INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque

THESE

pour obtenir le grade de

DOCTEUR DE L'INPG

Spécialité : Informatique
Option : Systèmes et communication

préparée au laboratoire **TIMA** dans le cadre de l'Ecole Doctorale

"MATHEMATIQUES, SCIENCE ET TECHNOLOGIE DE L'INFORMATION"

Présentée et soutenue publiquement par

Ferid Gharsalli

le 1 juillet 2003

Conception des interaces logiciel-matériel pour l'intégration des mémoires globales dans les systèmes monopuces

Directeur de thèse
Ahmed Amine Jerraya
Co-directeur de thèse
Frédéric Rousseau

JURY

Guy Mazaré	**Président**
Anne Mignotte	**Rapporteur**
Habib Mehrez	**Rapporteur**
Ahmed Jerraya	**Directeur de thèse**
Frédéric Rousseau	**Co-Directeur de thèse**

AVANT-PROPOS

Le travail de cette thèse a été réalisé au sein du groupe SLS du laboratoire TIMA de Grenoble. Je remercie Monsieur Bernard Courtois, Directeur de recherche au CNRS et Directeur du laboratoire TIMA pour m'avoir donné la chance d'effectuer mon travail de recherche.

Je tiens à exprimer mes plus sincères remerciements à Monsieur Ahmed Amine Jerraya, Directeur de recherche au CNRS et responsable du groupe SLS du laboratoire TIMA pour m'avoir accepté dans son groupe et pour avoir encadré cette thèse. Je le remercie pour sa disponibilité et pour ses conseils pendant les moments les plus difficiles tout au long de ces années de recherche. Qu'il trouve ici, l'expression de ma profonde reconnaissance.

De même, j'exprime ma plus grande gratitude à Monsieur Frédéric Rousseau, Maître de Conférences à l'université Joseph Fourier de Grenoble pour son encadrement, sa patience et sa disponibilité. Je le remercie également pour l'aide qu'il m'a apporté durant quatre ans de travail. J'espère que cette thèse soit à la hauteur de ses efforts.

Je remercie Monsieur Guy Mazaré, Professeur à l'école ENSIMAG de m'honorer en présidant le jury de ma thèse. Merci à Madame Anne Mignote, Professeur à l'université de Lyon, et Monsieur Habib Mehrez, Professeur à l'université de Pierre et Marie Curie pour avoir bien voulu juger cette thèse en acceptant d'en être les rapporteurs.

Je souhaite exprimer ma reconnaissance à Monsieur Mounir Zrigui, Maître de Conférences à la faculté des sciences de Monastir pour m'avoir encouragé à effectuer mon doctorat et pour son soutien moral.

Ensuite, je tiens à remercier tous mes amis et mes collègues du laboratoire TIMA pour leur leurs encouragements et leur aide.

Enfin, je dédie cette thèse à mes parents, à Imen et à toute ma famille.

RESUME

Grâce à l'évolution de la technologie des semi-conducteurs, aujourd'hui on peut intégrer sur une seule puce ce qu'on mettait sur plusieurs puces ou cartes il y a une dizaine d'années. Dans un futur proche, cette évolution permettra l'intégration de plus de 100 Mbits de DRAM et 200 millions de portes logiques dans la même puce. D'après les prévisions de l'association d'industrie de semi-conducteur et d'ITRS, les mémoires embarquées continueront de dominer la surface des systèmes monopuces dans les années qui viennent, à peu près 94 % de la surface totale en 2014.

La conception à base de réutilisation d'IP mémoire est survenue pour réduire le fossé entre cette grande capacité d'intégration et la faible production de mémoire. Cette solution peut être idéale dans le cas d'une architecture homogène où tous les éléments ont les mêmes interfaces et utilisent les mêmes protocoles de communication, ce qui n'est pas le cas pour les systèmes monopuces. Pour rendre cette solution efficace, le concepteur doit consacrer beaucoup d'efforts pour la spécification et l'implémentation des interfaces logiciel-matériel. Vu la pression du temps de mise sur le marché (" time to market "), l'automatisation de la conception de ces interfaces d'adaptation est devenue cruciale.

La contribution de cette thèse concerne la définition d'une méthode systématique permettant la conception des interfaces logiciel-matériel spécifiques aux mémoires globales. Ces interfaces correspondent à des adaptateurs matériels flexibles connectant la mémoire au réseau de communication, et à des pilotes d'accès adaptant le logiciel de l'application aux processeurs cibles. Des expériences sur des applications de traitement d'images ont montré un gain de temps de conception important et ont prouvé la flexibilité de ces interfaces ainsi que leur faible surcoût en surface et en communication.

TITRE EN ANGLAIS

HARDWARE-SOFTWARE INTERFACE DESIGN FOR GLOBAL MEMORY INTEGRATION IN SYSTEM ON A CHIP

ABSTRACT

Embedded memory is becoming a new paradigm allowing entire systems to be built on a single chip. In the near future, new developments in process technology will allow the integration of more than 100 Mbits of DRAM and 200 millions gates of logic onto the same chip. According to Semiconductor Industry Association and ITRS prevision, embedded memory will continue to dominate SoC content in the next several years, approaching 94% of the die area by year 2014.

Memory Reuse based design is emerging to close the gap between this steadily increasing capacity and the design productivity in terms of designed transistors per time unit. This solution can be ideal in the case of homogeneous architecture where all the components have the same interfaces and use the same communication protocols, which is not the case for system on chip. However, the integration of several memory IP into a system on a chip makes specification and implementation of hardware-software interfaces a dominant design problem. Indeed, memory interface design is still hand-made, it is time-consuming and it is error prone. For these raisons, the design automation of these memory interfaces becomes crucial.

The contribution of this thesis consists on systematic method of hardware-software interfaces design for global memory. These interfaces correspond to flexible hardware wrappers connecting the memory to the communication network, and software drivers adapting the application software to the target processors. Experiments on image processing applications confirmed a saving of significant design time and proved the flexibility as well as the weak communication and the area overhead.

SPECIALITE Informatique

MOTS CLES

Systèmes monopuces, mémoire, architecture logiciel-matériel, interface logiciel-matériel, adaptateur matériel, pilote logiciel, conception à base de composants, génération automatique, bibliothèque générique.

INTITULE ET ADRESSE DU LABORATOIRE

Laboratoire **TIMA**, Techniques de l'Informatique et de la Micro-électronique pour l'Architecture des ordinateurs.
46 Avenue Félix Viallet, 38031 Grenoble, France.

Sommaire

Chapitre 1 : INTRODUCTION 12

1. Contexte de la thèse 13
   1.1. Les systèmes multiprocesseurs monopuces spécifiques à une application donnée 13
   1.2. Intégration des mémoires dans un système multiprocesseur monopuce 16
2. Problématique d'intégration des mémoires dans un système monopuce 18
   2.1. L'insuffisance du niveau RTL pour la conception des interfaces logiciel-matériel 18
   2.2. Variété des interfaces mémoire et manque de flexibilité d'utilisation 19
   2.3. Faible production et réutilisation des pilotes d'accès 19
   2.4. Validation des interfaces logiciel-matériel 20
3. Causes et conséquences des problèmes d'intégration mémoire 21
   3.1. Conception de haut niveau sans lien avec le niveau RTL 21
   3.2. Utilisation des interfaces mémoire " produits maison " malgré les efforts de standardisation des interfaces matérielles 21
   3.3. Satisfaction de la conception des systèmes multiprocesseurs classiques par une faible production du logiciel dépendant du matériel 21
   3.4. Faiblesse de la validation d'interfaces 22
4. Solutions proposées pour la résolution des problèmes d'intégration mémoire 22
   4.1. Modèle de description mémoire de haut niveau 22
   4.2. Abstraction de la communication et génération automatique des interfaces 23
   4.3. Abstraction du matériel pour augmenter la portabilité des pilotes d'accès mémoire 23
   4.4. Génération automatique des programmes de tests pour la validation des interfaces 23
5. Contributions 23
   5.1. Proposition d'un modèle mémoire de haut niveau 24
   5.2. Génération automatique d'adaptateurs mémoire matériels 24
   5.3. Génération automatique des programmes de tests des interfaces 25
   5.4. Génération automatique d'adaptateurs mémoire logiciels 26
6. Plan de la thèse 26

Chapitre 2 : LES MEMOIRES DANS LES SYSTEMES MONOPUCES ET L'ETAT DE L'ART SUR LEUR INTEGRATION 27

1. Introduction : flot de conception système 28
2. Etat de l'art sur l'intégration mémoire aux différents niveaux d'abstraction du flot de conception mémoire 29
   2.1. Optimisations mémoire au niveau fonctionnel : transformation de code indépendamment de l'architecture 30
   2.2. Synthèse d'une architecture mémoire : assignation et allocation 31
   2.3. Transposition des mémoires logiques en mémoires physiques : choix de composants 33
   2.4. Adaptation des mémoires au reste d'un système monopuce 34
3. Conclusion 44

Chapitre 3 : INTRODUCTION A LA CONCEPTION DES INTERFACES LOGICIEL-MATERIEL POUR LES SYSTEMES MONOPUCES 45

1. Introduction : les interfaces logiciel-matériel 46
   1.1. Définition des interfaces logiciel-matériel 46
   1.2. Les différents modèles d'interfaces et de signalisations entre le logiciel et le matériel 47

2. Niveaux d'abstraction pour la conception d'interfaces logiciel-matériel 49
2.1. Niveau transactionnel 49
2.2. Niveau architecture virtuelle 49
2.3. Niveau micro-architecture 49
3. Abstraction des détails de réalisation du matériel 50
4. Modèle d'architecture virtuelle pour les systèmes multiprocesseurs monopuces 50
4.1. Module virtuel 52
4.2. Port virtuel d'un module mémoire 52
4.3. Canal virtuel de communication 53
5. COLIF : modèle de représentation pour la spécification des interfaces logiciel-matériel 53
5.1. Modélisation de Colif 54
5.2. Implémentation de Colif 54
6. Flot de conception SLS des systèmes monopuces avant son extension 56
6.1. Spécification d'entrée du flot de conception (décrite au niveau transactionnel) 57
6.2. Sortie du flot 58
6.3. Les différentes étapes du flot de conception 58
7. Extension du flot de conception "SLS" 60
8. Modèles mémoire aux différents niveaux d'abstraction 62
8.1. Modèle mémoire au niveau transactionnel 62
8.2. Modèle mémoire au niveau architecture virtuelle 63
8.3. Modèle mémoire au niveau micro-architecture 64
9. Conclusion 65

Chapitre 4 : ARCHITECTURE GENERIQUE DES ADAPTATEURS MEMOIRE MATERIELS 66

1. Introduction 67
1.1. Vue conceptuelle des adaptateurs mémoire matériels 67
1.2. Architecture générique de l'adaptateur mémoire matériel 68
2. Adaptateur de port mémoire (MPA) 69
2.1. Rôle 69
2.2. Fonctionnalité 69
2.3. Implémentation 74
3. Adaptateur de canal (CA) 74
3.1. Rôle 75
3.2. Fonctionnalité 76
3.3. Implémentation 77
4. Bus interne 80
4.1. Rôle 80
4.2. Fonctionnalité 80
4.3. Implémentation 81
5. Les avantages et les inconvénients du modèle d'adaptateur mémoire matériel 81
6. Conclusion 82

Chapitre 5 : GENERATION AUTOMATIQUE DES ADAPTATEURS MEMOIRE MATERIELS 83

1. Introduction 84
1.1. Rappel sur les objectifs de la génération automatique des adaptateurs mémoire matériels 84
1.2. Principe de la génération automatique 84
1.3. Les objets requis pour la réalisation des adaptateurs mémoire matériels 84
1.4. Résultats attendus 85
2. Outil de génération d'adaptateurs matériels pour les processeurs 85
2.1. Entrées de l'outil 85

2.2. Sortie de l'outil ... 88
2.3. Etapes de l'outil ... 88
3. Les limites de l'environnement de génération d'interfaces ... 89
4. Génération automatique des adaptateurs mémoire matériels ... 90
4.1. Générateur d'architectures internes en Colif ... 92
4.2. Génération du code pour les mémoires, les CA et les MPA ... 95
4.3. Génération automatique des programmes de tests ... 98
5. Application : filtre d'image de bas niveau ... 99
5.1. Choix de l'application ... 99
5.2. But de l'application ... 100
5.3. Domaine d'utilisation ... 100
5.4. Description de l'application ... 101
5.5. Architecture abstraite de l'application ... 101
5.6. Architecture matérielle ... 102
5.7. Spécification VADeL et Architecture virtuelle ... 102
5.8. Génération automatique ... 108
5.9. Analyse des résultats ... 108
6. Conclusion ... 111

Chapitre 6 : GENERATION DES PILOTES EN COUCHES D'ACCES MEMOIRE ... 113

1. Introduction : conception des pilotes logiciels ... 114
1.1. Rappel de la nécessité des adaptateurs mémoire logiciels ... 114
1.2. Les objets requis pour la conception des pilotes logiciels d'accès mémoire ... 114
1.3. Résultats attendus ... 115
2. Architecture en couches d'un pilote logiciel monopuce ... 115
2.1. Architecture logicielle d'un système monopuce ... 115
2.2. Architecture du HAL : pilotes en couches ... 116
2.3. Exemple d'un pilote d'accès mémoire ... 119
3. Flot de génération systématique des pilotes logiciels ... 120
3.1. Entrée du flot ... 120
3.2. Bibliothèque du flot ... 121
3.3. Sorties du flot ... 122
3.4. Etapes du flot ... 122
3.5. Génération automatique ... 122
4. Application ... 123
4.1. Description de l'application ... 123
4.2. Architecture cible ... 123
4.3. Spécification ... 124
4.4. Génération automatique des pilotes en couches ... 127
4.5. Analyse des résultats ... 127
5. Conclusion ... 128

Chapitre 7 : CONCLUSION ET PERSPECTIVES ... 129

1. Conclusion ... 129
2. Perspectives ... 132

Bibliographie ... 133

PUBLICATIONS ... 141

Liste des Figures

Figure 1. Architecture d'un système multiprocesseur monopuce 13
Figure 2. Structure en couches d'un système multiprocesseur monopuce spécifique à une application donnée 14
Figure 3. Prévision d'ITRS 2000 pour la capacité d'intégration mémoire dans les systèmes monopuces 16
Figure 4. Les contributions liées aux interfaces logiciel-matériel des mémoires : (a) architecture sans mémoire, (b) architecture avec mémoire 24
Figure 5. Une vue globale sur les quatre types de travaux liés aux mémoires 29
Figure 6. Transformation de données et de flot de contrôle avant la compilation 30
Figure 7. Synthèse de mémoires logiques spécifiques à une application donnée par la méthode DTSE 32
Figure 8. Flot de génération d'architecture de Coware 35
Figure 9. Chinook : synthèse d'interfaces 38
Figure 10. Les interfaces matérielles utilisant le standard VCI 40
Figure 11. Les différents types de pilotes de périphérique classiques utilisés pour les PC 41
Figure 12. Architecture I2O d'un pilote logiciel 43
Figure 13. Les adaptateurs mémoire logiciels et matériels 46
Figure 14. Exemple de signalisation logiciel-matériel par interruption matérielle 48
Figure 15. Architecture logicielle portable et flexible 50
Figure 16. Les concepts de la spécification et de l'implémentation des interfaces pour les systèmes hétérogènes : (a) concept conventionnel, (b) concept d'enveloppe, (c) implémentation 51
Figure 17. Vue conceptuelle des modules virtuels : (a) module de calcul, (b) module de mémorisation 52
Figure 18. Vue interne d'un port virtuel de module mémoire 53
Figure 19. Diagramme de classes de Colif 55
Figure 20. Une vue globale et simplifiée du flot de conception du groupe SLS 56
Figure 21. Flot de conception des systèmes monopuces du groupe SLS : (a) avant l'extension, (b) extensions matérielles, (c) extensions logicielles 61
Figure 22. Modèle mémoire au niveau transactionnel 62
Figure 23. Modèle mémoire au niveau architecture virtuelle 63
Figure 24. Modèle mémoire au niveau micro-architecture 64
Figure 25. Vue globale d'un adaptateur mémoire matériel 67
Figure 26. Architecture interne d'un adaptateur mémoire matériel 68
Figure 27. Adaptation des interfaces entre la mémoire et le protocole du bus interne 70

Figure 28. Exemples de configuration de la machine d'états finis du module MPA : (a) exemple 1, (b) exemple 2 et (c) exemple 3 72
Figure 29. Vue conceptuelle de la fonctionnalité d'un CA 75
Figure 30. Les différents modèles d'adaptateur de canal : (a) CA_W, (b) CA_R, et (c) CA_RW 76
Figure 31. La partie contrôle d'un adaptateur de canal 77
Figure 32. Architecture interne d'un adaptateur de canal d'écriture de type CA_W_MemFullHndshk_Register 78
Figure 33. Une partie du code SystemC d'un CA_W_ Register_MemFullHndshk générée automatiquement. 79
Figure 34. Vue conceptuelle du bus interne de l'adaptateur mémoire matériel 81
Figure 35. Outil de génération d'adaptateurs matériels pour les processeurs 86
Figure 36. Architecture interne d'un nœud de calcul 87
Figure 37. Architecture interne d'un nœud de mémorisation globale 90
Figure 38. Environnement de génération des interfaces adapté au cas des mémoires globales 91
Figure 39. Architecture interne spécifique à une mémoire SDRAM 93
Figure 40. Décomposition fonctionnelle du générateur d'architectures internes 94
Figure 41. Programme de test d'un CA : (a) macro-modèle, (b) code généré 99
Figure 42. Image médicale avant et après l'utilisation d'un filtre 100
Figure 43. Architecture fonctionnelle d'un JAMCAM 101
Figure 44. Architecture fonctionnelle de l'application 102
Figure 45. Architecture cible 102
Figure 46. Architecture virtuelle de la spécification VADeL : (a) expérience 1, (b) expérience 2 103
Figure 47. Code VADeL décrivant le canal virtuel VC1 106
Figure 48. Programme principal VADeL 107
Figure 49. Micro-architecture des adaptateurs mémoire matériels générés : (a) expérience 1, (b) expérience 2 109
Figure 50. Résultat de synthèse d'un adaptateur de canal 110
Figure 51. Résultat de la synthèse d'un adaptateur de port mémoire 111
Figure 52. Architecture logiciel-matériel d'un système monopuce 116
Figure 53. Architecture en couches d'un pilote logiciel 117
Figure 54. Pilote en couches d'accès mémoire 119
Figure 55. Flot de génération de pilote logiciel 120
Figure 56. Exemple d'un élément macro du pilote écrit en RIVE 121
Figure 57. Architecture cible de l'application 123
Figure 58. Architecture virtuelle de l'application 124
Figure 59. Appels des APIs d'accès mémoire par les tâches de l'application 125

Figure 60. Un extrait de la spécification de l'application.....126

Liste des tableaux

Tableau 1. Description de l'interface d'un adaptateur de port mémoire spécifique à une mémoire SDRAM... 71
Tableau 2. Paramètres de configuration de macro-modèles mémoire... 96
Tableau 3. Paramètres de configuration d'un macro-modèle d'adaptateurs de canal... 97
Tableau 4. Paramètres de configuration d'un macro-modèle d'adaptateurs de port mémoire... 98
Tableau 5. Paramètres de configuration du module virtuel VM1... 104
Tableau 6. Paramètres de configuration spécifiques aux ports externes des ports virtuels mémoire... 105
Tableau 7. Paramètres de configuration spécifiques aux ports internes des ports virtuels mémoire dans les deux expériences... 105
Tableau 8. Résultats de la génération du pilote DMA... 127

Chapitre 1 : INTRODUCTION

Dans ce chapitre, on évoque les systèmes multiprocesseurs monopuces spécifiques à une application donnée ainsi que l'importance des mémoires globales pour ces systèmes. Par ailleurs, la nécessité des adaptateurs logiciels et matériels pour l'intégration des mémoires dans un système monopuce est montrée. Les problèmes et les difficultés de l'intégration mémoire sont évoqués dans la deuxième section. Dans la troisième section, on explique pourquoi ces problèmes d'intégration n'ont pas été résolus avant et quelles sont les conséquences s'ils demeurent non résolus. Les solutions à ces problèmes d'intégration mémoire sont proposées dans la quatrième section. On finit par indiquer nos contributions dans la cinquième section et le plan de ce manuscrit dans la sixième section.

1. Contexte de la thèse

1.1. Les systèmes multiprocesseurs monopuces spécifiques à une application donnée

Un système monopuce, appelé encore SoC venant du terme anglais " System-on-Chip " ou système sur puce, est un système complet intégrant plusieurs composants complexes et hétérogènes sur une seule pièce de silicium. Ce type de système doit être quasi autonome en fonctionnant indépendamment des éléments externes à la puce. Pour assurer son autonomie, un système monopuce est composé de plusieurs éléments logiciels et matériels dont la fonctionnalité est très complexe. Comme le montre la Figure 1, les éléments qui composent un système multiprocesseur peuvent être des processeurs de nature différente (des microprocesseurs, des processeurs du traitement de signal (DSP), des microcontrôleurs (MCU)), des mémoires, des réseaux de communication complexes et des adaptateurs de communication matériels.

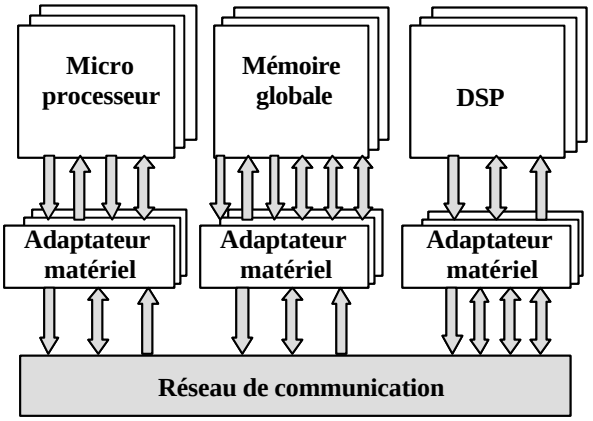


Figure 1. Architecture d'un système multiprocesseur monopuce

A la différence des systèmes multiprocesseurs classiques [Cul99][Pat98], les systèmes monopuces sont dédiés à des applications spécifiques et taillés sur mesure pour satisfaire seulement les besoins de l'application. Par rapport à un système-sur-carte ordinaire, un SoC emploie une seule puce réduisant le coût total d'encapsulation qui représente environ 50% du coût global du processus de fabrication de la puce [Sed00][Rea00], ce qui implique une réduction du coût total de fabrication. Ces caractéristiques ainsi que la faible consommation et la courte durée de conception permettent une mise sur le marché rapide de produits plus économiques et plus performants.

1.1.1. Architecture logiciel-matériel d'un système monopuce

L'architecture d'un système monopuce peut être vue comme une structure en couches. Cette structuration permet de définir les différents métiers de conception et la distribution du travail entre les différentes équipes [Jer02].

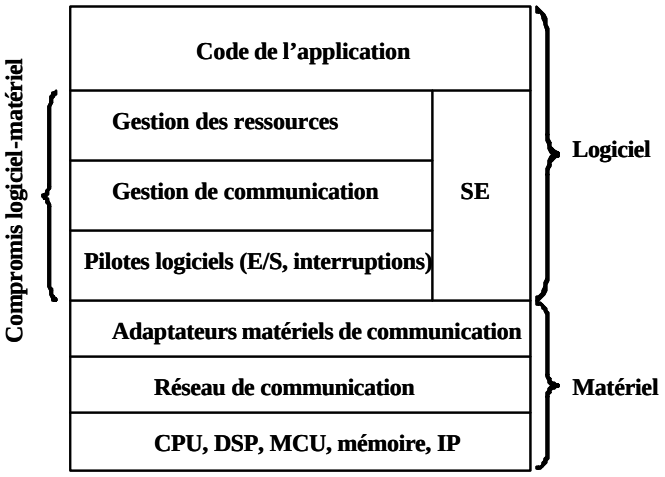


Figure 2. Structure en couches d'un système multiprocesseur monopuce spécifique à une application donnée

La Figure 2 montre une vue conceptuelle de l'architecture logiciel-matériel d'un système monopuce. Cette dernière est structurée en couches afin de maîtriser la complexité.

La partie matérielle est composée de trois couches essentielles :

- La couche basse contient les principaux composants utilisés par le système. Il s'agit de composants standards tels que des processeurs embarqués, des processeurs de traitement du signal (DSP), des microcontrôleurs, des mémoires et des IPs matériel.
- Le réseau de communication matériel embarqué sur la puce allant du simple pont (bridge) entre deux processeurs au réseau de communication par paquets. Parmi les réseaux de communication les plus utilisés pour la conception des systèmes monopuces : le bus AMBA de ARM [Arm99], le bus CoreConnect d'IBM [Ibm02] et le réseau de communication par paquets SPIN de l'université de Marie Curie [Gue00].
- Bien que le réseau de communication lui-même soit composé d'éléments standards, il est souvent nécessaire d'ajouter des couches d'adaptation matérielle entre le réseau de communication et les composants de la première couche. Il s'agit d'une couche de logique qui adapte les interfaces ainsi que les protocoles de communication utilisés des composants à ceux du réseau de communication.

Le partie logicielle embarquée est aussi découpée en couches :

- La couche basse contient les pilotes logiciels et les contrôleurs d'entrées-sorties permettant l'accès au matériel. Cette couche permet également d'isoler le matériel du reste du logiciel. Le code correspondant à cette couche est intimement lié au matériel.
- La couche de gestion de ressources permet d'isoler l'application de l'architecture. Bien que la plupart des systèmes d'exploitation (SE) fournissent une telle couche, il est souvent utile d'en réaliser une spécifique à l'application et ce pour des raisons de taille et/ou de performances. Les applications embarquées utilisent souvent des structures de données particulières avec des accès non standards (manipulation de champs de bits ou parcours rapides de tableaux) qui ne sont généralement pas fournis par les SEs standards. De plus, les couches de gestion de ressources fournies par les SEs standards sont généralement trop volumineuses pour être embarquées.
- Le code de l'application représente le logiciel développé pour être exécuté sur les processeurs de l'architecture du système. Ce code est généralement décrit à un haut niveau d'abstraction d'une manière complètement indépendante de l'architecture.

1.1.2. Les mémoires dans les systèmes multiprocesseurs monopuces à flot de données intensif

Pour accommoder les exigences de performances des domaines d'applications telles que xDSL, applications de jeu, etc., les architectures multiprocesseurs monopuces sont de plus en plus utilisées. Comme ces systèmes exigent des processeurs hétérogènes, des protocoles de communication complexes et plusieurs mémoires de différents types, leur conception est devenue longue et difficile. Une part importante de la complexité de leur conception est due à la mémoire puisqu'elle occupe une grande partie de la surface de la puce. La mise en œuvre de tels systèmes, contenant plusieurs mémoires hétérogènes, demande des efforts de conception considérables. Une façon de réduire ces efforts et de répondre aux contraintes du temps de mise sur le marché (" time to market ") est la réutilisation des composants. La conception à base de réutilisation de composants existants est survenue pour réduire le fossé matériel entre la capacité d'intégration importante en terme de transistors par surface et la faible production en terme de nombre de transistors produits par unité de temps. Ceci s'applique notamment pour la mémoire.

Cette solution peut être idéale dans le cas d'une architecture homogène où tous les éléments de l'architecture ont les mêmes interfaces et utilisent les mêmes protocoles de communication, ce qui n'est pas le cas pour les systèmes monopuces. Pour rendre cette solution efficace, le concepteur doit consacrer beaucoup d'efforts pour la spécification et l'implémentation des interfaces logiciel-matériel. Ces interfaces assurent l'adaptation des IPs mémoire au réseau de communication.

Cependant, l'intégration d'un composant mémoire fait que la spécification et la réalisation des adaptateurs logiciel-matériel est un vrai problème de conception car cela nécessite une connaissance

multidisciplinaire couvrant le domaine logiciel (application, pilote, SE) et le domaine matériel (processeur, mémoire, réseau). Ainsi, la section suivante aborde l'intégration des mémoires dans un système multiprocesseur monopuce.

1.2. Intégration des mémoires dans un système multiprocesseur monopuce

1.2.1. Nécessité des mémoires globales dans un système multiprocesseur monopuce

Les mémoires évoluent comme un nouveau paradigme dans le domaine des applications multimédia (audio, vidéo, jeux) qui consomment beaucoup de données. Pour satisfaire les besoins de ces applications en volume de données, l'utilisation des mémoires globales dans les systèmes monopuces est devenue nécessaire. Plusieurs facteurs sont à l'origine de cette nécessité :

a- Evolution de la technologie d'intégration

Grâce à l'évolution de la technologie des semi-conducteurs, aujourd'hui, on peut intégrer sur une seule puce ce qu'on mettait sur plusieurs puces ou cartes il y a une dizaine d'années. Dans un futur proche, cette évolution de la technologie permettra l'intégration de plus de 100 Mbits de DRAM et 200 millions de portes logiques dans la même puce. Comme le montre la Figure 3, d'après les prévisions de l'association d'industrie de semi-conducteurs et d'ITRS, les mémoires embarquées vont continuer de dominer la surface des systèmes monopuces dans les années qui viennent, à peu près 94 % de la surface totale en 2014 [Roa01].

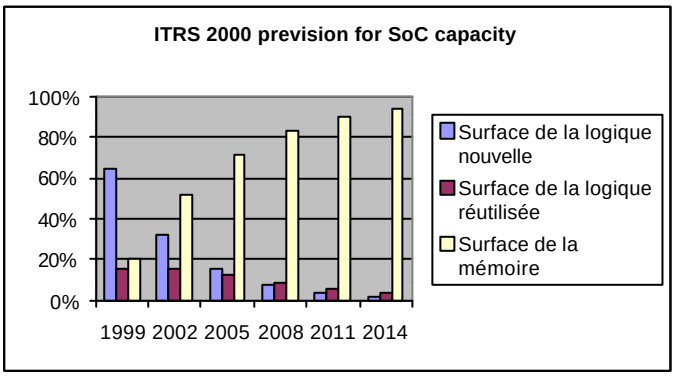


Figure 3. Prévision d'ITRS 2000 pour la capacité d'intégration mémoire dans les systèmes monopuces

b- Grande performance et faible consommation

Pour satisfaire les exigences des applications, les architectures monopuces doivent garantir une grande performance en terme de temps et une faible consommation d'énergie [Abr02]. L'intégration des mémoires sur la puce ne fait que répondre à ces exigences. En effet, le temps d'accès à une mémoire interne à la puce est beaucoup plus court que celui d'accès à une mémoire externe. De plus, la mémoire étant physiquement très proche, on réduit la consommation d'énergie. Ceci ne fait qu'optimiser le temps de calcul en évitant les opérations inutiles et réduire le temps d'accès global à la mémoire en évitant les protocoles d'accès lents et les transferts de données avec l'extérieur de la puce. Si la mémoire est embarquée sur la puce, le nombre de broches peut également être réduit et l'utilisation de bus sur carte devient obsolète.

c- Applications à flot de données intensif

Les applications multimédia traitent des volumes de données importants. Elles manipulent plusieurs images stockées dans des tableaux de grande taille et beaucoup de blocs de données. Un bloc de données peut être un secteur, un paquet, une trame ou une autre désignation, ce qui implique une utilisation intensive de la mémoire.

d- Logiciel en croissance

Pour maîtriser la complexité de la conception et pour réduire le coût de réalisation, le concepteur a recours au logiciel plutôt qu'au matériel (facilité de mise au point, modifications aisées, etc.). Cette utilisation massive du logiciel ne fait qu'augmenter l'utilisation de mémoire pour satisfaire les besoins du logiciel utilisé.

1.2.2. Nécessité des adaptateurs logiciels et matériels pour l'intégration mémoire dans un système multiprocesseur monopuce

a- Connexion des mémoires aux réseaux de communication

Un système monopuce contient généralement plusieurs mémoires différentes. Pour connecter ces mémoires au réseau de communication, il est d'une part nécessaire d'adapter les protocoles de communication différents utilisés par la mémoire au réseau. D'autre part, il faut générer les signaux physiques nécessaires au bon fonctionnement de la mémoire et effectuer des transformations sur les données selon la nature et les caractéristiques des informations transitant sur le réseau et celles acceptées par la mémoire.

b- Contrôle logiciel des accès mémoire

Le code de l'application d'un système monopuce est décrit à un haut niveau d'abstraction où toutes les informations sur l'architecture mémoire sont cachées. A cause de cette abstraction, le code de l'application est incapable d'accéder directement à l'élément de mémorisation. Pour rendre la mémoire accessible par ce

code de haut niveau, l'utilisation d'un adaptateur logiciel appelé aussi "pilote d'accès" entre le code de l'application et le processeur exécutant ce code est nécessaire.

c- Réutilisation et portabilité du code de l'application

L'adaptateur logiciel permet la portabilité du code de l'application sur différents types de processeurs. En effet, si le concepteur change le type de processeur durant la phase d'exploration d'architecture, il n'a pas à modifier le code de l'application pour l'adapter au nouveau processeur. Seul l'adaptateur logiciel doit être réécrit. L'adaptateur logiciel assure le ciblage logiciel de l'application sur le processeur utilisé. Cette portabilité permet la réutilisation du logiciel pour accélérer le processus de conception et réduit le temps de mise sur le marché.

d- Conception concurrente

L'adaptateur logiciel peut également être vu comme une couche de séparation entre le monde logiciel et le monde matériel qui permettra une conception concurrente. Cette séparation de conception définit trois différents métiers principaux : conception du logiciel, conception du matériel et conception de la communication logiciel-matériel.

2. Problématique d'intégration des mémoires dans un système monopuce

Pour intégrer une mémoire dans un système monopuce, le concepteur ou l'assembleur de composants doit faire face aux problèmes de nature logicielle et matérielle liés à la conception (spécification et implémentation) des interfaces. Les problèmes logiciels sont liés essentiellement à la conception des pilotes d'accès mémoire qui adaptent le code de l'application à l'architecture matérielle. Les problèmes matériels sont liés à l'adaptation entre l'interface matérielle de la mémoire et les protocoles de communication. La validation de ces adaptateurs logiciels et matériels est un souci qui rend la conception des interfaces logiciel-matériel de plus en plus difficile.

2.1. L'insuffisance du niveau RTL pour la conception des interfaces logiciel-matériel

Pour pouvoir accommoder les besoins exacts de communication requis par l'application [Arm99][Ibm02], les réseaux de communication sont devenus de plus en plus complexes. La spécification et la validation de la communication au niveau transfert de registre (RTL) est donc devenue de plus en plus difficile. Elle nécessite un travail manuel fastidieux et beaucoup de temps. En effet, à ce niveau d'abstraction, la structure d'un média de communication (bus, réseau) doit être détaillée au niveau du cycle pour vérifier les contraintes logiques et électriques du système. Cependant, le niveau RTL est trop bas pour la conception de la communication qui correspond essentiellement à la conception des interfaces logiciel-matériel. Pour ces raisons, un niveau

d'abstraction plus élevé que le niveau RTL est nécessaire pour la spécification et la validation des interconnexions entre les composants d'un système monopuce.

2.2. Variété des interfaces mémoire et manque de flexibilité d'utilisation

L'aspect matériel des problèmes d'intégration mémoire est lié aux interfaces mémoire qui sont très variées et qui correspondent rarement à l'interface du réseau de communication auquel elles sont connectées. Vue la diversité des applications à flot de données intensif, les protocoles d'accès mémoires se diversifient de plus en plus, et il devient de plus en plus difficile de les adapter aux réseaux de communication. Plusieurs facteurs sont à l'origine de cette difficulté d'adaptation :

- **Système hétérogène :** un système monopuce est généralement composé de plusieurs composants hétérogènes en terme de niveaux d'abstraction et de protocoles de communication. Pour intégrer une mémoire dans un tel système, les ports physiques de la mémoire doivent être adaptés aux ports abstraits (logiques) du reste du système qui sont généralement spécifiés à un haut niveau d'abstraction et qui peuvent utiliser des protocoles de transfert de données différents des protocoles d'accès mémoire.
- **Un large espace d'exploration d'architecture mémoire :** la conception d'une logique appelée aussi adaptateur matériel est nécessaire pour chaque nouveau type de mémoire. La spécification et l'implémentation de chaque nouveau type d'adaptateurs mémoire sont devenues une tâche de conception difficile. En effet, ces adaptateurs dépendent de l'interface de la mémoire et de l'interface du réseau de communication. Dans le cas où l'exploration de différents types de réseaux de communication est nécessaire, la complexité de la conception des adaptateurs mémoire ne fait qu'augmenter. Pour pouvoir réutiliser ces adaptateurs matériels, une implémentation modulaire et flexible est alors exigée.
- **Système monopuce multi-maîtres :** un système multiprocesseur monopuce permet l'intégration de plusieurs processeurs maîtres sur la même puce qui partagent une ou plusieurs mémoires. Pour résoudre le conflit des accès mémoire concurrents, l'implémentation des services de synchronisation sophistiqués est nécessaire.
- **Pression du temps de mise sur le marché et absence d'automatisation d'intégration mémoire dans les systèmes monopuces :** la réutilisation d'IP mémoire a permis de réduire le temps de conception, mais la conception des adaptateurs est encore un processus long en l'absence d'outils automatiques.

2.3. Faible production et réutilisation des pilotes d'accès

L'aspect logiciel des problèmes d'intégration mémoire est lié à la nécessité des pilotes logiciels qui doivent faire correspondre les accès mémoire du code de haut niveau aux processeurs sur lesquels les tâches de l'application sont exécutées. Le développement de ces pilotes est une tâche difficile. Cette difficulté est causée par plusieurs facteurs :

- **Forte dépendance des pilotes logiciels du matériel :** les pilotes d'accès mémoire sont généralement trop liés à l'architecture du processeur sur lequel ils s'exécutent (taille du bus de données, taille du bus d'adresse, le mode d'adressage, type de données transférées, etc.). Cette forte dépendance ne fait que réduire la flexibilité des pilotes d'accès. En effet, avant la mise sur silicium d'un système monopuce, le concepteur doit explorer plusieurs choix d'architectures (processeurs, mémoires, etc.). Un nouveau choix architectural nécessite l'écriture d'un nouveau pilote d'accès. Ceci nécessite une bonne connaissance du processeur sur lequel le pilote est exécuté, une bonne connaissance du matériel dont l'accès est contrôlé par le pilote et une bonne connaissance du système d'exploitation qui fournit les services d'accès requis par le pilote. Pour rendre ces pilotes d'accès moins indépendants du matériel et réutilisables, une implémentation modulaire et flexible s'impose.
- **Des pilotes d'accès spécifiques à une application donnée :** les pilotes classiques d'accès mémoires sont conçus essentiellement pour les systèmes multiprocesseurs généraux. Les outils de développement de pilotes classiques réduisent considérablement le temps de conception, mais ils restent trop généraux pour fournir des pilotes efficaces à des applications spécifiques. En effet, les systèmes monopuces imposent des fortes contraintes sur la performance (haut débit, faible consommation, etc.) et sur les ressources (petite surface, taille de la mémoire, etc.). Les pilotes doivent alors respecter ces contraintes et ne fournir que les services nécessaires à l'application.
- **Conception manuelle :** à cause de la complexité des systèmes monopuces et du large espace d'exploration d'architecture, la conception manuelle des pilotes d'accès s'avère très coûteuse en temps. Pour répondre à la pression de la mise sur le marché, il est devenu primordial d'avoir des outils de développement systématique des pilotes d'accès.

2.4. Validation des interfaces logiciel-matériel

Dans les approches de conception basées sur des composants ("Component Based Design"), les outils et les flots de conception utilisent des bibliothèques de composants différents. Ces approches supposent que le comportement de ces composants est déjà validé à l'extérieur de leurs flots. Mais, la construction d'un système à base d'assemblage de composants aboutit généralement à un échec. Ceci est dû à la diversité des composants et à l'incompatibilité de leurs interfaces qui empêchent le concepteur de valider toutes les combinaisons d'interfaces possibles entre les composants d'un système. Par conséquent, l'automatisation de la validation des interfaces de communication est devenue cruciale.

3. Causes et conséquences des problèmes d'intégration mémoire

3.1. Conception de haut niveau sans lien avec le niveau RTL

Plusieurs équipes de recherche ont traité l'utilisation de mémoire pour les systèmes monopuces à des niveaux d'abstraction plus élevés que le niveau RTL [Cat98][Cat02][Fra01][Tom96][Pan96]. La majorité de leurs travaux traite l'aspect logiciel des accès mémoire sans atteindre les problèmes matériels qui apparaissent au niveau RTL. Beaucoup d'efforts ont été consacrés pour l'optimisation d'allocation mémoire et la réduction des transferts de données en effectuant des transformations du code logiciel de l'application (essentiellement sur des boucles).

Si on se limite à une conception à des niveaux d'abstraction plus élevés que le niveau RTL, on ne peut pas implémenter des systèmes fidèles à la spécification. En effet, au niveau RTL, le concepteur peut valider la communication au cycle près et valider la fonctionnalité sur des simulateurs (ISS) de processeurs très proches des processeurs cibles.

Beaucoup d'études restent à faire pour traiter l'aspect communication des mémoires monopuces dont l'implémentation au niveau RTL nécessite beaucoup d'efforts.

3.2. Utilisation des interfaces mémoire " produits maison " malgré les efforts de standardisation des interfaces matérielles

Le problème de variété d'interfaces mémoire n'a pas été résolu parce que chaque concepteur utilise son propre flot de conception et ses propres bibliothèques matérielles et logicielles. Dans la majorité des flots de conception, l'utilisation d'un élément de la bibliothèque du flot (mémoire, processeur, pilote, etc.) ne pose pas de problèmes d'adaptation. Mais si le concepteur décide d'intégrer un nouvel élément externe à la bibliothèque (composant mémoire par exemple), il doit alors faire face manuellement aux problèmes d'adaptation. Beaucoup d'efforts ont été consacrés à la standardisation d'interfaces matérielles par VSIA [Vsi02], néanmoins il faut répondre à deux points :

- Comment peut-on implémenter ces interfaces matérielles standards d'une manière générique pour qu'elles puissent être utilisées facilement dans différents flots de conception ?
- Comment peut-on générer ces interfaces matérielles d'une manière automatique pour accélérer la production et l'intégration du matériel ?

3.3. Satisfaction de la conception des systèmes multiprocesseurs classiques par une faible production du logiciel dépendant du matériel

Les architectures multiprocesseurs classiques sont conçues indépendamment de l'application, les mêmes composants matériels de l'architecture peuvent être utilisés pour des applications différentes. Ceci ne fait que

limiter la diversité du matériel utilisé et par conséquent réduire la diversité du logiciel dépendant du matériel comme les pilotes de périphériques.

