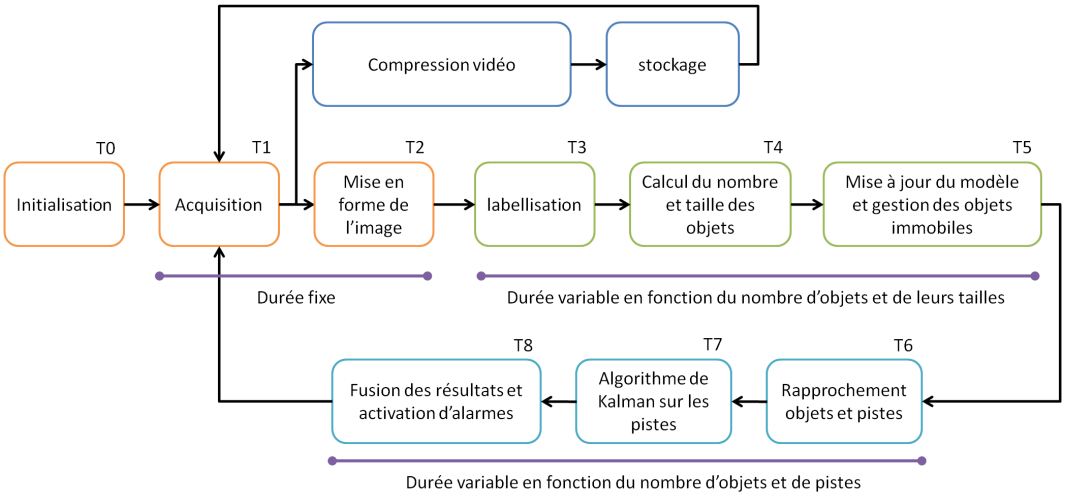


FIG. 5.7 – Pipeline de l'application de surveillance de carrefour. Pour chaque tâche, il est mentionné si celle-ci est dynamique (ou non) et selon quels critères.

Nous allons détailler les différentes tâches qui composent la branche réalisant la détection. La tâche T0 permet l'initialisation du système comme la mémorisation du fond qui représente le carrefour vide. Elle est réalisée périodiquement afin actualiser régulièrement les informations. Cependant, elle ne rentre pas dans la boucle principale. La tâche T1 récupère l'image acquise par le capteur. Cette opération est un traitement fixe de transfert de donnée. Quant à T1 son temps de traitement est constant et fixé par l'architecture.

La chaîne de traitements, à proprement parlé, commence par la tâche T2. Cette tâche réalise des traitements bas-niveau sur l'image. Ceux-ci se composent de filtres destinés à améliorer l'image ainsi que d'une transformation géométrique pour rectifier la forme de l'image. Enfin, une soustraction du fond est réalisée afin de préparer l'image pour l'étape suivante qui consiste en un étiquetage des objets présents dans la scène. Tous ces traitements sont fixes. La puissance de calcul nécessaire ainsi que le temps de traitement sont donc dépendants uniquement de la taille de l'image.

Au contraire à partir de la tâche T3, les traitements sont variables en fonction des données contenues dans l'image ainsi que de l'historique de l'application. Typiquement, cela correspond aux nombres d'objets trouvés sur chaque image.

La première tâche dynamique est une tâche d'étiquetage. Celle-ci permet de détecter les zones contiguës de pixels et ainsi d'identifier l'ensemble des objets à traquer dans l'image. Ensuite la tâche T4 répertorie ces objets et détermine leur taille. Cette tâche supprime également les objets trop petits et refocalise les objets autour des zones à fort gradient qui sont caractéristiques. La durée de traitement de ces deux tâches dépend donc du nombre d'objets et de leur taille. La tâche T5 met à jour le modèle de suivi en fonction des informations reçues sur ces objets et traite les objets immobiles. Son temps d'exécution varie donc également en fonction

du nombre d'objets.

Les trois tâches restantes s'occupent de la partie pistage du traitement global. En effet, pour chaque objet traqué, une piste est créée et maintenue à jour en fonction des informations obtenues. La tâche T6 a pour but de raccrocher les objets aux pistes existantes en fonction de prédictions faites sur les pistes répertoriées. Si un objet n'appartient à aucune piste répertoriée, une nouvelle piste est créée pour cet objet. La tâche T7 permet d'appliquer l'algorithme de Kalman aux différentes pistes afin de prédire la position et la vitesse des différents objets. Le pipeline se termine sur la tâche T8. Cette tâche fusionne les informations des tâches précédentes afin de déterminer s'il y a un risque de collision entre les objets. Elle va également activer une alarme si besoin. Afin d'accélérer le traitement, pour chaque piste, une sous tâche est créée dans chacune des tâches. Ces trois tâches ont donc un nombre de sous tâches variables en fonction du nombre de pistes présentes dans la scène. De plus chaque sous-tâche a un temps d'exécution variable en fonction du nombre d'objets.

L'ensemble de la chaîne a été présenté et pour chaque tâche le dynamisme a été précisé. L'ensemble de cette chaîne de traitements a un temps d'exécution qui varie d'un facteur 4, en moyenne, en fonction du nombre d'objets dans la scène. Afin de réaliser nos tests il n'est pas nécessaire d'utiliser l'ensemble du pipeline de détection. Il est simplement utile de choisir une des tâches représentatives de l'application. Nous avons choisi la tâche d'étiquetage. Celle-ci est une tâche dynamique de l'application, composée d'un flot de contrôle complexe. De plus, son traitement ne dépend pas de l'historique de l'application, comme le nombre de pistes suivies, ce qui va permettre de réaliser des tests sur des images seules.

Afin de comprendre les tests, cette tâche d'étiquetage va être détaillée ci-après. Dans la suite du mémoire, elle sera considérée comme une application à part entière.

L'application d'étiquetage [99] utilisée est une application orientée calcul intensif. Elle permet à partir d'une image d'entrée de déterminer les zones de pixels blancs contiguës. Elle produit une image où chaque zone possède un niveau de gris différent.

Elle se découpe en différentes phases successives. La majorité des phases peuvent être parallélisées afin d'accélérer l'exécution. La figure 5.8 représente l'application pour un parallélisme de 256, sous la forme d'un graphe de tâches composé de 32 branches identiques. L'image de base est découpée en 256 morceaux. Chacune des 32 branches de l'application traite 8 imagettes. La figure 5.9 détaille la forme d'une branche ainsi que les tâches qui s'y rapportent. Chaque branche se décompose donc en un étiquetage de chaque imagette, suivie d'une fusion horizontale puis verticale des labels obtenus afin de rendre cohérents les labels sur les 8 imagettes. Ceci est réalisé avant de ré étiqueter tous les labels. Les images 5.10(a) et 5.10(b) montrent un exemple d'image d'entrée et le résultat en sortie. On obtient donc une image où chaque forme est identifiée par un niveau de gris.

Dans cette section, nous avons choisi un exemple d'application qui va permettre de réaliser nos tests d'architecture. A cet effet, nous avons besoin d'un environnement complet de simulation. La section suivante détaille l'environnement du laboratoire SESAM puis l'implantation de notre architecture dans le simulateur.

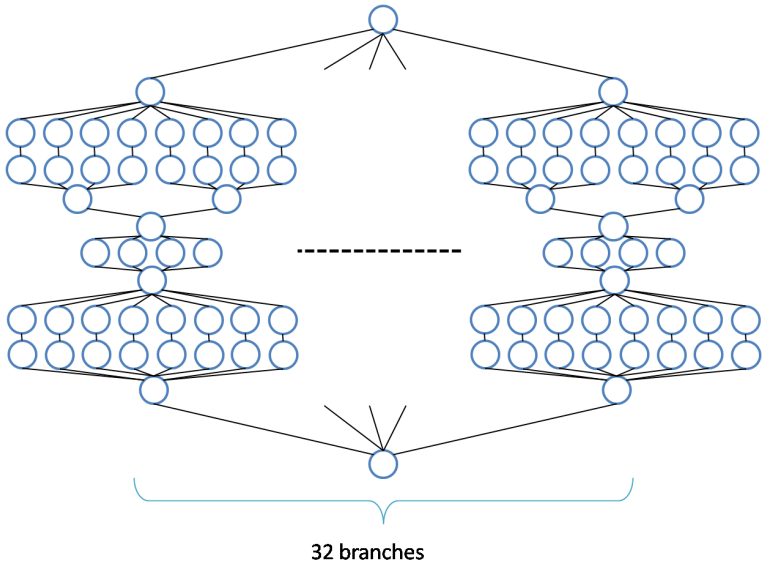


FIG. 5.8 – Représentation du graphe de tâches de l'application d'étiquetage. Cette application est découpée en 32 branches identiques.