Dans le cas des systèmes monopuces, l'architecture est conçue pour répondre aux besoins d'une application donnée. Ceci est la cause majeure de la diversité du matériel qui est proportionnelle à la diversité des applications, et par conséquent la cause de la diversité du logiciel dépendant du matériel y compris les pilotes d'accès mémoire.

Ce problème de diversité de matériel a engendré un problème de production de logiciel dépendant du matériel qui reste encore faible par rapport aux besoins des applications. Ce problème est connu sous le terme "fossé logiciel".

Récemment, des équipes de recherche ont commencé à penser à une standardisation du logiciel dépendant du matériel pour faciliter sa réutilisation et garantir une production élevée [Eco02][Vsi02][I2O02].

Si ce problème demeure non résolu, la conception des systèmes monopuces risque d'affronter un nouveau fossé de type logiciel. Ce fossé est lié à la grande marge entre le besoin croissant du logiciel embarqué et sa faible production.

3.4. Faiblesse de la validation d'interfaces

La validation des interfaces de communication est un point clef qui manque aux approches de conception basées sur des composants. Les approches de validation utilisées antérieurement, comme la simulation et la co-simulation, permettent de valider la fonctionnalité globale du système, mais elles souffrent toujours de problèmes de synchronisation entre les interfaces de composants. Ceci est causé par l'absence d'une validation locale[1] des interfaces de communication entre chaque paire de composants du système. La conséquence de ce problème est la perte du temps pendant la validation globale du système et éventuellement l'échec de toute la conception.

4. Solutions proposées pour la résolution des problèmes d'intégration mémoire

4.1. Modèle de description mémoire de haut niveau

Comme le niveau RTL est insuffisant pour la conception de la communication entre la mémoire et le reste d'un système monopuce, nous proposons dans cette thèse un modèle de spécification mémoire à un niveau d'abstraction plus élevé que le RTL. Ce niveau cache les détails des interfaces mémoire et des protocoles de transfert de données. Ces abstractions permettent au concepteur de valider la fonctionnalité de ses choix

[1] La vérification est dite locale si elle ne s'intéresse qu'aux interfaces de communication entre deux composants et non pas aux interfaces de tous les modules du système en même temps.

architecturaux rapidement sans se préoccuper des précisions temporelles des accès mémoire et des détails des protocoles d'accès.

4.2. Abstraction de la communication et génération automatique des interfaces

Pour résoudre la complexité de la conception des interfaces logiciel-matériel des mémoires, nous nous basons sur une abstraction de la communication qui nous permet de cacher les détails des interfaces logiciel-matériel. Nous remontons à un haut niveau d'abstraction pour spécifier un modèle d'interface virtuel qui sépare l'interface de la mémoire de celui du réseau de communication. Cette abstraction facilite l'implémentation générique des interfaces au niveau RTL d'une manière systématique. La génération automatique de ces interfaces raccourcit le temps de conception. En plus, le découplage entre l'interface mémoire et celle du média de communication permet la séparation des responsabilités de conception. Le concepteur des mémoires et celui des réseaux de communication peuvent donc travailler séparément et en même temps.

4.3. Abstraction du matériel pour augmenter la portabilité des pilotes d'accès mémoire

Pour accélérer la production des pilotes d'accès mémoire et les rendre réutilisables, nous proposons dans cette thèse une méthode systématique de développement des pilotes d'accès mémoire. La méthode proposée est basée sur l'abstraction du matériel pour rendre les pilotes d'accès moins dépendants du matériel. Cette indépendance facilite la portabilité des primitives d'accès mémoire sur différents types de processeurs et permet au concepteur leur réutilisation pour des applications différentes.

4.4. Génération automatique des programmes de tests pour la validation des interfaces

Pour valider les interactions entre l'interface d'un composant et toutes les interfaces possibles des autres composants, nous proposons une méthode de génération automatique de programmes de tests d'interfaces. Chaque test généré est spécifique à une interface de communication donnée. Cette méthode permet de vérifier les interfaces localement avant la validation globale du système. L'aspect automatique de cette méthode permet au concepteur de couvrir toutes les possibilités d'interfaces dans un temps minimal.

5. Contributions

Comme l'indique la Figure 4 a, l'architecture cible du flot SLS de conception des systèmes monopuces ne tenait pas compte des mémoires globales. L'intégration de ce type de mémoire nécessite l'extension du flot de conception pour concevoir des interfaces logiciel-matériel spécifiques à la mémoire. Ces interfaces

correspondent à des adaptateurs logiciels et à des adaptateurs mémoire matériels (Figure 4 b). Pour concevoir ces interfaces et les valider, nous proposons **quatre contributions** qui sont liées aux quatre problèmes d'intégration mémoire déjà décrits dans la section 2 de ce chapitre.

La première contribution correspond à la proposition de modèles mémoire de haut niveau.

La deuxième contribution est liée à la génération automatique des adaptateurs mémoire matériels.

La troisième contribution est liée à la validation de ces adaptateurs d'interfaces matériels.

La quatrième contribution correspond à une méthode de génération systématique des adaptateurs logiciels.

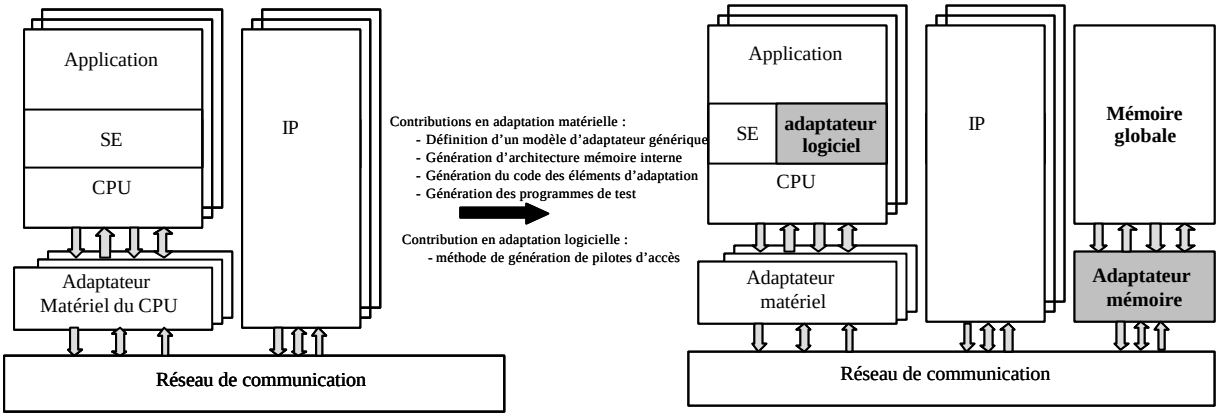


Figure 4. Les contributions liées aux interfaces logiciel-matériel des mémoires : (a) architecture sans mémoire, (b) architecture avec mémoire

5.1. Proposition d'un modèle mémoire de haut niveau

Notre première contribution consiste à réduire la complexité de la conception RTL des mémoires. Nous proposons des modèles mémoire abstraits décrits à des niveaux plus élevés que le RTL. La disponibilité de tels modèles mémoire permet de réduire l'hétérogénéité des systèmes monopuces en terme de niveau d'abstraction et par conséquent permet de faciliter l'intégration mémoire. Ces modèles mémoire font l'objet de la section 8 du chapitre 3.

5.2. Génération automatique d'adaptateurs mémoire matériels

Pour pouvoir générer automatiquement les adaptateurs mémoire matériels, il nous a fallu agir en trois points :

1. **Proposition d'un modèle d'adaptateur mémoire flexible :** cette contribution est liée au problème de flexibilité d'utilisation des interfaces. Elle se concrétise par la définition d'une architecture

générique d'adaptateur mémoire matériel. Cette architecture sépare la partie d'adaptation spécifique à la mémoire de celle qui est spécifique à la communication. Cette séparation permet une utilisation flexible de ces adaptateurs car ils peuvent être réutilisés pour plusieurs applications. Cette contribution est détaillée dans le chapitre 4.

2. **Génération automatique des architectures internes de la mémoire :** cette contribution est liée à la construction systématique d'une architecture mémoire appelée architecture interne[2]. La construction de cette architecture est une étape indispensable imposée par l'outil de génération d'interfaces que nous utilisons. Dans le flot de conception SLS, la conception de ce type d'architecture interne était une étape manuelle très difficile. En effet, cela nécessite la connaissance du langage de construction d'architecture (Colif [Ces01]) et la connaissance des langages cibles (SystemC et VHDL). Pour surmonter cette difficulté, nous avons développé un générateur d'architecture mémoire interne. Ce générateur permet à l'utilisateur de produire ses propres modèles d'architectures internes sans être obligé de connaître les langages de programmation et l'environnement du flot de génération d'interfaces. Cette contribution est présentée dans la section 4.1 du chapitre 5.
3. **Génération automatique du code pour les mémoires et pour les composants d'adaptation matériels :** cette contribution est liée à l'implémentation systématique du comportement de modèles mémoire et de leurs adaptateurs matériels génériques. Le code des différents composants est généré automatiquement à partir de bibliothèques de macro-modèles. Ces derniers peuvent être configurés spécifiquement aux caractéristiques de l'application. Cette contribution est présentée dans la section 4.2 du chapitre 5.

5.3. Génération automatique des programmes de tests des interfaces

Comme dans toutes les approches de conception à base de composants, l'approche du groupe SLS ne permettait pas une validation locale, systématique des interfaces de communication avant la validation globale du système par Co-simulation. Nous avons développé une méthode permettant de générer des programmes de tests spécifiques à chaque élément d'adaptation de la bibliothèque de macro-modèles. Cette méthode nous a permis de valider les interfaces de communication entre les différents types d'interfaces de mémoires et de réseaux de communication. Cette contribution est présentée dans la section 4.3 du chapitre 5.

[2] Nous utilisons le mot architecture mémoire interne pour désigner une structure hiérarchique composée d'un module mémoire connecté à un adaptateur de communication. L'interface de communication de cet adaptateur est encore abstraite. Le choix de cette architecture interne est imposé par l'outil de génération d'interfaces que nous utilisons (chapitre 5).

5.4. Génération automatique d'adaptateurs mémoire logiciels

Pour concevoir des pilotes logiciels qui adaptent les accès mémoire de l'application à l'architecture matérielle, nos contributions du côté logiciel ont concerné trois points :

1. Définition d'une couche modulaire et incrémentale d'abstraction du matériel.
2. Spécification et implémentation de pilotes en couches.
3. Extension de l'environnement de l'outil ASOG[3] pour la génération automatique de ces pilotes d'accès mémoire.

Ces trois contributions nous ont permis de concevoir systématiquement des pilotes d'accès qui assurent la portabilité d'une application donnée sur une architecture cible. Ces pilotes peuvent être réutilisé pour plusieurs types de processeurs. Cela fait l'objet du chapitre 6.

6. Plan de la thèse

Le reste de ce manuscrit est composé de six chapitres. Le chapitre 2 présente les travaux antérieurs spécifiques aux mémoires.

Le chapitre 3 présente les concepts de base utilisés pour l'abstraction des interfaces logiciel-matériel. Le flot de conception ainsi que notre méthode de conception des adaptateurs mémoire matériels et logiciels seront abordés.

Le chapitre 4 propose une architecture générique des adaptateurs mémoire matériels.

Le chapitre 5 détaille la génération automatique de ces adaptateurs mémoire matériels.

Le chapitre 6 présente une architecture en couches de pilotes d'accès mémoire ainsi qu'une méthode de génération systématique.

On finit par une conclusion et des perspectives dans le chapitre 7.

[3] ASOG: c'est un outil de génération de système d'exploitation développé dans le groupe SLS pendant les travaux de thèse de Lovic Gauthier [Gau01].

Chapitre 2 : LES MEMOIRES DANS LES SYSTEMES MONOPUCES ET L'ETAT DE L'ART SUR LEUR INTEGRATION

Ce chapitre présente les différents travaux liés à la mémoire dans un flot de conception système. Dans un tel flot, on distingue plusieurs étapes concernant la mémoire, allant des optimisations fonctionnelles des accès mémoire jusqu'à l'intégration de composants. La section 1 détaille ces différentes étapes dans le flot de conception système classique. Dans la section 2, nous étudions les travaux antérieurs liés à chacune de ces étapes. Une conclusion est donnée dans la section 3.

1. Introduction : flot de conception système

Un flot de conception système est souvent une succession d'étapes appliquées sur une spécification de haut niveau pour fournir une implémentation finale du système, décrite au niveau de transfert de registre (RTL). L'intégration des mémoires dans un flot de conception de systèmes monopuces peut être faite à différents niveaux d'abstraction allant d'un niveau fonctionnel jusqu'à un niveau RTL. Pour comprendre la problématique de cette intégration, nous avons étudié les différents travaux antérieurs liés à la mémoire dans le domaine de la conception système. Cette étude nous a permis de classer ces travaux dans un flot de conception mémoire composé de quatre étapes principales (Figure 5) :

1. **Optimisations fonctionnelles :** cette étape correspond à l'optimisation des accès mémoire de la spécification de haut niveau de l'application. Ces optimisations sont qualifiées de fonctionnelles parce qu'elles sont indépendantes de l'architecture du système. L'entrée de cette étape est souvent le code de l'application ou un graphe de flot de données (DFG). La sortie est un code ou un DFG optimisé. Les optimisations correspondent essentiellement à la réduction du nombre de transferts de données et de la taille de la mémoire utilisée.
2. **Choix de l'architecture mémoire :** cette étape consiste à trouver l'architecture mémoire qui répond aux besoins de l'application (nombre, taille, etc.). Elle prend en entrée le code ou le DFG optimisé pour produire une "netlist" de mémoires logiques. Ces mémoires sont qualifiées de logiques parce qu'elles ne sont pas toujours réalisables et ne correspondent pas toujours à des modèles mémoire physiques existants ou synthétisables.
3. **Choix de composants :** cette étape consiste à trouver la meilleure implémentation physique de l'architecture mémoire logique. Les composants mémoire d'implémentation sont souvent choisis à partir d'une bibliothèque. La sortie de cette étape est donc une "netlist" de composants mémoire qui correspond à des types de mémoires réelles.
4. **Intégration de composant :** l'intégration d'un composant mémoire de la "netlist" précédente dans un système monopuce consiste à concevoir les interfaces d'adaptation logiciel-matériel. Ces interfaces assurent l'adaptation de protocoles entre la mémoire et le réseau de communication, mais aussi la portabilité du code des accès mémoire de haut niveau sur l'architecture matérielle cible. La sortie de cette étape est une architecture d'implémentation RTL avec des mémoires.

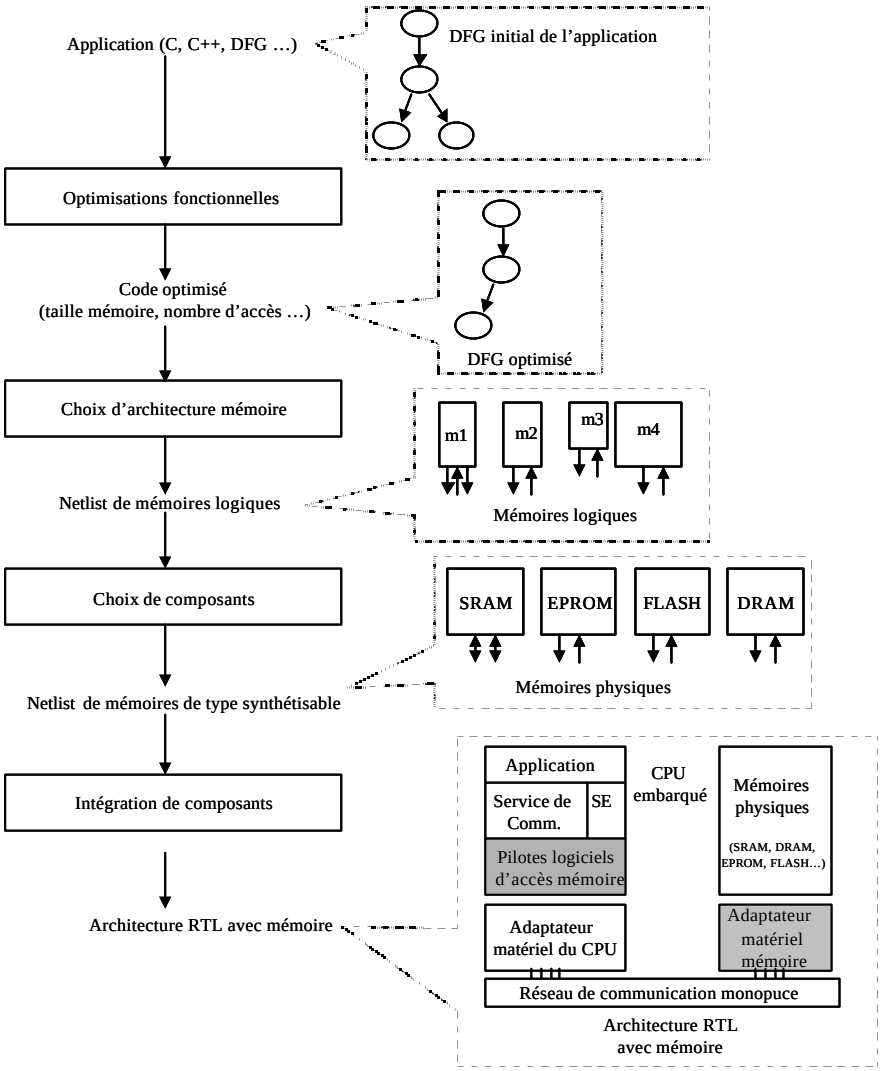


Figure 5. Une vue globale sur les quatre types de travaux liés aux mémoires

2. Etat de l'art sur l'intégration mémoire aux différents niveaux d'abstraction du flot de conception mémoire

Tous les travaux de recherche liés à la mémoire dans le domaine de la conception système peuvent être classés dans une ou plusieurs étapes du flot de conception mémoire. Cette section décrit les travaux antérieurs spécifiques à la mémoire durant chaque étape de ce flot.

2.1. Optimisations mémoire au niveau fonctionnel : transformation de code indépendamment de l'architecture

Plusieurs travaux ont traité les mémoires à un haut niveau d'abstraction indépendamment de la plateforme architecturale. Ces travaux visent essentiellement l'optimisation de la mémoire (taille, nombre d'accès, consommation, …) en effectuant des transformations sur le code de l'application. Celles-ci consistent essentiellement à :

- Augmenter la localité des données et la régularité des accès mémoires.
- Optimiser la taille des tableaux et des variables scalaires utilisées par l'application en éliminant les données temporaires inutiles et les données redondantes.
- Réduire les transferts de données avec les mémoires lointaines en utilisant des hiérarchies contenant des mémoires caches et des mémoires locales rapides.

Les techniques les plus utilisées sont généralement la fusion, l'alignement et l'éclatement de boucles. Ces techniques appelées également transformation source à source ont déjà été utilisées dans le domaine des compilateurs parallèles pour augmenter le degré de parallélisme des programmes séquentiels [Lov77][Wol91][Wol96][Kel92][Mck98][Ama95][Ban95]. En utilisant ces techniques, les compilateurs ne peuvent faire que des transformations locales liées à une seule boucle ou à une seule procédure à la fois. Ces optimisations locales ne sont pas très efficaces dans le cas des applications complexes qui contiennent plusieurs modules logiciels concurrents, et qui peuvent partager la même structure de données. Une première solution a été proposée par McKinley en 1998 [Mck98] en regroupant les boucles du code de l'application et en faisant une analyse inter-procédurale avant la compilation. Le but de ce travail a été essentiellement l'augmentation du parallélisme pour les architectures multiprocesseurs à mémoire partagée. Cette solution reste locale car le regroupement de toutes les boucles du programme n'est pas toujours possible. Une deuxième solution a été proposée par plusieurs travaux ; Franssen en 1994 [Fra94], Greef en 1995 [Gre95] et Masselos en 1999 [Mas99]. Comme l'indique la Figure 6, cette approche peut être considérée comme une phase de pré-compilation. Elle consiste essentiellement à faire des transformations globales sur toutes les boucles du programme ainsi que sur le flot de contrôle avant la compilation.

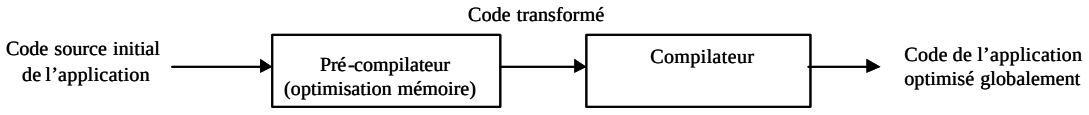


Figure 6. Transformation de données et de flot de contrôle avant la compilation

Récemment, l'automatisation de cette phase de pré-compilation a fait l'objet de plusieurs travaux de recherche [Cat98][Dan00][Fra01]. Dans [Cat98], une méthode d'exploration de l'organisation et de transfert de données est proposée. Cette méthode inclut une phase de nettoyage du code encore manuelle.

L'inconvénient de ces transformations de code est le fait qu'elles sont totalement indépendantes de l'architecture mémoire disponible et ne prennent donc pas en compte l'architecture du processeur sur lequel le code de l'application est exécuté. Un autre inconvénient est que la majorité de ces travaux ont traité l'aspect mémoire sans tenir compte des autres facteurs qui entrent en jeu dans la conception logiciel-matériel des architectures multiprocesseurs comme l'aspect de la communication.

2.2. Synthèse d'une architecture mémoire : assignation et allocation

La synthèse mémoire de haut niveau consiste à allouer les éléments de mémorisation et à assigner les données sur l'architecture mémoire allouée d'une façon optimale. Une assignation optimale est une organisation de données qui permet de réduire le nombre d'accès, la taille mémoire et la consommation. L'allocation mémoire consiste à choisir l'architecture mémoire sur laquelle les données sont assignées. Selon le type de mémoire allouée, on distingue deux types d'assignation mémoire :

- Assignation sur une architecture mémoire fixe.
- Assignation sur une architecture mémoire spécifique à une application donnée.

2.2.1. Assignation de données sur une architecture mémoire fixe

L'assignation mémoire consiste à trouver l'organisation optimale des données sur une architecture mémoire fixe en respectant les contraintes physiques (nombre de ports, taille, types d'accès supportés).

Le problème d'assignation de variables scalaires à des éléments de mémorisation a été traité dans [Bal88][Ahm91][Mef01a]. [Bal88] propose une méthode d'assignation de registres à des mémoires multi-ports. [Ahm91] propose une formalisation du problème sous forme d'un système d'équations linéaires. [Mef01a] propose une solution à ce problème linéaire en utilisant la simulation pour déterminer les coefficients des fréquences d'accès à chaque variable.

L'avantage de l'approche d'assignation utilisant des modèles mémoire fixes, est que l'architecture mémoire allouée respecte les caractéristiques physiques des mémoires disponibles ainsi que les modes d'accès supportés. L'inconvénient est que le type de mémoires utilisées est figé d'avance et il ne répond pas toujours aux besoins de l'application. Cette approche n'est pas très performante pour les systèmes monopuces qui nécessitent une architecture mémoire optimale spécifique à l'application.

2.2.2. Assignation des données sur une architecture mémoire spécifique à une application donnée

La deuxième approche consiste à affecter les données sur une architecture mémoire spécifique à l'application. Les travaux basés sur cette approche consistent à synthétiser une architecture mémoire à partir d'une spécification abstraite [Pan01]. La spécification d'entrée de cette approche est composée d'un graphe de dépendance de données extrait du code de l'application et d'une architecture mémoire abstraite. La sortie de cette méthode est une architecture mémoire logique dont la taille est spécifique aux données de l'application. Il y a eu d'autres méthodes qui permettent l'exploration de l'assignation des données sur des architectures mémoires différentes [Cat02][Mef01a].

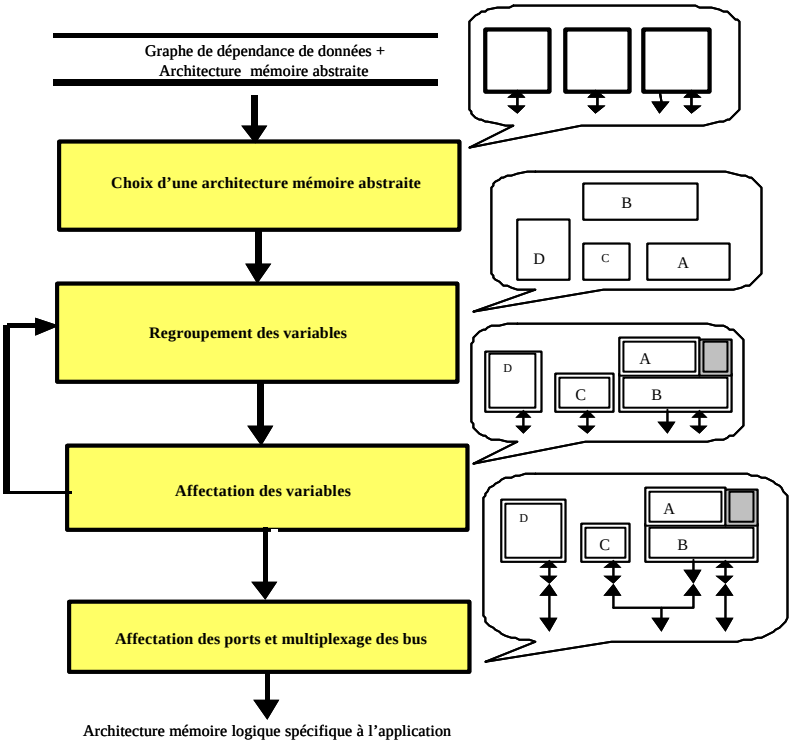


Figure 7. Synthèse de mémoires logiques spécifiques à une application donnée par la méthode DTSE

La méthode la plus avancée dans ce domaine est probablement DTSE ("Data Transfer and Storage Exploration") [Cat02]. Elle permet une exploration rapide d'architecture mémoire. Comme l'indique la Figure 7, cette méthode est composée de quatre étapes essentielles :

1. **Choix de l'architecture mémoire abstraite :** consiste à fixer le type de la mémoire et le nombre de ports. Cette étape est dite abstraite parce que le protocole de communication n'est pas encore détaillé. La taille de la mémoire choisie n'est pas fixée ainsi que le mode d'accès. La mémoire ne sera vraiment dimensionnée qu'au cours de l'affectation des données, car ceci est directement lié à la taille des variables.
2. **Regroupement de variables :** consiste à regrouper les variables ayant le même type et le même ordre de fréquence d'accès ou à regrouper les données.
3. **Affectation de variables :** consiste à décider dans quelles mémoires seront affectées les variables. Selon les caractéristiques des variables (taille, type …), la mémoire est dimensionnée. Dans le cas où la longueur du mot d'une variable dépasse la longueur du mot de la mémoire choisie lors de l'affectation, un retour en arrière peut invoquer la restructuration des données afin de les adapter aux contraintes mémoires (nombre de ports, largeur maximale d'un mot…).
4. **Interconnexion :** on décide dans cette dernière étape d'adapter les connexions des composants mémoires et des autres éléments de l'architecture. Il s'agit alors de définir les liens entre les ports de la mémoire et le bus de communication ainsi que le multiplexage des ports dans le cas des mémoires multi-ports.

L'avantage de la synthèse mémoire est qu'elle permet de trouver l'architecture mémoire adéquate à l'application. L'inconvénient est que cette architecture mémoire n'est pas toujours synthétisable ou réalisable par des modèles mémoire réels.

2.3. Transposition des mémoires logiques en mémoires physiques : choix de composants

On appelle mémoire logique un groupement de variables scalaires ou de tableaux dans un module décrit au niveau comportemental. Ce type de mémoire ne correspond toujours pas à une réalisation physique existante ou synthétisable. Une mémoire logique peut par exemple avoir plusieurs ports, tandis que les mémoires physiques réelles ont un nombre de ports très limité. La largeur d'un mot d'une mémoire logique peut dépasser un millier de bits tandis qu'elle ne dépasse pas une centaine dans le cas d'une mémoire réelle (8, 16, 32, 64, 128).

La transposition (mapping en anglais) des mémoires logiques sur des mémoires physiques est la partie dorsale (front-end) de la synthèse mémoire de haut niveau. Ceci consiste à générer à partir d'une mémoire logique un modèle de réalisation physique.

Kim et Lin [Kim93] sont parmi les premiers qui ont abordé ce sujet. Ils modélisent le problème de transposition sous forme d'un découpage en deux dimensions. Le nombre de ports et le nombre de registres

du module mémoire sont les deux dimensions sur lesquelles les mémoires logiques sont découpées et regroupées en une ou plusieurs mémoires physiques de même type.

Les travaux de Karchmer et Rose [Kar94] consistent à diviser une mémoire logique en plusieurs sous modules et de les affecter à une mémoire physique fixe. L'affectation des sous modules logiques est basée sur un algorithme de *brach and Bound*.

Bakshi et Gajski [Bak95] appliquent une séquence d'expansion sur les modèles mémoire disponibles pour les adapter aux mémoires logiques. Cette expansion inclue l'élargissement d'un mot mémoire, l'augmentation du nombre de mot et le multiplexage des ports.

Jha et Dutt [Jha97] proposent une méthode d'exploration de l'espace des solutions de la transposition de mémoires logiques en mémoires physiques. Pour choisir une mémoire physique qui répond au mieux à la mémoire logique, ils utilisent trois axes orthogonaux : le nombre de ports, le nombre de mots mémoire et la largeur d'un mot. Ils proposent deux algorithmes de transposition : le premier est un algorithme exhaustif très lent dans le cas d'un espace de solutions large. Il ne peut être utilisé que dans le cas d'un nombre de choix mémoire limité. Une deuxième solution correspond à un algorithme linéaire qui est plus rapide que le premier mais plus coûteux en terme de surface générée.

Tous ces travaux ont essayé de compléter la synthèse mémoire de haut niveau en adaptant l'architecture mémoire logique requise par l'application ou en synthétisant de nouveaux types de mémoire qui correspondent aux caractéristiques des mémoires logiques. L'intégration de ces mémoires physiques dans un système n'est pas toujours possible car aucun de tous ces travaux n'a traité l'aspect de la communication des mémoires avec le reste du système (processeur, SE, réseau de communication). La conception de système monopuce avec mémoire nécessite une vue globale sur les interactions entre le code des accès mémoire et le processeur cible, mais aussi sur les interactions de contrôle et de transfert de données entre l'interface de la mémoire physique et le réseau de communication. Ceci nécessite une maîtrise des adaptateurs logiciels et matériels. La section suivante traite l'adaptation des mémoires au reste du système.

2.4. Adaptation des mémoires au reste d'un système monopuce

Dans le schéma classique de l'architecture d'un système monopuce, un processeur accède à la mémoire globale à travers un réseau de communication. Il est donc nécessaire d'avoir une adaptation physique du composant mémoire au réseau de communication car les protocoles et les interconnexions de ces deux types de composants sont différents. Mais il est aussi nécessaire d'adapter le code de l'application pour que celui ci puisse accéder à la mémoire. Ces accès mémoire sont abstraits et décrits à un haut niveau d'abstraction. De plus, les informations sur l'architecture et les protocoles de communication utilisés sont eux également abstraits.

A cause de cette abstraction, le code de l'application est incapable d'accéder directement à l'élément de mémorisation. Pour rendre la mémoire accessible par ce code de haut niveau, l'utilisation d'un adaptateur

logiciel appelé aussi pilote d'accès, entre le code de l'application et le processeur exécutant ce code, est nécessaire. Ces adaptateurs doivent aussi fournir un certain degré de portabilité au code de l'application pour qu'il puisse être exécuté sur différents processeurs.

2.4.1. Adaptation matérielle des mémoires aux réseaux de communication

L'intégration d'un composant matériel (comme les mémoires) dans une architecture monopuce nécessite une adaptation matérielle entre l'interface mémoire et le protocole de communication employé. Selon le protocole de communication (standard ou spécifique), on distingue deux types d'adaptateurs : des adaptateurs spécifiques et des adaptateurs standards.

La conception des adaptateurs matériels spécifiques consiste à raffiner les interfaces initialement abstraites du composant jusqu'à obtenir une implémentation finale spécifique à un protocole de communication donné.

La conception des adaptateurs matériels standards consiste à utiliser des protocoles de communication et des interfaces qui suivent un standard donné.

a- Conception des adaptateurs matériels spécifiques à un protocole de communication donné

Coware [Ver96][Bol97] est un environnement de conception et de simulation des systèmes hétérogènes. Comme l'indique la Figure 8, le flot de génération d'architectures de Coware prend en entrée une architecture abstraite, une bibliothèque de composants (processeurs, mémoires, protocoles de communication) et une bibliothèque d'adaptateurs spécifiques aux scénarios de communication. L'architecture cible de ce flot est une architecture fixe composée d'un seul processeur connecté à plusieurs composants matériels via un réseau de communication point à point.

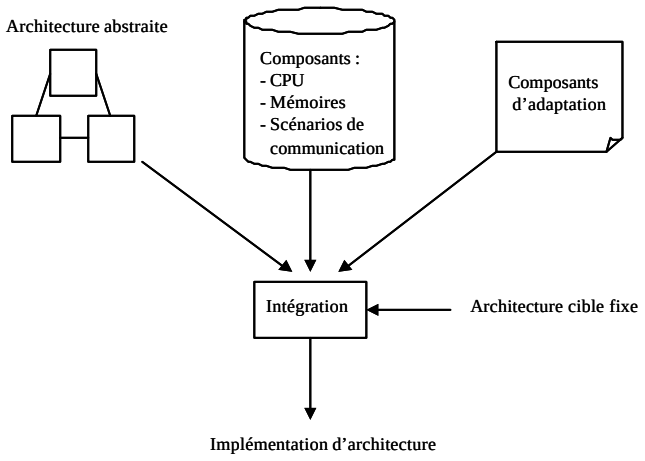


Figure 8. Flot de génération d'architecture de Coware

L'architecture abstraite est décrite en langage de spécification logiciel-matériel N2C (*Napkin-to-chip*). Elle spécifie l'application sous forme d'un réseau de modules communicants. La sémantique de la communication entre les tâches est basée sur des appels de procédures à distance (RPC). La communication entre les modules est assurée par des canaux de communication reliant les ports des modules. L'implémentation de cette architecture abstraite consiste à :

- Transposer les modules sur un modèle de processeur et des modèles de matériels déjà existants dans la bibliothèque.
- Synthétiser la communication entre les différents modules.
- Générer les adaptateurs de communication.

La transposition consiste à assigner chaque module de la spécification abstraite soit à un composant logiciel (processeur logiciel), soit à un composant matériel (accélérateur, contrôleur, périphérique, …) de l'architecture cible.

La synthèse de communication consiste à raffiner les canaux de communication abstraits et à choisir un scénario de communication pour chaque canal. La bibliothèque des protocoles de communication contient plusieurs scénarios basés sur le protocole à "poignée de main" :

- *NoHndshk* : c'est un simple signal pour transférer des données sans aucune notion de contrôle ou de synchronisation.
- *EnHndshk* : ce protocole est composé de deux signaux ; un pour les données et l'autre (" *Enable* ") pour synchroniser le transfert des données. Le transfert n'est autorisé que lorsque le signal "*Enable* " est valide. Ce protocole suppose que la communication est toujours initiée par le producteur et que la latence de la réponse du consommateur est connue d'avance.
- *FullHandshk* : ou rendez vous est un protocole qui possède en plus un signal requête et un signal acquittement. Ces signaux sont utilisés pour l'initiation et la synchronisation des transferts de données entre des modules maîtres et esclaves.
- *MemEnHndshk* : c'est un protocole *EnHndshk* avec un signal adresse. Il peut être utilisé pour accéder à un élément de mémorisation.
- *MemFullHndshk* : c'est un protocole *MemEnHndshk* synchronisé par un signal d'acquittement.

Dans Coware un élément de l'architecture est connecté au réseau de communication à travers un adaptateur de communication. Cet élément peut être généré automatiquement en sélectionnant dans une bibliothèque les bons éléments à partir des caractéristiques du réseau et du composant à connecter.

Avec cette approche, l'architecture complète du système est synthétisée et elle peut être validée à différents niveaux d'abstraction. Cette approche est toutefois insuffisante pour intégrer tous ces composants mémoire dans un système multiprocesseur monopuce. Les causes de cette limite de Coware sont :

- Coware se limite à des protocoles d'accès mémoire de type "poignée de main". Dans le cas d'un système monopuce, des protocoles d'accès plus sophistiquées sont nécessaires, tels que les accès en mode "burst" ou via un DMA.
- Un composant matériel ne peut être facilement intégré par Coware que si la structure de son interface est proche de celle des protocoles de communication existant dans la bibliothèque. Si l'utilisateur veut intégrer un composant externe dont l'interface ne supporte pas les protocoles disponibles, il doit écrire, manuellement, une cellule d'adaptation supplémentaire entre le composant et le protocole le plus proche de son interface.

Polis [Bal97] est une approche de co-conception logiciel-matériel orientée vers les applications de contrôle. L'architecture cible est formée d'un seul processeur entouré par une bibliothèque de composants matériels. La communication entre le logiciel et le matériel est basée sur une mémoire partagée. Le but de cette méthode est de faciliter la réutilisation des composants de communication. Le principe de base de cette approche est de traiter le comportement et la communication d'une manière orthogonale. Ceci consiste à raffiner continuellement la communication initialement abstraite entre des blocs fonctionnels sans tenir compte du comportement des blocs jusqu'à une implémentation finale. Chaque composant matériel doit suivre un protocole basé sur un simple signal de validité de données afin d'être interfacé avec d'autres composants.

Lycos [Jen97][Knu98] et Cosyma [Ern94] proposent une méthode de synthèse de communication logiciel-matériel basée sur une bibliothèque implémentant une diversité de protocoles de communication. Cependant, ceci déplace seulement les problèmes de conception des interfaces du concepteur au fournisseur de la bibliothèque. Dans la même direction, SpecSyn [Gaj98] et Archimate [Ism96][Val01] utilisent la technique de raffinement des protocoles de communication et des bibliothèques de communication pour arriver à une implémentation finale. Des bibliothèques de composants matériels et logiciels contenant des unités d'adaptation sont également utilisées.

Toutes ces approches ont traité la synthèse des interfaces pour des composants et des protocoles de communication spécifiques. Le principe commun de ces approches est le raffinement continu des interfaces et des protocoles de communication. Ce raffinement aboutit à une implémentation spécifique dont l'interface ne correspond à aucun standard. L'automatisation de ces méthodes est basée essentiellement sur des bibliothèques logicielles et matérielles implémentant les unités de communication spécifiques.

L'inconvénient de ces approches est qu'elles ne supportent pas des protocoles de communication standard et des IPs définis à l'extérieur de leur environnement.

Une autre méthode de génération d'interfaces plus ouverte aux protocoles de communication spécifiques et standards ainsi que les IP, a été développée pendant les travaux de thèse de Damien Lyonnard dans le groupe SLS. Cette méthode a aboutit à un outil de génération d'interfaces (ASAG) basé sur l'assemblage de composants de bibliothèques. Il permet la génération de co-processeurs de communication

matériels capables d'adapter un processeur donné à un réseau de communication. La génération des interfaces mémoire par cet outil n'a pas été prise en compte. Nous avons étendu l'environnement de cet outil pour qu'il puisse être réutilisé pour les mémoires. Les détails de cette partie fait l'objet du chapitre 5.

Dans la section suivante, on évoquera les efforts de recherche liés à la standardisation des interfaces et des protocoles de communication.

b- Conception des adaptateurs matériels standard

Récemment, beaucoup de travaux de recherche se sont intéressés à la standardisation des protocoles de communication. Le but de ces travaux est de faciliter la réutilisation des composants. Le concept de base de ces approches est de standardiser les interfaces des bus de communication. Un tel standard permet au concepteur d'IP de supporter un nombre limité d'interfaces. Dans Chinook [Bor98][Cho99], une méthode systématique de génération d'interfaces a été proposée. Cette méthode consiste à générer une logique séquentielle qui permet d'adapter un port d'entrées-sorties d'un microcontrôleur à un périphérique.

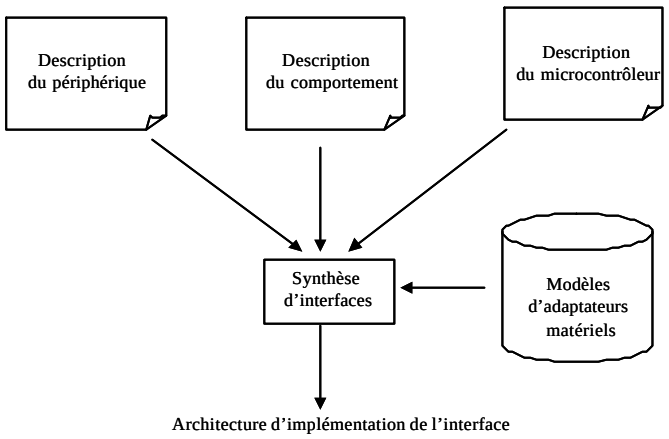


Figure 9. Chinook : synthèse d'interfaces

Comme l'indique la Figure 9, l'entrée de cette méthode est composée de :

- **Une description du comportement du système :** c'est un programme écrit par l'utilisateur en utilisant un langage de haut niveau. Cette description est composée d'une partie déclarative qui alloue les éléments de mémorisation statiques et une partie opérationnelle qui inclus les fonctions de communication avec le périphérique via des appels de pilotes. Pour éviter les problèmes de synchronisation, cette spécification est supposée mono-processus.
- **Une description du périphérique :** c'est une classe à trois attributs. Elle est composée d'une liste des ports de périphérique, une représentation matérielle du périphérique sous forme de règles statiques associées à chaque broche et une représentation logicielle sous forme de séquences de comportement.

- **Une description du microprocesseur :** elle contient une liste de ports du microcontrôleur avec des informations sur la direction et le type de données supportées par chaque port. Chaque port peut avoir des instructions de communication multiples. Une instruction de communication représente la sémantique d'accès de chaque port par le contrôleur

La sortie de cette méthode est un réseau de modules matériels synthétisables connectés au microcontrôleur via des parties logiques ou des adaptateurs. Ces adaptateurs interfacent chaque port de périphérique au port du microcontrôleur. Ils sont sélectionnés à partir d'une bibliothèque matérielle tout en respectant les paramètres de la spécification d'entrée.

Bien que cette approche puisse être étendue à l'assemblage de plusieurs IPs, elle se limite à un seul processeur et elle suppose que la spécification de l'application d'entrée ne contienne pas de tâches ou de processus concurrents. Cette hypothèse est assez forte pour les systèmes multiprocesseurs monopuces qui sont dédiés à des applications parallèles et multitâches.

O'Nils [Oni98][Oni01] présente une approche de conception d'architecture logiciel-matériel à base d'IP fixe. L'assemblage des différents éléments de l'architecture est basé sur des adaptateurs matériels (contrôleurs DMA) et logiciels (pilotes de périphérique). La conception de ces adaptateurs est basée sur une analyse grammaticale d'une spécification décrite sous forme d'expressions régulières écrites en langage ProGram [Obe96]. L'inconvénient de cette approche est qu'elle n'est pas complètement automatisée.

Parmi les autres travaux de standardisation les plus connus dans le domaine des interfaces logiciel-matériel, on trouve l'approche de Virtual Socket Interface Alliance (VSIA) qui définit un standard d'abstraction d'interfaces de bus nommé *Virtual Component Interface* (VCI). Dans le cadre du projet *ESPRIT COSY* qui résulte d'une collaboration entre l'université de Pierre-Marie Curie de Paris et les laboratoires de Philips, le standard d'interfaces VCI est utilisé pour la conception des adaptateurs matériels génériques [Bru00]. La Figure 10 montre la structure d'une interface de communication matérielle utilisant la norme VCI aux différents niveaux d'abstraction. Cette interface matérielle permet à un coprocesseur de s'adapter à plusieurs bus de nature différente en utilisant le standard *VCI-OCB*[4]. Au plus haut niveau d'abstraction, une interface système est définie comme des unités de vecteurs qui transforment les appels systèmes en un simple protocole producteur-consommateur implémenté par des FIFO. A un niveau plus bas, l'interface correspond à une unité de service qui transforme le protocole FIFO en un protocole VCI. Cette unité fournit des services de contrôle pour les FIFO et des services de configuration du matériel. Le dernier niveau correspond à un adaptateur du bus qui adapte l'interface VCI à l'interface du bus de communication. Cet adaptateur est implémenté en VHDL d'une manière paramétrable.

[4] VCI-OCB (Virtual Component Interface-On Chip Bus) : c'est une spécification qui définit les interfaces et les protocoles d'un bus de communication monopuce. Cette spécification a été développée par le groupe OCB de VSIA.

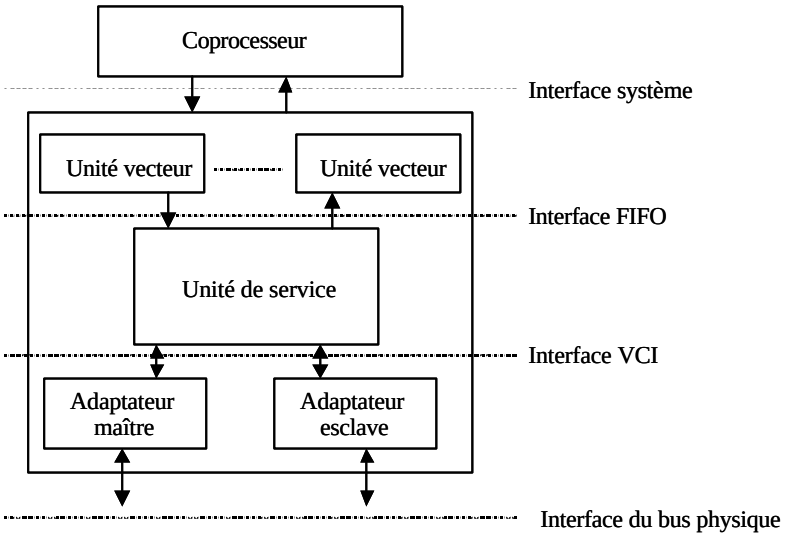


Figure 10. Les interfaces matérielles utilisant le standard VCI

D'autres efforts de standardisation consistent à offrir un bus système standard auquel plusieurs types de composants peuvent être connectés. Comme exemple de bus standard, on peut citer :

- Peripheral Interconnect Bus d'OMI [PIBus].
- AMBA d'ARM Inc [Arm99].
- CoreConnect d'IBM [Ibm02].
- OpenCore de Sonics.

Parmi les inconvénients de ces approches de standardisation :

- Les protocoles standards ne répondent pas toujours aux exigences de l'application.
- Les interfaces de communication standards sont parfois trop générales et utilisent des protocoles non nécessaires.
- Ces standards ne sont pas trop utilisés à cause des conflits politiques entre les utilisateurs et les entreprises qui parfois représentent elles-mêmes l'organisme du standard.

2.4.2. Adaptation logicielle des accès mémoire aux processeurs

En plus de la nécessité de l'adaptation matérielle, l'intégration d'une mémoire dans un système monopuce nécessite une adaptation de type logiciel. Ces adaptateurs logiciels appelés aussi pilotes d'accès assurent la portabilité du code d'accès mémoire sur différentes architectures utilisant différents processeurs et différents systèmes d'exploitation. Plusieurs outils commerciaux de développement logiciel fournissent les pilotes classiques aux processeurs à usage général (PC) [Jungo]. Ces pilotes sont assez généraux pour être utilisés par les systèmes monopuces. En effet, ils ne sont pas conçus spécifiquement à l'application et ils

peuvent implémenter des fonctionnalités non nécessaires. Récemment, des travaux de recherche académiques ont traité le sujet de conception des pilotes pour les systèmes monopuces. Dans cette section, on abordera les différents types de pilotes classiques et les travaux de recherche concernant la conception des pilotes pour les systèmes monopuces.

a- Les différents types des pilotes classiques

Dans les architectures des ordinateurs personnels (PC), on distingue quatre types principaux de pilote de périphérique :

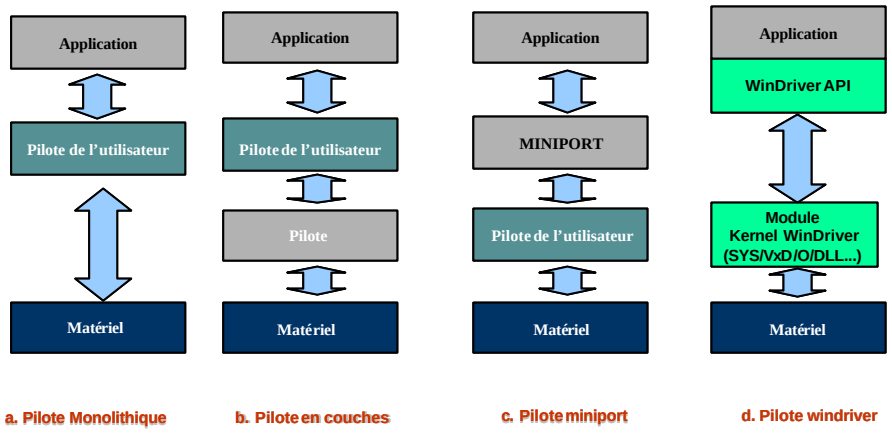


Figure 11. Les différents types de pilotes de périphérique classiques utilisés pour les PC

a-1 Pilote monolithique

Ce type de pilote est utilisé essentiellement pour un type de matériel donné. Ce pilote permet à l'application d'accéder au matériel directement. Comme l'indique la Figure 11 (a), un pilote monolithique est très proche de l'application et de la plateforme architecturale. La communication avec l'application est assurée par des commandes de contrôle d'entrées-sorties. La communication avec le matériel est assurée par l'appel des fonctions systèmes. La dépendance directe de ces pilotes avec le matériel et l'application est la cause de leur faible portabilité sur différentes architectures.

Les pilotes monolithiques existent dans divers systèmes d'exploitation commerciaux comme Windows et Unix.

a-2. Pilote en couches

Les pilotes en couches font partie d'une pile de pilotes (Figure 11 (b)). Le comportement de chaque pilote de cette pile est complémentaire au pilote de la couche de dessus. Un pilote de contrôle d'accès

à un disque est un exemple de pilote en couches. En effet, il est composé d'un pilote d'interception de requêtes d'accès et d'un autre pilote chargé de crypter-décrypter les données avant leur transfert.

Les pilotes en couches sont aussi appelés les pilotes filtres et sont supportés par Windows 95/98/Me.

a-3. Pilote Miniport

Certains fabricants fournissent pour chaque type de matériel une classe de pilotes. Chaque classe implémente les fonctions communes nécessaires pour accéder à tous les périphériques de la même famille. Le pilote miniport correspond au code d'accès spécifique à un matériel donné. Pour chaque périphérique, l'utilisateur doit écrire les fonctions d'entrées-sorties qui n'ont pas été fournies par le pilote du fabricant du matériel (Figure 11 (a)).

Windows NT/2000/XP fournissent plusieurs classes de pilotes appelées aussi ports. Le pilote NDIS ("Network Driver Interface Specification") est un exemple de pilote miniport utilisé pour les accès réseaux. Le noyau du système d'exploitation NT fournit toutes les fonctionnalités communes aux différentes piles de communication.

a-4. Pilote windriver

Les pilotes windriver sont plus flexibles que les autres types de pilotes déjà mentionnés. Ils sont supportés par plusieurs systèmes d'exploitation (Windows, VxWorks, Linux, …) et par plusieurs ressources de communication (PCI, ISA, USB …). Cette flexibilité est assurée par la séparation entre l'application, le système d'exploitation et le matériel. Comme l'indique la Figure 11 (c), un pilote windriver est composé de deux parties :

- Une API permettant à l'application d'accéder directement au périphérique sans le recours aux services du noyau.
- Un module fournissant des services systèmes d'accès au périphérique.

Les pilotes de périphériques classiques sont conçus essentiellement pour les architectures multiprocesseurs générales et les ordinateurs personnels (PC). Les outils de développement des pilotes classiques fournissent une réduction importante en temps de conception, mais ils restent trop généraux pour fournir des pilotes spécifiques. Dans le cas des systèmes monopuces spécifiques à une application donnée, ces pilotes classiques peuvent dégrader les performances du système. La conception de nouveaux pilotes d'accès spécifiques à une application donnée est devenue cruciale.

b- Conception des pilotes logiciels pour les systèmes monopuces

Plusieurs travaux de recherche ont été menés pour trouver de nouvelles méthodes de conception de pilotes monopuces.

[Bor98][Cho99][Nie98] traitent la génération automatique des pilotes de périphérique. Les pilotes sont configurés et générés en utilisant une bibliothèque logicielle contenant des procédures de communication d'entrées-sorties. L'implémentation de ces pilotes est très liée au matériel. Cette forte dépendance limite la flexibilité des pilotes qui ne peuvent pas être réutilisés pour d'autres matériels.

[Kat99] présente une méthode de génération de pilotes plus flexible. La spécification d'un pilote a été découpée en trois parties indépendantes : une partie liée au système d'exploitation, une partie liée au matériel en question et une partie liée à la classe du pilote. L'inconvénient de cette approche est qu'elle génère des pilotes généraux et non spécifiques à l'application.

Dans les outils de conceptions systèmes comme Coware [Bol97] et Polis [Bal97] , la génération des pilotes consiste à capturer les fonctions d'entrées-sorties d'une bibliothèque écrite en C. Chaque fonction est associée à un scénario de communication donné et à un contrôleur d'entrées-sorties d'un processeur. La flexibilité de ces outils reste limitée. En effet, l'ajout d'une nouvelle fonction d'entrées-sorties dans la bibliothèque par l'utilisateur est très difficile. Cette difficulté vient du fait que l'utilisateur doit connaître les détails du matériel, les caractéristiques du scénario de communication spécifique à chaque port et le système d'exploitation utilisé.

O'Nills [Oni01] présente une méthode de génération de pilotes logiciels basée sur la séparation du comportement et de la communication. Le comportement des pilotes est spécifié sous forme d'expressions régulières en Langage ProGram [Obe96]. A partir de cette spécification, un code C est généré spécialement au matériel en question. L'écriture d'un nouveau pilote pour un nouveau matériel avec cette méthode est plus facile que d'utiliser les autres méthodes déjà mentionnées. Malheureusement cette méthode n'est pas complètement automatisée.

Dans I2O (Intelligent Input Output) [I2O02], l'architecture logicielle d'un pilote de périphérique est présentée sous forme de trois modules superposés (Figure 12) :

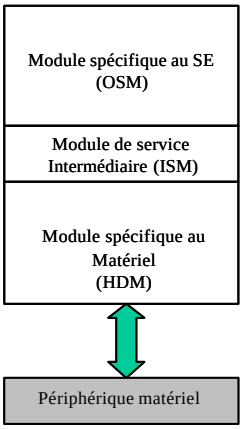


Figure 12. Architecture I2O d'un pilote logiciel

- Un module qui dépend du système d'exploitation. Ce module est aussi appelé *OSM* ("Operating system Service Module").
- Un module dépendant du matériel. Ce module est généralement appelé *HDM* ("Hardware Device Module").
- Un média de communication entre les deux autres modules appelés *ISM* ("Intermediate Service Module"). Cette couche de communication est basée sur un protocole d'envoi de message (*message passing*). Ce protocole est choisi pour sa flexibilité. Le concepteur a le libre choix d'implémentation.

Cette conception modulaire sert à fournir une interface standard entre les systèmes d'exploitation et les pilotes de périphériques. Le but de cette séparation modulaire est la création de pilotes uniques pour tous les systèmes d'exploitation du marché. Les développeurs des systèmes d'exploitation fourniront pour chaque classe de périphériques un pilote I2O OSM. Les constructeurs de périphériques, quant à eux, développeront un pilote unique HDM, capable de fonctionner avec n'importe quel système d'exploitation supportant la norme I2O. La limite de cette approche est qu'elle n'apporte aucune réalisation.

ASOG [Gau01] est un outil de génération de système d'exploitation développé pendant les travaux de thèse de Lovic Gauthier dans le groupe SLS. Nous avons profité de la flexibilité de l'environnement de cet outil pour l'utiliser à la génération des pilotes logiciels spécifiques à la gestion mémoire. Plus de détails sur cette partie font l'objet du chapitre 6.

3. Conclusion

Après l'introduction de la conception système, nous avons classé les travaux antérieurs spécifiques à la mémoire en quatre étapes essentielles dans le flot de conception mémoire. Ces étapes sont :

- Optimisation mémoire au niveau fonctionnel.
- Choix d'architecture mémoire.
- Transposition des mémoires logiques en mémoires physiques.
- Intégration des mémoires dans un système monopuce hétérogène.

Dans toutes ces étapes, aucune méthode ne permet une conception automatique et flexible des interfaces logiciel-matériel pour les mémoires. Une nouvelle méthode de conception d'interfaces spécifiques à la mémoire est alors nécessaire.

Chapitre 3 : INTRODUCTION A LA CONCEPTION DES INTERFACES LOGICIEL-MATERIEL POUR LES SYSTEMES MONOPUCES

Ce chapitre introduit la conception des interfaces logiciel-matériel. Il est composé de neuf sections. La première section définit les interfaces logiciel-matériel ainsi que leurs techniques de réalisation. La deuxième et la troisième section présentent les différents niveaux de description d'interfaces et l'abstraction du matériel. Cette abstraction est assurée par un modèle d'architecture virtuelle présenté dans la section quatre. Quant à la cinquième section, elle présente le langage Colif décrivant cette architecture virtuelle. Les sections six et sept détaillent le flot de conception SLS avant et après son extension. La section huit expose notre première contribution en matière de modèles mémoire aux différents niveaux d'abstraction. Une conclusion est donnée dans la neuvième section.

1. Introduction : les interfaces logiciel-matériel

1.1. Définition des interfaces logiciel-matériel

Une interface logiciel-matériel est un adaptateur de communication entre le logiciel exécuté par le processeur et une ressource matérielle (unité de communication, mémoire, contrôleur, …). L'implémentation de ces interfaces peut être faite en logiciel, en matériel, mais elle peut être aussi mixte (logiciel et matériel). Dans le cas des mémoires, le terme interface logiciel-matériel désigne l'adaptation entre le code des accès mémoire de l'application exécutés par le processeur ou par un coprocesseur de communication et le composant mémoire. En d'autres termes, c'est la partie qui permet d'adapter les transferts de données entre les deux parties.

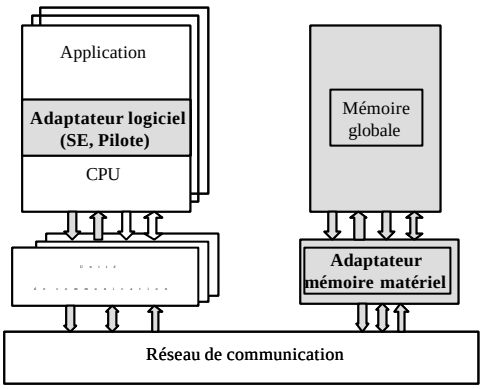


Figure 13. Les adaptateurs mémoire logiciels et matériels

La solution d'implémentation retenue est une solution mixte (Figure 13). Une première adaptation de type logiciel est utilisée entre l'application et le processeur. Cette adaptation correspond aux pilotes d'accès et aux services d'entrées/sorties du système d'exploitation qui permettent à l'application d'accéder à l'unité de communication du processeur. Cette unité peut être un simple contrôleur d'entrées/sorties embarqué dans le cœur du processeur ou un coprocesseur de communication externe.

Une deuxième adaptation de type matériel est utilisée entre le réseau de communication et le composant mémoire. Ceci permet d'adapter le protocole de transfert de données du réseau à celui de la mémoire.

1.2. Les différents modèles d'interfaces et de signalisations entre le logiciel et le matériel

1.2.1. Les modèles d'interfaces logiciel-matériel "memory mapped IO"

Dans ce modèle, l'interface entre la partie logicielle et la partie matérielle est basée sur l'utilisation d'un espace mémoire partagé. Ce modèle d'interface peut être implémenté par l'utilisation d'une mémoire partagée ou par l'utilisation d'un espace d'entrées-sorties dans une mémoire virtuelle.

a- Utilisation de mémoires partagées

La communication est assurée par l'utilisation des instructions d'accès mémoire du processeur du type *load/store*. Le processeur écrit le message dans une mémoire partagée et le périphérique matériel vient lire le message dans cette mémoire. Cette solution suppose que le processeur et les périphériques matériels sont connectés au même bus mémoire. Ceci suppose aussi qu'il n'y a pas de problème d'adaptation entre l'interface de la mémoire et celle des périphériques utilisés, ce qui n'est pas toujours vrai dans le cas des systèmes monopuces. Un autre inconvénient de cette solution est le coût de la taille mémoire qui est assez important.

b- Utilisation de mémoire virtuelle d'E/S

Cette solution consiste à utiliser un espace mémoire virtuel spécifique aux périphériques matériels d'entrées/sorties. L'adresse envoyée par le processeur contient une partie qui correspond à un périphérique donné. Cette adresse ne correspond à aucune adresse physique dans la mémoire du système. Un comparateur logique est généralement utilisé pour décoder l'adresse envoyée par le processeur, sélectionner le périphérique approprié et lui transférer les données. L'avantage de cette solution est qu'il n'y a pas un surcoût de mémoire durant le transfert des données entre la source et la destination.

1.2.2. Les modèles de signalisations pour la synchronisation logiciel-matériel

La façon de faire remonter les événements générés par un périphérique matériel (contrôleur mémoire, contrôleur DMA, …) au niveau logiciel de l'application, nécessite des mécanismes de signalisations. On distingue deux mécanismes de signalisations principaux : la signalisation par scrutation (*polling*) et la signalisation par interruption matérielle.

a- La scrutation

Ce modèle de signalisations consiste à une scrutation directe et périodique (par le processus logiciel utilisateur) des registres d'état du périphérique lorsqu'ils sont en attente d'un événement venant de ce dernier. Ces registres contiennent des drapeaux indiquant la fin d'une émission ou d'une réception d'une donnée. L'accès à ces registres peut se faire soit par accès direct à la mémoire physique, soit par l'accès à la mémoire

virtuelle des E/S projetées dans l'espace mémoire du système d'exploitation ou/et dans l'espace utilisateur du processus. L'inconvénient de la scrutation est qu'elle peut occuper les ressources du processeur dans le cas de plusieurs périphériques. Dans le cas d'un système monopuce de calcul intensif et contenant plusieurs mémoires, des contrôleurs DMA et un grand nombre d'autres périphériques, la solution de scrutation ne semble pas la solution idéale. Ceci est à cause de la surexploitation des ressources du processeur pendant la scrutation.

b- Les interruptions matérielles

Pour éviter la surexploitation des ressources du processeur pendant la scrutation, certains processeurs utilisent un contrôleur d'interruptions matérielles qui se charge de la détection d'une interruption matérielle. Dans ce modèle de signalisation, le périphérique matériel génère une interruption matérielle pour signaler un événement. Ce signal est pris en compte directement par le gestionnaire d'interruptions du système d'exploitation ou par le biais d'un contrôleur d'interruptions matérielles embarqué dans le cœur du processeur. Une fois détecté, le signal est donc remonté au processus utilisateur concerné en lui envoyant un signal ou par la modification d'un drapeau dans la mémoire virtuelle du noyau projeté éventuellement sur l'espace utilisateur.

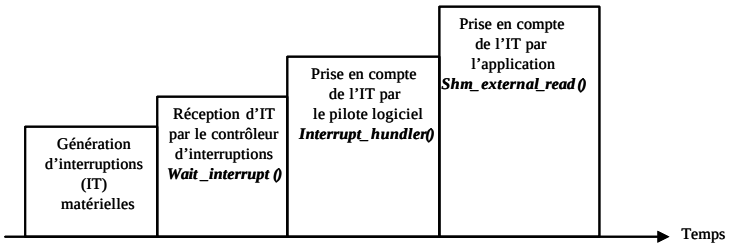


Figure 14. Exemple de signalisation logiciel-matériel par interruption matérielle

La Figure 14 représente un exemple simple de signalisation logiciel-matériel en utilisant des interruptions matérielles. Lors d'un accès en lecture à une mémoire, le processus source de cet accès doit être averti lorsque la donnée est valide sur le bus de données du processeur. Quand la donnée est lue dans la mémoire, l'adaptateur mémoire matériel place la donnée sur le bus de données et génère une interruption matérielle indiquant que la donnée est prête. Le contrôleur d'interruptions est en attente active d'un événement matériel externe (*Wait_interrupt*). Quand l'événement se produit, le contrôleur d'interruptions active le pilote logiciel approprié (*Interrupt_handler*) qui va propager le signal jusqu'au processus source de l'application via l'utilisation des services de communication (*Shm_external_read()*). A cet instant, le flot courant d'exécution est interrompu (changement de contexte) et le processeur peut lire le bus de données.

2. Niveaux d'abstraction pour la conception d'interfaces logiciel-matériel

Avec le grain fin des interfaces et de la communication à un bas niveau de description, la conception RTL des interfaces de communication entre le logiciel et le matériel est devenue plus complexe et plus difficile à maîtriser. L'abstraction de ces interfaces est la clef de voûte qui permet la maîtrise de cette complexité. En effet, nous pouvons réduire la difficulté de la conception des interfaces en faisant abstraction des détails de la communication et du comportement de bas niveau. Le concepteur peut partir d'une spécification abstraite des interfaces pour arriver à une implémentation au niveau RTL après des raffinements progressifs. Pour abstraire les interfaces dans le cas d'intégration d'une mémoire dans un système monopuce, nous nous basons sur les niveaux d'abstraction définis dans les travaux de thèse de Gabriela Nicolescu [Nic02]. Parmi tous les niveaux d'abstraction définis, nous ne tenons compte que des trois niveaux principaux. On distingue le niveau transactionnel, le niveau architecture virtuelle (appelé aussi macro-architecture) et le niveau RTL appelé aussi niveau micro-architecture.

2.1. Niveau transactionnel

Les différents modules du système communiquent via un réseau explicite de canaux de communication qui sont dits actifs. En plus de la synchronisation, ces canaux peuvent aussi disposer d'un comportement complexe, comme par exemple la conversion des protocoles spécifiques aux différents modules communicants. Les détails de la communication sont englobés par des primitives de communication de haut niveau (par exemple send/receive) et aucune hypothèse sur la réalisation des protocoles de communication n'est faite. Des exemples de langages modélisant les concepts spécifiques à ce niveau sont SDL (System Description Language) [Sdl87], UML (Unified Modeling Language) [Uml02] et ObjecTime [Obj00].

2.2. Niveau architecture virtuelle

Ce niveau est spécifique à la communication par des fils abstraits, englobant des protocoles de pilotes d'entrées/sorties (registre, file). Un modèle de ce niveau implique par conséquent le choix d'un protocole de communication et la topologie des interconnexions. Des exemples typiques de primitives de communication sont : l'envoi d'un message, l'écriture d'une valeur ou l'attente d'un événement ou message. Les langages caractérisant le mieux un système à ce niveau d'abstraction sont : CSP [Hoa85], SystemC 1.1 [Sys00], Cossap [Syn02] et StateCharts [Har87].

2.3. Niveau micro-architecture

A ce niveau d'abstraction la communication est réalisée par des fils et des bus physiques. La granularité de l'unité de temps devient le cycle d'horloge et les primitives de communication sont set/reset sur des ports et l'attente d'un nouveau cycle d'horloge. Les langages les plus utilisés pour la modélisation des systèmes à ce niveau sont SystemC 0.9-1.0, Verilog [Moo98] et VHDL [Iee93].

3. Abstraction des détails de réalisation du matériel

Pour réduire la complexité de la conception des adaptateurs logiciels, une abstraction des détails de la réalisation du matériel est nécessaire. Comme l'indique la Figure 15, notre architecture logicielle est définie comme une superposition de plusieurs couches. Cette architecture est qualifiée de portable parce que le code de l'application peut être exécuté sur différents processeurs sans changement. Elle est aussi qualifiée de flexible grâce à la modularité de l'implémentation. En effet, si on change le type du processeur ou le type du système d'exploitation, on ne réécrit pas tout le code logiciel, mais on ne change que quelques modules logiciels et les autres restent intacts.

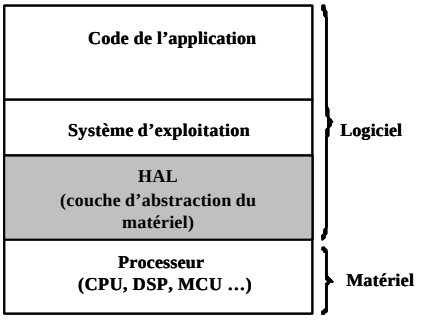


Figure 15. Architecture logicielle portable et flexible

La communication entre les couches d'application et le système d'exploitation sont séparés de l'architecture matérielle par une couche logicielle appelée "HAL" ("Hardware Abstraction Layer"). Cette couche représente la projection des informations sur l'architecture matérielle. Ce découplage entre le logiciel de l'application et le matériel permet d'avoir une conception concurrente et permet d'assurer la portabilité du logiciel sur différents processeurs. La spécification et l'implémentation de cette couche d'abstraction de matériel seront détaillées dans le chapitre 6.

4. Modèle d'architecture virtuelle pour les systèmes multiprocesseurs monopuces

Au niveau RTL, la représentation de données est au niveau du bit et la fonctionnalité des interfaces d'entrées/sorties est décrite au niveau du cycle d'horloge. Cette granularité fine qui caractérise une architecture RTL ne fait qu'augmenter la complexité de la conception des interfaces logiciel-matériel au niveau RTL. L'abstraction de ces interfaces à un niveau plus élevé que le RTL tel que le niveau architecture virtuelle est alors nécessaire. Cette abstraction est basée sur la séparation entre le comportement et la communication à partir d'un haut niveau de description. Cette séparation avancée dans le flot de conception permet d'alléger les

charges des modules de calcul en leur évitant la lourde tâche de communication. Aussi, cette séparation permet-elle d'avoir une grande flexibilité architecturale. En effet les interfaces des modules deviennent complètement indépendantes de celles des ressources de communication utilisées ; ce qui facilite l'intégration de modules prédéfinis comme les mémoires ou autres.

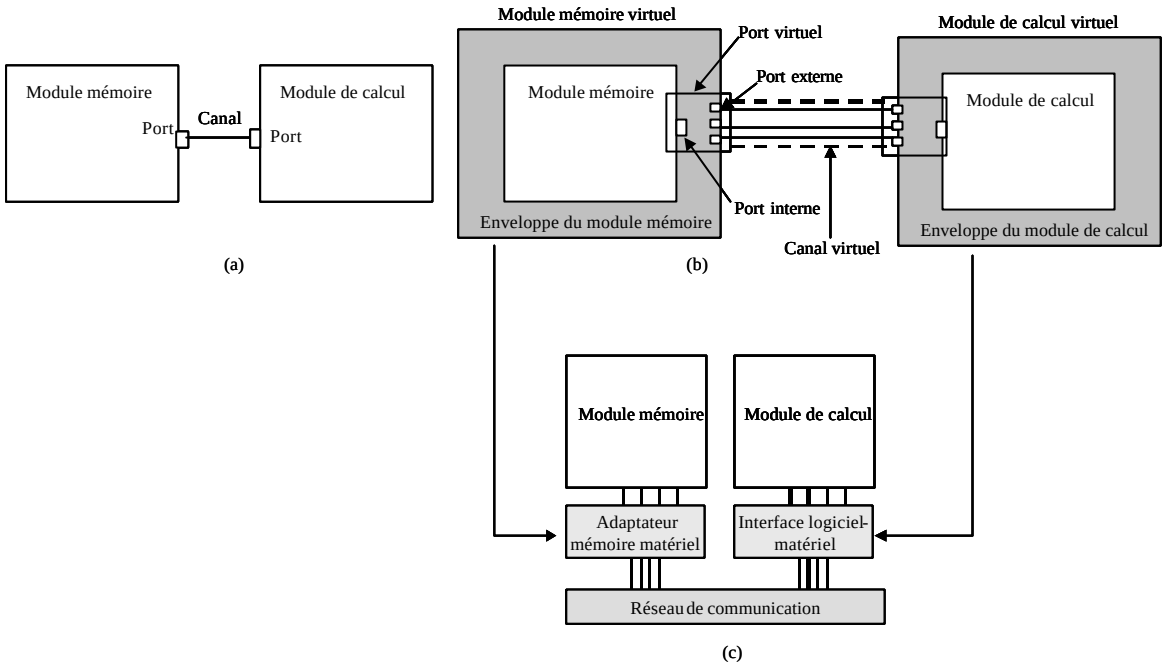


Figure 16. Les concepts de la spécification et de l'implémentation des interfaces pour les systèmes hétérogènes : (a) concept conventionnel, (b) concept d'enveloppe, (c) implémentation

Pour réaliser cette séparation, nous réutilisons les concepts conventionnels (module, port, canal) de la conception d'interfaces logiciel-matériel (Figure 16 a), et nous ajoutons une enveloppe d'adaptation entre un port d'un module et un canal de communication (Figure 16 b). Cette enveloppe permet de découpler l'interface du module de celle de la communication à un haut niveau d'abstraction. Elle a la charge d'adapter les ports du module aux différents protocoles du canal de communication. L'implémentation de cette enveloppe de haut niveau correspond à son raffinement en des adaptateurs de communication décrits au niveau RTL (Figure 16 c). L'implémentation de l'enveloppe du module mémoire correspond à un adaptateur matériel, tandis que l'implémentation de l'enveloppe du module de calcul correspond à un adaptateur logiciel-matériel (SE, pilotes logiciels et contrôleur de communication matériel).

L'ajout de ce modèle d'enveloppe nécessite la définition de trois nouveaux concepts pour la spécification d'interfaces logiciel-matériel : module virtuel, canal virtuel, et port virtuel.

4.1. Module virtuel

Un module virtuel est l'association d'un module et de son enveloppe. Comme l'indique la Figure 17 (a), la vue conceptuelle des modules virtuels de calcul est représentée comme la superposition d'une couche de calcul réalisant la fonctionnalité d'une application donnée et d'une couche de services de communication logiciel-matériel implémentée par l'enveloppe. Cette dernière isole complètement les modules de calcul des ressources de communication hiérarchiques qui peuvent être composées de plusieurs canaux de communication.

Un module mémoire virtuel est représenté comme la superposition d'une couche de mémorisation et d'une couche d'adaptation matérielle assurant la connexion de l'interface de la mémoire à celle des ressources de communication (Figure 17 b).

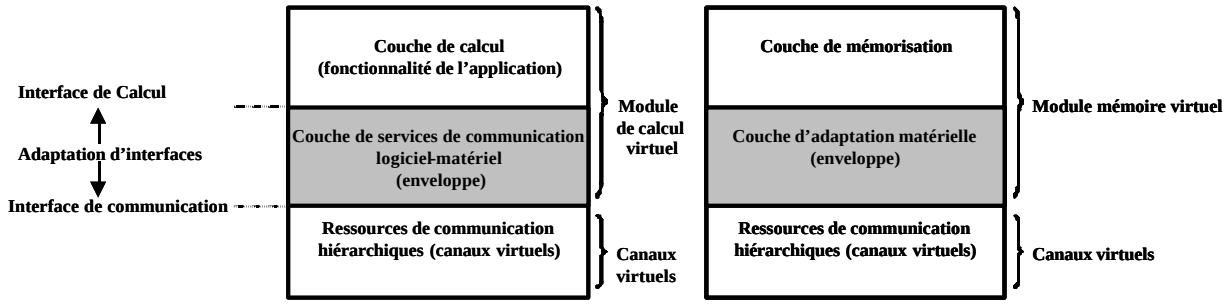


Figure 17. Vue conceptuelle des modules virtuels : (a) module de calcul, (b) module de mémorisation

L'adaptation des interfaces des modules mémoire et des interfaces des ressources de communication est assurée par l'enveloppe qui fournit des services de communication aux deux couches séparées. Dans cette thèse, nous réalisons cette enveloppe de communication en matériel et en logiciel. D'une part, la partie matérielle correspond aux adaptateurs matériels qui adaptent les modules mémoires aux réseaux de communication. D'autre part, la partie logicielle correspond aux pilotes d'accès qui adaptent le code de l'application aux processeurs cibles.

4.2. Port virtuel d'un module mémoire

Un port virtuel d'un module mémoire est le regroupement des ports internes et des ports externes de l'enveloppe de la mémoire. Un port interne correspond à un port du module et un port externe correspond à celui du canal de communication. Le nombre de ports internes est égal au nombre de ports du module, tandis que le nombre de ports externes est égal au nombre de canaux de communication. Une enveloppe connecte des ports internes et externes en nombres différents. Dans un port virtuel, nous pouvons avoir *m* ports internes et *n* ports externes (n, m sont des entiers naturels). Les ports internes et externes d'un port virtuel

peuvent être spécifiques à des niveaux d'abstraction différents ou liés à des scénarios de communication divers. Le contrôle de partage des ports et l'adaptation des protocoles de transfert de données entre ces ports sont assurés par la fonctionnalité de l'enveloppe. Comme l'indique la Figure 18, si la direction du port du module est "*out*" alors le port interne du port virtuel est de direction " *in* ". De la même façon, si le port d'un canal est de direction "*in*", alors la direction du port externe du port virtuel est l'opposé ("*out*"). Si un port d'un module ou d'un canal est maître (resp. esclave), alors le port correspondant du port virtuel est esclave (resp. maître).

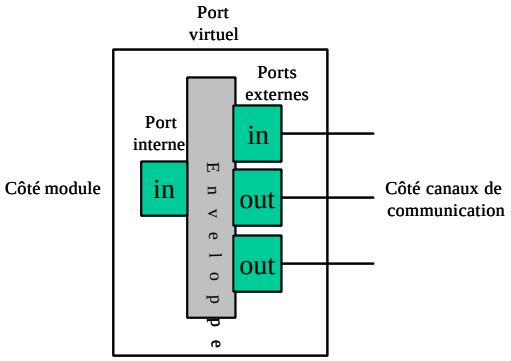


Figure 18. Vue interne d'un port virtuel de module mémoire

4.3. Canal virtuel de communication

A chaque port virtuel est associé un et un seul canal hiérarchique appelé canal virtuel. Ce dernier regroupe tous les canaux de communication reliés aux ports externes d'un port virtuel. Un canal peut contenir des canaux abstraits ou/et des fils physiques. Les données véhiculées par un canal virtuel peuvent être une structure de données abstraites (trame d'entiers) ou/et une suite de valeurs binaires.

5. COLIF : modèle de représentation pour la spécification des interfaces logiciel-matériel

Colif ("COdesign Language Independant Format") est un format de description d'architecture logiciel-matériel permettant de supporter des spécifications hétérogènes. En effet, Colif permet de décrire des spécifications multi-langages contenant des modèles de communication de nature différente et spécifiques à des niveaux d'abstraction différents. Le modèle de représentation Colif est basé sur la séparation entre le comportement et la communication. Le modèle Colif est le résultat d'un travail de groupe qui ne fait pas partie des contributions de cette thèse [Ces01]. Dans cette section, on donne juste un aperçu sur la modélisation et l'implémentation de Colif.

5.1. Modélisation de Colif

L'environnement logiciel de Colif est basé sur le concept des objets du langage C++. Sa modélisation est basée sur le concept des diagrammes UML. Colif utilise une syntaxe uniforme pour représenter les systèmes hétérogènes impliquant plusieurs niveaux d'abstraction ou langages de spécification. Chaque module est défini par son interface (qui consiste en un ensemble de ports) et son contenu. Le contenu d'un module est une liste des instances de modules ou un comportement. Les modules sont représentés comme des boîtes blanches s'ils contiennent d'autres instances ou comme des boîtes noires si leur contenu est inconnu.

La Figure 19 illustre une représentation simplifiée du diagramme des classes Colif, en utilisant des notations UML. Dans ce diagramme, le modèle d'objets Colif est divisé en deux parties principales : une partie déclarative et une partie d'instanciation. La partie déclarative représente des objets définissant un modèle ("template") réutilisable avec des définitions de propriétés génériques ("PARAM_DEFS") et des valeurs par défaut pour certaines propriétés ("PARAMETER"). La partie d'instanciation représente les objets utilisés pour la construction d'un arbre d'instances.

Comme principe général, chaque concept de base de Colif – MODULE, PORT, NET est divisé en deux parties : une interface (ENTITY) et le contenu (CONTENT). L'entité a un type (TYPE) qui encapsule des propriétés définissables par l'utilisateur. Le contenu CONTENT contient une référence vers un comportement (interne ou externe) défini et/ou une liste des déclarations DECL d'objets.

5.2. Implémentation de Colif

Pour la réalisation de Colif, le langage XML (Extensible Markup Language) a été utilisé [XML00]. XML, métalangage standardisé par W3C, connaît aujourd'hui un grand succès dans le monde de l'informatique grâce à sa souplesse, sa clarté et sa facilité à être analysé. Il facilite l'intégration des outils et l'échange d'informations entre les différents environnements de synthèse.

XML est basé sur le concept de balises : toute information est délimitée par une balise ouvrante et une balise fermante. Les balises XML peuvent avoir des attributs et peuvent être hiérarchiques. Pour obtenir un langage dérivé de XML, il faut définir des balises particulières et préciser les contraintes qui leur sont associées. Ces contraintes concernent leurs attributs ou la hiérarchie. Ces informations sont à donner dans un fichier à part nommé "*dtd*". Ce langage possède deux caractéristiques intéressantes pour le but fixé : c'est un méta-langage qui, grâce au système de *dtd*, n'a besoin que d'un seul analyseur quelque soit le langage dérivé et c'est un standard recommandé pour la représentation de données. XML est cependant plus une notation qu'un langage, en particulier il ne dispose pas de sémantique.

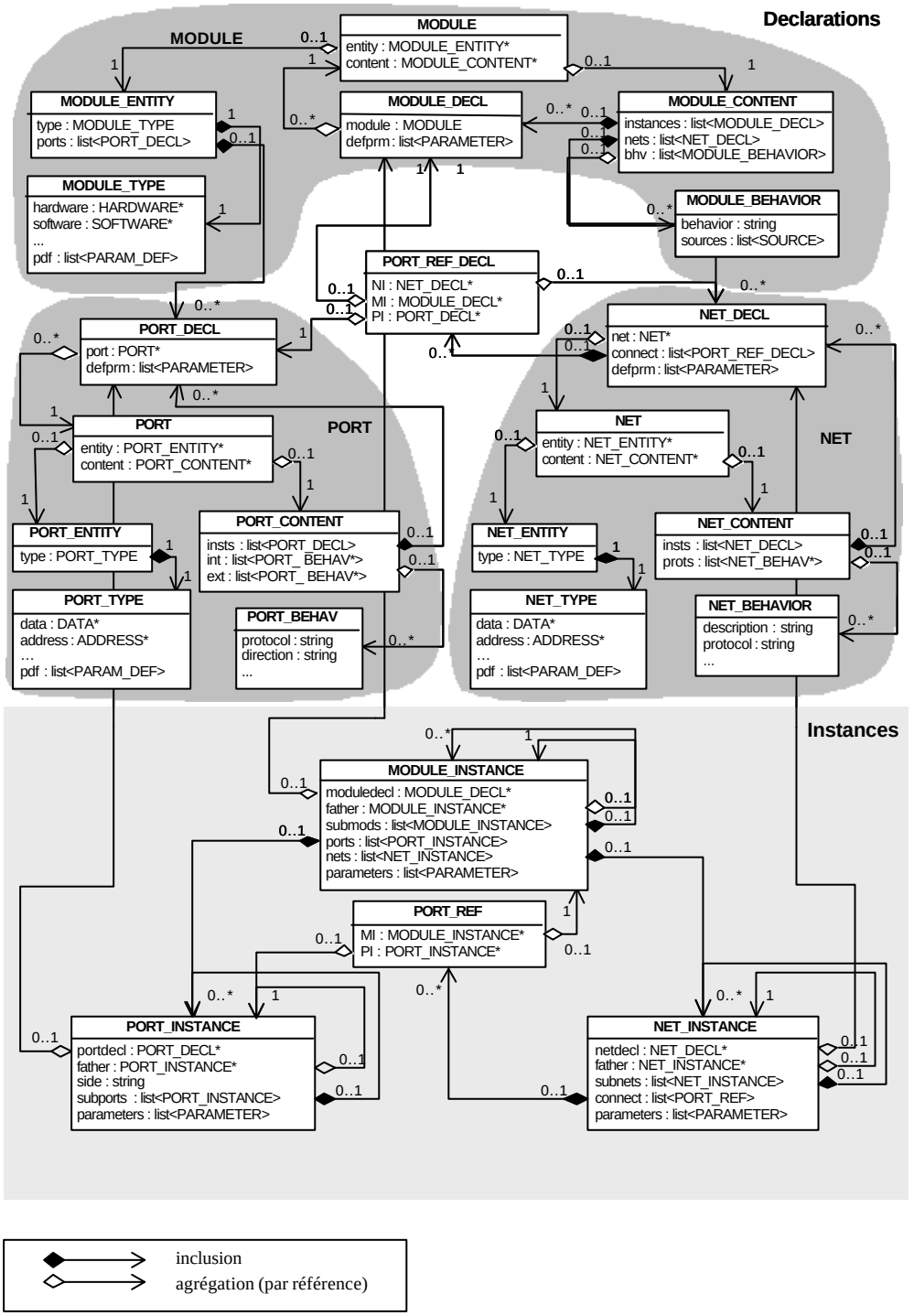


Figure 19. Diagramme de classes de Colif

6. Flot de conception SLS des systèmes monopuces avant son extension

Le flot de conception du groupe SLS débute au niveau transactionnel, après que le partionnement logiciel/matériel ait été décidé. Il se termine au niveau micro-architecture, où une classique étape de compilation et de synthèse logique permet d'obtenir la réalisation finale du système. C'est un flot descendant qui permet de simuler le système à tous les niveaux d'abstraction et de revenir en arrière à chaque étape. En entrée, le flot prend une description de la structure et du comportement de l'application au niveau transactionnel décrite en langage VADeL[5]. La sortie du flot est une réalisation d'une architecture logiciel-matériel au niveau RTL. Cette architecture finale est une architecture multiprocesseur avec des mémoires locales et une mémoire globale.