5.3 Implémentation de notre architecture dans SESAM

Cette section a pour but de présenter l'implémentation réalisée afin de permettre la validation du modèle d'exécution. Pour cela nous allons, dans un premier temps, présenter SESAM. Puis nous détaillerons les modifications apportées afin d'implémenter nos architectures.

5.3.1 SESAM : Simulation Environment for Scalable Asymmetric Multiprocessors

Cette sous section présente SESAM [100] qui est l'environnement de simulation réalisé au sein du laboratoire LCE. Cet environnement a été conçu afin de permettre l'exploration d'architectures multiprocesseurs asymétriques. Cet environnement de simulations est basé sur la librairie SystemC [68]. Il permet aussi bien d'utiliser la librairie TLM que de réaliser des simulations RTL ainsi que du codesign entre les différents niveaux de simulation avec ModelSim. Ce simulateur modulable donne accès à une large librairie de composants permettant de décrire entièrement un multiprocesseur. Il va donc permettre l'exploration matérielle aussi bien que logicielle.

SESAM possède également une caractéristique intéressante pour l'exploration. Il est possible de modifier une longue liste de paramètres avant chaque exécution ce qui va permettre de modifier les caractéristiques de la simulation sans avoir à recompiler le simulateur. Par exemple, il est possible de changer le nombre de processeurs, leur type, le mapping mémoire, la quantité de mémoires ou encore les timings du réseau. Cette caractéristique va permettre de créer plus facilement des scripts afin d'explorer

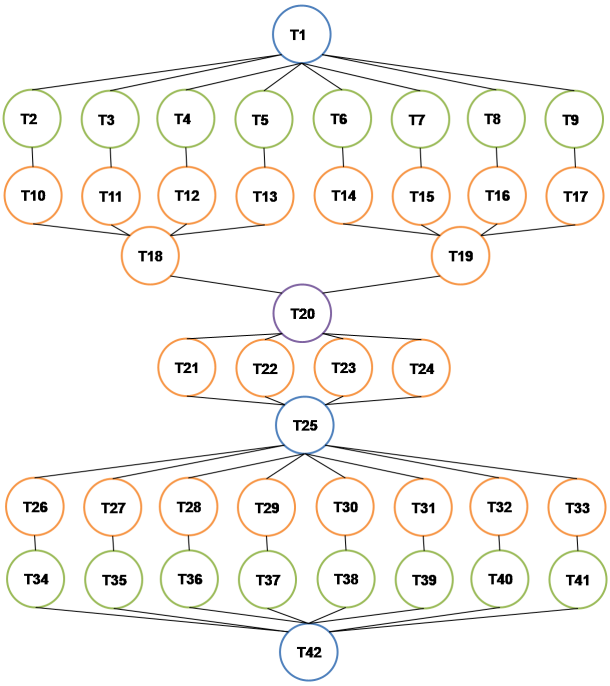


FIG. 5.9 – Représentation d'une branche du graphe de tâches de l'application d'étiquetage. La tâche 1 calcule les paramètres destinés aux tâches DMA T2 à T9. Ces tâches rapatrient les données depuis la mémoire externe vers la mémoire locale. Les tâches T10 à T17 labellisent chacune une imagette. Les tâches T18 et T19 réalisent la fusion verticale alors que les tâches T21 à T24 exécutent la fusion horizontale. La tâche T20 permet une synchronisation avant la fusion horizontale. La tâche T25 regroupe les informations des fusions et prépare les paramètres de T34 à T41. Les tâches T26 à T33 recopient les bons labels dans l'ensemble des imagettes. Les tâches T34 à T41 transfèrent l'image labellisée dans la mémoire extérieure. T42 finalise la branche.

FIG. 5.10 – (a) Exemple d'image d'entrée pour l'application d'étiquetage . (b) Image de sortie de l'application d'étiquetage pour l'image a en entrée.

l'architecture. La sous-section suivante décrit l'implémentation de notre architecture dans l'environnement SESAM.

5.3.2 Implémentation de notre architecture dans SESAM

Il existe, dans SESAM, un certain nombre de modules déjà disponibles permettant de simuler une architecture de multiprocesseurs asymétriques. C'est pourquoi, cette sous-section présente les modifications apportées aux modules existant dans SESAM pour recevoir notre architecture hiérarchisée ainsi que les paramètres choisis.

De base, l'environnement SESAM n'étant pas capable de gérer la hiérarchisation, nous avons dû développer les mécanismes de création des clusters. Nous avons également modifié les gestionnaires mémoire afin qu'ils soient capables de communiquer entre eux et d'ajouter la notion de cluster dans les TLB. Enfin, seul un contrôleur centralisé est disponible dans cet environnement, c'est pourquoi, nous avons dû l'étendre en un contrôle hiérarchisé.

Pour l'ensemble de l'implémentation, l'architecture est décrite au niveau TLM. Celle-ci est formée de 8 clusters contenant chacun 4 processeurs, 2 DMA et 176 bancs mémoire. Cette combinaison a été choisie afin d'avoir une architecture représentative de la solution contenant 256 processeurs mais en limitant le temps de simulation. La figure 5.11 présente l'architecture implémentée où seulement deux clusters sont représentés pour des raisons de lisibilité de la figure. Cette description permet de visualiser les différents composants et les connexions.

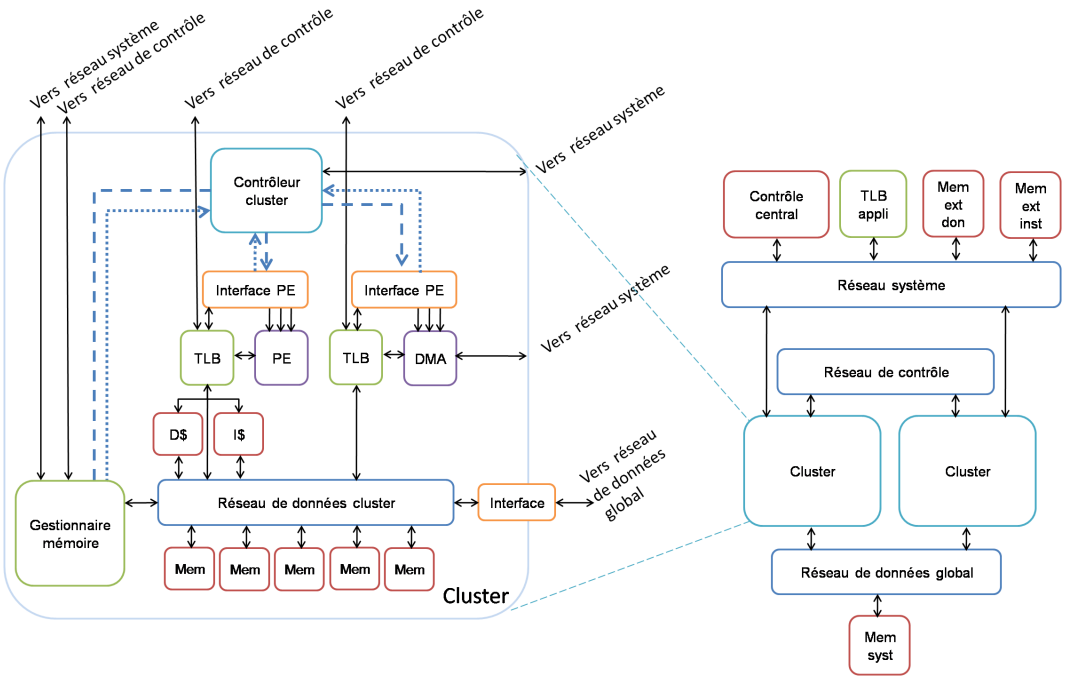


FIG. 5.11 – Représentation de l'architecture contenant un contrôle hiérarchique.

Dans l'architecture simulée, chaque banc mémoire est de 16 Kilo octets soit un

total de 22 Mo de mémoire partagée dans l'architecture. Les mémoires sont cadencées à 500 MHz. Les processeurs sont implémentés par des Instruction Set Simulator (ISS) fonctionnels. Ils sont également cadencés à 500 MHz. Les interconnexions utilisées sont les mêmes que pour l'étude des réseaux du chapitre 2. Nous considérons que l'ensemble des interconnections est cadencé à 500 MHz. Les Multi-bus ont à un temps de traversée d'un cycle. Le réseau Anneau entre les clusters a pour sa part un temps de traversée de quatre cycles entre deux routeurs (routeur + lien).

La figure 5.11 fait apparaître quelques modifications par rapport à l'architecture décrite dans la section précédente. Les processeurs sont tous accompagnés d'un cache de donnée et d'un autre d'instruction de un kilo octets chacun. L'Interface PE joue le rôle du contrôleur d'interruption pour chaque processeur de calcul et pour chaque DMA. Enfin les commandes entre le contrôleur cluster et les processeurs passent par des liens dédiés. Afin de raccourcir le temps de développement, nous avons choisi de garder certains mécanismes déjà implémentés dans l'environnement SESAM.

Dans cette implémentation de l'architecture, tous les services proposés dans le modèle d'exécution n'ont pas été implémentés. Ainsi le contrôle central alloue statiquement les différents groupes sur les clusters. Ce choix a permis de simplifier l'implémentation tout en gardant en vue les tests à réaliser.

Toute cette section a présenté l'environnement SESAM. Puis nous avons détaillé l'implémentation de notre architecture afin de mieux comprendre les résultats. La section suivante explique les tests et les comparaisons effectués et présente les résultats et les conclusions associés.

5.4 Caractérisation de notre architecture hiérarchique

Dans cette section, différents aspects majeurs de notre modèle vont être caractérisés. Pour cela des tests vont être réalisés représentant des scénarii d'utilisation.

Il est nécessaire, pour permettre une validation de notre solution, de la mettre en parallèle avec une autre architecture. Nous avons choisi de comparer notre architecture avec un contrôle hiérarchisé, à une architecture à contrôle centralisé. La base de l'architecture reste la même et seul le contrôle est modifié. Le contrôle centralisé traite l'ensemble de l'application au niveau tâche. Il réalise donc seul la répartition de l'ensemble des tâches sur les processeurs de l'architecture. La configuration des tâches dans l'architecture se fait de manière Round-Robin sur l'ensemble des clusters afin de les équilibrer. L'assignement des tâches aux processeurs possède deux versions pour cette architecture à contrôle centralisé. Une première version naïve où le contrôle n'a pas conscience des clusters. Il attribue donc en fonction des disponibilités n'importe quelle tâche sur n'importe quel processeur sans faire attention dans quel cluster a été configuré la tâche. Une deuxième version permet de choisir un processeur dans le cluster où a eu lieu la configuration de la tâche afin de limiter les pénalités de transport. Les deux versions seront utilisées ultérieurement.

La caractérisation de notre architecture est réalisée en la comparant à cette architecture au contrôle central. Les tests consistent donc en l'exécution de l'application

d'étiquetage sur les deux architectures dans différentes situations.

Nous allons, dans cette section, caractériser le gain de la gestion dynamique, les pertes possibles dans le cas d'une mauvaise utilisation du dynamisme et enfin le gain dû à la hiérarchisation du contrôle et au modèle d'exécution.

5.4.1 Dynamique versus Statique

Dans cette sous section, le gain du fait de la gestion dynamique de l'application va être discuté ainsi que les pertes de performance dues à une mauvaise gestion du dynamisme. Le placement dynamique doit permettre de placer les tâches en prenant en compte l'état actuel de l'architecture. Ceci va permettre de gagner en performance grâce à un placement au mieux des tâches et de répartir la charge de calcul sur les processeurs.

On peut estimer que dans une situation normale de l'application de surveillance de carrefour, le nombre d'objets, en moyenne dans l'image, est de trois ou quatre objets, de différentes tailles disposés dans la scène. Ces objets, qui sont des piétons et des véhicules, ne seront jamais placés régulièrement dans la scène étant donné la forme du carrefour surveillé. On peut donc envisager que dans la majorité des cas l'image à étiqueter aura une répartition inégale d'objets dans les différentes imagettes constituant la scène.

Un scénario standard est donc envisagé afin de montrer les avantages de la gestion dynamique par rapport à une gestion statique lorsque l'application est dynamique. On utilise pour ce test l'architecture hiérarchisée pour les deux cas comparés. On se place dans la situation où un piéton seul traverse un carrefour surveillé. Lors de l'étiquetage, on obtient une image à traiter ressemblant à la figure 5.12. Au vue du dimensionnement de l'architecture, l'image est découpée en huit fois quatre sous-images qui vont être placées chacune sur une branche de l'application. Chaque sous-image sera ensuite découpée par la branche qui la traite en huit imagettes.

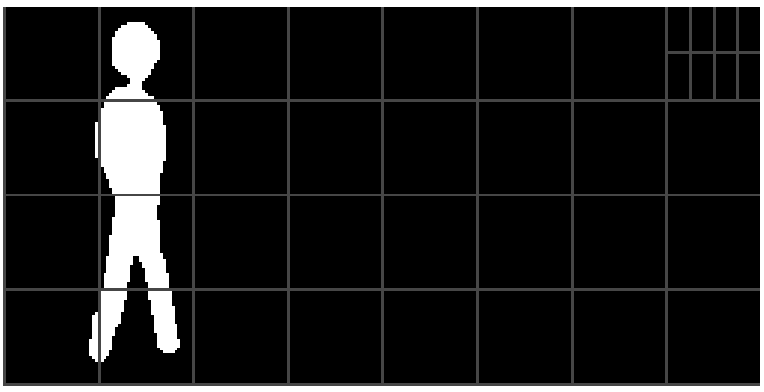


FIG. 5.12 – Image à labelliser contenant une fausse silhouette de piéton. On voit également apparaître la découpe des sous-images destinées à chaque branche. Dans la sous-imagette en haut à droite, on retrouve la découpe finale en imagette.

Si l'on envisage un placement statique de l'application sur la structure clustérisée et que l'on distribue statiquement chaque colonne sur chaque cluster, il en résulte qu'un des clusters va récupérer la majorité de la charge de calcul. Ceci peut être évité en plaçant dynamiquement les branches et surtout en les migrant en fonction de la charge des clusters. La migration n'étant pas implémentée, nous avons simulé le mécanisme en chargeant une demi-ligne par cluster au lieu d'une colonne. Il en résulte que la charge de calcul est mieux répartie dans l'architecture.

Le tableau 5.1 montre le temps total d'exécution de l'application dans les deux cas de mapping. On voit que le cas le plus favorable permet de gagner 30% sur le temps d'exécution. Ceci dépend grandement de la scène à labelliser. Ce gain est variable en fonction du nombre d'éléments dans l'image et de leur taille. Plus l'image sera chargée d'objets et moins le gain sera important. Ceci s'explique étant donné que pour cette application les temps d'exécution varient en fonction des objets dans les imagettes. Et donc dans le cas extrême où toutes les imagettes contiennent des objets, les différences dans les temps d'exécution ne sont plus assez grandes pour avoir un gain significatif avec le placement dynamique.

Cependant, en moyenne, à un carrefour, l'image n'est pas entièrement remplie d'objets à étiqueter. On peut donc supposer que le gain obtenu sur l'exemple est représentatif du gain moyen obtenu à l'aide d'une répartition dynamique.

type de placement	temps total d'exécution (million cycles)
placement statique	13
placement dynamique	8.94

TAB. 5.1 – Temps d'exécution de l'application de test pour un placement statique ou dynamique.

Ce test montre que dans les cas où l'application est dynamique, le placement dynamique des tâches dans l'architecture permet d'avoir une répartition de la charge de calcul et donc un gain de performance. Cependant la répartition de charges et donc le placement des tâches peuvent avoir un effet négatif dans certaines conditions.

En effet, si la configuration de la tâche n'a pas été faite dans le cluster où est exécutée la tâche, les performances peuvent chuter. En effet, les coûts de communication vont être supérieurs au gain de la répartition de charge. On se place dans un scénario où les choix des placements envoient les tâches aléatoirement dans des clusters.

Pour réaliser ce test, l'architecture possédant un contrôle central est utilisée. On considère cette architecture dans ces deux implémentations : soit avec un placement aléatoire, soit avec un placement qui prend en compte la configuration des tâches. Rappelons que les configurations sont réalisées de façon Round-Robin dans les clusters. Dans le cas du placement intelligent, le placement se fait toujours dans le cluster de la configuration. Dans l'autre cas, on obtient un placement des tâches qui a 1/8 de chance d'être au bon endroit. L'image de test reste celle du premier test.

Le tableau 5.2 montre la différence entre les temps d'exécution avec placement près de la configuration de la tâche ou non. On voit apparaître une différence de 13% entre les deux solutions. On peut estimer que cela serait la perte en performance obtenue dans le cas où les décisions de placement seraient erronées. On aurait alors les tâches allouées loin du code d'instruction et des données, comme le simule ce test. Cette perte maximale reste inférieure au gain dû au dynamisme. Cela pousse à penser qu'il y a un vrai gain à prendre en compte au niveau de l'utilisation du dynamisme.

type de placement	temps total d'exécution (million cycles)
placement intelligent	17
placement aléatoire	19.7

TAB. 5.2 – Temps d'exécution de l'application de test en fonction du type de placement.

Ces tests ont permis de mettre en évidence le gain de la gestion dynamique dans un fonctionnement courant ainsi que la perte potentielle des performances due à de mauvais placements. Le dernier test réalisé montre l'intérêt d'utiliser un contrôle hiérarchisé dans le cas de la gestion d'un grand nombre de tâches. La sous section suivante permet d'évaluer le gain qu'apporte un contrôle hiérarchisé dans notre structure.