Comme l'indique la Figure 20, le flot est composé de quatre étapes essentielles.

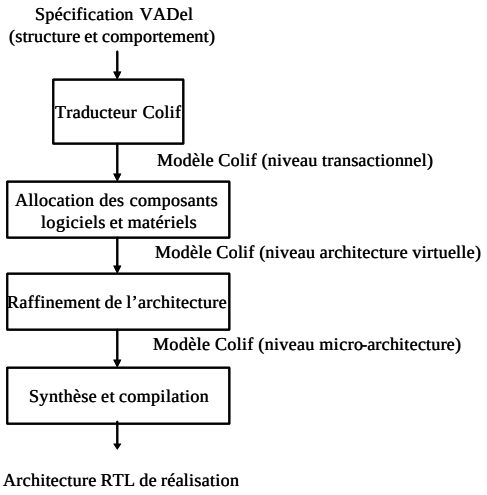


Figure 20. Une vue globale et simplifiée du flot de conception du groupe SLS

- **Traduction en format Colif :** la description d'entée est traduite dans le format intermédiaire Colif pour découpler le flot du langage de la spécification d'entrée. La sortie de cette étape de traduction est un modèle Colif décrit au niveau transactionnel. Ce modèle correspond à une architecture abstraite composée de plusieurs modules communicant via des canaux de communication abstraits. La nature des modules ainsi que celle de la communication n'est pas explicite. En effet, les types des composants spécifiques aux modules de calcul et de mémorisation ainsi que les protocoles spécifiques à la communication ne sont pas encore fixés.

[5] VADeL ("Virtual Architecture Description Language") est une extension du langage SystemC.

- **Allocation des composants logiciels et matériels :** cette étape prend en entrée la description transactionnelle et produit une architecture virtuelle en format Colif. Elle consiste à allouer les composants logiciels et matériels nécessaires au calcul, à la mémorisation et à la communication. A ce niveau, les éléments de mémorisation sont explicites et correspondent à des modules mémoire dont le type est fixe (SRAM, DRAM, …). La communication entre ces modules mémoire et le reste de l'architecture est encore abstraite, mais elle est explicite. En effet, elle correspond à des protocoles de communication spécifiques (FIFO, MemFullHndshk, …).
- **Raffinement de l'architecture virtuelle :** cette étape consiste à raffiner la communication entre les différents modules de l'architecture virtuelle au niveau micro-architecture. Ceci inclut la génération des interfaces logicielles et matérielles. Ces interfaces correspondent aux adaptateurs matériels et logiciels qui ne sont que la réalisation des enveloppes de modules virtuels. Malheureusement, le flot ne prenait pas en compte la conception des interfaces nécessaires à l'intégration de mémoires. La solution de ce problème est la contribution clef de cette thèse. Elle consiste à étendre le flot SLS pour qu'on puisse intégrer les mémoires dans les systèmes monopuces.
- **Synthèse et compilation :** cette étape consiste à employer les outils de synthèse et de compilation classiques. Le code RTL de la partie matérielle (adaptateurs matériels, contrôleurs, …) est synthétisé et le code du comportement des parties logicielles (tâches, SE, pilotes, …) est compilé.

6.1. Spécification d'entrée du flot de conception (décrite au niveau transactionnel)

La spécification d'entrée du flot décrit la structure et le comportement du système logiciel-matériel au niveau transactionnel ou aux niveaux inférieurs. Il est aussi possible qu'elle décrive un système hétérogène combinant plusieurs niveaux d'abstraction. Cette spécification est annotée par des informations nécessaires à la génération des interfaces logiciel-matériel.

Cette spécification est décrite en langage VADeL ("Virtual Architecture Description Language"). Ce langage est une extension du langage SystemC. Ce dernier est basé sur le langage C++, étendu avec des bibliothèques permettant la modélisation et la simulation de systèmes logiciel-matériel globalement synchrones ou asynchrones, avec un modèle à événements proche de celui du VHDL. La raison d'utilisation de VADel est qu'il permet de spécifier les interfaces logiciel-matériel des systèmes hétérogènes en utilisant les concepts : module virtuel, canal virtuel et port virtuel. L'autre raison est que VADeL permet d'annoter la spécification de l'application par des paramètres architecturaux décrivant les interfaces du système à un haut niveau d'abstraction dans le flot.

6.2. Sortie du flot

La sortie du flot correspond à une implémentation RTL de la spécification raffinée de l'architecture virtuelle. Les modules virtuels de calcul sont remplacés par des cœurs de processeur et les canaux de communication virtuels sont raffinés jusqu'à une réalisation d'un réseau de communication (Bus partagé, point à point, …). Les interfaces logiciel-matériel correspondent aux codes du système d'exploitation, aux pilotes logiciels et aux adaptateurs matériels.

6.3. Les différentes étapes du flot de conception

Les différentes étapes du flot de conception sont détaillées dans la Figure 21 (a).

6.3.1. Traduction vers Colif

Pour découpler le flot de conception du langage de spécification, une étape de traduction de la description d'entrée (VADeL) vers le format Colif est nécessaire. La description Colif contient des informations structurelles sur la décomposition hiérarchique du système ainsi que les interconnexions entre les modules. L'information sur les chemins des fichiers externes réalisant le comportement des modules est également contenue dans cette description. Ces informations guident les outils utilisés pour la réalisation de l'architecture finale tout au long du flot. L'étape de traduction a été automatisée grâce aux outils développés dans l'équipe par Wander Cesàrio [Ces01].

6.3.2. Allocation mémoire et synthèse de la communication

L'entrée de cette étape est la spécification initiale du système (au niveau transactionnel) en format Colif. La sortie de cette étape est une architecture virtuelle (en format Colif) dont les éléments de calcul, de mémorisation et de communication sont explicites. Cette étape permet de raffiner la spécification initiale du système décrite au niveau transactionnel jusqu'au niveau architecture virtuelle en allouant les modules mémoires et en synthétisant la communication entre les différents éléments de l'architecture.

a- Allocation mémoire

La question qui se pose au concepteur est de savoir quel type de mémoire est le mieux adapté pour stocker les données de l'application. Dans le cas des variables locales à un module, la donnée est généralement stockée dans la mémoire locale privée. Par contre, dans le cas d'une variable partagée, échangée par plusieurs processeurs (variable de communication) ou d'une variable globale, le choix d'une architecture mémoire nécessite le développement d'algorithmes judicieux. Une méthode d'allocation mémoire basée sur la programmation linéaire en nombre entier a été présentée dans [Mef01b].

b- Synthèse de la communication

La synthèse de la communication consiste à sélectionner les protocoles de communication et les éléments de calcul (processeurs, ASIC etc.) qui seront utilisés. Dans l'état actuel du flot, aucun outil ne permet d'effectuer cette synthèse de communication automatiquement. Cette opération est donc encore faite manuellement.

Des résultats de simulation au niveau architecture permettent de guider les choix pour les protocoles. À l'avenir, il est prévu d'intégrer une méthodologie et des outils permettant d'automatiser les choix à l'aide d'une bibliothèque de résultats de simulation.

6.3.3. Affectation mémoire

Cette étape consiste à trouver l'organisation "idéale " des données dans les mémoires allouées. Une bonne organisation de données permet d'améliorer le temps des accès mémoires et de réduire la taille de la mémoire utilisée. Cette étape a fait l'objet des travaux de thèse de Samy Meftaly [Mef01a].

6.3.4. Génération des adaptateurs matériels pour les processeurs

La génération d'interfaces matérielles permet d'interconnecter les éléments de calcul aux éléments de communication. Car ces éléments ne sont pas tous compatibles entre eux. Ces interfaces permettent aussi d'implémenter des protocoles de communication non supportés nativement par les éléments. Un outil de génération d'interfaces matérielles a fait partie des travaux de thèse de Damien Lyonnard [Lyo01]. Malheureusement, cet outil ne supportait pas la génération des adaptateurs mémoire matériels.

6.3.5. Génération des adaptateurs logiciels

Les parties logicielles ne peuvent pas être exécutées directement sur les processeurs. L'exécution concurrente de plusieurs tâches sur un même processeur doit être gérée par une couche logicielle appelée système d'exploitation. La génération des systèmes d'exploitation est réalisée par l'assemblage de composants logiciels organisés sous forme de bibliothèque. Cette étape a fait l'objet des travaux de thèse de Lovic Gauthier [Gau01]. Malheureusement, le système d'exploitation seul est incapable de gérer les accès mémoire et les transferts de données. D'où la nécessité d'une autre entité logicielle spécifique à la gestion mémoire.

6.3.6. La validaion

Dans notre flot de conception, on distingue principalement trois types de validation différents : simulation, cosimulation et émulation. Généralement, ces approches de validation rencontrent des problèmes de synchronisation et de perte de données pendant la communication. Ceci est causé par l'absence d'un outil de validation locale des interfaces de communication.

a- La simulation

Le flot permet d'effectuer des simulations du système à tous les niveaux d'abstraction, mais aussi en mélangeant les niveaux d'abstraction. La technique de base consiste à encapsuler les divers composants à simuler dans des enveloppes de simulation qui adaptent le niveau de leurs communications à celui de la simulation globale. Cette étape a fait partie des travaux de thèse de Gabriela Nicolescu [Nic02].

b- La cosimulation

Grâce aux enveloppes, il est possible d'effectuer des validations globales par cosimulation pour toutes les étapes du flot (ou même pour des systèmes composés de modules décrits à différents niveaux d'abstraction). Plus le niveau d'abstraction est élevé plus la simulation est rapide et plus le niveau est bas et plus la simulation est précise. Cette co simulation est multiniveaux, mais elle peut être aussi multilangage. Elle est basée sur un processus de contrôle décrit en SystemC qui gère l'ensemble des communications et synchronisations entre les simulateurs par le biais de mémoires partagées.

c- L'émulation

En utilisant l'approche de cosimulation, la validation des systèmes complexes à un niveau d'abstraction relativement bas (RTL) s'avère trop lente et non acceptable. En effet l'utilisation des instances logicielles (simulateur) pour simuler le comportement d'un processeur ne permet pas la simulation proche du temps réel. L'émulation est une technique apparue pour accélérer la validation des systèmes logiciel-matériel en utilisant des unités matérielles (FPGA, …). Une méthodologie d'émulation utilisant une plateforme ARM, fait l'objet des travaux de thèse de Arif Sasongko.

7. Extension du flot de conception "SLS"

Pour intégrer les mémoires dans les systèmes monopuces, il nous a fallu étendre le flot de conception SLS. Cette extension correspond à quatre contributions :

1. Définition de plusieurs modèles mémoire aux différents niveaux d'abstraction (Figure 21 (b)). Ces modèles sont simulables et synthétisables (seulement pour les modèles mémoire RTL). Ils sont présentés dans la section 9 de ce chapitre.
2. Le développement d'un outil de génération d'architecture mémoire interne en Colif (Figure 21 (b)). Cet outil permet à l'utilisateur de construire ses propres architectures mémoire sans le besoin de connaître le langage Colif. Ceci offre au flot une flexibilité et une ouverture importante.
3. Egalement, nous ajoutons au flot une étape de génération du code des éléments d'adaptation matériels (Figure 21 (b)).

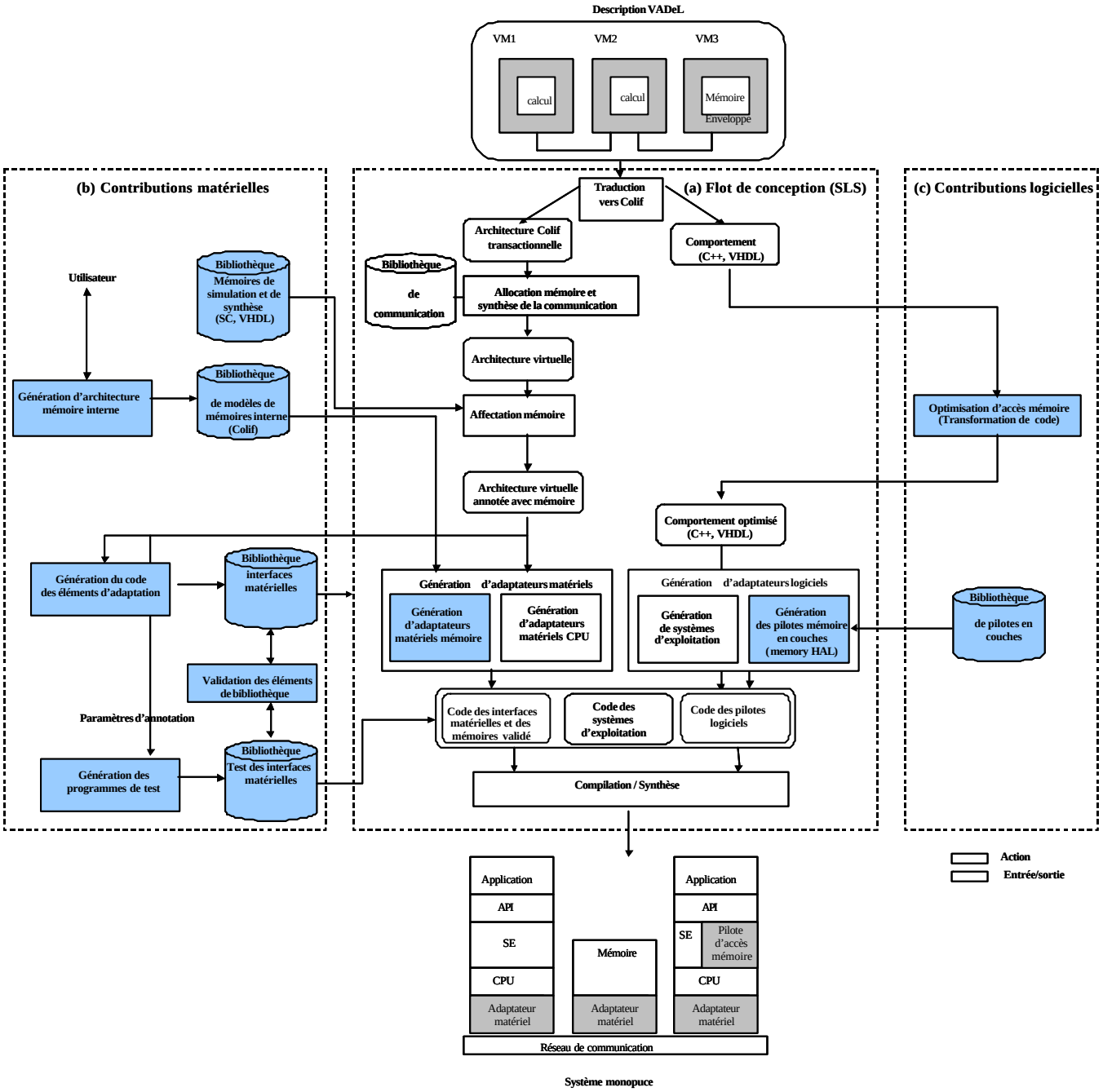


Figure 21. Flot de conception des systèmes monopuces du groupe SLS : (a) avant l'extension, (b) extensions matérielles, (c) extensions logicielles

4. Une étape de génération de programmes de tests spécifiques à la bibliothèque d'adaptation matérielle est aussi ajoutée. Cette étape permet de valider le comportement des interfaces de communication avant la validation globale du système par Co-simulation ou par émulation.

5. Une étape de génération de pilotes d'accès mémoire (appelés aussi : pilotes de gestion mémoire) (Figure 21 (c)).
6. Une étape d'optimisation d'accès mémoire (Figure 21 (c)) consiste à réduire le nombre d'accès mémoire et à éliminer les transferts de données redondants. Ceci est effectué par une transformation du code de l'application après que l'affectation des données soit faite. Ces optimisations ont fait l'objet de mes travaux de DEA [Gha00]. Ces travaux ne seront pas détaillés dans ce manuscrit

8. Modèles mémoire aux différents niveaux d'abstraction

L'utilisation de plusieurs niveaux d'abstraction nous a encouragé à définir des modèles de mémoire spécifiques à chaque niveau d'abstraction. La disponibilité de tels modèles de mémoire permet de réduire le degré d'hétérogénéité (en terme de niveaux d'abstraction) et par conséquent permet de faciliter l'intégration des mémoires. En effet, si le système est décrit au niveau architecture virtuelle, la disponibilité d'un modèle mémoire à ce niveau d'abstraction évite de se soucier de l'interface à cette étape de la conception. Ces modèles mémoire spécifiques à chaque niveau d'abstraction représentent la première contribution de cette thèse.

8.1. Modèle mémoire au niveau transactionnel

A ce niveau d'abstraction, les éléments de mémorisation sont généralement implicites. On trouve dans la spécification de l'application des mémoires implicites sous forme de variables locales aux fonctions, variables globales à plusieurs fonctions et des variables utilisées pour la communication entre les fonctions.

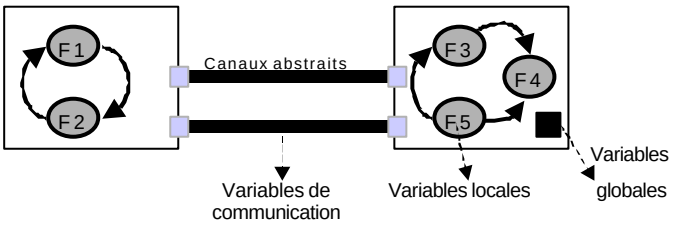


Figure 22. Modèle mémoire au niveau transactionnel

Les interfaces des modules mémoire sont composées des ports d'accès aux canaux actifs. Les ports d'accès fournissent des appels de procédures de haut niveau (par exemple "send", "receive", "put" "get"). Par contre, chaque canal peut envelopper un comportement complexe. Le comportement des tâches élémentaires est un (ou plusieurs) processus qui communiquent avec l'extérieur par des transactions. Le temps de communication n'est pas nul et imprévisible. L'abstraction du temps est réduite à un ordre partiel des envois de messages.

La Figure 22 montre un exemple d'une application au niveau transactionnel. Cette application est composée de deux modules A et B. Le premier contient les fonctions F1 et F2 et le second contient les

fonctions F3, F4 et F5. Le module B possède des variables globales aux fonctions F3, F4, et F5, des variables locales à la fonction F5, ainsi que des variables de communication servant à échanger des valeurs entre les modules A et B. Ces variables de communication sont échangées via les canaux abstraits reliant les deux blocs.

8.2. Modèle mémoire au niveau architecture virtuelle

Au niveau architecture virtuelle, on suppose que la partition logiciel-matériel est déjà faite. Les éléments de mémorisation correspondent à des modules de l'architecture logiciel-matériel qui communiquent avec les autres modules via des canaux logiques. A ce niveau d'abstraction, on trouve des modules mémoires globales explicites qui peuvent être utilisés pour la communication ou/et pour le stockage des données partagées par les autres modules. Les variables globales ou variables de communication sont généralement affectées aux mémoires globales qui ont été allouées. Par contre les variables locales aux modules de calcul sont généralement affectées à des mémoires locales qui correspondent aux mémoires locales des processeurs.

A ce niveau d'abstraction, l'interface mémoire est composée de ports logiques qui ne peuvent être connectés qu'à des canaux de communication abstraits composés de fils abstraits.

Les opérations effectuées sur ces ports sont des envois de messages de type fixe (ex. entier, réel). Ces opérations peuvent cacher le décodage des adresses et la gestion des interruptions. A ce niveau, les opérations d'accès à un port mémoire peuvent durer un temps non nul mais prévisible. A ce niveau, le comportement d'un module mémoire correspond à des machines d'états finis abstraites où chaque transition peut cacher un calcul complexe pouvant prendre plusieurs cycles d'horloge au niveau micro-architecture (traduction d'adresses, conversion de protocoles d'accès, …).

Dans l'exemple de la Figure 23, la mémoire globale est connectée aux fils d'un bus de communication logique via des ports logiques. Cette mémoire contient les variables globales et les variables de communication entre les deux modules logiciel et matériel. Par contre les variables locales restent dans le même module de calcul correspondant, pour être affectées à sa mémoire locale après le raffinement du système.

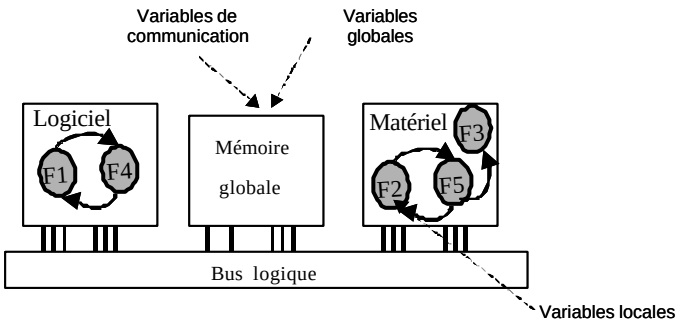


Figure 23. Modèle mémoire au niveau architecture virtuelle

8.3. Modèle mémoire au niveau micro-architecture

Après raffinement de l'architecture virtuelle, les modules de calculs sont remplacés par des cœurs de processeurs (ARM, Motorola, Mips, …), les canaux abstraits deviennent un réseau de communication physique (bus partagé, point à point) et les mémoires logiques deviennent des modules physiques correspondant à des réalisations telles que des SRAMs, DRAMs ou SDRAMs. A ce niveau d'abstraction, les interfaces mémoire correspondent à des ports d'accès physiques. Les opérations effectuées sur ces ports sont du type "*set/reset* " sur les signaux de contrôle et des opérations de lecture et d'écriture d'une structure de données, qui correspond à une suite binaire, sur les lignes d'adresses et de données. La connexion des ports physiques de la mémoire au réseau de communication est assurée via des adaptateurs matériels. A ce niveau d'abstraction, la gestion des interruptions, le décodage des adresses etc. sont explicites. Dans ce cas, l'unité temporelle d'un accès mémoire devient le cycle d'horloge et le comportement de la mémoire correspond à des machines d'états finis où chaque transition est réalisée en un cycle d'horloge.

La Figure 24 donne un exemple de la micro-architecture obtenue en raffinant le système décrit au niveau architecture virtuelle. La mémoire globale est devenue une mémoire SDRAM et les fonctions du module A sont exécutées par un processeur ARM7. Le module B a été remplacé par un processeur MC68000. Une mémoire locale (ROM et/ou RAM) est associée à chaque processeur. Elle est généralement connectée aux bus d'adresses et de données du processeur, ainsi qu'à quelques signaux de contrôle. Un contrôleur peut être nécessaire pour sélectionner et valider la ou les mémoires, mais aussi pour générer les signaux physiques adaptés à la mémoire utilisée. L'adaptation entre les processeurs, la mémoire globale et le réseau de communication est assurée par des adaptateurs matériels (interface de communication) et par des adaptateurs logiciels sous forme de pilotes d'accès exécutés avec le code de l'application par le processeur cible. Les adaptateurs gèrent aussi les conflits d'accès.

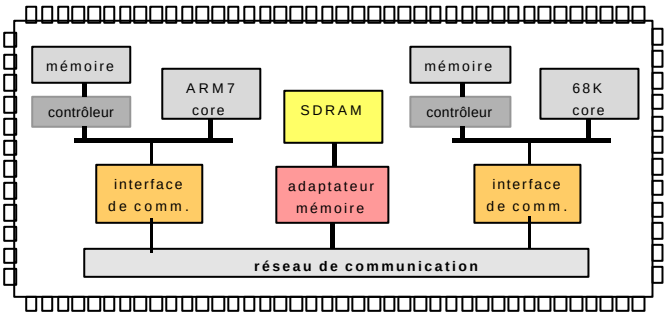


Figure 24. Modèle mémoire au niveau micro-architecture

9. Conclusion

La conception des interfaces logiciel-matériel est le point clef de la conception des systèmes monopuces. A cause de l'hétérogénéité de ces systèmes, l'utilisation des concepts de base conventionnels (module, port et canal) est devenue insuffisante pour spécifier les interfaces logiciel-matériel. Pour cela nous avons défini de nouveaux concepts basés sur l'abstraction des interfaces et la séparation entre le comportement et la communication. Nous avons étendu le flot de conception SLS pour qu'il supporte la génération des interfaces logiciel-matériel spécifiques à la mémoire. La contribution de cette thèse dans ce flot est liée à la génération des adaptateurs matériels et des pilotes logiciels.

Chapitre 4 : ARCHITECTURE GENERIQUE DES ADAPTATEURS MEMOIRE MATERIELS

Pour faciliter la génération automatique des adaptateurs mémoire matériels, nous avons défini une architecture générique qui peut être configurée selon l'application. Ce chapitre détaille cette architecture générique. Il est composé de six sections. La première section introduit la vue conceptuelle et architecturale du modèle de l'adaptateur mémoire matériel que nous proposons ainsi que les motivations du choix de ce modèle. Dans les trois sections suivantes, nous détaillons le rôle, la fonctionnalité et l'implémentation de chacun des composants de cet adaptateur. La section cinq analyse le choix de ce modèle d'adaptateur avec ses avantages et ses inconvénients. Une conclusion est donnée dans la sixième section.

1. Introduction

1.1. Vue conceptuelle des adaptateurs mémoire matériels

Un adaptateur matériel permet d'interfacer la mémoire et le réseau de communication. Il peut être vu comme une superposition de couches implémentant chacune une fonctionna lité donnée. Ce choix conceptuel est motivé par le besoin de la séparation entre l'interface du média de communication externe et celle de la mémoire. Une telle séparation assure la flexibilité d'utilisation ainsi que la réutilisation des adaptateurs mémoire. Comme l'indique la Figure 25, on distingue trois couches :

- **Gestion des protocoles de communication :** cette couche est spécifique au média de communication externe. Elle permet de convertir l'interface de ce demier en une interface intermédiaire fixe qui correspond à celle du média de communication interne de l'adaptateur. Cette couche permet également de préparer les données venant du média externe avant leur envoi.
- **Gestion des accès mémoires :** cette couche est spécifique à la mémoire. Elle assure l'adaptation d'interface entre la mémoire et le média interne. Elle permet également la préparation des adresses et des données lors d'un accès en écriture ou en lecture dans la mémoire. Cette préparation consiste essentiellement à la traduction des adresses virtuelles en adresses mémoires physiques et au transfert de données en utilisant des protocoles d'accès divers (simple, "burst", etc.).
- **Transfert de données :** cette couche correspond au média de communication interne de l'adaptateur mémoire matériel. Elle est la frontière séparatrice entre l'interface de la communication et celle de la mémoire. Elle assure l'acheminement des données entre la couche de gestion des protocoles de communication et la couche de gestion d'accès mémoire.

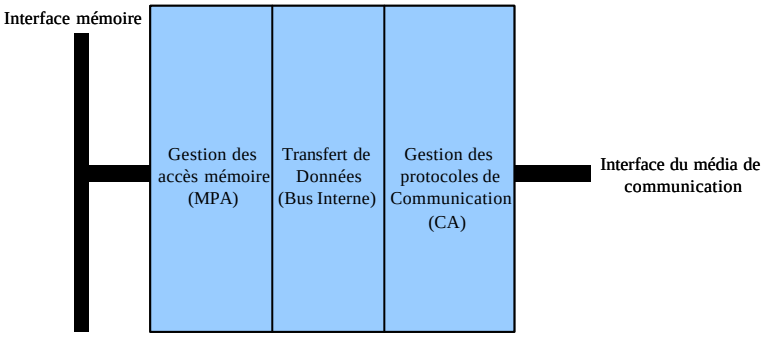


Figure 25. Vue globale d'un adaptateur mémoire matériel

1.2. Architecture générique de l'adaptateur mémoire matériel

Pour découpler les mémoires globales et les médias de communication, nous proposons une architecture d'adaptateurs mémoire matériels générique et flexible [Gha02b]. Conformément avec les trois couches conceptuelles de l'adaptateur mémoire matériel, cette architecture est composée de trois parties essentielles (Figure 26) :

- Une partie spécifique aux ports d'accès mémoire appelée adaptateur de port mémoire ("Memory Port Adaptor").
- Une partie spécifique au média de communication externe appelée adaptateur de canal ("Channel Adaptor").
- Un bus de communication interne à l'adaptateur matériel qui connecte les CAs et les MPAs.

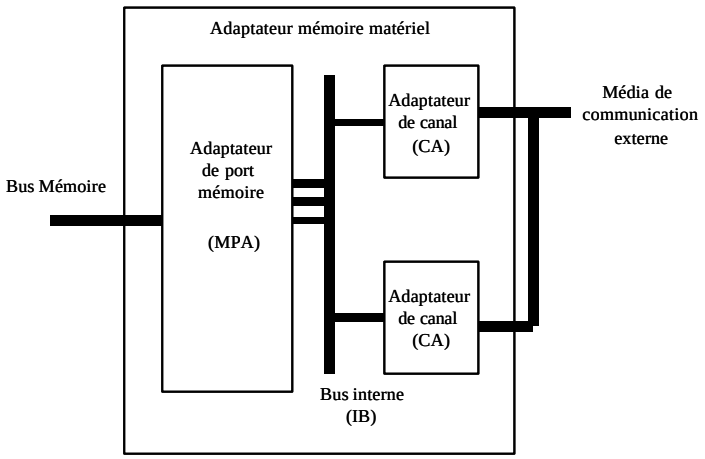


Figure 26. Architecture interne d'un adaptateur mémoire matériel

Nous optons à cette décomposition architecturale pour les raisons suivantes :

- **Séparation entre le comportement et la communication :** cette décomposition architecturale correspond parfaitement au concept de la séparation entre le comportement et la communication. En effet, les adaptateurs de canaux (CA) contrôlent les protocoles de communication tandis que les adaptateurs de ports mémoire (MPA) gèrent les transferts de données spécifiques au comportement des interfaces mémoire.
- **Flexibilité :** le fait que l'adaptateur mémoire contient une partie spécifique à la mémoire et une autre spécifique au réseau de communication, l'exploration architecturale de plusieurs types de mémoire et de réseaux de communication devient moins complexe. Car, si le concepteur veut utiliser un nouveau type de mémoire par exemple, il doit alors seulement changer le module d'adaptation spécifique à la mémoire (MPA). Les adaptateurs de canaux restent les mêmes.

- **Standard d'adaptation :** l'utilisation d'un bus interne à l'adaptateur mémoire matériel assure le découplage entre les interfaces des modules spécifiques à la communication et les interfaces des modules spécifiques à la mémoire (MPA). Ce découplage permet d'adapter des protocoles de communication de nature différente à des composants mémoire venant de différents fabricants.
- **Facilité d'implémentation et de validation :** une décomposition modulaire facilite l'implémentation ainsi que la validation des adaptateurs mémoire matériels. En effet, on peut implémenter et compiler chaque module séparément sans se soucier des autres fonctionnalités. Ceci permet de réduire le temps de développement et d'avoir un code facile à maintenir.
- **Réutilisation :** la décomposition modulaire facilite la réutilisation des adaptateurs de canaux et des adaptateurs de ports mémoire dans plusieurs applications.

2. Adaptateur de port mémoire (MPA)

2.1. Rôle

Le rôle d'un MPA est d'adapter le protocole d'accès mémoire natif du bus mémoire à celui utilisé par le bus interne de l'adaptateur pour le transfert de données.

2.2. Fonctionnalité

La fonctionnalité d'un adaptateur de port mémoire consiste à :

- Traduire les adresses virtuelles envoyées par les CAs à des adresses mémoire physiques.
- Adapter l'interface du bus interne à celui de la mémoire.
- Gérer les transferts de données entre le bus interne et celui de la mémoire.

2.2.1. Traduction d'adresses

Indépendamment des niveaux d'abstraction, les mots d'un modèle mémoire de simulation correspondent à des cases d'un tableau linéaire. Chaque indice du tableau correspond à l'adresse du mot contenu dans la case du tableau. L'utilisation de ce type d'adressage permet de faciliter la tâche du programmeur de l'application. En effet, le programmeur code son application sans trop se méfier de l'architecture mémoire utilisée. Il ne voit qu'un tableau de mots contigus. Les adresses utilisées par le programmeur ne correspondent pas aux adresses envoyées par le processeur après la compilation du code de l'application. Dans le cas des mémoires locales, ce problème de correspondance entre les adresses reste invisible pour le concepteur. En effet, l'espace adressable de la mémoire locale est connu par le compilateur. Ce dernier attribue des adresses logiques arbitraires à chaque donnée. La traduction de ces adresses se fait par le processeur en utilisant des modes de codage dans les jeux d'instruction du processeur ou des unités de gestion mémoire " MMU " (Memory Management Unit). Dans le cas des mémoires globales, l'espace

adressage n'est pas vu par le processeur et la répartition des données dans la mémoire globale est faite par le concepteur d'une manière non aléatoire contrairement à ce qui fait le compilateur. Comme dans le cas des mémoires locales, les adresses attribuées aux données de la mémoire globale ne correspondent pas à celles envoyées par le processeur. Mais dans ce cas, le processeur est incapable de coder les liens entre les adresses générées par le compilateur et celles de la mémoire globale puisque qu'il ne voit ni l'espace adressable, ni la répartition de données.

2.2.2. Adaptation des interfaces entre le bus interne et le bus mémoire.

Un des rôles du MPA est d'adapter l'interface du bus interne et celle du bus mémoire qui ne sont pas forcément identiques. L'interface d'un MPA est composée de deux parties : une partie spécifique aux signaux du bus interne et une partie spécifique à l'interface de la mémoire. Dans notre modèle d'adaptateur matériel, nous utilisons une interface fixe pour le bus interne tandis que les interfaces des mémoires sont variables, car elles dépendent du type utilisé.

La Figure 27 montre l'interface d'un module MPA spécifique à une mémoire Micron SDRAM 256 Mo (64 Mo x 4 bancs). L'interface du MPA est composée de deux parties : des signaux spécifiques à la mémoire SDRAM et des signaux spécifiques au protocole du bus interne.

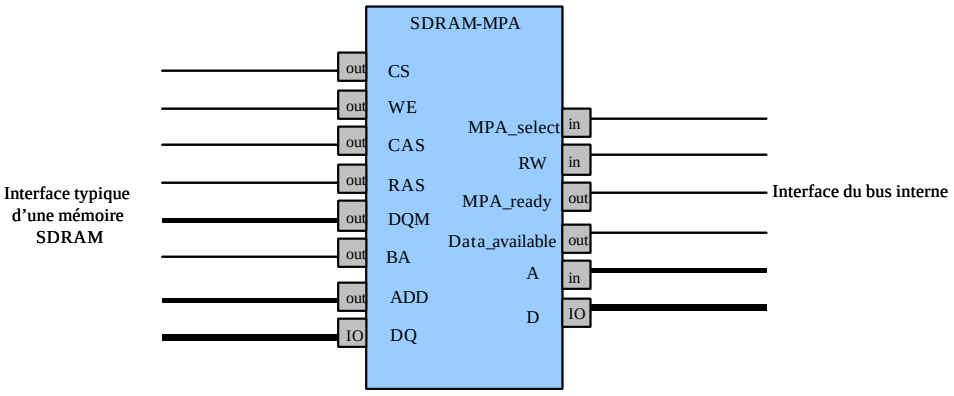


Figure 27. Adaptation des interfaces entre la mémoire et le protocole du bus interne

Le Tableau 1 décrit les ports de chaque partie de l'interface du MPA. La partie spécifique au bus interne est composée de six ports spécifiques aux signaux du bus interne. La taille du port de données du côté du bus interne est générique, elle peut être configurable selon le type de données utilisé par l'application et selon le mode de transfert de données. Par exemple, pour un type de données de 8 bits (un pixel d'une image) et un mode de transfert simple (un pixel par accès), on utilise un port de données de 8 bits. Par contre, si le mode de transfert est un mode "burst " de huit (8 pixels par accès), on utilise un port de données de 64 bits. On constate que la taille du port d'adresse du MPA (côté bus interne) n'est pas fixe, elle dépend du choix du

concepteur. Ce choix peut dépendre de la taille du bus d'adresses du canal et de l'espace mémoire physiquement adressable.

	Nom du port	*Type du port*	*Direction*	*Largeur en bits*	*Description*
Interface spécifique au bus interne	MPA_select	Ctrl	In	1	Demande d'accès par un CA
	RW	Ctrl	In	1	Pour faire la distinction entre les accès en écriture (RW = 0) et en lecture (RW = 1)
	MPA_ready	Ctrl	Out	1	Confirmer au CA que le module MPA est prêt à accepter un accès
	Data_available	Ctrl	Out	1	Indique au CA que la donnée est prête à lire sur le bus interne
	A	Add	In	-	Bus d'adresses
	D	Data	IO	8, 16, 32, 64	Bus de données
Interface spécifique à la mémoire	CS	Ctrl	Out	1	Active la mémoire
	WE	Ctrl	Out	1	0 : écriture, 1 : lecture
	CAS	Ctrl	Out	1	Sélection d'une colonne
	RAS	Ctrl	Out	1	Sélection d'une ligne
	DQM	Ctrl	Out	2	Masque
	BA	Ctrl	Out	2	Sélection du banc mémoire
	ADD	Add	Out	13	Bus d'adresses mémoire
	DQ	Data	IO	16	Bus de données mémoire

Tableau 1. Description de l'interface d'un adaptateur de port mémoire spécifique à une mémoire SDRAM

La partie d'interface du MPA spécifique à la mémoire varie d'une mémoire à une autre. Dans le cas de la mémoire Micron SDRAM, elle est constituée de huit ports. La taille des ports est fixée selon le " Datasheet " de la mémoire fournie par le constructeur ou par le concepteur lui-même.

La construction des interfaces des MPAs se fait automatiquement en configurant des modèles de bibliothèques génériques par des paramètres d'architecture spécifiques au bus interne et à la mémoire. Les paramètres utilisés pour la construction des interfaces du MPA correspondent au nom du port, son type, sa direction et sa largeur. Ces paramètres annotent la spécification d'entrée du flot de conception globale. Leur extraction se fait d'une manière systématique en analysant la description Colif de l'application. Les détails de ces paramètres seront donnés dans la section 4.2 du chapitre 5.

2.2.3. Gestion des transferts de données entre le bus interne et celui de la mémoire

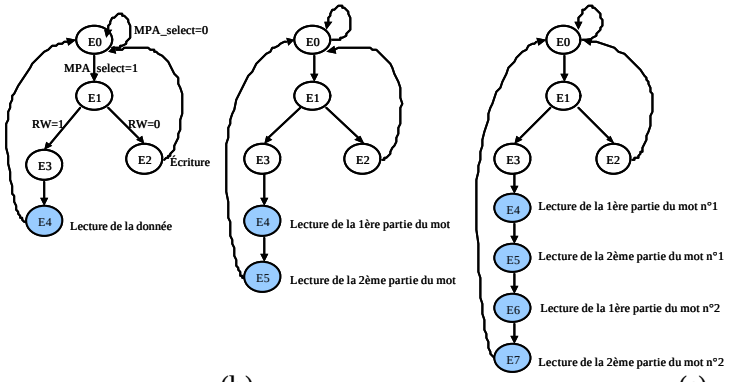
Le MPA permet de résoudre l'hétérogénéité entre le protocole de communication utilisé par le bus interne et le protocole de transfert de données spécifique à la mémoire (section 2.2.2 du chapitre 3). Le MPA est composé d'une machine d'états finis qui permet de:

- Traduire les signaux du protocole du bus interne aux signaux de la mémoire. Par exemple si le MPA reçoit un signal " MPA_select " du bus interne, il active la mémoire par un signal de sélection tel que le signal " CS " dont le nom est spécifique au type mémoire.
- Adapter la taille du bus de données interne à celle du bus de la mémoire.
- Transférer les données selon le mode d'accès désigné dans la spécification. On distingue un mode d'accès simple et un mode d'accès " burst " séquentiel.

Le modèle générique du MPA est configurable par cinq paramètres :

1. " BURST " indiquant si l'accès est simple (BURST=1) ou en mode " burst " (BURST>1).
2. " DATA_INT_WIDTH " indiquant la taille du bus de données interne.
3. " ADDR_INT_WIDTH " pour la taille du bus d'adresse interne.
4. " MEM_BUS_DATA_WIDTH " indiquant la taille du bus de données de la mémoire utilisée.
5. " MEM_BUS_ADDR_WIDTH " pour la taille du bus d'adresse mémoire.

Le nombre d'états de la machine du MPA dépend de ces paramètres. Le nombre de configurations possibles de cette machine d'état est égal à trente deux (2^5). Pour des raisons de simplicité et de clarté nous donnons trois exemples de configuration basés sur trois paramètres seulement. Nous supposons aussi que l'arbitrage des accès parallèles est fait à l'extérieur du MPA et que les accès en mode " burst " sont seulement en lecture.



(a) (b) (c)

Figure 28. Exemples de configuration de la machine d'états finis du module MPA : (a) exemple 1, (b) exemple 2 et (c) exemple 3

Exemple 1 (BURST = 1, DATA_INT_WIDTH = 16, MEM_BUS_DATA_WIDTH = 16)

Dans cet exemple, la taille du bus interne de données est égale à la taille de celui de la mémoire. L'accès à la mémoire est en mode simple (un mot mémoire par accès). Comme l'indique la Figure 28 (a), le nombre d'états de la machine du MPA est égal à cinq.

- *E0 : c'est l'état initial. Tant que le module MPA ne reçoit pas un signal d'activation "MPA_select " d'un CA, la machine reste en attente à l'état E0. Dans le cas contraire la machine passe à l'état E1.*
- *E1 : dans cet état, la machine teste un registre d'état indiquant si le module MPA est prêt à accepter un accès ou non. Dans le cas où il est prêt, il confirme son état au CA par un signal "MPA_ready ". La machine passe à l'état E2 dans le cas d'écriture (RW=0) ou à l'état E3 dans le cas de lecture (RW=1). Dans le cas où le module MPA n'est pas prêt, la machine reste en attente dans l'état E1.*
- *E2 : dans cet état la machine lit l'adresse et la donnée envoyées par un CA via le bus interne. La machine d'états finis renvoie l'adresse et la donnée systématiquement au bus de la mémoire pour finir cet accès en écriture. Tous les signaux de contrôle de la mémoire sont également envoyés. La machine passe systématiquement à l'état E0.*
- *E3 : cet état correspond à un accès en lecture. L'adresse est lue et envoyée à la mémoire. Les signaux de contrôle de la mémoire sont également envoyés à cet état. L'état suivant est E4.*
- *E4 : la transition entre l'état E3 et cet état correspond à la latence mémoire. Comme la latence dépend du type de la mémoire, nous utilisons un signal de synchronisation appelé "data_available " qui indique que la donnée est prête à être consommée. Pour faciliter la compréhension de cet exemple, nous supposons que la latence mémoire est égale à un cycle dans le cas d'une lecture et elle est nulle dans le cas d'une écriture. Après le temps de latence le module MPA lit la donnée du bus mémoire et il l'envoie avec le signal "data_available " au CA via le bus interne. Par la suite la machine passe à l'état initial pour recommencer un nouvel accès.*

Exemple 2 (BURST = 1, DATA_INT_WIDTH = 32, MEM_BUS_DATA_WIDTH = 16)

Dans ce cas de figure, la taille du bus de données de la mémoire est deux fois supérieure à la taille de celui du bus interne. Ceci pose un problème de transfert de données lors des accès en lecture car la taille du mot mémoire à lire est plus large que la taille supportée par le bus interne. La machine d'états finis du module MPA permet de résoudre ce problème en transférant le mot mémoire sur deux fois. Comme l'indique la Figure 28 (b), la machine d'états finis contient un état de plus par rapport au cas de configuration précédent.

- *E4 correspond à la lecture et l'envoi de la première moitié du mot mémoire et son envoi au CA.*
- *E5 correspond à la lecture et l'envoi de la deuxième moitié du mot mémoire.*

Exemple 3 (BURST = 2, DATA_INT_WIDTH = 32, MEM_BUS_DATA_WIDTH = 16)

Comme dans le dernier cas de figure, la taille du bus de données de la mémoire est supérieure à celle du bus interne, mais l'accès en lecture est en mode " burst " de taille deux. Ceci implique que le module MPA doit lire deux mots mémoire à partir de l'adresse

envoyée. Comme l'indique la Figure 28 (c), la machine d'états finis contient huit états dont quatre sont consacrés à la lecture du premier et du deuxième mot mémoire.

Ces trois exemples montrent la complexité et la diversité de l'implémentation des adaptateurs de ports mémoire (MPA). Pour cela un modèle d'implémentation générique, qui doit couvrir touts les cas de configuration possibles, est nécessaire. Une méthode de génération automatique des machines d'états finis est également primordiale.

2.3. Implémentation

Nous utilisons une bibliothèque de macro-modèles, écrite en langage RIVE, implémentant la fonctionnalité générique d'un adaptateur de port mémoire (MPA). Ces modèles génériques sont configurables par des paramètres d'architecture spécifiques à la mémoire d'une part et au bus interne d'autre part. Actuellement la bibliothèque contient deux classes de MPA : des MPA spécifiques à des mémoires statiques et d'autres spécifiques à des mémoires dynamiques. Le code des MPA généré à partir de cette bibliothèque de macro-modèles est en SystemC et en VHDL.

3. Adaptateur de canal (CA)

Un adaptateur de canal est l'unité matérielle de l'adaptateur mémoire qui est spécifique au média de communication externe. La Figure 29 montre une vue conceptuelle d'un CA. Il peut être vu comme une superposition de six couches. L'extension de ce modèle en couche des adaptateurs de canaux de communication fait partie des travaux de thèse de Giedrius Majauskas.

- Une couche de lecture/écriture externe qui permet de recevoir l'adresse et/ou la donnée venant du média de communication externe.
- Une couche de traitement externe permet de convertir le type de données et d'empaqueter/dépaqueter les données à transférer avant leur mémorisation.
- Une couche de mémorisation permet de sauvegarder les adresses et/ou les données avant leur envoi au bus interne.
- Une couche de traitement interne est utilisée pour adapter la représentation de données supportée par l'élément de mémorisation et celle supportée par le bus interne.
- Une couche de lecture/écriture interne permet d'envoyer/recevoir les données au/du bus interne.
- Une couche de contrôle permet de contrôler toutes les couches de fonctionnalité tout au long d'un transfert de données de la part du média de communication externe ou de la part du bus interne. Cette couche de contrôle synchronise la communication entre le média externe et le bus interne

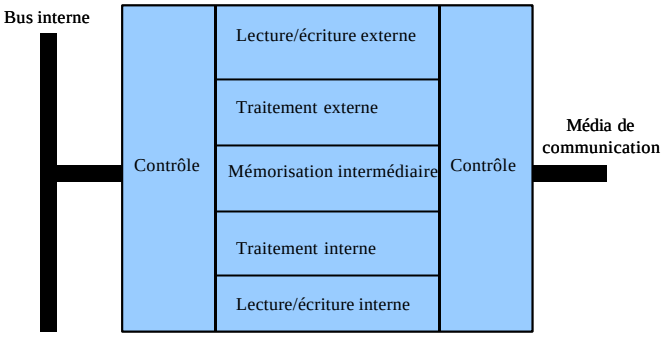


Figure 29. Vue conceptuelle de la fonctionnalité d'un CA

3.1. Rôle

Un adaptateur de canal permet d'adapter et de contrôler la communication entre un canal de communication et le bus interne de l'adaptateur mémoire. Un CA implémente le comportement des ports externes d'un port virtuel. Comme un port virtuel d'un module mémoire peut avoir trois directions (in, out et in-out), un CA doit donc tenir compte de tous les modes d'accès mémoire : en mode écriture, lecture et les deux à la fois. La réalisation d'un seul CA couvrant tous les modes d'accès est assez complexe et réduit la flexibilité de son utilisation. En effet, dans une application, on peut trouver des canaux de communication utilisés pour l'écriture ou pour la lecture. Pour ces raisons, nous réalisons un CA hiérarchique et modulaire.

On propose trois modèles d'adaptateur de canal : un pour l'écriture (CA_W), un pour la lecture (CA_R) et un pour les deux (CA_RW) (Figure 30) :

- Un CA_W, implémentant un port mémoire virtuel de direction " in ", contient un seul module d'écriture appelé " WU " (Write Unit) (Figure 30 (a)). Le rôle de ce module est de lire les adresses et les données venant du canal de communication, de convertir et empaqueter les données avant leur éventuelle mémorisation et leur envoi sur le bus interne.
- Un CA_R implémente un port mémoire virtuel de direction " out ". Il contient un seul module appelé " RU " (Read Unit) n'implémentant que la fonctionnalité de lecture (Figure 30 (b)). Son rôle est de recevoir l'adresse du bus interne, la transmettre au bus mémoire et d'attendre que la donnée soit valide sur le bus de données interne pour la lire et enfin l'envoyer au média de communication externe.
- Dans le cas d'un module mémoire contenant un port virtuel bidirectionnel, le CA_RW implémentant ce port est constitué de deux modules d'écriture et de lecture (resp. WU et RU). Le rôle de ce type de CA est de répondre aux requêtes d'accès mémoire en mode d'écriture comme en mode de lecture (Figure 30 (c)).

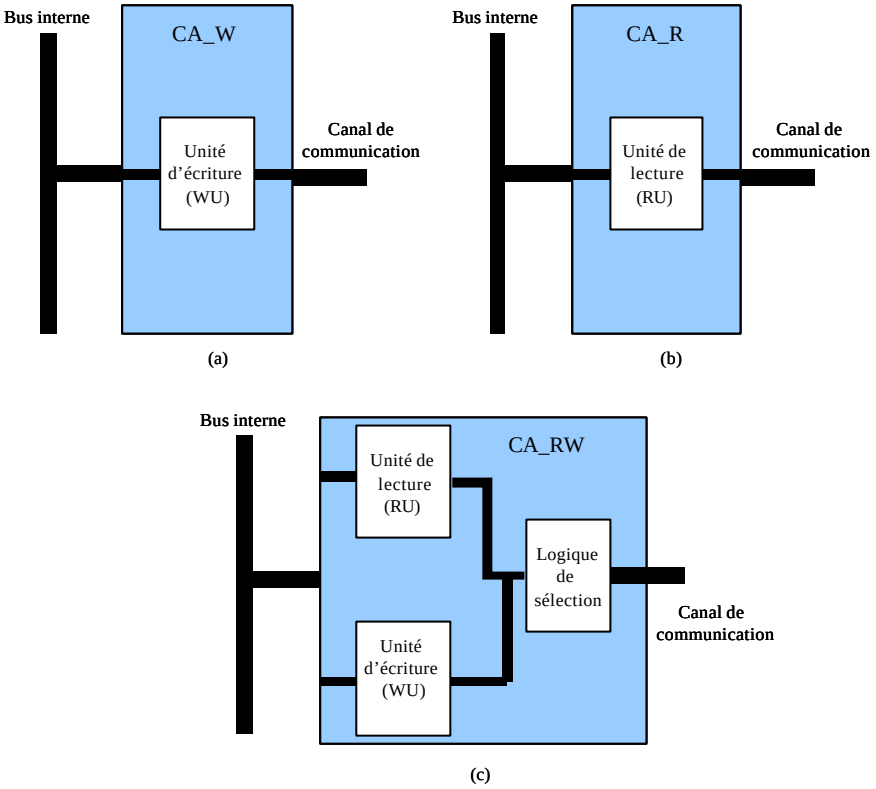


Figure 30. Les différents modèles d'adaptateur de canal : (a) CA_W, (b) CA_R, et (c) CA_RW

3.2. Fonctionnalité

Chaque couche de l'adaptateur de canal correspond à une fonctionnalité donnée :

- **Lecture/écriture externe :** c'est la réception/émission de l'adresse et/ou la donnée venant du média de communication externe. Le CA reste inactif tant qu'il ne reçoit pas un signal d'activation venant de la partie contrôle du média externe. À la réception de ce signal, le CA peut lire l'adresse et/ou la donnée.
- **Traitement externe :** ceci consiste à adapter le type de données envoyé par le média externe au type de données supporté par l'élément de mémorisation du CA. Par exemple, les données véhiculées par le média de communication peuvent être de type entier " int " tandis que l'élément de mémorisation est un registre de vecteur logique. L'empaquetage/dépaquetage de données consiste à adapter la taille des données véhiculées par le média de communication à la taille d'un mot de l'élément de mémorisation utilisé.
- **Mémorisation interne :** l'utilisation d'un élément de mémorisation est motivée par le besoin des registres de stockage intermédiaires pour les couches de traitement. Elle est aussi motivée par la nécessité d'éviter un goulot d'étranglement sur le bus interne. L'utilisation d'une FIFO permet au CA de recevoir des données du média de communication externe même si le bus interne n'est pas encore prêt.

- **Traitement interne :** c'est l'adaptation de la représentation des données supportée par l'élément de mémorisation et celle supportée par le bus interne. Dans le cas où la taille du bus de données interne est inférieure à la taille d'un mot de l'élément de mémorisation, le CA doit diviser ces mots en plusieurs mots de plus petite taille avant de les envoyer consécutivement au bus interne.
- **Lecture/écriture interne :** permet d'envoyer/recevoir les données au/du bus interne.
- **Contrôle :** c'est la synchronisation de la communication entre le média externe et le bus interne. Comme l'indique la Figure 31, le CA ne peut lire une adresse ou/et une donnée venant du média externe que si la partie contrôle reçoit une requête d'activation externe (signal "req "). Après la réception de ce signal, la partie contrôle du CA scrute son port lié à l'arbitre " IB_status " pour vérifier l'état du bus interne. Le CA n'envoie l'adresse et/ou la donnée que si le bus interne est libre et que s'il reçoit un acquittement du module MPA indiquant qu'il est prêt à accepter l'accès. Dans le cas d'une lecture, le CA reste inactif jusqu'à la réception d'un signal de contrôle " data_available " indiquant que la donnée à lire est prête sur le bus de données. Le contrôle des accès concurrents peut être implémenté par le module MPA ou par un module d'arbitrage externe spécifique au bus interne.

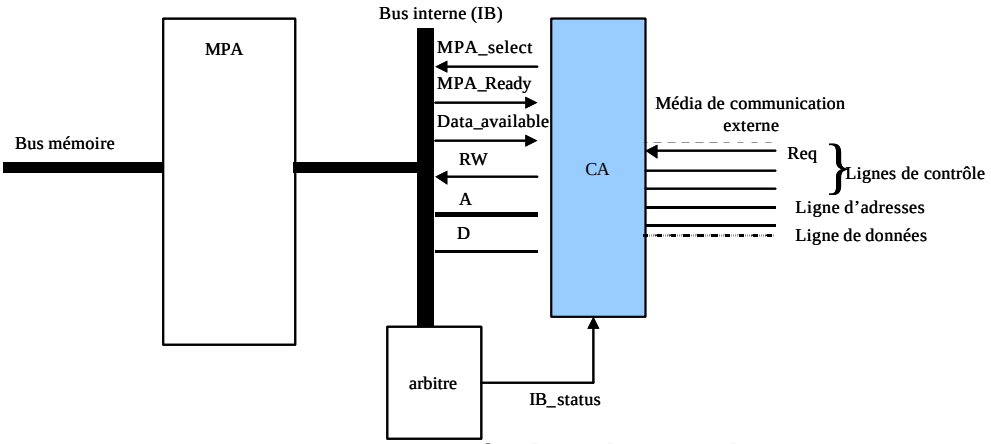


Figure 31. La partie contrôle d'un adaptateur de canal

3.3. Implémentation

Nous utilisons une bibliothèque générique d'adaptateurs de canal contenant différentes classes de CA. Lors de la génération automatique des adaptateurs mémoire matériels, la configuration des CAs génériques permet d'avoir le code final synthétisable. Nous utilisons des modèles de CA en VHDL et en SystemC. Les modèles générés peuvent implémenter des protocoles de communication différents et peuvent utiliser des éléments de mémorisation divers allant du simple registre à une FIFO.

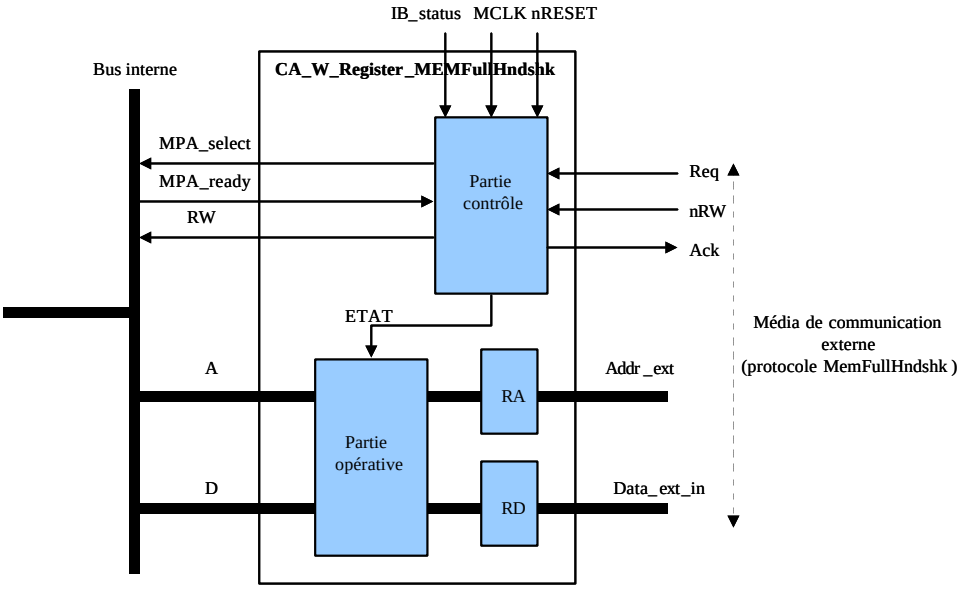


Figure 32. Architecture interne d'un adaptateur de canal d'écriture de type CA_W_MemFullHndshk_Register

La Figure 32 montre l'architecture interne d'une réalisation d'un simple adaptateur de canal d'écriture (*CA_W_MemFullHndshk_Register*). Ce CA adapte un protocole de communication "*MemFullHndshk*[6] " du média externe aux signaux du bus interne. Son élément de mémorisation contient deux registres pour stocker l'adresse et la donnée envoyées lors d'une opération d'écriture. Les éléments de traitement de ce CA sont réalisés par une seule unité d'écriture (WU) contenant :

- Une partie contrôle réalisée par une machine d'états finis qui permet de lire les signaux de contrôle du média de communication externe pour gérer la transition des états et pour contrôler les parties de traitement et de mémorisation.
- Une partie opérative commandée par la partie contrôle. Selon l'état de la machine, elle permet de lire l'adresse et la donnée des registres et de les envoyer sur le bus interne quand l'état de la machine le permet.
- L'élément de mémorisation de ce CA correspond à deux simples registres utilisés pour stocker l'adresse et la donnée venant du média externe. Ces registres gardent leurs données tant que la partie opérative ne les a pas envoyées au bus interne.

[6] MemFullHndshk est un protocole à base de poignée de main qui possède une ligne d'adresse pour supporter les accès mémoire.

La Figure 33 montre le code SystemC de l'adaptateur de canal *CA_W_MemFullHndshk_Register* généré automatiquement à partir d'une bibliothèque matérielle de composants génériques. Le comportement de cet adaptateur de canal est assez simple car :

- Le mode d'accès mémoire est simple (pas d'accès burst).
- La taille des bus de données et d'adresses est la même pour le média externe et le bus interne.

Une machine d'états finis à quatre états permet de réaliser le comportement de cet adaptateur de canal. La partie contrôle est assurée par un processus SystemC appelé " ctrl_Part : ligne 3 de la Figure 33 a ". Tandis que la partie opérative est réalisée par un autre processus SystemC appelé " operation_part : ligne 1 de la Figure 33 b ".

```
1.  #include"systemc.h"
2.  #include"ca_w_MemFullHndsk_Register.h"
3.  void CA_W_MemFullHndsk_Register:: ctrl_part(){
4.
5.      if(nRESET == 0)          ETAT = 0;
6.      while(true){
7.          if(MCLK.event() && MCLK ){
8.           switch (ETAT){
9.              case (0):
10.                 if (req)     ETAT = 1;
11.                 else         ETAT = 0;
12.                 break;
13.             case (1):
14.                 if (IB_status && MPA_ready)  ETAT = 2
15.                 else         ETAT = 1;
16.                 break;
17.             case (2):
18.                              ETAT = 3;
19.                 break;
20.             case (3):
21.                              ETAT = 0;
22.                 break;
23.             default:
24.                              ETAT= 0;
25.         }
26.       }
27.     wait();
28.     }
29.
30. }
```

(a)

```
1.  void CA_W_MemFullHndsk_Register:: operation_part() {
2.
3.      while(true){
4.          switch (ETAT){
5.              case (0):
6.                  D  = Z_32;
7.                  A  = Z_32;
8.                  ack=0;
9.                  MPA_select=0;
10.                 RW=0;
11.                 break;
12.             case (1):
13.                 /* lecture et sauvegarde de l'adresse et de la donnée */
14.                 Data_ Register = data_ext_in;
15.                 Address_Register = addr_ext;
16.                 ack=0;
17.                 MPA_select=1;
18.                 break;
19.             case (2):
20.                 /* envoie de l'adresse et de la donnée sur le bus interne *
21.                 D  = Data_Register;
22.                 A  = Address_Register;
23.                 ack=1;
24.                 MPA_select=1;
25.                 break;
26.             case (3):
27.                 //fin du cycle d'accès en écriture
28.                 D  = Z_32;
29.                 A  = Z_32;
30.                 ack=0;
31.                 MPA_select=0;
32.                 break;
33.          }
34.     wait();
35.     }
36. }
```

(b)

Figure 33. Une partie du code SystemC d'un CA_W_ Register_MemFullHndshk générée automatiquement

- Etat 0 : tant que l'adaptateur de canal ne reçoit pas un signal d'activation " req " du média de communication externe, la machine reste en attente dans l'état 0. Après la réception de ce signal, la machine passe à l'état 1.

- Etat 1: dans cet état, la partie opérative lit l'adresse et la donnée venant du média externe et les sauvegarde dans les registres correspondants (ligne 14 et 15 de la Figure 33 b). La partie de contrôle teste si le bus interne est libre (IB_status : ligne 14 de la Figure 33 a) et si l'adaptateur de port mémoire MPA est prêt à accepter l'accès " MPA_ready: ligne 14 ". Le signal " IB_status " peut venir d'un arbitre externe spécifique au bus interne ou du MPA, si l'arbitrage est géré par ce dernier. Si le test est validé, la machine passe à l'état 2, sinon elle reste à l'état 1.
- Etat 2 : dans cet état, l'adaptateur de canal place l'adresse et la donnée sur le bus interne de l'adaptateur mémoire matériel (ligne 21 et 22 de la Figure 33 b) et acquitte le média externe (ligne 23 de la Figure 33 b) pour qu'il puisse commencer un nouvel accès à partir du prochain cycle. Puis, le passage à l'état 3 est systématique.
- Etat 3: c'est l'état qui permet de mettre les bus de l'adaptateur du canal en haute impédance avant d'entamer un nouvel accès.

La synchronisation entre les différents états de la machine de chaque processus SystemC est assurée grâce à la fonction *wait* (ligne 27 et 34) qui permet au moteur de simulation SystemC d'attendre le prochain top d'horloge du système.

4. Bus interne

4.1. Rôle

Le rôle du bus interne de l'adaptateur mémoire matériel est très simple, il permet de véhiculer les données, les adresses et les signaux de contrôle entre les CAs et un module MPA.

4.2. Fonctionnalité

La fonctionnalité du bus se concrétise par une connexion des différents interlocuteurs via des signaux de données, d'adresses et de contrôle. Comme l'indique la Figure 34, l'architecture interne de ce bus est composée de quatre parties :

- Un ensemble de ports d'entrées/sorties qui permettent d'intercepter/d'envoyer les données, les adresses et les signaux de contrôle du/au CA.
- Un ensemble de ports d'entrées/sorties qui permettent la lecture et l'écriture venant du MPA.
- Un média de communication qui correspond aux fils physiques de communication qui relient les différents ports. Selon le degré du parallélisme de l'application et selon le nombre de ports d'accès mémoire, le bus interne peut être composé d'un seul média de communication partagé ou de plusieurs médias parallèles pour véhiculer les accès concurrents des CA maîtres.
- Un arbitre de communication qui contrôle les accès concurrents à un média de communication partagé.

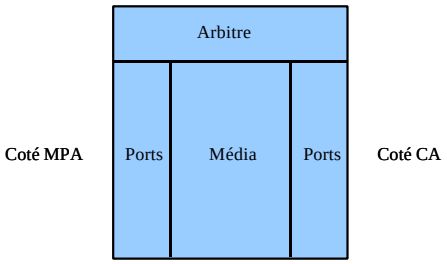


Figure 34. Vue conceptuelle du bus interne de l'adaptateur mémoire matériel

4.3. Implémentation

Nous utilisons une réalisation simple pour implémenter la fonctionnalité du bus interne. Nous utilisons les ports et les signaux SystemC. La résolution du partage d'un signal est assurée par l'utilisation des ports trois états spécifiques au langage SystemC ("sc_rv : SystemC resolved vector").

L'implémentation de l'arbitre est basée sur un algorithme de priorité fixe. Chaque CA maître possède un niveau de priorité attribué par le concepteur. L'arbitre met à jour périodiquement un registre indiquant l'état du bus interne (prêt ou occupé). Dans le cas où le bus est prêt, l'arbitre active le CA de plus haute priorité par un signal " IB_ready ".

5. Les avantages et les inconvénients du modèle d'adaptateur mémoire matériel

L'architecture matérielle de l'adaptateur mémoire que nous proposons permet bien la séparation entre l'interface du média de communication et celle de la mémoire. L'avantage de cette séparation est qu'elle assure la séparation des responsabilités de conception. En effet, le concepteur de la communication et celui des mémoires peuvent travailler séparément. Ceci ne fait qu'accélérer le cycle de conception. Un autre facteur de gain de temps de conception est la nature modulaire et flexible des adaptateurs mémoire. Les modules d'adaptation (MPA et CA) peuvent être réutilisés pour plusieurs applications différentes. Pour couvrir un large domaine d'applications, nous avons implémenté une bibliothèque de macro-modèles implémentant des modèles génériques d'adaptateurs mémoire. Ces macro-modèles sont configurés suivant les besoins de l'application. Nous utilisons cette bibliothèque pour la génération systématique de ces adaptateurs par assemblage de composants.

Comme toutes les approches basées sur des bibliothèques, la limite de notre approche est qu'elle ne peut pas couvrir tous les types de mémoires et de médias de communication existants.

6. Conclusion

La nature hétérogène des systèmes monopuces impose la nécessité d'adaptateurs matériels pour connecter l'interface de la mémoire à celle du réseau de communication. Pour faciliter la réalisation systématique et améliorer la flexibilité d'utilisation de ces adaptateurs, nous avons proposé une architecture d'adaptateurs mémoire modulaire composée de trois modules indépendants. Un premier module d'adaptation de canal (CA), spécifique au canal de communication, permet de convertir les protocoles des médias de communication à un protocole unique utilisé par le bus interne de l'adaptateur mémoire. Un deuxième module d'adaptation de port mémoire (MPA), spécifique à la mémoire, permet d'adapter le protocole du bus interne à celui du port d'accès mémoire. Un troisième module correspond au bus interne de l'adaptateur qui relie les CAs et les MPAs. Le concepteur peut réutiliser les CAs pour connecter différents types de mémoire à un réseau de communication, comme il peut réutiliser les MPAs pour connecter des mémoires à différents médias de communication. Ceci ne fait qu'accélérer la productivité de ces adaptateurs mémoire et par conséquent faciliter l'intégration rapide des mémoires dans les systèmes monopuces.

Chapitre 5 : GENERATION AUTOMATIQUE DES ADAPTATEURS MEMOIRE MATERIELS

Ce chapitre détaille la génération automatique des adaptateurs mémoire matériels qui ont été présentés dans le chapitre quatre. Nous présentons trois contributions liées à l'implémentation et à la validation automatique des adaptateurs mémoire matériels dans la section 4 :

1. La génération d'architecture mémoire interne (en Colif).
2. La génération du code des mémoires, des CA et des MPA.
3. La génération des programmes de tests.

Notre approche est validée dans la section 5 par une application de traitement d'images. Une conclusion est donnée dans la section 6.

Chapitre 5 : GENERATION AUTOMATIQUE DES ADAPTATEURS MATERIELS POUR LES MEMOIRES

1. Introduction

1.1. Rappel sur les objectifs de la génération automatique des adaptateurs mémoire matériels

L'objectif de la génération automatique des adaptateurs mémoire matériels est de faciliter l'intégration de composants mémoire (IP) dans une architecture de système monopuce. La génération automatique de ces adaptateurs accélère la phase de conception ou le prototypage, mais elle facilite aussi l'exploration d'architecture mémoire. En effet, le concepteur peut "essayer" plusieurs mémoires différentes sans se soucier de la compatibilité des protocoles et des signaux. Ceci est géré par les adaptateurs.

1.2. Principe de la génération automatique

La génération automatique des adaptateurs mémoire matériels est basée sur l'assemblage de composants de bibliothèque. Un outil d'assemblage systématique d'adaptateurs matériels a été développé au sein du groupe SLS lors des travaux de thèse de Damien Lyonnard [Lyo03]. Cet outil a été utilisé pour la génération des interfaces matérielles spécifiques aux processeurs et non pas pour les mémoires. Dans cette thèse, nous avons profité de la flexibilité de l'environnement de conception pour étendre cet outil à la génération des interfaces matérielles spécifiques aux mémoires. Dans une première partie, nous détaillons l'environnement de l'outil de génération des interfaces pour les processeurs avant son extension. Dans une deuxième partie, nous proposons d'étendre cet environnement à la génération des adaptateurs mémoire matériels.

1.3. Les objets requis pour la réalisation des adaptateurs mémoire matériels

La génération des adaptateurs mémoire matériels nécessite les langages suivants :

- Le langage VADeL ou SystemC pour décrire la structure et le comportement de l'application. La description VADel de l'application doit être annotée avec des paramètres de configuration qui vont guider l'outil de génération des interfaces.
- Le langage de description de macro-modèles RIVE pour décrire le comportement générique des composants de bibliothèque (composant mémoire et composants de base de l'adaptateur matériel). Bien entendu la connaissance du langage cible est nécessaire. Nous utilisons SystemC et VHDL.

- Le langage Colif pour décrire l'architecture interne[7] de chaque type de composant. Pour éviter à l'utilisateur la connaissance de Colif, nous avons développé un générateur d'architecture interne.

1.4. Résultats attendus

Les résultats attendus de notre approche de réalisation d'adaptateurs mémoire matériels sont :

- Le comportement des adaptateurs doit être correct.
- La flexibilité du flot de génération. L'utilisateur doit pouvoir générer l'adaptateur de son modèle mémoire facilement.
- L'architecture des adaptateurs mémoire doit être flexible pour qu'elle puisse être réutilisée par plusieurs applications.
- Le résultat de la synthèse logique de ces adaptateurs doit être d'une complexité optimale. Nous utilisons le nombre de portes logiques comme un critère de complexité.
- Le surcoût de temps de communication introduit par un adaptateur mémoire doit être acceptable.
- Le temps de génération des adaptateurs doit être minimal.

2. Outil de génération d'adaptateurs matériels pour les processeurs

Cet outil génère le code RTL des adaptateurs permettant de connecter les processeurs au réseau de communication. La fonctionnalité de l'outil consiste à assembler les composants de base d'adaptation qui sont décrits dans un environnement spécifique aux processeurs. L'environnement ainsi que les étapes de l'outil sont détaillés dans les sous sections suivantes.

2.1. Entrées de l'outil

Comme l'indique la Figure 35, l'outil de génération des adaptateurs matériels pour les processeurs prend en entrée trois éléments :

- Une architecture virtuelle annotée.
- Une bibliothèque de modèles Colif, appelée aussi bibliothèque d'architecture interne.
- Une bibliothèque de comportement.

[7] Le terme architecture interne désigne un modèle Colif qui décrit la structure hiérarchique et l'interface d'un ou de plusieurs modules. Le comportement n'est pas inclus dans ce modèle.

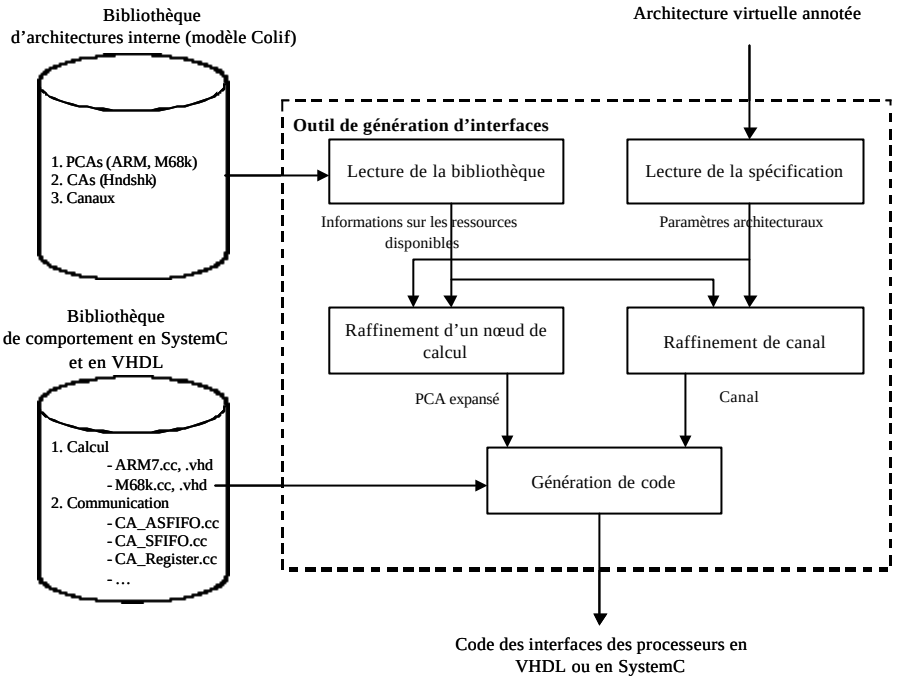


Figure 35. Outil de génération d'adaptateurs matériels pour les processeurs

2.1.1. Architecture virtuelle annotée

Une spécification logiciel-matériel de l'application annotée avec des paramètres architecturaux. Ces paramètres caractérisent :

- Les besoins du logiciel de l'application (type de données, taille de mémoire, SE, …)
- La plateforme matérielle utilisée (type de processeur, type de mémoire, …).
- Les interfaces logiciel-matériel. Plus précisément, les services de communication nécessaires à l'application. L'architecture virtuelle fixe le producteur et le consommateur de chaque service ainsi que le type de réalisation de ces services (logiciel ou matériel). Une implémentation logicielle correspond à des pilotes d'accès tandis qu'une implémentation matérielle correspond à l'adaptateur matériel.

2.1.2. Bibliothèque d'architectures internes

Une bibliothèque d'architectures internes au format Colif décrit la décomposition hiérarchique et l'interface de chaque module. Cette bibliothèque contient une structure d'un module appelée architecture interne ou PCA ("Processor Concentric Architecture"). Ce module représente une architecture générique d'un processeur.

Le PCA est composé de (Figure 36) :

- Un module " Proc " qui représente le processeur.
- Un décodeur d'adresse.
- Un gestionnaire d'interruption (IC).
- Une mémoire locale RAM pour stocker les variables de calcul.
- Une mémoire ROM pour stocker l'image du programme à exécuter.
- Un coprocesseur de communication (CC) qui correspond à une architecture abstraite de l'adaptateur du processeur qui connecte ce dernier au média de communication. Il est composé d'un adaptateur de module (MA) spécifique au processeur, d'un bus interne et d'un adaptateur de canal virtuel (VCA). Ce dernier est abstrait parce que l'interface, du côté média externe, n'est pas encore définie et parce qu'il ne correspond encore à aucune implémentation.

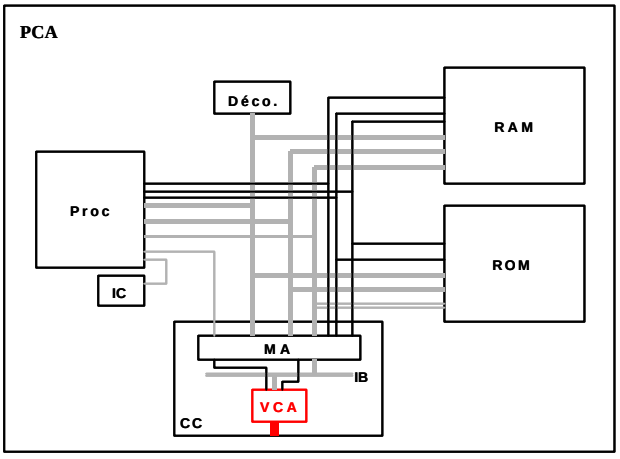


Figure 36. Architecture interne d'un nœud de calcul

Cette bibliothèque contient aussi des modèles structurels de canaux de communication et de leurs adaptateurs (CA). Les interfaces de ces modèles sont généralement basées sur des protocoles de poignée de main qui ne supportent pas les accès à des mémoires globales. Tous les éléments de cette bibliothèque sont écrits à la main par le concepteur en langage Colif. Ceci s'avère très coûteux en temps et très difficile pour un utilisateur non expert en Colif qui veut intégrer ses propres modèles dans la bibliothèque (IP, modules standard, …).

2.1.3. Bibliothèque de comportement

Une bibliothèque de comportement réalisant les fonctionnalités des nœuds de calcul (PCA) et des composants de communication.

2.2. Sortie de l'outil

La sortie de l'outil de génération des interfaces est le code des adaptateurs matériels du processeur cible. Le code généré est en SystemC ou en VHDL.

2.3. Etapes de l'outil

La génération automatique des adaptateurs matériels de processeurs est composée de cinq étapes essentielles.

2.3.1. Lecture de la spécification de l'application

L'outil prend en entrée une description de l'application en format Colif sous forme d'un fichier XML. Cette étape consiste à récupérer les paramètres de configuration qui annotent la spécification d'entrée. La représentation XML que nous utilisons, est basée sur le langage MIDDLE ("Mark-up Internal Data-Description Language Extension"). Cette étape correspond à :

- Une analyse syntaxique de MIDDLE pour reconnaître la définition des objets Colif à partir du fichier XML.
- Une analyse syntaxique de Colif permet de construire un arbre contenant tous les objets Colif (module, port, net, instance,…).

2.3.2. Lecture de la bibliothèque d'architecture interne

Cette étape consiste à charger la bibliothèque architecturale. Cette lecture permet de connaître la disponibilité des ressources architecturales nécessaires pour la réalisation de l'architecture cible. Comme il a été expliqué, cette bibliothèque contient différents composants de l'architecture cible. La structure de chaque composant de cette bibliothèque est décrite en Colif. La fonctionnalité est décrite sous forme d'un chemin vers l'implémentation fournie par la bibliothèque de comportement.

2.3.3. Raffinement d'un nœud

Un nœud de calcul est décrit comme un module virtuel dont l'interface est composée de ports virtuels hiérarchiques regroupant deux types de ports : des ports internes spécifiques à la fonctionnalité du nœud de calcul et des ports externes spécifiques aux canaux de communication. Le raffinement d'un nœud de calcul consiste à :

- Choisir et configurer une architecture interne PCA à partir de la bibliothèque d'architecture en Colif en utilisant les paramètres d'annotation attachés au module de calcul virtuel. Cette étape utilise deux paramètres : "CPU " pour identifier le processeur et " ADDRESS_INT " qui correspond à l'adresse physique du gestionnaire d'interruption.

- Définir la structure de l'adaptateur de canal virtuel (VCA) de l'architecture interne PCA. En effet, selon les paramètres d'annotation attachés aux ports internes et aux ports externes, on fixe le nombre et l'interface des adaptateurs de canaux (CA) qui vont implémenter le VCA. Le choix d'un CA dépend des paramètres liés au protocole de communication et au bus interne. Un CA peut être répliqué selon un paramètre appelé " proliferation " qui peut annoter un module ou un port ou un canal.

2.3.4. Raffinement de canal virtuel

Le raffinement d'un canal virtuel est composé de deux parties :

- Une analyse de tous les ports liés au canal permet de donner une liste d'attributs caractérisant la spécificité du canal à sélectionner à partir de la bibliothèque.
- Une partie sélection utilise la liste précédente pour choisir un modèle de canal parmi les différents modèles disponibles dans la bibliothèque. Selon le paramètre "proliferation", le canal comme son adaptateur peuvent être également répliqués.

2.3.5. Génération du code

Pour chaque instance d'élément de l'architecture interne PCA (sauf le processeur et les mémoires locales), correspond un modèle d'implémentation dans une bibliothèque générique décrite en macro-langage (RIVE). Une implémentation générique est ni simulable ni synthétisable, car elle n'est pas encore configurée. Le type de données est abstrait, le nombre, la taille ainsi que la direction des ports sont encore génériques. Pour une application donnée, on utilise le paramètre de lien vers les sources du comportement et d'autres paramètres de configuration (type de données, taille du port, etc.) pour configurer le code générique sélectionné à partir de la bibliothèque comportementale. Cette expansion du code génère le code final de l'adaptateur de processeur (CC) qui peut être simulable et synthétisable.

3. Les limites de l'environnement de génération d'interfaces

L'environnement de génération d'interfaces présente quelques limites :

- **Construction manuelle d'architecture interne :** la réalisation systématique des adaptateurs mémoire par l'outil de génération d'interfaces nécessite une bibliothèque contenant les modèles Colif d'une architecture interne et des éléments d'adaptation de base (CA et MPA). L'écriture de ces modèles Colif se faisait manuellement. Ceci ne fait que ralentir le temps de conception et rendre l'utilisation de l'outil plus difficile pour les non experts en Colif. La solution que nous proposons à cette limite correspond à un générateur d'architecture interne. Celui–ci peut être également utilisé pour générer les modèles Colif des autres composants.
- **Bibliothèque d'adaptation limitée :** cette bibliothèque ne contenait que le code des éléments d'adaptation spécifiques aux processeurs, et non des éléments spécifiques à la mémoire. Nous

proposons de nouvelles bibliothèques implémentant la fonctionnalité des mémoires, des CA et des MPA [Gha02a].

- **Validation du comportement des composants de bibliothèque :** la validation des éléments d'adaptation de la bibliothèque se faisait d'une manière globale par Cosimulation ou d'une manière locale par des "testbench" écris manuellement. Il n'existait pas de programmes de tests qui permettaient de vérifier tous les cas possibles du comportement d'un élément d'adaptation. Ceci est à cause de la difficulté du développement manuelle des programmes de tests et de la complexité du comportement à valider. La solution qu'on propose consiste à générer automatiquement des programmes de tests pour chaque élément d'adaptation à partir des macro-modèles de test.

4. Génération automatique des adaptateurs mémoire matériels

La génération automatique des adaptateurs mémoire matériels est aussi basée sur l'assemblage de composants de bibliothèque. Pour réutiliser l'outil de génération des interfaces décrit précédemment, il nous a fallu modifier son environnement pour l'adapter au cas des mémoires [Gha02a]. L'outil de génération d'interfaces nécessite une bibliothèque d'architectures internes, en Colif, décrivant la décomposition hiérarchique et l'interface de chaque module. Parmi ces modules, l'outil requiert la structure d'une architecture interne du module à adapter. C'est le cas de l'architecture interne PCA dans le cas d'un module de calcul. Dans le cas d'un module de mémorisation (mémoire globale), il nous a fallu définir une nouvelle architecture interne spécifique à un nœud de mémorisation appelé " GMCA " ("Global Memory Concentric Architecture").