5.4.2 Caractérisation de la hiérarchisation du contrôle

Dans cette sous section, le but est d'évaluer le gain en performance d'un contrôle hiérarchique dans notre structure multiprocesseur. Pour ce test, on se place dans un scénario classique en utilisation normale des deux architectures. Nous comparons les temps d'exécution de l'application d'étiquetage 256 sur deux structures : une avec un contrôle central et une avec un contrôle hiérarchisé. Nous allons également mesurer le temps moyen d'ordonnancement des tâches sur les processeurs qui est directement lié au nombre de tâches à ordonnancer.

On compare les exécutions sur la même image 5.12 pour les deux architectures. L'image a peu d'impact sur notre test. Nous visualisons la différence entre les deux modèles d'exécution ainsi que la différence d'implémentation du contrôle.

Le tableau 5.3 présente le temps moyen que prend le processeur de contrôle (soit central, soit dans le cluster) pour ordonnancer et placer les tâches sur les processeurs. En effet ce temps varie en fonction du nombre de tâches à ordonnancer. On voit, sur ce tableau, une énorme différence entre les deux cas. La structure hiérarchique gagne plus de 90% sur le temps de contrôle moyen par rapport à la solution centrale. Ceci s'explique par le fait que dans le cas hiérarchique, chaque cluster doit s'occuper seulement de 4 branches de l'application alors que le contrôle central gère les 32 branches en parallèle. On parallélise ainsi le traitement des différentes tâches sur les différents processeurs de contrôle.

Plus particulièrement l'algorithme d'ordonnancement dans chaque cluster correspond à un ordonnancement par liste. Chacune de ces phases d'ordonnancement

commence par un tri de toutes les tâches en attente de traitement (qui sont déjà configurées) puis par le placement des tâches sur les processeurs. Ceci explique pourquoi, il y a un intérêt à répartir au mieux les tâches à traiter sur les différents processeurs de contrôle.

type de contrôle	temps de tick (cycles)
contrôle hiérarchique	7712
contrôle centralisé	111344

TAB. 5.3 – Temps moyen d'ordonnancement durant l'application de test en fonction du type de contrôle.

Afin d'évaluer le coût du contrôle, le temps minimal d'exécution de toute l'application est montré à titre de référence. Pour évaluer ce temps, la durée moyenne d'une tâche est mesurée durant l'exécution. Ensuite, ce temps est multiplié par le nombre d'étages de tâches qui ne peuvent pas être exécutées en parallèle.

Le tableau 5.4 montre les résultats du temps d'exécution de l'application en fonction du choix du contrôle. On voit apparaître que le coût du contrôle pour la solution hiérarchique par rapport à la solution idéale est d'environ 19%. Ce pourcentage, en plus sur le temps d'exécution, est composé du temps d'ordonnancement des contrôleurs cluster et des temps de configuration des différentes tâches.

Ce tableau fait également apparaître un gain de 46% pour la solution hiérarchisée par rapport à la solution centrale. Cette différence significative entre les deux solutions réside dans la répartition des tâches sur les différents processeurs de contrôle. Le gain apparaît alors car il y a un traitement en parallèle des tâches sur ces différents processeurs de contrôle et donc un gain de temps de contrôle.

type de contrôle	temps total d'exécution (million cycles)
idéal	7.2
contrôle hiérarchique	8.9
contrôle centralisé	19.7

TAB. 5.4 – Temps d'exécution de l'application de test en fonction du type de contrôle.

Ce test a permis de mettre en évidence le gain au niveau de la gestion hiérarchique des tâches dans la solution dynamique. Du fait que le processeur de contrôle traite moins de tâches, celui-ci réduit considérablement son temps de traitement à chaque ordonnancement ce qui permet un gain de temps sur l'ensemble de l'application. De plus, le placement des groupes sur les différents clusters permet de diminuer les communications entre clusters et participe à l'augmentation des performances. La section suivante réalise une estimation du surcoût en surface de la hiérarchisation du contrôle.

5.5 Impact de la hiérarchisation sur la surface

La hiérarchisation du contrôle offre un gain significatif sur le temps de traitement de l'application. Cependant, la hiérarchisation du contrôle impose de dupliquer les organes de contrôle dans la puce. Cette section propose une estimation de l'impact du surplus de contrôle dans une architecture contenant 256 processeurs, 256 Mbits de mémoire.

Afin d'estimer la surface, on se place dans le même cas que pour la synthèse du réseau sur puce soit en TSMC 40 nm low power. Nous considérons que l'estimation comprend seulement les processeurs, les bancs mémoire, le réseau d'interconnexions et un processeur pour chaque composant de contrôle. Le tableau 5.5 récapitule les différents chiffres de l'estimation.

éléments	surface (mm^2)
1 AntX de contrôle + 128Kbits	0.042
8 Mips de calcul	1.8617
8 Mbits	2.03
Multi-bus 8→8	0.00826
2 Anneaux (16 routeurs - 6 ports)	0.2649
cluster Hier (1 AntX + 8 Mips + 8 Mbits + Multi-bus)	3.94
cluster N-Hier (8 Mips + 8 Mbits + Multi-bus)	3.9
Archi Hier (1 AntX + 32 clusters Hier + Anneaux)	126.2
Archi N-Hier (1 AntX + 32 clusters N-Hier + Anneaux)	124.8

TAB. 5.5 – Estimation de la surface des architectures en mm^2. Le Mips considéré est un Mips 24 kec fonctionnant à 400 MHz. Il comporte 377 Kportes. Le processeur AntX fonctionne à 300 MHz et contient 16 Kportes. Les chiffres qui permettent d'estimer la taille d'un Mips, d'un AntX et de la mémoire proviennent de la documentation de TSMC. TSMC estime que 1620 portes mesurent un mm^2 et qu'il est possible de placer un Mbits de mémoire dans 0.2537 mm^2. Les abréviations Hier et N-Hier correspondantes respectivement à une architecture hiérarchisée ou non.

D'après les chiffres obtenus, le surcoût du contrôle reste inférieur à 10% par rapport au reste de l'architecture. Le surcoût exact est de 1,1% au niveau cluster ce qui représente 1.1% sur l'ensemble de l'architecture considérée. Cette estimation permet de voir l'impact de la hiérarchisation.

Cette section montre un gain de 47% sur le temps d'exécution de l'application test grâce à la hiérarchisation. Cette hiérarchisation se fait au dépend d'un ajout de 32 processeurs. Cependant ce surcoût de 32 processeurs représente 6% en surface totale silicium. De plus la parallélisation du contrôle permet un gain de 90% en moyenne sur le temps moyen d'ordonnancement.

5.6 Conclusion

Nous avons vu, à travers ce chapitre, l'environnement de test implémenté afin de valider et de caractériser notre proposition. Cet environnement, se basant sur le simulateur SESAM du laboratoire, a permis de réaliser deux architectures avec des contrôles différents afin de faire les tests. Ces trois tests distincts montrent l'impact de notre solution sur les performances.

Ainsi le premier test a permis de mesurer l'impact du placement dynamique des groupes de tâches dans une structure hiérarchique, dans un scénario donné. Cette répartition dynamique permet de distribuer dynamiquement la charge de calcul. Le test a fait apparaître un gain de 30% sur les performances entre une solution statique et dynamique.

Cependan,t il est important de maîtriser le placement dynamique des tâches et des groupes de tâches à travers l'architecture sous peine de dégrader les performances. En effet si l'on place loin les tâches ou les groupes de tâches des données et du code à exécuter, les communications risquent de prendre une part trop importante sur le temps d'exécution de la tâche. D'après le test effectué, plus de 13% du temps total d'exécution de l'application peut être perdu.

Enfin, le point important de ce chapitre a été de démontrer l'intérêt de la hiérarchisation du contrôle. Le dernier test a permis de constater un gain de 47% sur le temps total d'exécution de l'application avec un contrôle hiérarchique par rapport à un contrôle centralisé. Ce gain est dû principalement à la diminution du nombre de tâches à gérer dans chaque cluster. On constate un gain d'environ 90% sur le temps moyen d'ordonnancement moyen durant l'exécution de l'application d'étiquetage. Nous avons également évalué à 6% l'impact de la hiérarchisation du contrôle sur la surface estimée de la puce.

Conclusions et Perspectives

Sommaire

Synthèse des travaux **135**
Perspectives **137**
A court terme 137
A long terme 138

Les dispositifs embarqués sont devenus de plus en plus polyvalents. Cette nécessité d'être polyvalent impose à ces dispositifs d'être flexibles afin de pouvoir supporter des applications de domaines applicatifs différents. Ces applications sont de plus en plus gourmandes en puissances de calcul. De plus elles sont devenues dynamiques avec des exécutions variables en fonction du contexte de l'application. Toutes ces contraintes s'ajoutent aux contraintes de consommation électrique et de surface silicium.

Il est donc nécessaire que les dispositifs embarqués offrent une grande puissance de calcul quelque soit l'application. Cette puissance doit être gérée dynamiquement afin d'optimiser au mieux l'utilisation et ainsi permettre d'approcher la puissance crête du multiprocesseur avec la puissance effective. C'est pourquoi cette étude s'est penchée sur la gestion dynamique des tâches et des données dans une structure massivement parallèle.

Synthèse des travaux

La première partie de cette étude a permis, à l'aide d'un état de l'art sur les multiprocesseurs, de déterminer que l'infrastructure de communication ainsi que la gestion dynamique des applications sont deux des points durs de la conception des architectures massivement parallèles. Nous nous sommes donc concentrés, dans un premier temps, sur le choix d'une interconnexion adaptée à l'augmentation du nombre de processeurs. Puis,simulations dans un second temps, nous nous sommes penchés sur le problème de la gestion dynamique des applications.

Le choix d'une interconnexion a commencé par une étude bibliographique de différents types d'interconnexion. Ce premier état de l'art conclut sur le fait que les réseaux sur puce sont de bons candidats. Ensuite un deuxième état de l'art a permis d'explorer les différents paramètres des réseaux sur puce. La topologie s'est avérée compliquée à évaluer par une étude papier c'est pourquoi nous avons décidé de réaliser des comparaisons par simulations.

Nous avons mis en place, pour cette étude, un environnement de simulation basé sur la librairie SystemC. Celui-ci propose une façon innovante de simuler des réseaux sur puce dans un seul thread SystemC. De plus, il est capable de simuler

n'importe quelle topologie en prenant en compte les contentions dans le réseau. Cet environnement permet d'accélérer les simulations d'un facteur compris entre 10 et 100 en fonction de la taille du système simulé, et avec une perte de précision limitée à environ 10% par rapport à un simulateur précis au cycle près. Il a fait l'objet d'une publication à la conférence DSD en 2009 [DSD09].

Les différents réseaux développés ont été ajoutés à l'environnement SESAM du laboratoire. Cet ajout offre à SESAM de nouvelles possibilités d'exploration. Les travaux sur SESAM ont été publiés à la conférence ICESS en 2010 [ICESS10].

L'état de l'art sur les topologies de réseau, nous a permis de choisir une série de topologies à plat et hiérarchiques. Une comparaison d'un point de vue performance et surface silicium a été réalisée afin de choisir la topologie la plus pertinente. Nous avons introduit une nouvelle façon de comparer l'efficacité silicium en combinant les résultats de performance et de surface. Nous avons conclu de cette étude que les solutions hiérarchisées proposent en général une meilleure efficacité silicium. De plus, l'augmentation du degré de connectivité des routeurs permet également un gain en performance en diminuant le nombre de routeurs. Le réseau Multibus + Anneau amélioré (réseau MX) a été sélectionné car il offre la meilleure efficacité silicium dans le cas d'un trafic localisé. L'ensemble de cette étude a fait l'objet d'une publication à la conférence NOCS en 2010 [NOCS10].

Une fois la structure de communication choisie, les travaux se sont focalisés sur la gestion en ligne d'un grand nombre de tâches. Ceci dans le but de gérer correctement une application dynamique sur notre structure hiérarchisée.

Dans cette optique, une étude bibliographique a été réalisée sur les algorithmes provenant du calcul haute performance utilisable avec les contraintes du domaine de l'embarqué. Nous avons extrait un certain nombre d'algorithmes connus. Comme nous cherchons à augmenter le nombre de processeurs dans la puce, nous avons posé comme hypothèse que si l'on réduit les coûts de communication cela permettra d'améliorer les performances. C'est dans ce but que nous avons proposé de nouveaux algorithmes permettant de placer les tâches au plus près de leurs données. Nous avons comparé les performances, les taux d'utilisation ainsi que le nombre de communications entre clusters de tous ces algorithmes. Nous avons conclu que dans le cas de graphes contenant beaucoup de dépendances, les algorithmes par liste obtiennent de bons résultats. Alors que dans le cas de graphes avec peu de dépendances entre les tâches, les algorithmes par cluster obtiennent de meilleures performances. Ces résultats ont été publiés à la conférence PDCN en 2010 [PDCN10].

Cette étude a conclu sur l'utilisation d'un algorithme combinant les avantages des algorithmes par groupe et par liste. Celui-ci obtient des performances comprises entre les solutions par liste et par groupe seules dans les cas de graphes parallèles et de graphes hautement connectés. Il est donc facilement utilisable dans toutes les situations. En plus de cet algorithme, nous avons mis en place toute la gestion de l'application et des données afin de proposer un modèle d'exécution adapté à une structure hiérarchique. Ce modèle d'exécution se caractérise par une gestion automatique de création de groupes hors-ligne ainsi que d'une gestion dynamique des groupes et des tâches dans l'architecture. Nous avons, par ailleurs, mis en place

un mécanisme d'affinité. Celui-ci permet au programmeur de contraindre l'exécution et le placement des groupes sur les clusters afin de guider la gestion dynamique dans le cas où le graphe d'application ne met pas en évidence des propriétés de celle-ci.

Afin de permettre l'utilisation du modèle d'exécution proposé, une architecture massivement parallèle a été créée. Elle propose une découpe en clusters contenant chacun des processeurs de calcul, des bancs mémoire, une parti du contrôle et un module de gestion mémoire. Un contrôleur central répartit dynamiquement des parties de l'application dans les clusters.

Afin de vérifier nos hypothèses et ainsi de valider nos propositions, l'architecture a été implémentée dans l'environnement SESAM du laboratoire. Ensuite en vue d'évaluer les performances, notre architecture a été comparée à une seconde basée sur la même infrastructure de communication possédant un contrôle centralisé unique. Ces comparaisons entre les deux architectures a fait apparaître que la gestion hiérarchisée du contrôle permet un gain de 90% sur les temps d'ordonnancement moyens. De plus au total, cette solution offre un gain de 50% sur le temps d'exécution global de l'application de test. Ce gain est dû au contrôle mais aussi à la meilleure répartition des tâches qui permet de diminuer les coûts de communication.

Ces travaux s'inscrivent aujourd'hui dans les activités du laboratoire en ayant réalisé une étude sur l'extension des multiprocesseurs actuels en processeurs massivement parallèles. De plus, ces travaux ont permis d'aboutir à un modèle d'exécution et à une architecture massivement parallèle.

Perspectives

A court terme

Afin de finaliser cette étude, un prototype matériel est nécessaire afin de caractériser la surface totale. Une étude sur la consommation basée sur le prototypage matériel afin de montrer l'efficacité énergétique de notre solution et ainsi justifier son utilisation dans le domaine de l'embarqué serait également intéressante. Pour des questions de temps, nous n'avons en effet pas eu le temps de réaliser le prototypage complet de l'architecture.

Dans le modèle utilisé pour la gestion mémoire, les TLB sont capables de mémoriser toutes les translations mémoires demandées par toutes les tâches exécutées durant l'exécution. Cette hypothèse simplifie le modèle mais elle est peu réalisable étant donné la quantité mémoire nécessaire. Il faudrait donc envisager de placer des caches à l'intérieur du TLB pour mémoriser les translations. De ce fait, la mémoire nécessaire serait réduite. Cependant ce genre de remplacement créera automatiquement plus de communications sur le réseau de contrôle afin de recharger des translations d'adresses effacées.

Afin de réduire ces communications, il searit également possible d'envisager de hiérarchiser le gestionnaire mémoire en ajoutant un gestionnaire global qui permettrait de connaître l'ensemble des translations d'adresse. Ceci introduira cependant un problème de surface pour mémoriser l'ensemble des translations. On pourrait

donc imaginer de la même façon mettre un système de cache dans le gestionnaire mémoire global qui ne contiendrait que les translations les plus demandées

Enfin ce genre de choix aurait un impact sur le réseau de contrôle global. Il serait donc nécessaire de choisir la solution la plus viable afin de réaliser une étude sur le réseau de contrôle. En effet, nous n'avons pas réalisé d'étude sur le choix de l'interconnexion pour le réseau de contrôle, or ce réseau influe sur les performances générales de l'architecture. Cette étude doit être réalisée car ce réseau doit supporter plusieurs types de communications (message court de contrôle et message long depuis la mémoire centrale). Il est donc nécessaire de choisir une interconnexion adaptée.

A long terme

Notre solution contient 32 clusters de calcul, soit 256 processeurs. Afin d'augmenter ce nombre de clusters, il est important de vérifier le comportement de l'allocation des groupes en avance de phase. Plus le nombre de clusters est important et plus le contrôle va potentiellement effectuer des placements non optimaux. En effet, avec l'augmentation du nombre de clusters suit l'augmentation du nombre de migrations. Ces migrations vont remettre en cause les pré-placements effectués en avance de phase. Il sera nécessaire d'ajuster ce pré-placement afin de ne pas réduire les performances de l'architecture lorsque le nombre de clusters augmentera.