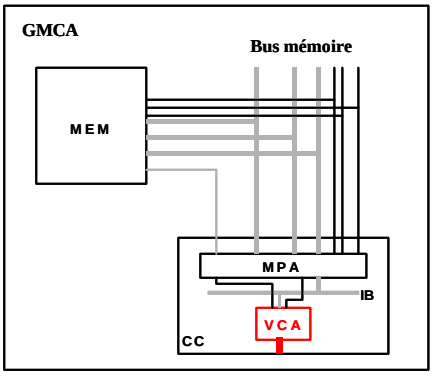


Figure 37. Architecture interne d'un nœud de mémorisation globale

Comme l'indique la Figure 37, un GMCA est composé de :

- Un module " MEM " représentant la mémoire globale.

- Un module hiérarchique " CC " représentant l'adaptateur mémoire matériel. Il est composé d'un adaptateur de port mémoire (MPA), un adaptateur de canal virtuel (VCA) qui sera raffiné en des adaptateurs de canaux (CA) par l'outil de génération des interfaces.
- Un bus interne qui supporte les accès mémoire. En plus du bus de données et de contrôle, ce dernier contient un bus d'adresses permettant l'adressage de la mémo ire globale.

La construction d'une telle architecture interne nécessite une bonne maîtrise du langage Colif. Pour permettre à l'utilisateur de concevoir ses propres architectures internes, un outil d'aide à la construction de modèles Colif est nécessaire.

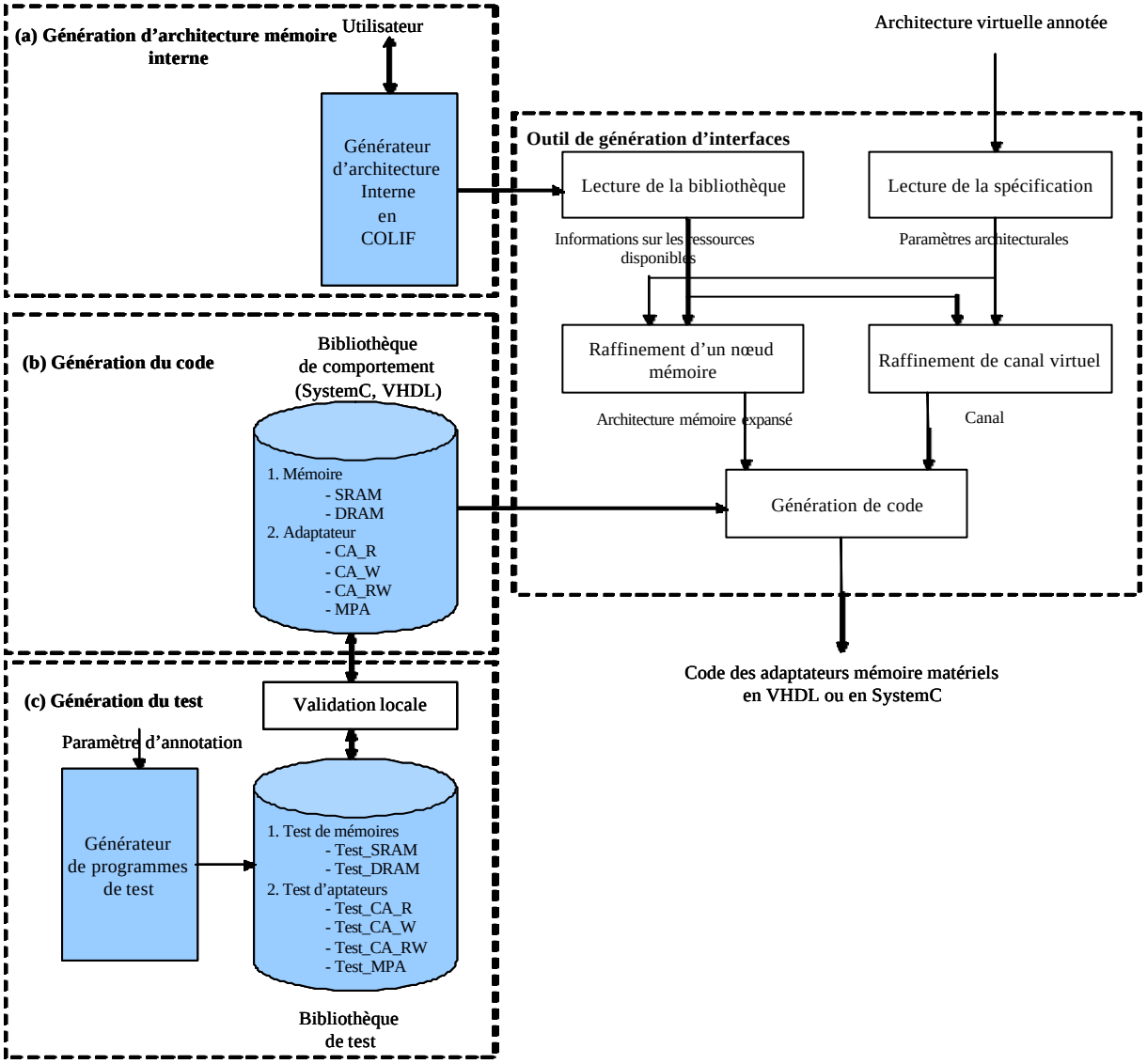


Figure 38. Environnement de génération des interfaces adapté au cas des mémoires globales

Comme l'indique la Figure 38, les extensions apportées à l'environnement de génération des interfaces sont :

- La génération d'architecture mémoire interne. Ceci correspond à la réalisation d'un outil d'aide à la construction d'architectures internes, en Colif, spécifiques aux mémoires et aux autres composants tels que le CA et le MPA. L'usager peut utiliser cet outil pour spécifier ses propres modules indépendamment du langage Colif.
- La génération du code des mémoires et de leurs adaptateurs matériels. Ceci est basé sur une bibliothèque de macro-modèles implémentant la fonctionnalité des mémoires, des adaptateurs de canaux (CA) et des adaptateurs de ports mémoire (MPA). Contrairement à la bibliothèque utilisée pour la génération des adaptateurs de processeur, cette bibliothèque implémente de nouveaux macro-modèles d'adaptateurs de canaux qui supportent l'accès à des mémoires globales.
- La génération de test. Elle correspond à la génération de programmes pour valider, localement, le comportement de chaque élément de la bibliothèque (mémoire, CA, MPA).

La suite de cette section détaille chacune de ces extensions apportées à l'environnement de génération des interfaces pour générer automatiquement les adaptateurs mémoire matériels.

4.1. Générateur d'architectures internes en Colif

L'architecture interne de chaque composant à intégrer, notamment de la mémoire, est écrite en Colif. L'utilisateur de l'outil de génération d'interfaces est donc obligé de connaître le langage Colif ou d'être limité aux éléments disponibles dans la bibliothèque. Ceci ne fait que limiter la flexibilité d'utilisation de l'outil. Pour résoudre ce problème, nous avons développé un générateur d'architectures internes (en Colif) permettant à l'utilisateur de construire l'architecture de ses propres modules sans être lié à l'environnement de l'outil de génération d'interfaces.

L'utilisation de ce générateur évite l'écriture manuelle de la bibliothèque d'architectures internes en Colif. L'utilisateur de l'outil de génération des interfaces devient alors complètement indépendant de l'environnement de génération d'interfaces. En effet :

- L'utilisateur n'est plus confronté au problème de disponibilité des éléments d'architecture dans la bibliothèque d'architectures internes. Si le type de la mémoire n'existe pas dans la bibliothèque, l'utilisateur peut construire l'architecture interne de cette mémoire en utilisant l'outil d'aide.
- L'utilisateur peut intégrer ses propres modèles sans connaître le langage Colif. C'est le générateur d'architectures internes qui se charge de la construction du squelette d'un composant ou d'un système entier.

4.1.1. Entrée du générateur

L'entrée du générateur correspond à des commandes en format texte fournis par l'utilisateur pour construire l'architecture structurelle de son système. L'interaction entre le générateur et l'utilisateur est assurée par une interface utilisateur implémentée dans le code du générateur. Cette interface demande à l'utilisateur d'entrer par exemple :

- Le nombre de modules de son système.
- Le nom et le type de chaque module. Un module peut être considéré comme : un système hiérarchique (" compound "), une boite noire (" blackbox "), un processeur (" CPU "), une mémoire (" MEM "), un module logiciel (" software "), ou un module matériel quelconque (" hardware "). L'utilisateur doit choisir un de ces types pour chacun de ses modules.
- Le nombre de ports de chaque module. Le nombre de ports supporté par le générateur est illimité.
- Le nom et la direction de chaque port. La direction de chaque port doit être " IN " ou " OUT " ou " INOUT ".
- Le type et la taille de données supportés par chaque port.
- Si le module est hiérarchique.
- Le nombre d'interconnexions entre les modules du système.
- Le nom, le type et la taille des données du média de connexion (" Net ") entre les ports.
- Les liens vers les fichiers sources de comportement de chaque module.

4.1.2. Sortie du générateur

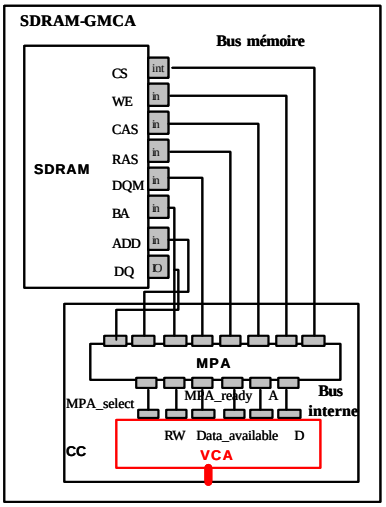


Figure 39. Architecture interne spécifique à une mémoire SDRAM

La sortie du générateur d'architectures internes est un ensemble de données correspondant à des objets Colif. Ces objets décrivent le système sous forme de modules, de ports et de "nets". Ils sont représentés sous forme d'une base de données par un fichier XML.

Le fichier de sortie sera lu par l'outil de génération des interfaces. Le contenu de ce fichier peut être également visualisé en mode graphique. La Figure 39 montre un exemple d'une architecture interne spécifique à une mémoire SDRAM.

4.1.3. Etapes du générateur

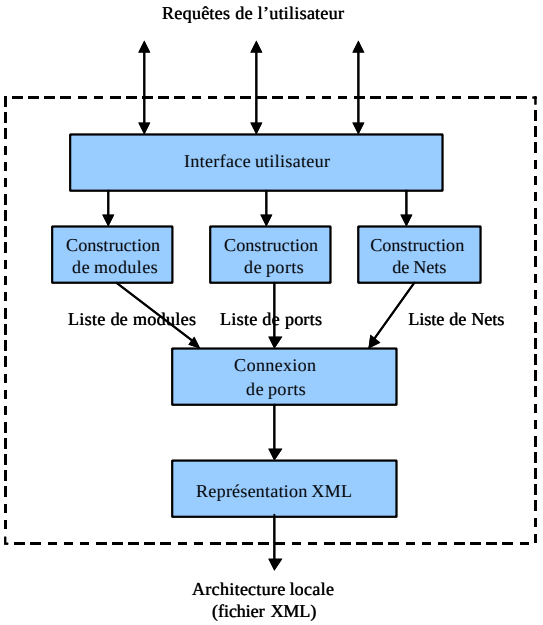


Figure 40. Décomposition fonctionnelle du générateur d'architectures internes

Comme l'indique la Figure 40, le générateur d'architectures internes est composé de six étapes principales :

a- Interface utilisateur

C'est la partie qui implémente les interactions entre l'utilisateur et le générateur. L'implémentation correspond à des fonctions d'entrées/sorties du langage C permettant la lecture des requêtes de l'utilisateur.

b- Construction de module

Cette étape consiste à construire un objet " ColifModule " et de l'instancier avec le nom et le type du module entrés par l'utilisateur. Dans le cas d'un module hiérarchique, le générateur crée une relation de

dépendance entre ce dernier et ces modules fils. Les liens vers les sources réalisant le comportement du module sont attachés à un objet " ColifModuleBehaviour " de la classe " ColifModule ".

c- Construction de ports

La construction d'un port consiste à construire un objet de la classe " ColifPort " avec les attributs donnés par l'utilisateur (nom, direction, taille, etc.). Chaque port est attaché à son module. Le générateur peut également annoter ces ports par des paramètres architecturaux fournis par l'utilisateur, qui vont guider l'outil de génération des interfaces. Cette étape fournit en sortie une liste de ports associés à chaque module.

d- Construction des "Nets"

Pour chaque niveau d'hiérarchie du système, l'interface utilisateur demande à l'utilisateur de choisir les connections d'une part, entre les différents ports des modules fils et d'autre part entre un module fils et son module père. Pour chaque type de connexion, le générateur crée un objet " ColifNet ". La sortie de cette étape est une liste de " Net " qui sert à connecter les ports des différents modules.

e- Connexion de port

Cette étape utilise la liste des modules à connecter, la liste des ports impliqués dans ces connexions et la liste des " Nets " de communication. La connexion de deux ports consiste à attacher un objet " ColifNet " de la liste des Net à chacun des ports. La sortie de cette étape est une liste contenant plusieurs objets Colif attachés.

f- Représentation XML

Cette partie prend en entrée la liste des données contenant les objets Colif interconnectés et fournit en sortie une copie de sauvegarde de ces données sous forme d'un fichier XML. Ce fichier sera l'une des entrées de l'outil de génération des interfaces.

4.2. Génération du code pour les mémoires, les CA et les MPA

La génération de code est basée sur des bibliothèques génériques de macro-modèles. Elle consiste à configurer ces macro-modèles par des paramètres de configuration. Ces paramètres sont extraits de l'architecture virtuelle annotée de l'application. Nous avons développé une bibliothèque matérielle contenant des macro-modèles de mémoires, de CA et de MPA.

4.2.1. Bibliothèque de macro-modèles de mémoire

Un élément de cette bibliothèque correspond à une implémentation de la fonctionnalité d'un modèle mémoire générique. Un macro-modèle de mémoire est une implémentation générique de mémoire qui peut

être configurable avec des paramètres qui dépendent généralement de l'application. Un macro-modèle n'est ni simulable, ni synthétisable car plusieurs caractéristiques mémoires sont encore abstraites. L'écriture de cette bibliothèque de mémoire dépend du langage cible. Les langages que nous avons ciblés sont SystemC et VHDL. Le langage d'écriture de cette bibliothèque (RIVE) est complètement indépendant du langage cible. La configuration de ces macro-modèles génère des mémoires en SystemC ou en VHDL qui sont simulables et synthétisables. La génération de ce code est obtenue en remplacement systématiquement des paramètres génériques du macro-modèle par des valeurs spécifiques à une application donnée. Ces valeurs sont extraites automatiquement de la spécification de l'application.

Nom du paramètre de configuration	*Valeur du paramètre*	*Description*
MEM_TYPE	SRAM, SDRAM	Désigne le type de la mémoire qui répond aux besoins de l'application. Selon la valeur du paramètre (SRAM ou SDRAM), une macro d'interfaces permet la génération de signaux d'interfaces spécifiques au type de la mémoire choisie.
BANC_NB	1, 2, 4, 8	Correspond au nombre de banc mémoire. Pour être cohérent avec les modèles mémoire physiques fournis par les constructeurs, le nombre de bancs utilisé est une puissance de deux qui varie entre 1 et 8.
PORT_NB	1, 2, 3	Indique le nombre de ports d'accès mémoire. Si la valeur de ce paramètre est égale à 1, alors il s'agit d'une mémoire simple port. Si elle est égale à 2, alors il s'agit d'une mémoire double ports. Dans ce cas la macro d'interface doit dupliquer les ports élémentaires de contrôle et des données.
DATA_TYPE	LV, int, short, char, …	Indique le type de données supporté par le modèle mémoire. Nous utilisons des données sous forme de vecteurs logiques (suite de 0 et 1) ou sous forme d'entiers (int, short, char …).
DATA_BIT_WIDTH	8, 16, 32, 64	Est utilisé pour fixer la représentation de données d'un port de données mémoire. On distingue une représentation sur 8, 16, 32 et 64 bits.
ADDR_BIT_WIDTH	Arbitraire	C'est comme le paramètre précédant, sauf qu'il est utilisé pour le port d'adresses de la mémoire. La valeur de configuration de ce paramètre est arbitraire.
MEM_SIZE	Arbitraire	Indique la taille totale de la mémoire.

Tableau 2. Paramètres de configuration de macro-modèles mémoire

Le Tableau 2 montre les paramètres de configuration que nous utilisons pour écrire un macro-modèle de mémoire.

4.2.2. Bibliothèque de macro-modèles de CA

Pour les CAs, nous avons implémenté des nouveaux modèles dont l'interface contient un bus d'adresses permettant l'accès à des mémoires globales. Parmi les adaptateurs de canaux actuellement disponibles dans les bibliothèques de comportement, on distingue :

- *CA_x_Register_MemFullHndshk* implémente un CA lié à un média externe utilisant le protocole de communication *MemFullHndshk*. L'élément de mémorisation interne à ce CA correspond à deux simples registres pour stocker l'adresse et la donnée.
- *CA_x_GuardedRegister_MemFullHndshk* adapte le même protocole de communication comme le CA précédant. Mais il utilise un élément de mémorisation assez spécial. Il s'agit d'un banc de registres utilisé comme une FIFO spéciale. En effet, le dépilement n'est autorisé que si la FIFO est complètement pleine. Nous utilisons ce type de registre pour les applications utilisant des accès mémoire par paquet ou par page. Ces registres sont gardés par des registres d'état.
- *CA_x_ASFIFO_MemFullHndshk* est un CA utilisant une FIFO asynchrone comme élément de mémorisation interne. Ce type de CA est utilisé dans les applications à haut débit. En effet, l'utilisation d'une FIFO asynchrone permet d'éviter le goulot d'étranglement du bus interne causé par la différence de débit entre le média externe et le bus de la mémoire.

Nom du paramètre de configuration	*Valeur du paramètre*
DATA_BIT_WIDTH[8]	8, 16, 32, 64
ADDR_BIT_WIDTH	Arbitraire
DATA_INT_WIDTH[9]	8, 16, 32, 64
ADDR_INT_WIDTH	Arbitraire
POOL_SIZE[10]	Arbitraire

Tableau 3. Paramètres de configuration d'un macro-modèle d'adaptateurs de canal

D'autres types de canaux sont également disponibles. Ils utilisent les mêmes éléments de mémorisation que les autre CA déjà cités, mais ils sont spécifiques à d'autres protocoles de communication tels que *MemEnHndshk* et *MemNoHndshk*.

Tous ces adaptateurs de canaux peuvent être configurés pour la lecture (CA_R) ou l'écriture (CA_W) ou les deux (CA_RW). On utilise les paramètres de configuration indiqués par le Tableau 3.

4.2.3. Bibliothèque de macro-modèles de MPA

Comme l'adaptateur de port mémoire (MPA) dépend du type de la mémoire, nous avons implémenté deux classes de macro-modèles de MPA : un SRAM-MPA spécifique à une mémoire SRAM et SDRAM-MPA

[8] _BIT_WIDTH : indique la taille du média externe.

[9] _INT_WIDTH : indique la taille du bus interne de l'adaptateur matériel.

[10] POOL_SIZE : indique la taille de l'élément de mémorisation du CA.

spécifique à une mémoire dynamique SDRAM. Chacun de ces macro-modèles peut générer un modèle spécifique à une interface mémoire donnée et à interface du bus interne de l'adaptateur mémoire matériel.

Le Tableau 4 montre les paramètres utilisés pour configurer un macro-modèle d'un MPA.

La machine d'états finis réalisant le comportement d'un MPA dépend des valeurs de configuration de ces paramètres. Le nombre d'états ainsi que le transfert de données sont fixés à partir de ces valeurs.

Nom du paramètre de configuration	*Valeur du paramètre*	*Description*
MEM_BUS_DATA_WIDTH	8, 16, 32, 64	Est spécifique à la taille du bus de données de la mémoire. Selon la valeur de ce paramètre (8, 16, 32, 64) la taille du port de données du MPA est fixée.
MEM_BUS_ADDR_WIDTH	Arbitraire	Indique la taille d'adresses de la mémoire.
DATA_INT_WIDTH	8, 16, 32, 64	C'est la taille du bus de données interne à l'adaptateur mémoire matériel.
ADDR_INT_WIDTH	Arbitraire	C'est la taille du bus d'adresses interne.
BURST	1, 2, 4, 8, …	Est lié au mode d'accès mémoire. Si la valeur de ce paramètre est égale à 1, alors il s'agit d'un accès simple. Dans les autres cas le mode d'accès est un accès " burst " séquentiel dont la taille est égale à la valeur du paramètre.

Tableau 4. Paramètres de configuration d'un macro-modèle d'adaptateurs de port mémoire

4.3. Génération automatique des programmes de tests

Nous validons la bibliothèque matérielle contenant le comportement des macro-modèles de mémoire, d'adaptateurs de canaux et d'adaptateurs de ports mémoire en utilisant des programmes de tests génériques. A chaque composant de la bibliothèque, est associé un macro-modèle d'un programme de simulation SystemC. Pour chaque application, on génère les composants d'adaptation nécessaires mais aussi le programme de test spécifique à cette application. On utilise les paramètres d'annotation pour configurer ces macro-modèles de tests.

La Figure 41 montre une partie du programme principal de test pour un adaptateur de canal (CA). La Figure 41 (a) est le code macro écrit en RIVE pour un langage cible SystemC. On utilise des macros pour les fichiers d'entête (ligne 2, 3) pour que le programme de test soit indépendant de la nomenclature utilisée par le programmeur de l'application. Ce programme implémente un module fournissant des vecteurs de test pour le CA via une interface générique. La taille des bus d'adresses et de données du module de test est générique

(lignes 12-15). Elle sera configurée selon le type de CA cible. La déclaration et l'instantiation des modules (objets C++, lignes 20 et 21) sont aussi configurables.

La Figure 41 (b) montre le code généré à partir du macro-modèle. Les paramètres de configuration utilisés correspondent aux :

- Noms des fichiers sources d'entêtes. (*CA_W_Register_MemFullHndshk.h* et *test_CA_W_Register_MemFullHndshk.h*).
- Noms des modules d'instantiation.
- La taille du bus de données du média externe est égale à 16 bits, celle du bus d'adresses est égale à 23 bits.
- La taille de la ligne de données et d'adresses du bus interne de l'adaptateur est égale à 32 bits.

```
1.  #include "systemch"
2.  #include "test_ @{entity_name}@.h "
3.  #include "../@{entity_name}@.h"

4.  int sc_main(int ac, char* av[])
5.  {
6.  //-------------------------------------------------------------------------
7.  //------      Net Declaration.                          ------
8.  //-------------------------------------------------------------------------
9.      sc_signal<bool> Req;
10.     sc_signal<bool> Ack;
11.     sc_signal<bool> nRW;
12.     sc_signal<sc_lv<@{DATA_BIT_WIDTH}@> >Data_ext_in;
13.     sc_signal<sc_lv<@{ADDR_BIT_WIDTH}@> >Addr_ext;
14.     sc_signal <sc_lv<@{DATA_INT_WIDTH}@ > > D;
15.     sc_signal <sc_lv<@{ADDR_INT_WIDTH}@ > > A;
16.     ….
17. //-------------------------------------------------------------------------
18. //------      Submodule Declaration                     ------
19. //-------------------------------------------------------------------------
20.     test_@{entity_name}@ *test;
21.     @{entity_name}@ *module;
22. //-------------------------------------------------------------------------
23. //------      Submodule Instanciation.                  ------
24. //-------------------------------------------------------------------------
25.     ….
26.     sc_start(200);
27.     return(0);
28.
```

```
1.  #include "systemch"
2.  #include " test_CA_W_Register_MemFullHndshk.h "
3.  #include "../CA_W_ Register_MemFullHndshk.h "

4.  int sc_main(int ac, char* av[])
5.  {
6.  //-------------------------------------------------------------------------
7.  //------      Net Declaration.                          ------
8.  //-------------------------------------------------------------------------
9.      sc_signal<bool> Req;
10.     sc_signal<bool> Ack;
11.     sc_signal<bool> nRW;
12.     sc_signal<sc_lv<16> >Data_ext_in;
13.     sc_signal<sc_lv<23 > >Addr_ext;
14.     sc_signal <sc_lv<32 > > D;
15.     sc_signal <sc_lv<32> >A;
16.     ….
17. //-------------------------------------------------------------------------
18. //------      Submodule Declaration                     ------
19. //-------------------------------------------------------------------------
20.     test_CA_W_ Register_MemFullHndshk *test;
21.     CA_W_ Register_MemFullHndshk  *module;
22. //-------------------------------------------------------------------------
23. //------      Submodule Instanciation.                  ------
24. //-------------------------------------------------------------------------
25.     ….
26.     sc_start(200);
27.     return(0);
28.
```

Figure 41. Programme de test d'un CA : (a) macro-modèle, (b) code généré.

5. Application : filtre d'image de bas niveau

5.1. Choix de l'application

Les raisons qui ont motivé ce choix d'application de traitement d'image sont :

- Le besoin d'une mémoire de grande taille pour stocker les images. Ceci nous a motivé pour utiliser une mémoire globale.

- La possibilité de paralléliser cette application. Ceci nous a permis d'appliquer notre approche dans un contexte multiprocesseur.

5.2. But de l'application

Le but de cette application est de valider notre approche de co nception des adaptateurs mémoire matériels. Plusieurs points doivent être vérifiés par cette application :

- Validation globale du comportement des adaptateurs mémoire.
- Flexibilité de l'architecture matérielle de l'adaptateur mémoire.
- Valider la génération automatique.

5.3. Domaine d'utilisation

Les applications de traitement d'images consomment de plus en plus d'espace mémoire pour stocker les images avant, pendant et après le calcul. Un filtre d'image est un exemple typique de ces applications. Il a pour but d'éliminer les aberrations et les bruits introduits lors de la capture de l'image. Il produit des images claires et faciles à interpréter. Il peut être utile dans le domaine médical, biophysique, astronomie ou autres domaines. Il offre aux scientifiques une meilleure interprétation de l'image.

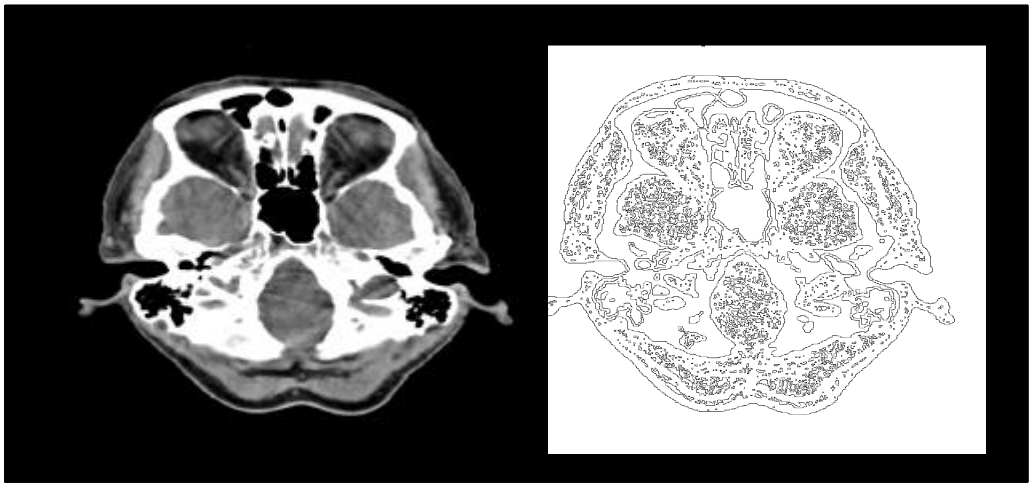


Figure 42. Image médicale avant et après l'utilisation d'un filtre

La Figure 42 montre l'utilisation d'un filtre d'image dans le domaine médical. La figure de gauche représente une image d'une coupe transversale d'un cerveau humain qui est difficile à interpréter à l'œil nu par un médecin. La figure de droite est l'image produite par le filtre. Le médecin peut alors détecter éventuellement les cellules cancéreuse ou les tumeurs facilement.

5.4. Description de l'application

Un filtre d'image fait partie de la fonctionnalité de plusieurs appareils photo digitals. La Figure 43 montre l'architecture d'un appareil photo commercial " JAM CAM " utilisant un filtre d'image [Pix03].

Le comportement de ce filtre consiste à appliquer des convolutions successives sur les pixels de l'image pour arrondir les pixels défectueux. L'algorithme original de ce filtre utilise deux convolutions successives : une première horizontale puis une deuxième verticale. L'architecture originale de ce système contient beaucoup de mémoire mais un seul processeur RISC 8 bits. Le système de mémoire utilisé contient une mémoire Flash de 2 Mo, une mémoire RAM statique de 256 Ko et une mémoire ROM de 2 Mo. L'image prise par l'appareil photo est sauvegardée dans la mémoire Flash avant d'être traitée par le processeur.

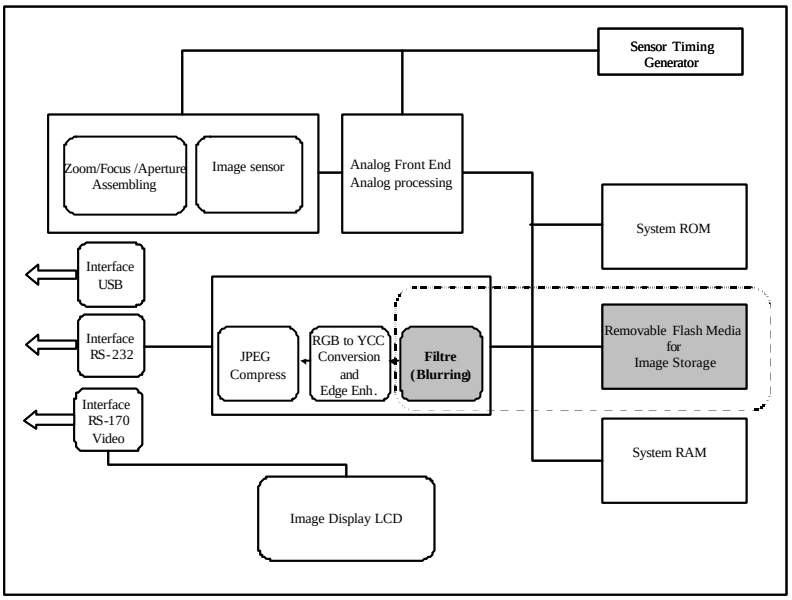


Figure 43. Architecture fonctionnelle d'un JAMCAM

5.5. Architecture abstraite de l'application

Chaque image est découpée en deux parties. Une tâche consiste à appliquer de façon séquentielle une convolution horizontale puis verticale sur une partie de l'image. Le traitement des deux parties de l'image peut donc être fait en parallèle (Figure 44).

Chaque partie de l'image est lue dans la mémoire, subit la convolution horizontale, est recopiée dans la mémoire, puis relue pour la convolution verticale, avant d'être encore une fois recopiée.

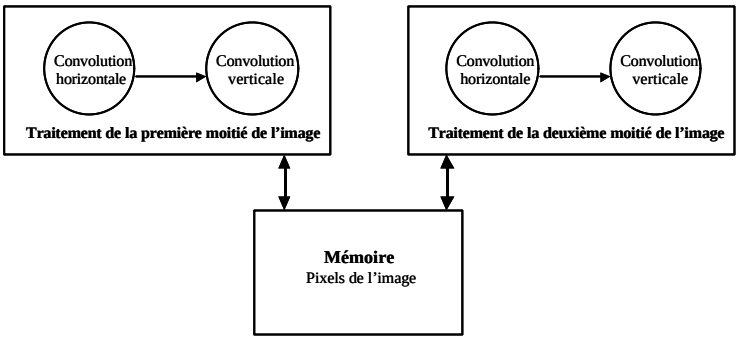


Figure 44. Architecture fonctionnelle de l'application

5.6. Architecture matérielle

Nous ciblons cette application sur une architecture composée de (Figure 45) :

- Deux processeurs ARM7TDMI (RISC 32 bits). Le choix du type de processeur dépend de la disponibilité.
- Une mémoire globale partagée par les deux processeurs. Pour prouver la flexibilité des adaptateurs mémoire, nous avons traité deux expériences utilisant chacune un type de mémoire. Pour la première expérience, nous utilisons une mémoire SRAM double ports, tandis que nous utilisons une mémoire SDRAM simple port pour la deuxième expérience.
- Un réseau de communication point à point qui relie les deux processeurs et la mémoire.

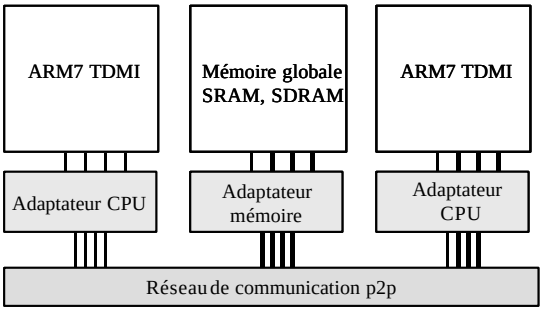
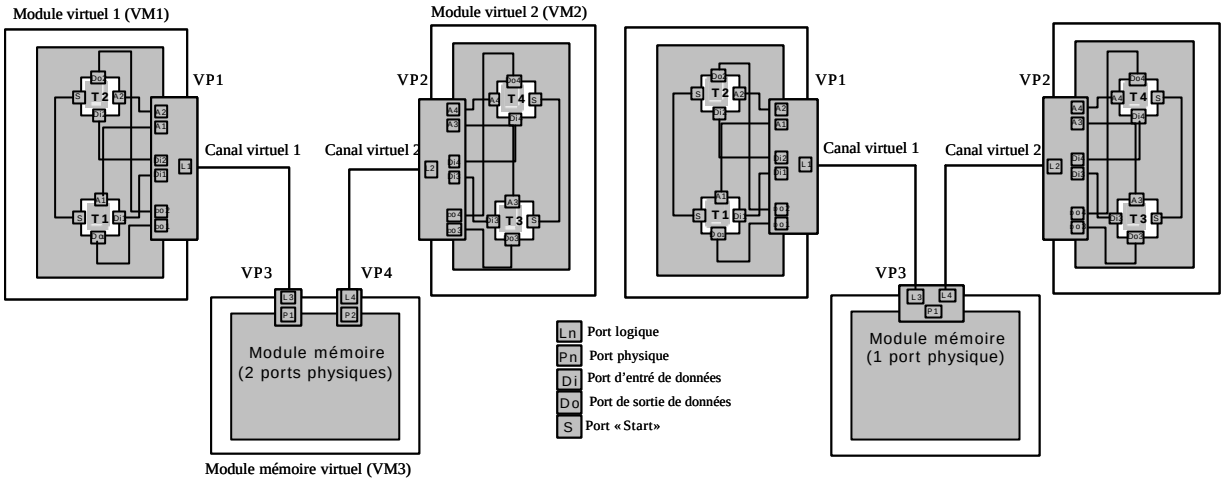


Figure 45. Architecture cible

5.7. Spécification VADeL et Architecture virtuelle

Dans les deux expériences, la spécification VADeL de l'application est constituée de deux modules virtuels de calculs symétriques VM1 et VM2 et un module de mémorisation VM3 (Figure 46).



(a) (b)

Figure 46. Architecture virtuelle de la spécification VADeL : (a) expérience 1, (b) expérience 2

La communication entre les deux modules est assurée par deux canaux virtuels de type "pipe ".

5.7.1. Modules de calcul virtuels VM1 et VM2

VM1 et VM2 sont symétriques, ils ont les mêmes interfaces et les mêmes comportements. Pour les deux modules, l'interface comme le comportement restent inchangés dans les deux expériences. Seuls ceux du module VM3 changent.

L'interface de VM1 est composée d'un port virtuel dont quatre ports internes et un port externe :

- A1 : est un port logique interne utilisé par le module M1 pour envoyer les adresses d'accès mémoire à partir de la tâche T1.
- A2 : similaire à A1, sauf qu'il est utilisé par la tâche T2 du module M1.
- Di1 : est un port logique utilisé pour la lecture des données par la tâche T1.
- Do1 : est un port logique de données utilisé par le module M1 lors des opérations d'écriture exécutées par la tâche T1.
- Di2 : est comme Di1, mais pour la tâche T2.
- Do2 : est comme Do1, mais pour la tâche T2.
- L1 : est le seul port externe du port virtuel VM1. C'est un port logique.

Le Tableau 5 résume les paramètres de configuration qui annotent le module virtuel de calcul VM1. Ces paramètres sont attachés au module, au port interne et au port externe du port virtuel.

	Module	*Ports internes*	*Port externe*
		A1	*L1*
CPUName	CPUARM7		
ADDRESS_INT	0x1000		
C_DATA_TYPE		long int	
IT_LEVEL		1	
MASK_GET		2	
MASK_PUT		1	
ADDRESS_STATE		0x000F0000	
ADDRESS_DATA		0x000F0004	
SoftPortType		GuardedRegister	
HardPortType			MemFullHndshk
DATA_INT_WIDTH			32
POOL_SIZE			16

Tableau 5. Paramètres de configuration du module virtuel VM1

Paramètres attachés au module VM1

- CPUName : est attaché au module VM1 avec une valeur (CPUARM7) indiquant qu'il s'agit d'un processeur ARM7.
- ADDR_INT: indique que les interruptions du module VM1 sont à l'adresse mémoire 0x1000.

Paramètres attachés au port interne A1 du port virtuel VM1

Pour simplifier la description de ces paramètres, on ne détaille qu'un seul port interne.

- C_DATA_TYPE : ce paramètre est attaché aux ports internes du port virtuel VP1. La valeur de ce paramètre est " char " pour tous les ports de données internes. En effet, l'application traite des pixels qui sont représentés sur 8 bits comme tous les données de type " char ". Pour les ports internes d'adresses (A0, A1), ce paramètre prend une valeur " long int " compatible avec le bus d'adresses d'un processeur ARM7 (32 bits).
- IT_LEVEL : est aussi attaché aux ports internes. Comme les deux tâches T1 et T2 sont séquentielles, les niveaux de leurs interruptions sont alors égaux. La valeur attribuée est 1.
- MASK_GET et MASK_PUT : ces deux paramètres positionnent les drapeaux de lecture et d'écriture du canal respectivement à 2 et 1.
- ADDRESS_STATE : indique au flot de conception des pilotes logiciels l'adresse d'un registre d'état du canal (0x000F0000).
- ADDRESS_DATA : sert à indiquer, aux entrées / sorties d'un processeur, l'adresse physique d'un canal (0x000F0004).
- SoftPortType: ce paramètre est aussi attaché aux ports internes. Il est utilisé pour spécifier le pilote logiciel associé au port. Dans ce cas, il s'agit d'un pilote d'accès à un registre gardé (GuardedRegister).

Paramètres attachés au port externe L1 du port virtuel VP1

- HardPortType : ce paramètre est attaché au port externe. Il est spécifique au protocole de communication qui va être implémenté par l'adaptateur de canal. Dans ce cas, la valeur attribuée au paramètre indique qu'il s'agit d'un protocole "MemFullHndshk ". L'outil de génération des interfaces sélectionne le CA dont le nom est la jonction entre les valeurs des paramètres SoftPortType et HardPortType.
- DATA_INT_WITH : ce paramètre est aussi attaché au port externe du port virtuel VP1. Il indique à l'outil de génération des interfaces que la taille du bus interne est 32 bits. Ce choix nous permet d'avoir des accès à des pixels mémoire en mode " burst " de taille 4 (4x8=32 bits).
- POOL_SIZE : le nombre de mots de l'élément de mémorisation du CA doit être égal à 16.

5.7.2. Module de mémorisation globale virtuel VM3

L'interface de ce module dépend du type de la mémoire utilisée dans chaque expérience. Dans l'expérience 1, VM3 est composé de deux ports virtuels (VP3 et VP4) tandis qu'il est composé d'un seul port virtuel (VP3) dans l'expérience 2.

Puisque le protocole de communication utilisé est le même pour les deux expériences, les paramètres de configuration spécifiques aux ports externes de ce module restent inchangés dans les deux expériences.

	Port externe (L3 ou L4)
DATA_INT_WIDTH	32
ADDR_INT_WIDTH	32
POOL_SIZE	16
HardPortType	MemFullHndshk

Tableau 6. Paramètres de configuration spécifiques aux ports externes des ports virtuels mémoire

	SRAM	*SDRAM*
PORT_NB	2	1
MEM_BUS_DATA_WIDTH	32	16
MEM_BUS_ADDR_WIDTH	32	16
BURST	4	4

Tableau 7. Paramètres de configuration spécifiques aux ports internes des ports virtuels mémoire dans les deux expériences

Le Tableau 6 résume les paramètres de configuration liés aux ports externes des ports virtuels. Ces paramètres sont les mêmes pour les ports logiques externes L3 et L4, car ces derniers sont symétriques. Les mêmes paramètres sont applicables dans les deux expériences.

Les paramètres de configuration des ports internes de la mémoire dépendent du type mémoire utilisé dans chaque expérience.

Le Tableau 7 résume les paramètres de configurations liés aux ports internes du module mémoire virtuel VM3 dans les deux expériences. Dans la première expérience, VM3 correspond à une SRAM double ports. La taille du bus de données et d'adresses est 32 bits pour la mémoire SRAM de l'expérience 1, tandis qu'elle est de 16 bits pour la mémoire SDRAM de l'expérience 2.

Nous utilisons le langage VADeL pour spécifier l'application et pour l'annoter avec les paramètres de configuration. Ces paramètres guideront l'outil de génération des interfaces et configureront les bibliothèques génériques. La Figure 47 montre le code VADel qui spécifie le canal virtuel VC1. Ce dernier est un canal hiérarchique contenant un canal d'adresses (ADDR_CH1 : ligne 6) et un canal de données (DATA_CH1 : ligne 7). Ils représentent un protocole " pipe[11] " décrit au niveau architecture virtuelle (désigné par " mac ").

```
//-------------------------------------------------------------------------------------
1.    // definition of the virtual channels
2.    //
3.
4.    VA_CHANNEL(vc1)
5.    {
6.     va_ch_mac_pipe        ADDR_CH1;
7.     va_ch_mac_pipe        DATA_CH1;
8.
9.     VA_CCTOR(vc1)
10.    {
11.     VA_CEND
12.    };
13.   };
14.
```

Figure 47. Code VADeL décrivant le canal virtuel VC1

La Figure 48 montre la partie du programme principal VADeL spécifique à la description du module mémoire virtuel VM3 dans le cas de l'expérience 1. Ce code contient plusieurs parties :

- Initialisation de l'environnement de VADeL (ligne 7).
- Instantiation des deux canaux virtuels VC1 et VC2 (lignes 10 et 11).
- Instantiation du module mémoire virtuel VM3 (ligne 14).
- Définition du niveau d'abstraction des deux ports virtuels VP3 et VP4 du module mémoire (lignes 17 et 18). L'attribut " DL " désigne le niveau architecture virtuelle.