L'aspect outil qui accompagne l'architecture est un point à ne pas négliger afin d'utiliser au mieux l'architecture. Le problème de la compilation d'applications pour des architectures massivement parallèles est un problème complexe et en partie lié à l'architecture visée. Dans notre cas de figure, cette compilation devra être capable de transformer le code initial et un graphe de tâches parallèles. Actuellement, les outils peuvent facilement extraire le parallélisme de données mais il est encore compliqué d'extraire le parallélisme dans le cas d'applications dynamiques.

A long terme, les solutions avec un contrôle centralisé sont vouées à disparaître si le nombre de processeurs continue à augmenter. En effet, notre solution hiérarchisée reste une solution intermédiaire vers des systèmes contenant plus de processeurs, car même s'il reste la possibilité d'augmenter le nombre de couches de hiérarchie dans le système, la latence induite par le passage de couche en couche risque de pénaliser les performances globales du système.

Il est donc plus envisageable de passer à un système distribué du contrôle qui propose une solution collaborative pour remplacer le modèle centralisé. Cette solution distribuée peut être répartie sur chaque processeur ou plus vraisemblablement par groupe de processeurs. Le passage du centralisé au distribué change complètement le modèle à envisager. Même si dans le domaine du calcul haute performance, cette problématique n'est pas récente et des solutions existent, ces solutions ne sont qu'en partie applicable au domaine de l'embarqué. Il reste donc encore beaucoup de recherches afin de trouver des solutions viables dans ce domaine.

Au sujet de la structure hiérarchisée des communications, il reste à observer comment elle se comporte avec un modèle d'exécution distribué. Mais on peut supposer qu'une structure de communication hiérarchisée pourra fournir de bonnes perfor-

mances. Cependant avec l'avènement du 3D dans le monde de l'embarqué, il sera nécessaire de reconsidérer les topologies pour prendre en compte ce nouveau degré de liberté.

L'avenir du multiprocesseur se fera certainement avec un modèle d'exécution distribué. Il est donc nécessaire qu'aussi bien les programmeurs que les concepteurs passent d'une vue séquentielle des problèmes à résoudre à une vue parallèle afin d'utiliser au mieux toutes les capacités offertes par les processeurs massivement parallèles.

Bibliographie

[1] B. C. Mochocki, K. Lahiri, S. Cadambi, and X. S. Hu, "Signature-based workload estimation for mobile 3D graphics," in *Proceedings of the 43rd annual Design Automation Conference*, 2006, pp. 592–597.

[2] S. Borkar, "Thousand core chips : a technology perspective," in *the 44th annual Design Automation Conference*, 2007, p. 746—749.

[3] Semiconductor Industry Association, "International Technology Roadmap for Semiconductors," 2007.

[4] N. Ventroux and R. David, "Les architectures parallèles sur puce," *journal Technique et Science Informatiques*, vol. 29, no. 3, pp. 345–378, 2010.

[5] A. Duller, G. Panesar, and D. Towner, "Parallel Processing – the picoChip way !" *Communicating Processing Architectures*, pp. 125–138, 2003.

[6] D. Truong, W. Cheng, T. Mohsenin, Z. Yu, A. Jacobson, G. Landge, M. Meeuwsen, C. Watnik, A. Tran, Z. Xiao, E. Work, J. Webb, P. Mejia, and B. Baas, "A 167-Processor Computational Platform in 65 nm CMOS," *IEEE Journal of Solid-State Circuits*, vol. 44, no. 4, pp. 1130–1144, 2009.

[7] T. R. Halfhill, "Ambric's New Parallel Processor," *Microprocessor Report*, 2006.

[8] N. Bayer and R. Ginosar, "High Flow-Rate Synchronizer/Scheduler Apparatus And Method For Multiprocessors," *United States Patent*, no. 5,202,987, 1993.

[9] TILERA, 2009. [Online]. Available : http://www.tilera.com/

[10] Y. P. Zhang, T. Jeong, F. Chen, H. Wu, R. Nitzsche, and G. Gao, "A study of the on-chip interconnection network for the IBM Cyclops64 multi-core architecture," in *20th International Symposium on Parallel and Distributed Processing*, April 2006, pp. 10 pp.–.

[11] ARM, "Amba specification and multi layer ahb specification (rev2.0)," Tech. Rep., 2001. [Online]. Available : http://www.arm.com

[12] STMicroelectronics, "Stbus communication system : Concepts and definitions," Tech. Rep., 2003.

[13] IBM, "On-chip coreconnect bus architecture specification (rev2.1)," Tech. Rep., 2001. [Online]. Available : http://www.chips.ibm.com/products/coreconnect/index

[14] W. Dally and B. Towles, "Route packets, not wires : on-chip interconnection networks," in *Design Automation Conference, 2001. Proceedings*, 2001, pp. 684–689.

[15] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-Vincentelli, "Addressing the system-on-a-chip interconnect woes through communication-based design," in *Design Automation Conference, 2001. Proceedings*, 2001, pp. 667–672.

[16] T. Bjerregaard and S. Mahadevan, "A survey of research and practices of network-on-chip," *ACM Comput. Surv.*, vol. 38, no. 1, p. 1, 2006.

[17] J.-M. Philippe, "Intégration des réseaux sur Silicium : optimization des performances des couches physique et liaison," Ph.D. dissertation, Ecole doctorale : mathématiques télécomunications informatique signal systèmes électronique, 2005.

[18] G. Stefănescu, *Network Algebra.* Secaucus, NJ, USA : Springer-Verlag New York, Inc., 2000.

[19] T. G. Robertazzi, *Computer networks and systems : queueing theory and performance evaluation.* New York, NY, USA : Springer-Verlag New York, Inc., 1990.

[20] J.-Y. L. Boudec and P. Thiran, *NETWORK CALCULUS : A Theory of Deterministic Queuing Systems for the Internet.* Springer-Verlag New York, Inc., 2004.

[21] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks.* San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2003.

[22] A. Duller, D. Towner, G. Panesar, A. Gray, and W. Robbins, "picoarray technology : the tool's story," in *Design, Automation and Test in Europe, 2005. Proceedings*, March 2005, pp. 106–111 Vol. 3.

[23] J. Liang, S. Swaminathan, and R. Tessier, "Asoc : a scalable, single-chip communications architecture," in *Parallel Architectures and Compilation Techniques, 2000. Proceedings. International Conference on*, 2000, pp. 37–46.

[24] K. Goossens, J. Dielissen, and A. Radulescu, "Aethereal network on chip : concepts, architectures, and implementations," *Design & Test of Computers, IEEE*, vol. 22, no. 5, pp. 414–421, Sept.-Oct. 2005.

[25] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar, "An 80-tile 1.28tflops network-on-chip in 65nm cmos," in *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International*, Feb. 2007, pp. 98–589.

[26] E. Salminen, A. Kulmata, and T. D. Hämäläinen, "Survey of network-on-chip proposals," March 2008.

[27] D. Sigüenza-Tortosa, T. Ahonen, and J. Nurmi, "Issues in the development of a practical noc : the proteo concept," *Integr. VLSI J.*, vol. 38, no. 1, pp. 95–105, 2004.

[28] S. Evain, "Spider :Environnement de Conception de Réseau sur Puce," Ph.D. dissertation, Institut Nationale des Sciences Appliquées de Rennes, 2006.

[29] M. Palesi, R. Holsmark, S. Kumar, and V. Catania, "A methodology for design of application specific deadlock-free routing algorithms for noc systems," in *CODES+ISSS '06 : Proceedings of the 4th international conference on Hardware/software codesign and system synthesis.* New York, NY, USA : ACM, 2006, pp. 142–147.

[30] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee : a many-core x86 architecture for visual computing," *ACM Trans. Graph.*, vol. 27, no. 3, pp. 1–15, 2008.

[31] T. Ainsworth and T. Pinkston, "Characterizing the cell eib on-chip network," *Micro, IEEE*, vol. 27, no. 5, pp. 6–14, Sept.-Oct. 2007.

[32] F. Karim, A. Nguyen, S. Dey, and R. Rao, "On-chip communication architecture for oc-768 network processors," in *DAC '01 : Proceedings of the 38th annual Design Automation Conference*. New York, NY, USA : ACM, 2001, pp. 678–683.

[33] M. Coppola, R. Locatelli, G. Maruccia, L. Pieralisi, and A. Scandurra, "Spidergon : a novel on-chip communication network," in *System-on-Chip, 2004. Proceedings. 2004 International Symposium on*, Nov. 2004, pp. 15–.

[34] E. Beigne, F. Clermidy, P. Vivet, A. Clouard, and M. Renaudin, "An asynchronous noc architecture providing low latency service and its multi-level design framework," in *Asynchronous Circuits and Systems, 2005. ASYNC 2005. Proceedings. 11th IEEE International Symposium on*, March 2005, pp. 54–63.

[35] A. Jantsch and H. Tenhunen, *Networks on Chip*. Kluwer Academic Publishers, February 2003.

[36] S. Kumar, A. Jantsch, J.-P. Soininen, M. Forsell, M. Millberg, J. Oberg, K. Tiensyrja, and A. Hemani, "A network on chip architecture and design methodology," in *VLSI, 2002. Proceedings. IEEE Computer Society Annual Symposium on*, 2002, pp. 105–112.

[37] B. Feero and P. Pande, "Networks-on-chip in a three-dimensional environment : A performance evaluation," *Computers, IEEE Transactions on*, vol. 58, no. 1, pp. 32–45, Jan. 2009.

[38] P. Guerrier and A. Greiner, "A generic architecture for on-chip packet-switched interconnections," in *Proceedings of the conference on Design, automation and test in Europe*. New York, NY, USA : ACM, 2000, pp. 250–256.

[39] M. Dall'Osso, G. Biccari, L. Giovannini, D. Bertozzi, and L. Benini, "Xpipes : a latency insensitive parameterized network-on-chip architecture for multi-processor socs," in *Computer Design, 2003. Proceedings. 21st International Conference on*, Oct. 2003, pp. 536–539.

[40] Arteris, "http ://www.arteris.com/."

[41] M. Hosseinabady, M. Kakoee, J. Mathew, and D. Pradhan, "Reliable network-on-chip based on generalized de bruijn graph," in *High Level Design Validation and Test Workshop, 2007. HLVDT 2007. IEEE International*, Nov. 2007, pp. 3–10.

[42] X. Leng, N. Xu, F. Dong, and Z. Zhou, "Implementation and simulation of a cluster-based hierarchical NoC architecture for multi-processor SoC," in *IEEE International Symposium on Communications and Information Technology*, vol. 2, 12-14 2005, pp. 1203–1206.

[43] A. Lankes, T. Wild, and A. Herkersdorf, "Hierarchical nocs for optimized access to shared memory and i/o resources," in *12th Euromicro Conference on Digital System Design*, 2009, pp. 255–262.

[44] D. Tutsch and M. Malek, "Comparison of network-on-chip topologies for multicore systems considering multicast and local traffic," in *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, 2009, pp. 1–9.

[45] S. Murali and G. De Micheli, "SUNMAP : a tool for automatic topology selection and generation for NoCs," in *Proceedings of the 41st Design Automation Conference*, 2004, pp. 914–919.

[46] L. Ni and P. McKinley, "A survey of wormhole routing techniques in direct networks," *Computer*, vol. 26, no. 2, pp. 62–76, Feb 1993.

[47] C. Glass and L. Ni, "The turn model for adaptive routing," in *Computer Architecture, 1992. Proceedings., The 19th Annual International Symposium on*, 1992, pp. 278–287.

[48] G.-M. Chiu, "The odd-even turn model for adaptive routing," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 11, no. 7, pp. 729–738, Jul 2000.

[49] A. Mello, L. Tedesco, N. Calazans, and F. Moraes, "Virtual channels in networks on chip : Implementation and evaluation on hermes noc," in *Integrated Circuits and Systems Design, 18th Symposium on*, Sept. 2005, pp. 178–183.

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

[51] J. Hu, U. Y. Ogras, and R. Marculescu, "System-level buffer allocation for application-specific networks-on-chip router design," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 12, pp. 2919–2933, December 2006. [Online]. Available : http://www.gigascale.org/pubs/978.html

[52] A. Mello, L. Tedesco, N. Calazans, and F. Moraes, "Evaluation of current qos mechanisms in networks on chip," in *System-on-Chip, 2006. International Symposium on*, Nov. 2006, pp. 1–4.

[53] A. A. Jerraya, A. Bouchhima, and F. Pétrot, "Programming models and HW-SW interfaces abstraction for multi-processor SoC," in *DAC '06 : Proceedings of the 43rd annual Design Automation Conference*, 2006, pp. 280–285.

[54] Synopsys, "Using Virtual Platforms for Pre-Silicon Software Development," Tech. Rep., 2008.

[55] T. Grötker, *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[56] L. Yu, S. Abdi, and D. Gajski, "Transaction level platform modeling in systemc for multi-processor designs," Tech. Rep., 2007.

[57] L. Cai and D. Gajski, "Transaction level modeling : an overview," in *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS)*, 2003.

[58] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics : A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50–58, Feb 2002.

[59] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta, "Complete computer system simulation : The simos approach," *IEEE Parallel and Distributed Technology*, vol. 3, pp. 34–43, 1995.

[60] V. S. Pai, P. Ranganathan, and S. V. Adve, "RSIM : An Execution-Driven Simulator for ILP-Based Shared-Memory Multiprocessors and Uniprocessors," in *Proceedings of the Third Workshop on Computer Architecture Education*, 1997.

[61] D. Perez, G. Mouchard, and O. Temam, "Microlib : A case for the quantitative comparison of micro-architecture mechanisms," in *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*, Dec. 2004, pp. 43–54.

[62] J. Renau, B. Fraguela, W. Tuck, M. Prvulovic, and L. Ceze, "SESC simulator," 2005. [Online]. Available : http://sesc.sourceforge.net

[63] D. August, J. Chang, S. Girbal, D. Gracia-Perez, G. Mouchard, D. Penry, O. Temam, and N. Vachharajani, "UNISIM : An Open Simulation Environment and Library for Complex Architecture Design and Collaborative Development," *Computer Architecture Letters*, 2007.

[64] V. Reyes, T. Bautista, G. Marrero, P. Carballo, and W. Kruijtzer, "CASSE : a system-level modeling and design-space exploration tool for multiprocessor systems-on-chip," *Digital System Design, 2004. DSD 2004. Euromicro Symposium on*, 2004.

[65] W. Cesario, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A. Jerraya, L. Gauthier, and M. Diaz-Nava, "Multiprocessor soc platforms : a component-based design approach," *Design & Test of Computers, IEEE*, vol. 19, no. 6, pp. 52–63, Nov/Dec 2002.

[66] K. Huang, S.-i. Han, K. Popovici, L. Brisolara, X. Guerin, L. Li, X. Yan, S.-l. Chae, L. Carro, and A. A. Jerraya, "Simulink-based mpsoc design flow : case study of motion-jpeg and h.264," in *DAC '07 : Proceedings of the 44th annual Design Automation Conference*, 2007, pp. 39–42.

[67] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *SIGARCH Comput. Archit. News*, 2005.

[68] Open SystemC Initiative OSCI, "SystemC Documentation," 2004. [Online]. Available : http://www.systemc.org

[69] "Transaction-level Modeling Working Group." [Online]. Available : http://www.systemc.org/

[70] G. Beltrame, C. Bolchini, L. Fossati, A. Miele, and D. Sciuto, "ReSP : A non-intrusive Transaction-Level Reflective MPSoC Simulation Platform for design space exploration," 2008.

[71] M. Palesi, D. Patti, and F. Fazzino, "Noxim." [Online]. Available : http://noxim.sourceforge.net

[72] V. Puente, J. Gregorio, and R. Beivide, "Sicosys : an integrated framework for studying interconnection network performance in multiprocessor systems," in *Parallel, Distributed and Network-based Processing, 2002. Proceedings. 10th Euromicro Workshop on*, 2002, pp. 15–22.

[73] H. Hossain, M. Ahmed, A. Al-Nayeem, T. Islam, and M. Akbar, "Gpnocsim - a general purpose simulator for network-on-chip," in *Information and Communication Technology, 2007. ICICT '07. International Conference on*, March 2007, pp. 254–257.

[74] L.-S. P. Niket Agarwal, Tushar Krishna and N. K. Jha, "Garnet : A detailed on-chip network model inside a full-system simulator," in *IEEE International Symposium on Performance Analysis of Systems and Software*, April 2009.

[75] Z. Lu and A. Jantsch, "Traffic configuration for evaluating networks on chips," in *System-on-Chip for Real-Time Applications, 2005. Proceedings. Fifth International Workshop on*, July 2005, pp. 535–540. [Online]. Available : http://www.ict.kth.se/nostrum/NNSE/

[76] V. Puente, J. Gregorio, and R. Beivide, "Sicosys : an integrated framework for studying interconnection network performance in multiprocessor systems," in *Parallel, Distributed and Network-based Processing, 2002. Proceedings. 10th Euromicro Workshop on*, 2002, pp. 15–22.