[11] Protocole pipe : c'est un protocole qui permet l'envoi des données sans implémenter aucune synchronisation entre les interlocuteurs.

- Définition du type de la mémoire (ligne 21) et des niveaux d'abstraction des deux canaux. (lignes 24 et 25).
- Annotation du module mémoire (lignes 28 et 29). Avec l'API " setColifParam ", on annote le module mémoire par le niveau RTL ainsi que les liens vers les sources de comportement.
- Les ports sont également annotés (à partir de la ligne 32).
- Enfin, une traduction de la description VADeL en format intermédiaire Colif (ligne 39).

```
1.  //-----------------------------------------------------------------------------------------
2.  // main program
3.  //
4.  int sc_main( int argc, char *argv[])
5.  {
6.    // VADeL initialization
7.    va_init();
8.
9.    // instanciate virtual channels
10.   vc1 VC1("VC1");
11.   vc2 VC2("VC2");
12.
13.   // instanciate virtuals modules & sc_modules.
14.   vm3 VM3("VM3");
15.
16.   //Virtual ports
17.   VM3.VP3->level = "DL";
18.   VM3.VP4->level = "DL";
19.
20.   //Virtual modules
21.   VM3.MEMName = "SRAM";
22.

23.  //Virtual Nets
24.   VC1.level = "DL";
25.   VC2.level = "DL";
26.
27.   //Annotation of modules
28.   VM3.M3->setColifParam("module_level","RTL");
29.   VM3.M3->setColifParam("Source","SystemC","VM3/m3.cpp","VM3/m3.h");
30.
31.    //Annotation of ports
32.   VM3.L3.POOL_SIZE = "32";
33.   VM3.L3.HardPortType = "MemFullHndshk";
34.   VM3.L4.POOL_SIZE = "16";
35.   VM3.L4.HardPortType = " MemFullHndshk ";
36.    …..
37.
38.   // translate on COLIF
39.   sc_start(0);
40.
41.   return EXIT_SUCCESS;
42. };
```

Figure 48. Programme principal VADeL

5.8. Génération automatique

5.8.1. Génération d'architecture interne spécifique à la mémoire

Les étapes de la génération automatique de la GMCA correspondent à une succession d'interactions entre l'utilisateur et l'outil de génération d'architecture interne. L'outil demande à l'utilisateur d'entrer des informations spécifiques aux modules, aux ports et aux signaux de connexion.

5.8.2. Génération des programmes de tests

La validation locale des composants de base de l'adaptateur mémoire est faite en deux étapes essentielles. La première étape consiste à extraire les valeurs des paramètres de configuration à partir de la spécification de l'application. La deuxième étape consiste à configurer les macro-modèles de tests par les valeurs extraites. Pour cette application, nous avons généré un programme de test spécifique à un adaptateur de canal de type *CA_RW_GuardedRgister_MemFullHndshk*.

5.8.3. Génération de l'adaptateur mémoire

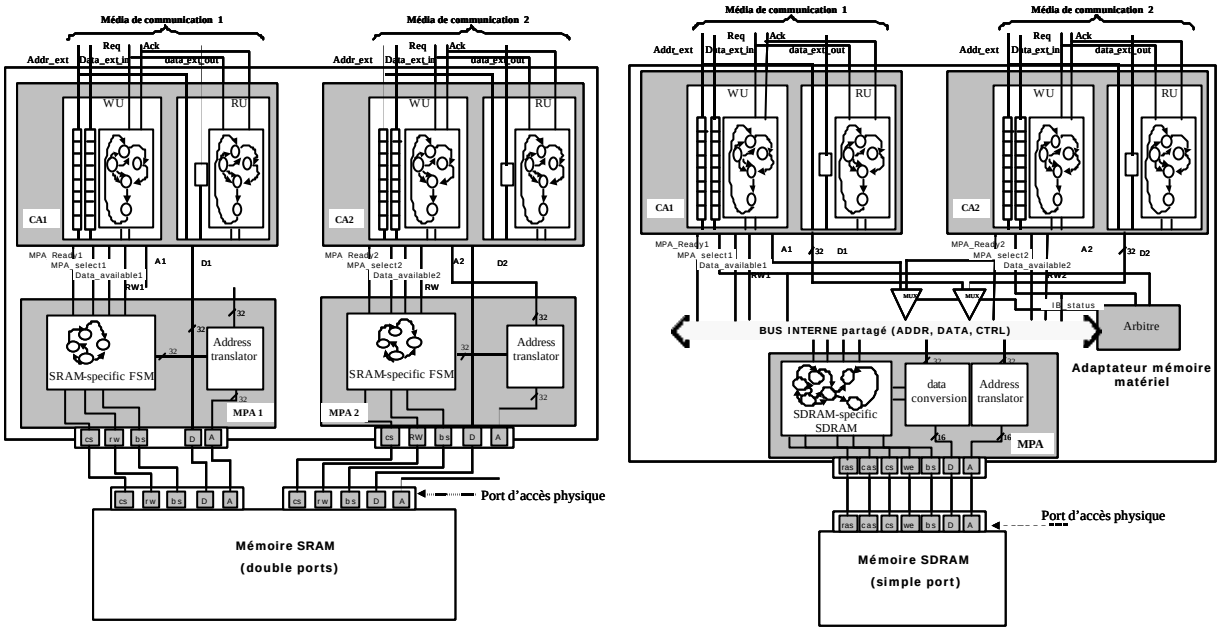
Les étapes de la génération de l'architecture globale de l'adaptateur mémoire pour cette application sont cinq :

- Extraction des paramètres de configuration.
- Lecture de la bibliothèque de modèles Colif produite par l'outil de génération d'architectures internes.
- Raffinement de l'adaptateur de canal virtuel de la GMCA.
- Raffinement des deux canaux virtuels qui connectent le module mémoire aux deux modules de calcul.
- Génération du code de la mémoire SRAM (expérience 1) et de la mémoire SDRAM (expérience 2) ainsi que la génération du code des CAs et des MPAs. Cette génération correspond à l'expansion de macro-modèles.

5.9. Analyse des résultats

5.9.1. Flexibilité des adaptateurs mémoire matériels

Le résultat de la génération des interfaces mémoire spécifiques à cette application est un adaptateur matériel synthétisable décrit en SystemC et en VHDL.



(a) **(b)**

Figure 49. Micro-architecture des adaptateurs mémoire matériels générés : (a) expérience 1, (b) expérience 2

Expérience 1: comme l'indique la Figure 49 a, l'adaptateur mémoire matériel spécifique à cette expérience est composé de :

- Deux adaptateurs de canaux de type CA_RW_GuardedRgister_MemFullHndshk. Chaque CA est composé d'une unité d'écriture (WU) et d'une unité de lecture (RW). L'unité d'écriture utilise deux FIFOs (pour les adresses et les données) contrôlées par la même machine d'état finie. La taille de ces FIFOs est égale à 16 mots. L'élément de mémorisation utilisé par l'unité de lecture correspond à un registre de stockage des données lues. La taille de ce registre est égale à 32 bits. Ceci permet de lire quatre pixels (8x4) d'un seul coup en utilisant un accès " burst ".
- Deux adaptateurs de ports mémoire spécifiques aux deux ports d'accès de la mémoire SRAM. Chaque MPA est composé d'un module de traduction d'adresses et d'une machine d'états finis qui adapte l'interface du bus interne à celle de la mémoire SRAM. Cette machine gère aussi le transfert de données entre les deux parties.

Expérience 2: comme l'indique la Figure 49 b, l'adaptateur mémoire matériel spécifique à cette expérience est composé des mêmes adaptateurs de canaux que l'expérience précédente, mais d'un seul adaptateur de port mémoire. Ce dernier est spécifique au port d'accès de la mémoire SDRAM. La machine d'états finis de cet MPA est complètement différente de celle de la mémoire SRAM. Ceci vient du fait que

l'interface de la mémoire SDRAM est très différente de la mémoire SRAM et que les tailles des bus mémoire ne sont pas les mêmes.

Malgré ces différences, nous avons pu réutiliser le code des adaptateurs de canaux de l'expérience précédente. La génération d'un nouveau MPA pour l'expérience 2 a nécessité une simple modification dans le code VADeL spécifique au module mémoire virtuel. Cette modification se concrétise par la définition d'un seul port virtuel composé d'un seul port interne et par l'annotation du code par les nouveaux paramètres de configuration caractérisant l'interface de la mémoire SDRAM. Le temps nécessaire pour cette modification est négligeable par rapport au temps total nécessaire pour coder toute l'application. Ceci ne peut que prouver la flexibilité de ces adaptateurs mémoire matériels.

5.9.2. Complexité en terme de portes logiques

Pour juger la complexité du code généré des adaptateurs mémoire matériels, nous avons utilisé la technologie AMS 3.20 (0.35 µm) pour synthétiser le code VHDL de l'adaptateur de canal et de l'adaptateur de port mémoire utilisés dans l'expérience 1.

Le résultat de cette synthèse est 549 portes pour le CA (Figure 50) et 430 portes pour le MPA (Figure 51). Si on tient compte de la complexité du bus interne (moins de 150 portes), la complexité totale de l'adaptateur mémoire de l'expérience 1 est de l'ordre de 2000 portes. Indépendamment de l'application, la complexité de l'adaptateur mémoire est généralement inférieure à 2500 portes. Nous estimons que cette complexité est très acceptable par rapport à des applications qui correspondent à des milliers voir des millions de portes.

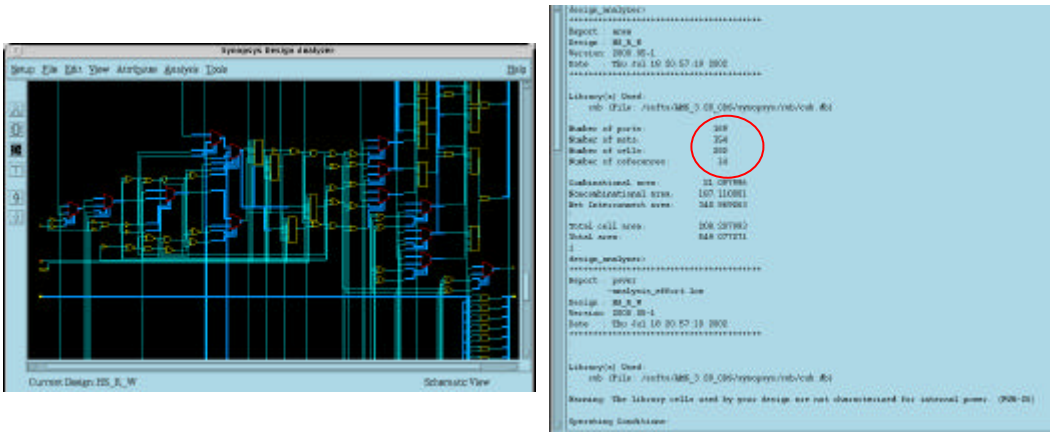


Figure 50. Résultat de synthèse d'un adaptateur de canal

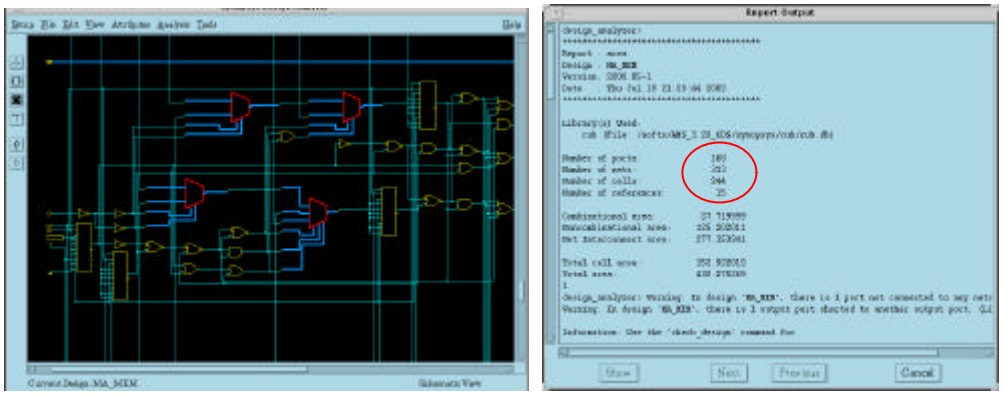


Figure 51. Résultat de la synthèse d'un adaptateur de port mémoire

5.9.3. Surcoût en temps

La pénalité temporelle due à un adaptateur matériel lors d'un accès mémoire est de l'ordre de 2 à 3 cycles du processeur. Sachant qu'un accès à une mémoire globale SDRAM peut atteindre une cinquantaine de cycles, ce surcoût temporel peut être considéré comme négligeable. Même dans le cas des mémoires globales plus rapides que la SDRAM (vingtaine de cycle CPU), nous estimons que ce résultat reste acceptable.

5.9.4. Temps de génération

Le temps de génération des adaptateurs mémoire ne prend que quelques minutes. Ce temps ne correspond qu'à la durée d'exécution interne de l'outil de génération d'interfaces. Le temps de l'écriture manuelle de la spécification VADeL est plus significatif, il est de l'ordre de 2 à 3 jours pour un connaisseur du langage. La tâche la plus lente est l'écriture de la bibliothèque de macro-modèles implémentant le comportement des éléments de base de l'adaptateur mémoire matériel. La cause de cette difficulté est que :

- L'écriture d'un macro-modèle nécessite un effort considérable pour couvrir toutes les configurations possibles.
- La validation globale au niveau cycle près d'un adaptateur matériel mémoire nécessite un temps important pour la vérification de la synchronisation entre les différents composants de l'architecture.

6. Conclusion

La génération automatique des adaptateurs mémoire matériels permet d'accélérer l'intégration des mémoires globales dans les systèmes monopuces. Ceci ne fait que faciliter la réutilisation des composants mémoire déjà existants (IPs mémoire) et par conséquent réduire le fossé entre la productivité et l'intégration. La génération de ces adaptateurs est basée sur l'assemblage de composants génériques à partir de bibliothèque de macro-

modèles. L'environnement de l'outil de génération a été étendu pour qu'il puisse supporter les mémoires. Ces extensions sont liées essentiellement aux bibliothèques d'architectures internes, de comportement et de test des adaptateurs de canaux, des adaptateurs de ports mémoire et aux composants mémoire.

L'application de traitement d'images, que nous avons développé, nous a permis de valider le comportement global des adaptateurs mémoire (par co-simulation) ainsi que leur flexibilité. Nous estimons que la complexité du code RTL généré pour un adaptateur mémoire est acceptable (2000 portes) et que le surcoût de communication est aussi raisonnable (2 à 3 cycles).

Chapitre 6 : GENERATION DES PILOTES EN COUCHES D'ACCES MEMOIRE

Ce chapitre présente des travaux toujours en cours sur la conception de la partie logicielle des interfaces logiciel-matériel nécessaire à l'intégration des mémoires globales dans les systèmes monopuces. Ce chapitre est composé de cinq sections. La première section introduit la conception des adaptateurs logiciels. Dans la deuxième section, nous proposons une architecture modulaire et portable des adaptateurs logiciels. Dans la troisième section, nous présentons un flot de génération de ces adaptateurs. Dans la quatrième section, nous traitons un exemple d'adaptateur logiciel spécifique à des accès mémoire via un protocole DMA. Une conclusion est donnée dans la cinquième section.

1. Introduction : conception des pilotes logiciels

1.1. Rappel de la nécessité des adaptateurs mémoire logiciels

Un adaptateur logiciel est l'implémentation logicielle des interfaces logiciel-matériel permettant au logiciel de l'application d'accéder au matériel. Dans le cas des mémoires, cet adaptateur correspond au pilote logiciel permettant au code d'accès mémoire de haut niveau d'être exécuté sur le processeur cible pour lire/écrire des données dans la mémoire. L'utilisation de ces pilotes s'avère nécessaire pour les raisons suivantes :

- Le code de l'application est généralement décrit à un haut niveau d'abstraction sans tenir compte des contraintes architecturales. Pour cette raison, il n'est pas toujours possible de l'exécuter directement sur le processeur cible. L'utilisation d'un pilote logiciel est alors indispensable.
- A cause du large espace de solutions architecturales, le concepteur a besoin d'une couche d'adaptation logicielle entre le code de l'application et chaque architecture cible. Ceci assure la portabilité du code de l'application.
- L'utilisation d'une couche logicielle (pilote) qui sépare le logiciel de l'application du matériel permet d'assurer une conception concurrente. En effet, le concepteur de l'application logicielle et celui du matériel peuvent travailler indépendamment sans se soucier de leurs problèmes respectifs. C'est à l'assembleur de composants de concevoir les adaptateurs entre le logiciel et le matériel (c'est le cas de cette thèse).

A la différence des pilotes classiques, les pilotes des systèmes monopuces sont spécifiques à une application donnée. Ils n'implémentent que les services nécessaires au fonctionnement de l'application. Ceci provient du fait que les ressources sont limitées (taille mémoire, surface, énergie, …).

La conception des pilotes logiciels nécessite une connaissance multidisciplinaire couvrant le domaine logiciel et matériel. Le code d'un pilote d'accès est intimement lié au système d'exploitation et au matériel sur lequel il est exécuté. Du côté logiciel, le concepteur doit connaître les services fournis par le système d'exploitation. Tandis que du côté matériel, il doit connaître les détails de l'architecture matérielle cible. Ceci ne fait qu'accroître les difficultés de réutilisation et d'automatisation de la conception des pilotes. Pour résoudre ces problèmes, nous proposons une architecture en couches permettant la réutilisation des pilotes (au moins quelques parties). Nous proposons aussi un flot de génération automatique permettant une conception rapide des pilotes spécifiques à une application donnée.

1.2. Les objets requis pour la conception des pilotes logiciels d'accès mémoire

Pour concevoir des pilotes spécifiques à une application donnée, nous avons besoin d'informations caractérisant les besoins de l'application et l'architecture logiciel-matériel de l'application (SE, processeur, communication, …). Ces informations sont utilisées pour configurer des éléments de base du pilote à partir

d'une bibliothèque générique. La spécification de ces informations et l'implémentation de cette bibliothèque nécessitent plusieurs éléments :

- Comme dans le cas des adaptateurs matériels, nous utilisons le langage VADeL pour décrire l'architecture logiciel-matériel de l'application. Cette description est annotée avec des paramètres de configuration qui sont utilisés pour la sélection et la spécialisation des éléments nécessaires à la construction du pilote. Ces paramètres spécifient les services requis et fournis par le pilote logiciel pour une application donnée.
- Notre méthode de génération de pilotes logiciels est basée sur des bibliothèques génériques implémentant des macro-modèles d'éléments de base nécessaires au fonctionnement du pilote. L'écriture de ces éléments nécessite la connaissance du langage de description de macro-modèles RIVE. Nous utilisons le langage C comme un langage cible pour ces macro-modèles.
- A la différence de la bibliothèque des adaptateurs mémoire matériels, la bibliothèque des pilotes logiciels est composée de plusieurs éléments hiérarchiques dépendants. En effet, un élément peut contenir d'autres composants élémentaires et il peut dépendre d'une fonctionnalité fournie par un autre élément. Pour décrire ces relations de dépendance entre les éléments de la bibliothèque des pilotes logiciels, nous avons utilisé un langage nommé Lidel basé sur XML [Gau01].

1.3. Résultats attendus

Les résultats attendus de notre approche de réalisation d'adaptateurs mémoire logiciels sont :

- L'architecture logicielle de ces pilotes doit être portable et flexible pour qu'elle puisse être réutilisée par plusieurs applications.
- Le code des pilotes doit être spécifique à l'application. Il ne doit contenir que les parties nécessaires aux besoins de l'application. Par conséquent, une faible taille du code est souhaitée.
- Le temps de réalisation des pilotes d'accès mémoire doit être minimal.

2. Architecture en couches d'un pilote logiciel monopuce

2.1. Architecture logicielle d'un système monopuce

Comme l'indique la Figure 52, notre vue conceptuelle d'un système monopuce est composée de trois couches logicielles :

- Une couche logicielle de haut niveau qui correspond aux tâches de l'application. L'utilisateur peut coder cette partie en utilisant des APIs de communication de haut niveau qui sont complètement indépendantes de la plateforme architecturale.
- Une couche représentant le système d'exploitation.

- Une couche logicielle d'abstraction du matériel appelée HAL (Hardware Abstraction Layer) qui permet d'abstraire les interfaces logiciel-matériel en cachant les détails de la plateforme matérielle. La réalisation de cette couche correspond au code logiciel dépendant du matériel appelé HdS ("Hardware-dependent Software"). Dans cette thèse, nous nous concentrons seulement sur la conception du HAL dont l'implémentation correspond aux pilotes d'accès mémoire.

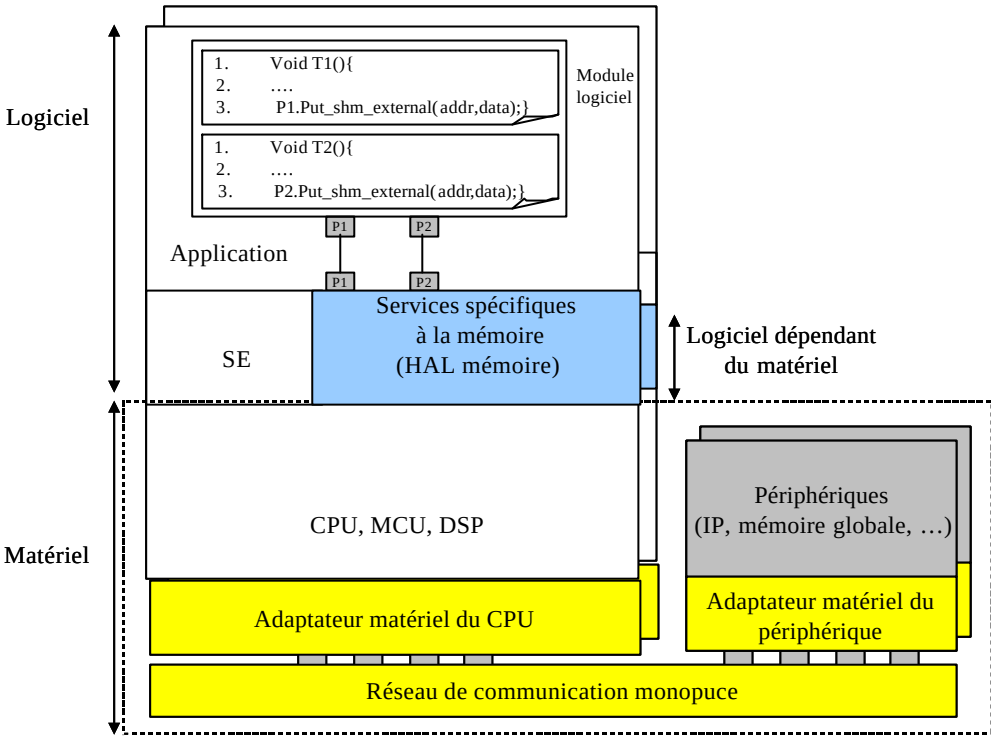


Figure 52. Architecture logiciel-matériel d'un système monopuce

La couche HAL permet au code de l'application et au système d'exploitation d'accéder à la plateforme matérielle qui peut contenir des processeurs (CPU, DSP, …), des périphériques (IP, mémoire globale, …), des adaptateurs d'interfaces matériels et des réseaux de communication.

2.2. Architecture du HAL : pilotes en couches

La couche HAL correspond aux pilotes logiciels. Elle adapte les accès mémoire de haut niveau à l'architecture cible. L'architecture de ces pilotes logiciels est structurée en quatre couches superposées pour assurer leur flexibilité et limiter leur complexité. En effet, cette structuration permet la séparation explicite entre le code des pilotes dépendant et indépendant du processeur. Ceci facilite la réutilisation du code et par conséquent, améliore la productivité du logiciel.

Comme l'indique la Figure 53, les quatre couches d'un pilote logiciel correspondent aux parties suivantes :

- API du pilote.
- Des services dépendants du logiciel de l'application[12].
- Des éléments de mémorisation internes au pilote.
- Une partie implémentant le logiciel dépendant du matériel.

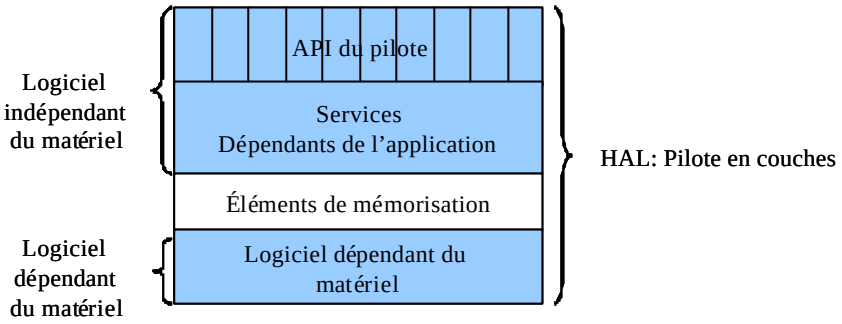


Figure 53. Architecture en couches d'un pilote logiciel

2.2.1. API du pilote

C'est la couche supérieure du pilote qui correspond à une interface de programmation standard entre le programmeur de l'application et le matériel en question. L'utilisateur peut coder les tâches de son application en utilisant ces APIs sans se soucier des détails de l'architecture (mémoire, SE, CPU, …). Les APIs cachent les détails d'implémentation des protocoles de communication et de transfert de données. Elles définissent les services de communication de haut niveau permettant l'accès à la mémoire globale. Ces services correspondent aux méthodes de transfert de données et de contrôle spécifiques aux protocoles de communication utilisés par l'application. Dans l'exemple de la Figure 54, les tâches de l'application (T1 et T2) utilisent une mémoire globale partagée (shm : shared memory) comme un protocole de communication. La communication entre les ports logiciels[13] de ces tâches (P1 et P2) et le reste de l'architecture (SE, HAL, …) est assurée par des APIs d'accès mémoire (ligne 3 : *Port_shm_external*, *Get_shm_external*).

[12] Logiciel de l'application : Il s'agit du code de l'application (code des tâches) plus d'autres entités logicielles comme le système d'exploitation et autres modules logiciels.

[13] On définit un port logiciel comme une instance d'un pilote logiciel qui correspond à une classe C++.

2.2.2. Services dépendants du logiciel de l'application

Cette partie implémente les services de contrôle et de synchronisation nécessaires à la réalisation des APIs de haut niveau. Ces services sont dépendants du logiciel de l'application car ils peuvent utiliser des fonctions fournies par le système d'exploitation ou par une autre entité logicielle. Généralement, dans les applications multi-tâches, on ne peut pas contrôler l'accès au matériel par un simple logiciel. Lorsque les tâches s'exécutent en parallèle, le pilote nécessite des services de gestion d'accès parallèles et de gestion d'interruptions fournis par le système d'exploitation.

Les services de cette couche implémentent des fonctions d'écriture et de lecture permettant d'accéder aux éléments de mémorisation internes au pilote.

2.2.3. Eléments de mémorisation

Cette couche représente la dernière frontière entre le logiciel indépendant du matériel et celui qui est dépendant. Elle permet la séparation totale entre ces deux parties. En effet, le logiciel indépendant du matériel écrit les données dans les éléments de mémorisation où le logiciel dépendant du matériel vient lire ces données.

Les éléments de mémorisation peuvent être des FIFOs, un tampon mémoire ou de simples registres. Leur taille est spécifique à l'application. Actuellement, nous utilisons de simples registres pour stocker les adresses et les données envoyées pendant une opération d'écriture ou de lecture dans une mémoire globale.

2.2.4. Logiciel dépendant du matériel

Cette partie implémente les fonctions d'écriture et de lecture spécifiques aux processeurs. Ces fonctions sont liées aux protocoles de transfert de données et de gestion d'interruptions supportés par le processeur. Lors d'une opération d'écriture dans la mémoire globale, cette couche récupère l'adresse et la donnée à partir des registres du pilote et elle les adapte[14] avant de les écrire dans les ports matériels du processeur cible. L'implémentation de cette couche dépend de plusieurs paramètres spécifiques à l'architecture :

- L'adresse de base des accès mémoire externes (EMBA) : elle correspond à l'adresse début de la plage mémoire virtuelle du processeur réservée aux accès mémoires externes. Dans le cas d'un accès externe, le pilote convertit les adresses codées par le programmeur de l'application pour qu'elles puissent être vues par le processeur comme des adresses externes.
- La transposition des ports d'entrée/sortie du processeur dans la mémoire. Dans le cas des interfaces de type " memory-mapped interface ", le pilote doit écrire la donnée dans l'adresse qui correspond au port de données du processeur.

[14] L'adaptation consiste à convertir les adresses logiques en adresses physiques, découper la donnée selon la taille du bus de données du processeur.

- Les types de données supportés par le compilateur du processeur cible. Dans le cas où le type de données utilisé par l'application n'est pas adapté au processeur, le pilote utilise des fonctions de conversion de type.
- La taille du bus d'adresses et de données du processeur. A cause du haut niveau d'abstraction, le programmeur code son application sans tenir compte de l'architecture. Les données de l'application correspondent généralement à des paquets dont la taille est largement supérieure à celle des bus du processeur. Le pilote doit alors découper ces données en petites trames selon la taille du bus du processeur.

Vu l'existence d'autres éléments de dépendance entre l'architecture et le logiciel de cette couche, l'écriture manuelle de ce dernier s'avère très compliquée et très lente. Pour rendre ce code réutilisable et facile à écrire, nous utilisons une bibliothèque logicielle générique implémentant plusieurs macro-modèles de fonction d'écriture et de lecture. Ces fonctions sont configurées par les paramètres spécifiques à l'architecture pendant la génération du pilote.

2.3. Exemple d'un pilote d'accès mémoire

Dans l'exemple de la Figure 54, deux tâches T1 et T2 accèdent à une mémoire globale en lecture et en écriture en utilisant des primitives d'accès de haut niveau. L'implémentation de ces primitives correspond au code d'un pilote d'accès à la mémoire.

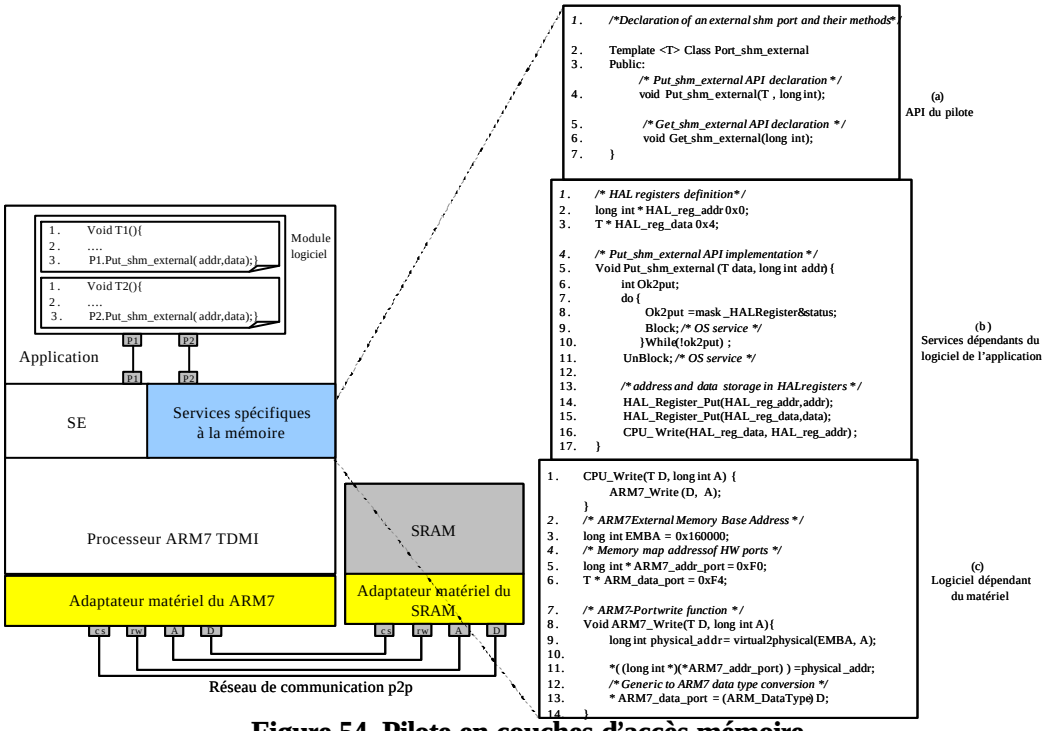


Figure 54. Pilote en couches d'accès mémoire

L'architecture matérielle est composée d'une mémoire globale SRAM, d'un processeur ARM7 TDMI et d'un réseau de communication point à point. L'implémentation de l'architecture logicielle correspond au code C des différentes couches du pilote.

API : comme le montre la partie (a) de la Figure 54, un port logiciel de communication de type mémoire partagée externe est déclaré comme une classe C++. Les méthodes de cette classe correspondent à deux APIs d'écriture et de lecture (ligne 4: *Put_shm_external*, ligne 6: *Get_shm_external*). On constate que le type de données est générique (ligne 2).

Service dépendant du logiciel de l'application : cette partie du code (Figure 54 (b)) correspond à l'implémentation indépendante du matériel des APIs d'écriture et de lecture. L'implémentation de l'API d'écriture par exemple utilise un service *HAL_Register_Put* (lignes 14 et 15) pour écrire l'adresse et la donnée dans les registres de la couche de mémorisation. L'écriture dans ces registres est gardée par des services de synchronisation (ligne 9: *Block*, ligne 11 : *UnBlock*) fournis par le SE. Pour que le logiciel dépendant du matériel puisse consommer les valeurs de ces registres, il faut qu'il soit activé par un appel de fonction (ligne 16 : CPU_Write).

Logiciel dépendant du matériel : c'est la partie du code qui correspond à la fonction d'écriture spécifique à un processeur donné. Pour un processeur ARM7 (Figure 54 (c)), la fonction *ARM7_Write* convertit l'adresse d'écriture de l'application en une adresse d'accès externe (ligne 9). La nouvelle adresse est écrite dans le port d'adresses spécifique au processeur (ligne 11). Le type de données de l'application est adapté à celui du processeur ARM7 avant l'écriture de la donnée dans le port du processeur correspondant (ligne 13).

3. Flot de génération systématique des pilotes logiciels

Le flot que nous utilisons a été utilisé initialement pour la génération des systèmes d'exploitation [Gau01]. Nous l'avons étendu pour la génération des pilotes logiciels en couches spécifiques à la mémoire.

3.1. Entrée du flot

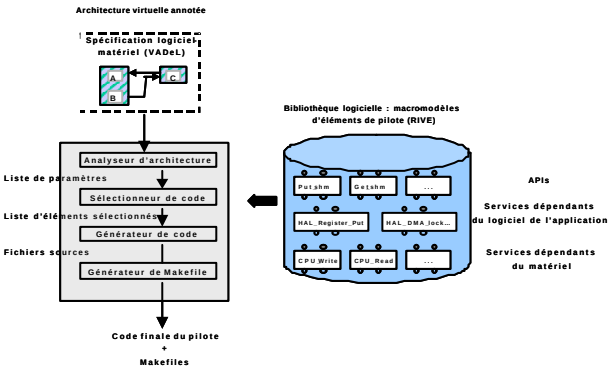


Figure 55. Flot de génération de pilote logiciel

Comme dans le cas de génération d'adaptateurs mémoire matériels, le flot de génération de pilotes logiciels (Figure 55) prend en entrée une architecture virtuelle qui décrit l'architecture logiciel-matériel abstraite de l'application (en VADeL). Cette architecture est annotée avec des paramètres de configuration spécifiques à l'application et à l'architecture (CPU, interface logiciel-matériel). Ces paramètres sont utilisés pour sélectionner et configurer les éléments spécifiques aux trois couches du pilote qui sont nécessaires au fonctionnement de l'application.

Paramètres spécifiques à l'application

Ces paramètres sont liés au logiciel de l'application (SE + code des tâches). Ils caractérisent les services de communication fournis ou requis par le pilote ainsi que le type de données manipulé par l'application. Ils servent essentiellement à sélectionner et à configurer les éléments de la deuxième couche du pilote et à fixer le type et la taille des éléments de mémorisation du pilote.

Paramètres spécifiques au matériel

Ces paramètres caractérisent la structure d'informations liées au périphérique en question et le processeur cible. Ces paramètres sont utilisés pour sélectionner les éléments du pilote qui dépendent du matériel.

3.2. Bibliothèque du flot

La génération des pilotes utilise une bibliothèque logicielle contenant des macro-modèles d'éléments de chaque couche du pilote. Ces éléments génériques sont configurés par les paramètres d'annotation pendant la génération du code du pilote. Chaque élément peut fournir/requérir un service à/de un autre élément. Nous utilisons le langage Lidel pour décrire cette relation de dépendance entre les services d'un pilote. Le code des macro-modèles de cette bibliothèque est écrit en langage RIVE, le langage cible étant du C.

```
1.   /* CPU-Port write function */
2.   Void @{CPUName}@_Write (T D, @{C_data_type}@ A){
3.        …….
4.        /* Address conversion */
5.        @{C_data_type}@ physical_addr = virtual2physical( @{EMBA}@, A);

6.        /* Address CPU port write */
7.        *( (@{C_data_type}@ *)(*CPU_addr_port) ) = physical_addr;

8.        /* Generic to CPU data type conversion and data CPU port write */
9.        * @{CPU_data_port}@  = (@{C_data_type}@) D;
10.       …….
11.  }
```

(a) Code macro en RIVE

```
1.   /* CPU-Port write function */
2.   Void ARM7_Write (T D, long int A){
3.        …….
4.        /* Address conversion */
5.        long int physical_addr = virtual2physical( 0x160000, A);

6.        /* Address CPU port write */
7.        *( (long int *)(*ARM7_addr_port) ) = physical_addr;

8.        /* Generic to CPU data type conversion and data CPU port write */
9.        * ARM7_data_port  = (long int) D;
10.       …….
11.  }
```

(b) Code C généré

```
1.   DEFINE CPUName = "ARM7" ENDDEFINE
2.   DEFINE C_data_type = "long int" ENDDEFINE
3.   DEFINE EMBA = 0x160000 ENDDEFINE
4.   DEFINE CPU_addr_port = "ARM7_addr_port" ENDDEFINE
5.   DEFINE CPU_data_port = "ARM7_data_port" ENDDEFINE
```

(c) Paramètres de configuration

Figure 56. Exemple d'un élément macro du pilote écrit en RIVE

La Figure 56 montre une partie du code macro de la fonction d'écriture spécifique au processeur, contenu dans la bibliothèque (a), les paramètres de configuration utilisés (c) et le code généré (b).

3.3. Sorties du flot

Les sorties du flot correspondent au code final (C) du pilote, après l'expansion des macro-modèles sélectionnés et au fichier de compilation (Makefile).

3.4. Etapes du flot

Le flot de génération des pilotes logiciels est composé de quatre étapes essentielles :

- Une analyse de l'architecture permet de parcourir le fichier XML de l'architecture virtuelle annotée pour extraire les paramètres de configuration. La sortie de cette étape est une liste de paramètres.
- Une étape de sélection consiste à utiliser les paramètres de la liste précédente pour choisir les éléments de bibliothèque spécifiques à l'application et à l'architecture. La sortie de cette étape est une liste de noms de fichiers de macro-modèles.
- La génération du code correspond à l'expansion des macro-éléments sélectionnés à partir de la bibliothèque. L'expansion du code RIVE utilise aussi la liste des paramètres de configuration extraite à la première étape. La sortie de cette étape est composée de plusieurs fichiers représentant le code final produit par l'expansion de chaque macro-élément.
- La génération du Makefile produit un fichier de compilation contenant tous les noms des fichiers sources ainsi que les règles de compilation et d'édition de liens spécifiques au processeur cible.

3.5. Génération automatique

Nous avons étendu l'environnement d'un outil de génération de système d'exploitation [Gau01] pour l'adapter à la génération des pilotes logiciels. Cet environnement est lié au langage de la spécification d'entrée, au traducteur de cette spécification en format intermédiaire Colif et au langage de représentation de dépendance entre les éléments de bibliothèque. Pour générer un nouveau type de pilote, l'ajout de son code dans la bibliothèque est loin d'être suffisant. Il est donc nécessaire de modifier l'environnement de génération pour que l'outil accepte le nouveau type de pilote.

On distingue trois modifications essentielles liées au langage de spécification VADeL, au traducteur Colif et au langage de description des relations de dépendance Lidel. Les détails d'implémentation de ces modifications sont dans l'annexe.

Extension de VADeL

L'utilisation d'un nouveau type de matériel dans le système nécessite la spécification d'un nouveau logiciel qui contrôle son accès. Cette spécification requiert la définition d'un nouveau type de port et d'un nouveau type de canal de communication dans le langage VADeL. Pour obtenir la nouvelle bibliothèque de VADeL

mise à jour, l'ajout des fichiers d'entête "headerfiles" définissant chaque nouveau port et chaque nouveau canal est nécessaire.

Extension du traducteur Colif

Le traducteur traduit chaque port et chaque canal de la spécification d'entrée (VADeL) en ports et "nets" dans le format intermédiaire Colif en se basant sur des API. Si le type de port ou le type de canal à traduire n'est pas reconnu par les API, la traduction échouera. Pour cette raison, l'ajout d'un nouveau type de port ou d'un nouveau type de canal doit être suivi systématiquement par leur définition au format Colif.

Extension des représentations Lidel

La bibliothèque de pilotes est composée d'éléments et de services. Chaque élément peut fournir des services à d'autres éléments et peut requérir d'autres services fournis par d'autres éléments. La relation de dépendance entre ces éléments est représentée par le langage Lidel. Pour cette raison, l'ajout d'un nouvel élément de pilote à la bibliothèque nécessite la définition de ces relations avec les autres éléments de la bibliothèque déjà existants. La description Lidel doit définir les services fournis et requis par l'élément ajouté, les paramètres d'appel de chaque méthode du pilote et les liens vers les sources d'implémentation.