[77] "Mentor Graphics." [Online]. Available : http://www.mentor.com/

[78] "Coware." [Online]. Available : http://www.coware.com/

[79] J. Emer, P. Ahuja, E. Borch, A. Klauser, C.-K. Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, and T. Juan, "Asim : A Performance Model Framework," *Computer*, vol. 35, no. 2, pp. 68–76, 2002.

[80] J. Cong, K. Gururaj, G. Han, A. Kaplan, M. Naik, and G. Reinman, "MC-Sim : An efficient simulation tool for MPSoC designs," in *Computer-Aided Design, 2008. ICCAD 2008. IEEE/ACM International Conference on*, Nov. 2008, pp. 364–371.

[81] L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivieri, "MPARM : Exploring the Multi-Processor SoC Design Space with SystemC," *J. VLSI Signal Process. Syst.*, vol. 41, no. 2, pp. 169–182, 2005.

[82] "SoClib." [Online]. Available : https://www.soclib.fr/

[83] L. Charest, E. Aboulhamid, C. Pilkington, P. Paulin, and C. STMicroelectronics, "SystemC performance evaluation using a pipelined DLX multiprocessor," in *IEEE Design Automation and Test in Europe (DATE)*, 2002.

[84] G. Schelle and D. Grunwald, "Onchip Interconnect Exploration for Multicore Processors Utilizing FPGAs," in *Proceedings of the 2nd Workshop on Architecture Research using FPGA Platforms*, 2006. [Online]. Available : http://ccis.colorado.edu/mediawiki/index.php/NoCem

[85] N. Ventroux, "Contrôle en ligne des systèmes multiprocesseurs hétérogènes embarqués : élaboration et validatioon d'une architecture," Ph.D. dissertation, Université de Rennes 1 (MATISSE), septembre 2006.

[86] O. Sinnen, *Task Scheduling for Parallel Systems.* John Wiley, 2007.

[87] J. Sgall, "On-Line Scheduling - A Survey," 1997.

[88] C. C. Lee and D. T. Lee, "A simple on-line bin-packing algorithm," *Journal ACM*, vol. 32, no. 3, pp. 562–572, 1985.

[89] N. Ventroux, F. Blanc, and D. Lavenier, "A Low Complex Scheduling Algorithm for Multi-processor System-on-Chip," in *Proceedings of Parallel and Distributed Computing and Networks*, 2005, pp. 540–545.

[90] E. Wenzel Briao, D. Barcelos, and F. Wagner, "Dynamic Task Allocation Strategies in MPSoC for Soft Real-time Applications," in *Proceedings of Design, Automation and Test in Europe*, 2008, pp. 1386–1389.

[91] J. J. Chew, "NUMA project : Memory Placement Optimzation." [Online]. Available : http://hub.opensolaris.org/bin/view/Project+numa/

[92] M. Rajagopalan, B. T. Lewis, and T. A. Anderson, "Thread scheduling for multi-core platforms," in *HOTOS'07 : Proceedings of the 11th USENIX workshop on Hot topics in operating systems.* Berkeley, CA, USA : USENIX Association, 2007, pp. 1–6.

[93] M. R. Garey and D. S. Jonhson, *Computers and Intractability - A Guide to the Theory of NP-Completeness.* W.H. Freeman, 1979.

[94] Y. Azar, A. Broder, and A. Karlin, "On-line load balancing," in *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science*, 1992, pp. 218–225.

[95] A. Zomaya, C. Ward, and B. Macey, "Genetic scheduling for parallel processor systems : comparative studies and performance issues," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 8, pp. 795–812, 1999.

[96] N. Auluck and D. Agrawal, "A scalable task duplication based algorithm for improving the schedulability of real-time heterogeneous multiprocessor systems," in *Proceedings of the International Conference on Parallel Processing Workshops*, 2003, pp. 89–96.

[97] R. Dick, D. Rhodes, and W. Wolf, "TGFF : task graphs for free," in *Proceedings of the Sixth International Workshop on Hardware/Software Codesign*, 1998.

[98] M. Rajagopalan, B. T. Lewis, and T. A. Anderson, "Thread scheduling for multi-core platforms," in *Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, 2007, pp. 1–6.

[99] F. Chang, C.-J. Chen, and C.-J. Lu, "A linear-time component-labeling algorithm using contour tracing technique," *Comput. Vis. Image Underst.*, vol. 93, no. 2, pp. 206–220, 2004.

[100] N. Ventroux, A. Guerre, T. Sassolas, L. Moutaoukil, C. Bechara, and R. David, "SESAM :an MPSoC Simulation Environment for Dynamic Application Processing," in *Proceedings of IEEE Internationnal Conference on Embedded Software and Systems*, 2010.

Publications personnelles

Conférences internationales

[DSD09] Alexandre Guerre, Nicolas Ventroux, Raphaël David, Alain Mérigot, *Approximate-Timed Transactional Level Modeling for MPSoC Exploration : A Network-on-Chip Case Study*, in the 12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools (DSD), pp 390-397, Patras, August2009

[PDCN10] Alexandre Guerre, Nicolas Ventroux, Raphaël David, Alain Mérigot, *Low-Complex Task Scheduling Algorithms for Hierarchical Embedded Many-Core Architectures and Dynamic Applications*, in the Ninth IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN), Innsbruck, February 2010.

[ICESS10] Nicolas Ventroux, Alexandre Guerre, Tanguy Sassolas, Larbi Moutaoukil, Charly Bechara and Raphaël David, *SESAM : an MPSoC Simulation Environment for Dynamic Application Processing*, in the IEEE International Conference on Embedded Software and Systems (ICESS), Bradford, July 2010

[NOCS10] Alexandre Guerre, Nicolas Ventroux, Raphaël David, Alain Mérigot, *Hierarchical Network-on-Chip for Embedded Many-Core Architectures*, in the 4th ACM/IEEE International Symposium on Networks-on-Chip (NOCS), Grenoble, Mai 2010

Résumé : Afin de fournir la puissance de calcul nécessaire, les monoprocesseurs ont atteint une limite au niveau du parallélisme d'instructions et au niveau de l'efficacité transistor. Il est devenu plus économique en consommation et en surface d'associer plusieurs éléments de calcul afin de fournir la puissance de calcul demandée.

Des solutions actuelles répondent au problème de l'embarqué, cependant un problème de mise à l'échelle apparait lorsque l'on veut augmenter le nombre d'éléments de calcul. On peut distinguer deux approches dans les structures actuelles : celles contenant un contrôle central et celles dont le contrôle est distribué. Les deux approches gèrent difficilement le dynamisme des applications lorsque le nombre de processeurs augmente.

Au cours de cette étude, nous nous sommes attardés sur deux points durs que sont la communication au sein de l'architecture ainsi que la gestion dynamique des applications dans le système. La résolution de ces deux points, nous a permis d'aboutir à un modèle d'exécution et à une architecture massivement parallèle.

Les communications au cœur d'un système massivement parallèle ont été étudiées à l'aide d'un simulateur réalisé au cours de cette thèse. Le simulateur a permis d'évaluer le comportement de différentes solutions de réseaux sur puce afin de déterminer lequel est le plus performant. Une comparaison au niveau de leur efficacité silicium a également été réalisée et a conclu sur l'efficacité des solutions hiérarchiques.

La gestion dynamique des applications sur un grand nombre de cœur est un problème connu dans le monde du calcul haute performance mais peu dans celui de l'embarqué. Une comparaison de différentes propositions visant à réduire les communications entre clusters a été réalisée. Cette étude fait apparaitre que la création et la gestion de groupe linéaire de tâche permet de réduire les communications entre clusters et ainsi augmente les performances.

Nous proposons donc un modèle d'exécution pour des structures massivement parallèles hiérarchiques contenant plus de 100 processeurs. Cette solution permet de réduire les coûts de contrôle par rapport à une solution centralisée et de répartir dynamiquement sur toute la puce les différentes tâches contrairement à une solution distribuée.

L'architecture associée se base sur l'assemblage de clusters composés de processeurs, de mémoires, d'un gestionnaire mémoire ainsi que d'un contrôleur cluster. Tous ces clusters sont assemblés autour d'une interconnexion centrale et commandés par un contrôleur central. L'ensemble de la mémoire est logiquement partagé et physiquement distribué, donc l'ensemble des processeurs ont accès à toutes les mémoires.

Nous validons les points durs du modèle d'exécution. Nous montrons également que la structure de communication hiérarchique permet d'augmenter les performances dans le cas d'un placement des tâches près des données. La hiérarchisation du contrôle permet un gain de 90% sur les temps d'ordonnancement du fait de la parallélisation de ce contrôle, malgré un surcout en surface de 1% sur l'architecture total dans le cas où elle contient 256 processeurs.

Mots clés : Multi-Processeur, Many-processeur, réseau sur puce, ordonnacement faible compléxité, architecture hérarchique