Actuellement, les trois modifications sont effectuées à la main. Leur automatisation fait partie de nos travaux futurs.

4. Application

4.1. Description de l'application

L'application consiste à transférer des images entre un processeur et une mémoire globale en utilisant un contrôleur DMA (DMAC). La fonctionnalité de cette application correspond à deux tâches d'écriture et de lecture d'une image dans la mémoire. Chaque tâche traite une moitié de l'image.

Le but de cette application est de réaliser un pilote logiciel en couches d'un contrôleur DMA et de valider la portabilité sur différents processeurs et la spécificité du pilote à l'application.

4.2. Architecture cible

Nous ciblons cette application sur une architecture composée de (Figure 57) :

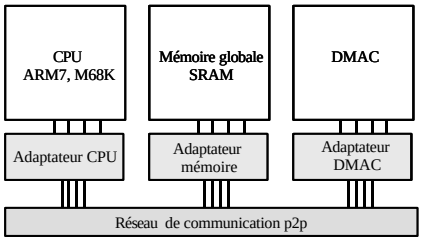


Figure 57. Architecture cible de l'application

- Un contrôleur DMA à deux canaux (IP). Les deux canaux permettent au contrôleur de transférer deux blocks de données en parallèle.
- Une mémoire globale SRAM double ports pour stocker l'image. Les deux ports permettent les accès concurrents du DMAC.
- Un seul processeur. La méthode de génération de pilotes reste également applicable dans un contexte multiprocesseur. Pour prouver la portabilité du pilote de DMAC, on utilise deux types de processeur. Dans une première expérience, on utilise un processeur ARM7TDMI. Dans une deuxième expérience, on utilise un processeur Motorola M68k.
- Un réseau de communication point à point qui relie le processeur, la mémoire et le contrôleur DMA.

4.3. Spécification

La spécification de l'application correspond à l'architecture virtuelle décrite dans la Figure 58. Elle est constituée de trois modules virtuels : un module de calcul (VM1), un module DMAC (VM2) et un module de mémorisation (VM3). Tous ces modules communiquent par l'intermédiaire de deux canaux virtuels hiérarchiques (VC1 et VC2). Ces canaux abstraits relient les ports logiciels du module M1 et les ports matériels du DMAC. Nous visons à générer le code du pilote logiciel qui adapte la communication entre ces deux types de port. Le pilote doit fournir les services de communication requis entre les ports logiciels du module M1 et les ports matériels du module M3.

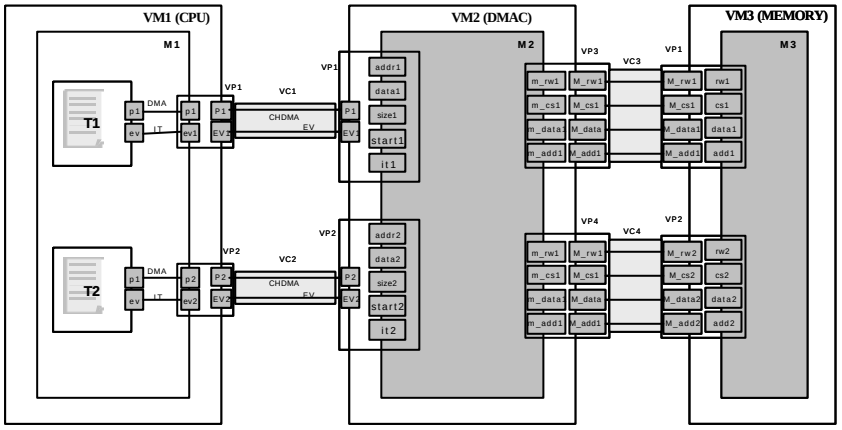


Figure 58. Architecture virtuelle de l'application

L'entrée du flot de génération des pilotes logiciels correspond à la description VADeL décrivant la spécification de cette application. Cette description est annotée par des paramètres de configuration qui seront utilisés pour sélectionner et spécialiser les macro-modèles de base nécessaires au pilote.

VM1 a deux ports virtuels symétriques VP1 et VP2. VP1 est composé de deux ports externes (*P1* et *EV1*) spécifiques au canal de communication VC1. *P1* est utilisé pour le transfert de données tandis que *EV1* est utilisé pour recevoir les interruptions du DMAC. VP1 est composé également de deux ports internes (*p1* et *ev1*) spécifiques au comportement du module M1. Ces ports correspondent aux ports logiciels de la tâche T1 (*p1* et *ev*). Ils sont utilisés par la tâche T1 respectivement pour les données et les interruptions.

```
1.    Void T1() {
2.    ….
3.    Buffer = p1.Get_dma(addr,size);
4.    ….
5.    p1.Put_dma(addr,data,size);
6.    ….
```

Figure 59. Appels des APIs d'accès mémoire par les tâches de l'application

La Figure 59 montre une section de code de la tâche T1. Cette tâche emploie deux APIs *Get_dma* (ligne 3) et *Put_dma* (ligne 5) pour transférer les données mémoire via le DMAC.

La Figure 60 montre une partie du programme principal VADeL de l'application. À partir de la ligne 5 jusqu'à la ligne 11, les paramètres spécifiques au contrôleur DMA sont attribués au module VM2. Ces paramètres correspondent aux chemins des fichiers sources, au type du module, au niveau d'abstraction et à l'adresse d'interruptions.

Entre la ligne 13 et 17, nous indiquons les paramètres liés au module logiciel VM1. La ligne 15 indique que le module VM1 correspond à un processeur ARM7. La ligne 17 prouve que le module VM1 est décrit à un niveau abstrait appelé architecture virtuelle ou DL.

Entre la ligne 19 et 36, nous avons annoté la spécification avec des paramètres spécifiques à l'application. Puisque l'application manipule des données de type *long int*, la largeur du port logiciel *p1* est de 32 bits. Ce paramètre est utilisé pour fixer la taille des registres du pilote. D'autres paramètres liés aux interruptions sont également donnés.

Le terme *DmaRegister* (ligne 20) correspond au type du port logiciel. Il indique que le port *p1* correspond à un port de DMAC transposé dans la mémoire comme un registre. Cette information indique aussi que le port *p1* nécessite des mécanismes de synchronisation pour accéder au port de DMAC.

Le terme *Interrupt* (ligne 34) indique que le port *ev2* du module VM1 doit gérer les interruptions du DMAC. Cette information est utilisée pour sélectionner la routine de gestion d'interruptions. Nous supposons qu'elle est fournie par le SE.

À partir de la ligne 38, nous décrivons les APIs de communication nécessaires à chaque tâche. Par exemple, le port *p1* de la tâche T1 nécessite les APIs *Put_dma* et *Get_dma* (ligne 39). Ces APIs sont

implémentées par le pilote. Le port *ev* de la tâche T1 qui est lié au port d'interruptions *ev1* du module VM1, nécessite les services *Sleep* et *WakeUp* (ligne 40) pour détecter la fin d'une opération de lecture. Ces services sont fournis par le SE.

```
1.  int sc_main(int argc, char* argv[])
2.  {
3.  …...
4.
5.      /* Hardware DMA controller parameters */
6.      VDMA.DMA->setColifParam("Source","SystemC","DMA/DMA.cpp","DMA/DMA.h");
7.      VDMA.DMA->setColifParam("Source","SystemC","DMA/DMA.cpp","DMA/DMA.h");
8.      VDMA.DMA->setColifParam("module_type", "blackbox");
9.      VDMA.BlackBox=true;
10.     VDMA.level="RTL";
11.     VDMA.ADDRESS_IT="0x000FF008";
12.
13.     /* Software module (M1) parameters */
14.     /* parameter related to module M1*/
15.     VM1.CPUName="CPUARM7";
16.     VM1.level="DL";
17.     VM1.ADDRESS_IT="0x000FF000";
18.
19.   /* application-specific parameter related to M1 ports (p1,  p2, ev1 and ev2) */
20.     VM1.M1->p1.SoftPortType="DmaRegister";
21.     VM1.M1->p1.DATA_BIT_WIDTH="32";
22.     VM1.M1->p1.ADDRESS_DATA="0x000F0000";
23.     VM1.M1->p1.ADDRESS_STATE="0x000F0004";
24.     VM1.M1->p1.IT_NUMBER="1";
25.     VM1.M1->p1.IT_LEVEL="1";
26.     VM1.M1->p1.CHAN_PRIO="0";
27.     VM1.M1->p1.C_DATA_TYPE="long int";
28.     VM1.M1->p1.MASK_GET="2";
29.     VM1.M1->p1.MASK_PUT="1";
30.
31.     VM1.M1->p2.SoftPortType="DmaRegister";
32.     …
33.     …
34.     VM1.M1->ev2.SoftPortType="Interrupt";
35.     VM1.M1->ev2.IT_NUMBER="4";
36.     VM1.M1->ev2.IT_LEVEL="1";
37.
38.   /* application-specific parameter related to the driver services for each task  */
39.     VM1.M1->T1->p1.SoftService=“HAL/Dma_Put; HAL/Dma_Get";
40.     VM1.M1->T1->ev.SoftService ="Kernel/Signal/Sleep;Kernel/Signal/WakeUp";
41.     VM1.M1->T2->p1.SoftService=“HAL/Dma_Put; HAL/Dma_Get";
42.     VM1.M1->T2->ev.SoftService ="Kernel/Signal/Sleep;Kernel/Signal/WakeUp";
43.
44.  sc_start(0);

45.     return EXIT_SUCCESS;
46.  };
```

Figure 60. Un extrait de la spécification de l'application

4.4. Génération automatique des pilotes en couches

Les étapes de la génération des pilotes d'accès mémoire spécifiques à cette application sont les suivantes :

- **Analyse de l'architecture virtuelle :** les paramètres extraits correspondent à ceux qui ont été décrits dans la section précédente.
- **Sélection du code :** selon les paramètres sélectionnés, seules les fonctions nécessaires aux pilotes sont choisies. Le Tableau 8 montre des fonctions sélectionnées comme *Mask_HAL_Register*, *HAL_Register_Put*, *HAL_Register_Get* et *CPU_Write*.
- **Génération du code :** le résultat de cette étape est le code source des fonctions dépendantes du processeur ARM7. Comme le montre le Tableau 8, parmi ces fonctions, nous avons cité la fonction de conversion d'adresses (*ARM7_virtual2physical*) et les fonctions d'écriture et de lecture (*ARM7_Write* et *ARM7_Read*).
- **Génération de Makefile :** le fichier de compilation inclut les règles de compilation et d'édition de lien spécifiques au processeur ARM7.

4.5. Analyse des résultats

4.5.1. Portabilité

Pour prouver la portabilité du pilote DMAC sur différents processeurs, nous avons utilisé deux types de processeurs dans deux expériences.

Les différences essentielles entre les deux expériences et la précédente correspondent à la génération du code et du Makefile. En effet, la partie dépendante du processeur correspond aux fonctions d'écriture et de lecture spécifiques au processeur M68K (M68K_*Write* et M68K_*Read*) et aux fonctions de conversion d'adresses (*M68K_virtual2physical*).

Eléments du pilote	*Composition*		*Taille du code (lines)*
API	Put_dma Get_dma,		4 4
Services dépendants de l'application	Mask_HALRegister HAL_Register_Put HAL_Register_Get CPU_write		71 113 63 82
Services dépendants du processeur	ARM7	M68K	Environ 220 lignes pour chaque CPU
	ARM7_virtual2physical ARM7_WRITE ARM7_Read …	M68K_virtual2physical M68K_Write M68K_Read …	

Tableau 8. Résultats de la génération du pilote DMA

4.5.2. Spécificité du pilote à l'application

Comme l'indique le Tableau 8, les tailles du code des fonctions du pilote sont petites. Ceci vient du fait qu'on n'a utilisé que les fonctions nécessaires à l'application.

4.5.3. Automatisation

La génération de ce pilote est semi-automatique. En effet, il a fallu définir à la main des nouveaux ports et canaux de type *dma* dans le code source de VADeL et nous avons mis à jour le traducteur Colif et la description de dépendance de la bibliothèque du pilote. La partie automatisée est liée à l'extraction des paramètres, à la sélection du code, à la génération du code et aux Makefiles. Ceci ne prend que quelques secondes. L'automatisation de la partie faite manuellement correspond aux travaux en cours.

5. Conclusion

La conception des pilotes logiciels figure parmi les tâches les plus difficiles de la conception des interfaces logiciel-matériel. En effet, ces pilotes sont fortement dépendants du matériel sur lequel ils sont exécutés. Ceci ne fait que réduire la portabilité et la réutilisation de ces pilotes. Pour résoudre ces problèmes, nous avons proposé une architecture de pilotes en couches qui permet de séparer le code dépendant de celui qui est indépendant du matériel. La génération de ces pilotes est actuellement semi-automatique. L'approche de génération est basée sur l'assemblage de composants logiciels de base à partir d'une bibliothèque générique. Nous obtenons le code du pilote après des mises à jour manuelles de l'environnement de génération. Une application d'un pilote de DMA nous a permis de comprendre les différentes mises à jour à faire et nous a donné l'occasion de valider le code généré ainsi que sa portabilité.

Chapitre 7 : CONCLUSION ET PERSPECTIVES

1. Conclusion

La plupart des systèmes monopuces spécifiques à des applications modernes telles que le traitement d'images et les jeux, requièrent une grande capacité de mémoire. Ces dernières occuperont plus de 95% des surfaces des puces à l'horizon 2014 selon des prévisions récentes (ITRS).

La mise en œuvre de tels systèmes consiste à concevoir des architectures multiprocesseurs intégrant plusieurs mémoires de différents types sur la même puce. Cette conception demande des efforts considérables et prend un temps qui n'est plus acceptable vu la pression exercée par le marché. Une façon de réduire ces efforts et de répondre aux contraintes du temps de mise sur le marché (" *time to market* ") est la réutilisation des composants. La conception à base de réutilisation d'IP mémoire permet de réduire le fossé entre l'importante capacité d'intégration en terme de transistors par unité de surface et la faible production en terme de transistors produits par unité de temps. Cette solution peut être idéale dans le cas d'une architecture homogène où tous les éléments ont les mêmes interfaces et utilisent les mêmes protocoles de communication, ce qui n'est pas le cas pour les systèmes monopuces qui sont par nature hétérogènes. Pour rendre cette solution efficace dans le cas des systèmes hétérogènes, le concepteur doit consacrer beaucoup d'efforts pour la spécification et l'implémentation d'interfaces logiciel-matériel. Ces interfaces doivent assurer d'une part, l'adaptation des ports physiques de la mémoire aux ports logiques des réseaux de communication et d'autre part, l'adaptation du code logiciel des accès mémoire de haut niveau à l'architecture matérielle cible.

Vu la diversité du matériel et la forte dépendance du logiciel au matériel, la spécification et la réalisation des adaptateurs logiciels et matériels sont devenues un vrai problème de conception. En effet, cela nécessite une connaissance multidisciplinaire couvrant le domaine logiciel et matériel.

La solution apportée par cette thèse à ce problème consiste à abstraire ces interfaces logiciel-matériel pour réduire la complexité de leur spécification et faciliter leur génération automatique. Nous avons proposé des architectures d'adaptateurs matériels et d'adaptateurs logiciels (pilote) génériques et flexibles permettant leur réutilisation. Nous avons présenté également une méthode de génération automatique de ces adaptateurs et une méthode de génération de programmes de tests.

Le chapitre 1 de ce manuscrit a introduit la conception des systèmes multiprocesseurs monopuces avec des mémoires globales. Nous avons ainsi présenté la problématique et les difficultés de conception des interfaces logiciel-matériel pour l'intégration des mémoires globales. Ces difficultés se résument en quatre points :

1. L'insuffisance du niveau RTL pour la conception de ces interfaces.

2. La variété des interfaces matérielles des mémoires et le manque de standard.
3. La forte dépendance des pilotes logiciels aux architectures matérielles.
4. Absence d'outil de validation d'interfaces de communication.

Pour dépasser ces difficultés, les contributions de cette thèse étaient :

1. La proposition de modèles mémoire de haut niveau.
2. L'abstraction de la communication pour générer automatiquement les adaptateurs mémoire matériels.
3. La définition d'une couche d'abstraction du matériel (HAL) permettant la réduction de la dépendance entre le logiciel et le matériel et permettant la génération systématique des pilotes en couches.
4. La génération de programmes de tests pour la validation des adaptateurs mémoire matériels.

Le chapitre 2 a présenté une étude pragmatique et synthétique sur la conception des systèmes monopuces avec des mémoires. Dans une première partie, nous avons introduit le flot de conception système avec mémoire. Dans une deuxième partie de ce chapitre, une étude a été faite sur les travaux antérieurs spécifiques à la mémoire à chaque étape du flot de conception. Cette étude a montré que peu de travaux ont abordé les problèmes d'adaptation matérielle et logicielle des mémoires et qu'il reste encore du travail pour les générations futures des systèmes électroniques.

Le chapitre 3 est lié à l'abstraction des interfaces logiciel-matériel pour les décrire à un niveau d'abstraction plus élevé que le RTL. Nous avons présenté de nouveaux concepts sur lesquels notre méthode de spécification d'interfaces logiciel-matériel a été basée. Le concept central est la séparation entre les interfaces de communication et celles du comportement. Pour ce faire, le modèle d'architecture virtuelle exposé propose d'envelopper chaque composant mémoire en séparant les ports physiques des ports logiques. Ainsi, les systèmes monopuces avec des mémoires globales sont représentés comme un ensemble de composants virtuels interconnectés via des canaux abstraits (canaux virtuels). La spécification de ces interfaces logiciel-matériel est décrite par le langage Colif. Pour l'implémentation, nous avons été amené à étendre l'environnement du flot SLS pour générer des interfaces spécifiques à la mémoire. Nous avons également défini des modèles mémoire spécifiques à chaque niveau d'abstraction du flot.

Dans **le chapitre 4**, nous avons défini une architecture flexible d'adaptateurs mémoire matériels. Celle-ci correspond à l'architecture de réalisation de l'enveloppe d'un module mémoire virtuel. Elle est composée de trois éléments : un adaptateur de canal (CA) qui implémente le port externe d'un port virtuel, un adaptateur de port mémoire (MPA) qui implémente le port interne du port virtuel de la mémoire et un bus interne qui assure le transfert de contrôle, d'adresses et de données entre le CA et le MPA. Cette structuration modulaire facilite la réutilisation de ces adaptateurs mémoire et par conséquent, elle permet d'accélérer l'intégration des mémoire. Les CAs sont spécifiques aux protocoles de communication, ils ne dépendent pas de l'interface de la mémoire. Les MPAs, quant à eux, dépendent de l'interface de la mémoire et ils sont complètement indépendants de celle du média de communication externe. Le rôle, la fonctionnalité et l'implémentation de chacun de ces éléments ont été détaillés à la fin de ce chapitre.

Le chapitre 5 a été consacré à la génération automatique des adaptateurs mémoire matériels. Cette génération est basée sur l'assemblage de macro-modèles de CA et de MPA à partir d'une bibliothèque matérielle générique. Nous avons adapté l'outil de génération d'interfaces utilisé par le groupe SLS aux mémoires globales. Pour cela, nous avons développé un nouvel outil d'aide permettant à l'utilisateur de l'outil de génération d'interfaces de réaliser les adaptateurs matériels de ces modules mémoire indépendamment du langage intermédiaire Colif. Les extensions apportées à l'environnement de génération des interfaces ont été :

- Réalisation d'une bibliothèque de mémoires génériques implémentant la fonctionnalité de plusieurs modèles mémoire. L'usager peut utiliser son modèle mémoire s'il est indisponible dans cette bibliothèque.
- Réalisation d'un générateur d'architectures internes en Colif permettant à l'utilisateur de spécifier ses propres modules indépendamment de l'environnement du flot.
- Génération du code des éléments d'adaptation. Cela a nécessité la réalisation d'une bibliothèque d'adaptateurs mémoire matériels implémentant la fonctionnalité des adaptateurs de canaux (CA) et des adaptateurs de ports mémoire (MPA). Contrairement à la bibliothèque utilisée pour la génération des adaptateurs de processeurs, cette bibliothèque implémente de nouveaux macro-modèles d'adaptateurs de canaux qui autorisent l'accès à des mémoires globales.
- Validation locale des éléments de base d'adaptateurs mémoire (CA et MPA) par la génération automatique de programmes de tests.

La validation du comportement global des adaptateurs matériels mémoire a été faite par une co-simulation basée sur SystemC. L'application utilisée est liée au traitement d'images. L'architecture spécifique à cette application est composée de deux processeurs ARM7, d'une mémoire globale et d'un réseau de communication point à point. Pour prouver la flexibilité des adaptateurs matériels de la mémoire, nous avons abordé deux expériences : la première utilise une mémoire SRAM double ports et la deuxième utilise une mémoire SDRAM simple port. Les résultats constatés après ces expériences montrent la flexibilité d'utilisation, la faible complexité (environ 2000 portes par adaptateur) et le faible surcoût de communication (2 à 3 cycles par rapport à un accès mémoire externe complet).

Le chapitre 6 a été consacré à la spécification et à la réalisation des adaptateurs logiciels mémoire qui correspondent à des pilotes d'accès implémentant la partie logicielle des interfaces logiciel-matériel. L'abstraction de ces adaptateurs est basée sur une couche d'abstraction du matériel (HAL) qui permet au programmeur de l'application d'être complètement indépendant de l'architecture cible. L'implémentation du HAL correspond à des pilotes logiciels en couches. L'architecture de ces pilotes est composée de quatre parties :

- Des API d'accès mémoire de haut niveau.
- Des services de communication dépendants du logiciel de l'application.
- Des éléments de mémorisation internes au pilote.

- Des fonctions d'écriture et de lecture qui dépendent du matériel.

Cette structuration en couche permet la conception concurrente et facilite la réutilisation du logiciel ainsi que sa portabilité. Nous avons proposé une méthode de génération de pilotes basée sur l'assemblage de macro-éléments logiciels à partir d'une bibliothèque générique. L'ajout d'un nouveau type de pilote dans ces bibliothèques nécessite encore des travaux manuels. L'étude de ces pilotes nous a permis de comprendre les différentes étapes dans le flot de conception globale nécessaires à l'ajout d'un nouveau type de pilote. La génération automatique de code et la portabilité des pilotes ont été validées par une application d'accès mémoire via un contrôleur DMA.

2. Perspectives

Vu que les systèmes monopuces contiennent des composants autres que les mémoires et les processeurs tels que les IPs, nous pensons à l'extension de la génération des interfaces logiciel-matériel à d'autres composants est nécessaire. Comme le principe de notre approche est basé sur des bibliothèques, nous estimons que la généralisation est possible, mais elle nécessite un grand effort de développement de bibliothèques dû à la diversité du logiciel et du matériel des systèmes monopuces.

Par ailleurs, nous sommes fortement motivés par les travaux liés aux systèmes d'exploitation temps réel et à la mémoire. Nous pensons que c'est un axe de recherche qui nécessite beaucoup d'exploration. En effet, il y a peu d'équipes qui ont traité la conception de gestionnaire mémoire spécifique à une application donnée. L'exploration des compromis logiciel-matériel est aussi nécessaire pour pouvoir juger l'impact du logiciel et du matériel sur les performance des mémoires (taille, temps, consommation).

Dans cette thèse, les adaptateurs logiciels et matériels ont été traités séparément avec un découpage fixe entre les fonctionnalités réalisées par le pilote et les fonctionnalités réalisées par l'adaptateur mémoire matériel. Ces deux entités sont dépendantes. Il sera donc intéressant de lier la conception des interfaces logiciel-matériel. Des compromis d'implémentation logicielle et matérielle peuvent être également étudiés.

Avec l'évolution des applications multimédia de codage/décodage vidéo (tel que OpenDivx), nous pensons que la conception d'adaptateurs mémoire ou de serveurs mémoire de haut débit est un point clef pour la performance des systèmes monopuces.

Bibliographie

[Abr02] A. Abril Garcia, J. Gobert, T. Dombek, H. Mehrez, F. Pétrot, "Energy Estimations in High Level Cycle-Accurate Descriptions of Embedded Systems", The 5th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS'2002), Brno, Czech Republic, pp. 228-235, April 2002.

[Ahm91] I. Ahmad, and C.Y.R. Chen, "Post-processor for data path synthesis using multiport memories", In Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCD '91, santa Clara, CA, Nov. 11-14). IEEE Computer Society Press, Los Alamitos, CA, 276-279, 1991.

[Ama95] S. Amarasinghe, J. Anderson, M. Lam and C.-W. Tseng, "A Overview of the suif compiler for scalable parallel machines", In Processings of the SIAM Conference on Parallel Processing for Scientific Computing (San Francisco, CA, Feb.). SIAM, Philadelphia, PA, 1995.

[Arm99] MBA specification (REV 2.0) ARM Limited, 13 mai 1999. http://arm.com

[Bak95] S. Bakshi. and D.D. Gajski, "A memory selection algorithm for high-performance pipelines", In Proceedings of the European Conference EURO-DAC '95 with EURO-VHDL '95 on Design Automation, G. Musgrave, Ed. IEEE Computer Society Press, Los Alamitos, CA, 124-129, 1995.

[Bal88] M. Balakrishnan, Banerji and all, "Allocation of multiport memories in datapath synthesis", IEEE Trans . comput.-Aided Des. 7, 4 (Apr.), 536-540, 1988.

[Bal97] F. Balarin, M. Chido, P. Giusto, H. Hsieh, A, Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara, "Hardware-Software Co-design of embedded System: The Polis Approach". Kluwer Academic Press, 1997. (polis)

[Ban95] P. Banerjee, J. Chandy, M. Gupta, E. Hodges, J. Holm, A. Lain, D. Palermo, S. Ramaswamy and E. SU, "The paradigm compiler for distributed-memory multicomputers", IEEE Computer 28, 10 (oct.), 37-47, 1995.

[Bol97] I. Bolsens, H. J. De Man, B. Linn, K. Van Rompaey, S. Vercautere, and D. Verkest, “Hardware/Software co-design of digital telecommunication systems, Proceedings of the IEEE, Vol.85, No. 3, pp. 391-418, 1997.

[Bor98] G. Borriello, L. Lavagno, Ross B. Ortega, “Interface synthesis: a vertical slice from digital logic to software components” ICCAD, pp. 693-695, 1998.

[Bru00] J.Y. Brunel, W. Kruijtzer, H. Kenter , F. Pétrot, L. Pasquier, E. de Kock and W. Smits, “COSY Communication IP’s”, Proc. Of Design Automation Conference 2000, Los Angeles, june 2000.

[Cat01] F. Catthoor, N. Dutt, K. Danckaert, S. Wuytack, “Code Transformation for Data Transfer and Storage Exploration Preprocessing in Multimedia Processor”, IEEE Design and Test of Computers, Mai-Juin 2001.

[Cat02] F. Catthoor, “Data Access and Storage Management for Embedded Programmable Processors”, Kluwer Academic Publishers, 01/01/2002.

[Cat98] F. Catthoor, S. Wuytack, E. Greef, F. Balasa, L. Nachtergaele and A. Vandecappelle, “Custom Memory Management Methodology : Exploration of Memory Organization for Embedded Multimedia System Design”, Kluwer Academic, Dordrecht, Netherlands, 1998.

[Ces01] W.O. Cesario, G. Nicolescu, L. Geauthier, D. Lyonnard and A.A. Jerraya, "Colif: a Multilevel Design Representation for Application-Specific Multiprocessor System-on-Chip Design", 12th IEEE International Workshop on Rapid System Prototyping, June 25-27, 2001, California, USA.

[Cho99] P. Chou, Ross B. Ortega, K. Hines, K. Patridge, G. Borriello, “ipChinook: an Integrated IP-based Design Framework for Distributed Embedded Systems”, DAC, pp. 44-49, 1999.

[Cul99] D.E. Culler, J. Pal Singh, "Parallel Computer Architecture", Morgan Kaufmann Publishers, 1999.

[Dan00] K. Danckaert, F. Catthoor and H.D. Man, “A preprocessing step for global loop transformations for data transfer and storage optimization”, In Proceedings of the International Conference on compilers, Architecture and Synthesis for Embedded Systems (San Jose, CA, Nov.), 2000.

[Eco02] eCos, http://sources.redhat.com/ecos/, 2002.

[Ern94] R. Ernst and Th. Benner, “Communication, Constraints and User Directives in COSYMA”, Technical Report CY-94-2, Techniche Universitat Braunschweig, June 1994.

[Fly72] M. Flyn, “Some computer organizations and their effectiveness”, IEEE Transactions on Computers, (21):948-960, Sept. 1972.

[Fra01] A. Fraboulet, "Optimisation de la mémoire et de la consommation des systèmes multimédia embarqués ", Thèse : Institut National des sciences appliquées de Lyon, 2001.

[Fra01] A. Fraboulet, K. Kodray, A. Mignotte, “Loop Fusion for Memory Space Optimization”, Procedings of Internal Syposium on System Synthesis, Montreal, Canada, October 2001.

[Fra94] F. Franssen, L. Nachtergaele, H. Samson, F. Catthoor and Man, H. D., “Control flow optimization for fast system simulation and storage minimization”, In Proceedings of the International Conference on Design and Test (Paris, Feb.). 20-24, 1994.

[Gaj98] D. Gajski, F. Vahid, S. Narayan, and J. Gong, “SpecSyn: An Environment Supporting the Specify-Explore-Refine Paradigm for Hardware/Software System Design”, IEEE Transactions on VLSI Systems, Vol. 6, No. 1, March 1998.

[Gau01] L. GAUTHIER, “Génération de système d’exploitation pour le ciblage de logiciel multitâche sur des architectures multiprocesseurs hétérogènes dans le cadre des systèmes embarqués spécifiques”, Thèse de doctorat, INPG, laboratoire TIMA, 2001.

[Gha00] F. Gharsalli, “Conception mixte logicielle/matérielle des systèmes multiprocesseurs avec mémoire”, DEA informatique systèmes et communication, Université Joseph Fourier de Grenoble, laboratoire TIMA, 2000.

[Gha02a] F. Gharsalli, D. Lyonnard, S. Meftali, F. Rousseau, A. A. Jerraya, “Unifying memory and processor wrapper architecture for multiprocessor SoC design”, In proceedings of International Symposium on System Synthesis (ISSS'02), Kyoto, Japan, October 2-4, 19-24, 2002.

[Gha02b] F. Gharsalli, S. Meftali, F. Rousseau, A. A. Jerraya, "Automatic generation of embedded memory wrapper for multiprocessor SoC", In proceedings of the 39th Design Automation Conference (DAC'02), New Orleans, USA, June 10-14, 596-601, 2002.

[Gre95] E. D Greef, F. Catthoor, and H. D Man, "Memory organization for video algorithms on programmable signal processors", In Proceedings of the IEEE International Conference on computer Design (ICCD '95, Austin TX, Oct.). IEEE Computer Society Press, Los Alamitos, CA, 552-557, 1995.

[Gue00] P. Guerrier, A. Greiner, "A generic architecture for on-chip packet switched interconnections", In proceedings of Design Automation and Test in Europe, 2000.

[Har87] Harel, "Statecharts: A Visual Formalism for Complex Systems", Science of Computer Programming, 1987, 8, p. 231-274.

[Hen99] J. Hennessy, M. Heinrich, A. Gupta, "Cache-Coherent Distributed Share Memory: Perspectives on Its Development and Future Challenges", Special issue on distributed Shared-Memory Systems, March 1999.

[Hoa85] C.A.R. Hoare, Communicating Sequential Processes, 1985, Prentice Hall.

[I2O02] I2O SIG : http://www.i2osig.org/, 2002.

[Ibm02] IBM, "The CoreConnect Bus Architecture" available at: http://www.chips.ibm.com/products/coreconnect/, 2002.

[Iee93] Institute of Electrical and Electronically Engineers, IEEE Standard VHDL Language Reference Manual, 1993, STD 1076-1993. IEEE

[Ism96] T.B. Ismail, J. Daveau, K. O'Brien, and A. A. Jerraya, "A system-level communication synthesis approach for hardware/software systems", Microprocessors and Microsystems, 20(3):149-157, May 1996.

[Jen97] D.C.R. Jensen, J. Madsen, and S. Pedersen, "The importance of Interfaces: A HW/SW codesign case study", Preceedings of 5th International Workshop on Codesign, 1997.

[Jer02] A. A Jerraya., "De l'ASIC au SoC, puis au réseau de composants sur puce", veille technologique n°25, décembre/janvier 2002.

[Jha97] P. K. Jha and N. Dutt, “Library mapping for memories”, In Proceedings of the conference on European Design and Test (Mar.). 288-29, 1997

[Jungo] Windriver and Kernel drivers development tools, available at www.jungo.com

[Kar94] D. Karchmer and J. Rose, “Definition and solution of the memory packing problem for field-programmable systems. In Proceedings of the 1994 IEEE/ACM International Conference on Computer-Aided Design (ICCAD ‘94, San Jose, CA, Nov. 6-10), J. A. G. Jess and R. Rudell, Eds. IEEE Computer Society Press, Los Alamitos, CA, 20-26, 1994.

[Kat99] T. Katayama, K. Saisho, and A. Fukuda “Proposal of a Support System for Device Driver Generation”, Proc. Asia-Pacific Softw. Eng. Conf. (APSEC'99), pp.494-497, 1999.

[Kel92] W. Kelly and W. Pugh, “Generating schedules and code within a unified reordering transformation framework”, UMIACS-TR-92-126. University of Maryland at College Park, College Park, MD, 1992.

[Kim93] T. Kim, and C. L. Liu, “Utilization of multiport memories in data path synthesis”, In Proceedings of the 30th ACM/IEEE International Conference on Design Automation (DAC ’93, Dallas, TX, June 14-18), A. E. Dunlop, ed. ACM Press, New York, NY, 298-302, 1993.

[Knu98] P. V. Knudsen and J. Madsen “Integrating Communication Protocol Selection with Hardware/Software Codesign”, in ISSS, 1998. (lycos)

[Kus94] J. Kuskin et al, “The Stanford FLASH Multiprocessor”, In Proceedings of the 21st Int’l Symposium On Computer Architecture, 1994.

[Lov77] D. B. Loveman, “Program improvement by source-to-source transformation”, J. ACM 24, 121-145, 1977.

[Lyo01] D. Lyonnard, S. Yoo, A. Baghdadi, A. A. Jerraya, "Automatic Generation of Application-Specific Architectures for Heterogeneous Multiprocessor System-on-Chip", Proceedings DAC, Las Vegas, USA, 2001.

[Lyo03] D. Lyonnard, “Approche d'assemblage systématique d'éléments d'interface pour la génération d'architectures multiprocesseurs”, Thèse de doctorat, INPG, laboratoire TIMA, 2003.

[M4] GNU implementation of the UNIX macro processor.

[Mas99] K. Masselos, F. Catthoor, C. Goutis, H. D. and Man, “A performance oriented use methodology of power optimizing code transformations for multimedia applications realized on programmable multimedia processors”, In Proceedings of the IEEE Workshop on Signal Processing Systems (Taipeh, Taiwan). IEEE Computer Society Press, Los Alamitos, CA, 270-272, 1999.

[Mck98] K. S., Mckinley, “A compiler optimization algorithm for shared-memory multiprocessors”, IEEE Trans. Parallel Distrib. Syst. 9, 8, 769-787, 1998.

[Mef01a] S. Meftali, “Exploration d’architectures et allocation/affectation mémoire dans les systèmes multiprocesseurs monopuce”, Thèse de doctorat, Université Joseph Fourier de Grenoble, laboratoire TIMA, 2001.

[Mef01b] S. Meftali, F. Gharsalli, F. Rousseau et A.A. Jerraya, “An Optimal Memory Allocation for Application-Specific Multiprocessor System-on-Chip”, In Proc. Int’l Symposium on System Synthesis (ISSS), 2001.

[Moo98] R. Philip R. MOORBY, DONALD E. THOMAS, “The Verilog Hardware Description Language”, May 1998, Hardcover.

[Nic02] E. G. N. Nicolescu, “Spécification et validation des systèmes hétérogènes embarqués”, Thèse de doctorat, INPG, laboratoire TIMA, 2002.

[Nie98] R. Niemann, P. Marwedel: “Synthesis of Communicating Controllers for Concurrent Hardware/Software Systems”, Design, Automation and Test in Europe (DATE), 1998.

[Obe96] J. Öberg, A. Kumar, and Hemani, “Grammer based hardware synthesis of data communication protocols”, Proceedings of the 9th International Symposium on System Synthesis, pp. 14-19,1996.

[Obj00] Jectime, available on-line at http://www.objectime.on.ca/, 2000.

[Oni01] M. O'Nils, A. J. Jantsch “Device Driver and DMA Controller Synthesis from HW/SW Communication Protocol Specifications”, in Design Automation for Embedded Systems, Vol. 6, No. 2, Kluwer Academic Publisher, 2001.

[Oni98] M. O'Nils, J. Öberg and A. J. Jantsch "Grammar Based Modelling and synthesis of device drivers and bus interfaces", in Proc. of the EuroMicro Workshop on Digital System Design, 1998.

[OpeCo] OpenCore, Protocol International Partnership, available at http://www.ocpip.org/home.

[Pan00] P. R. Panda, N. D. Dutt and A. Nicolau, "On-chip vs. off-chip memory: The data portioning problem in embedded processor-based systems", ACM Trans. Des. Autom. Electron. Syst. 5,3 (july), 682-704, 2000

[Pan01] P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, P. G. Kjeldsberg, "Data and memory optimization techniques for embedded systems", TODAES 6(2): 149-206, 2001.

[Pan96] P. R. Panda, N. D. Dutt and A. Nicolau, "Memory organization for improved data cache performance in embedded processors", In Proceedings of the ACMIIEEE International Symposium on System Synthesis, ACM Press, New York, NY, 90-95, 1996.

[Pat98] D.A. Patterson , J.L. Hennessy, "Computer Organization and Design - The Hardware/Software Interface", Morgan Kaufmann Publishers, 1998.

[Phili] Philips Semiconductor, Home Entertainment Engine: Nexperia pnx8500.

[PIBus] Open Microprocessor System Initiative, PI Bus VHDL Toolkit, Version 3.1.

[Pix03] http://www.pixelphysics.com/, 2003.

[Pro96] J. Protic, M. Tomasevic, V. Milutinovic, "Distributed Shared Memory : Concepts and Systems", IEEE Parallel & Distributed Technology; Summer 1996.

[Rea00] RealChip Custom communication Chips, Systems-on-Chips. http://www.realchip.com.systems-on-chips/ systems-on-chips, 2000.

[Roa01] International Technology Roadmap for Semiconductors http://public.itrs.net/Files/2001ITRS/FEP.pdf

[Sdl87] Computer Networks and ISDN Systems. CCITT SDL, 1987.

[Sed00] Semiconductor Encapsulation Development, http://www.netd.cn/english/t_zdyq/tz_zdyq_5.html, 2000.

[Syn02] Synopsys Eaglei, available on line at http://www.synopsys.com/products/hwsw/eagle_ds.html

[Sys00] Synopsys Inc., "SystemC", available at http://www.systemc.org/.

[Tag00] H. Tago, "CPU for Playstation 2", Workshop on synthesis and System integration pf Mixed technology (SASIMI), 2000.

[Tom96] H. Tomiyama H. Yassuura, "size-constrained code placement for cache miss rate reduction" In the proceedings of the ACM/IEEE International Symposium on system synthesis, ACM Press, New York, NY, 96-101, 1996.

[Uml02] UML, available on-line at http://www.rational.com/uml/, 2002.

[Val01] C.A. Valderrama, A. Changuel, P. V. Vijaya-Raghavan, M. Abid, T. Ben Ismail, and A. A. Jerraya, "A unified model for co-simulation and co-synthesis of mixed hardware/software systems", page 579-583. Morgan Kaufman Publshers, 2001.

[Ver96] S. Vercauteren, B. Lin and H. De Man, "Constructing Application-Specific Heterogeneous Embedded Architectures from Custom HW/SW Applications", in Proc. of DAC, June 1996.

[Vsi02] Virtual Socket Interface Alliance, http://www.vsi.org/

[Wil95] P. R Wilson, M. Jhonstone, M. Neely and D. Boles, "Dynamic storage allocation : A survey and critical review", In proceedings of the International Workshop on Memory Management (Kinross, Scotland, Sept.), 1995.

[Wol91] M. E. Wolfe and M. S. Lam, "A loop transformation theory and an algorithm to maximize parallelism", IEEE Trans. Parallel Distrib. Syst. 2, 4 (Oct.), 452-471, 1991.

[Wol96] M. Wolfe "High-Performance Compilers for Parallel Computing. Addison-Wesley", Reading, MA, 1996.

[XML00] XML 1.0 Specification, 2d ed., W3C Recommendation, Oct. 2000, http://www.w3c.org/XML

PUBLICATIONS

1. F. Gharsalli, S. Meftali, F. Rousseau, A. A. Jerraya, "Automatic generation of embedded memory wrapper for multiprocessor SoC", In proceedings of the 39th Design Automation Conference (DAC'02), New Orleans, USA, June 10-14, 596-601, 2002.

2. F. Gharsalli, D. Lyonnard, S. Meftali, F. Rousseau, A. A. Jerraya, "Unifying memory and processor wrapper architecture for multiprocessor SoC design", In proceedings of International Symposium on System Synthesis (ISSS'02), Kyoto, Japan, October 2-4, 19-24, 2002.

3. S. Meftali, F. Gharsalli, F. Rousseau, A. A. Jerraya, "An optimal memory allocation for application-specific multiprocessor system-on-chip", In proceedings of the 13th International Symposium on System Synthesis (ISSS 2001), Montreal, Canada, 19-24, 2001.

4. S. Meftali, F. Gharsalli, F. Rousseau, A. A. Jerraya, "Automatic code-transformations and architecture refinement for application-specific SoC", In proceedings of IFIP International Conference on Very Large Scale Integration Conference (VLSI-SOC 2001), Montpellier, France, December 3-5, 193-204, 2001.

5. S. Meftali, F. Gharsalli, F. Rousseau, A. A. Jerraya, Chapitre dans un livre : "Architecture refinement for application-specific multiprocessor SoC", Kluwer, 2002.

RESUME

Grâce à l'évolution de la technologie des semi-conducteurs, aujourd'hui on peut intégrer sur une seule puce ce qu'on mettait sur plusieurs puces ou cartes il y a une dizaine d'années. Dans un futur proche, cette évolution permettra l'intégration de plus de 100 Mbits de DRAM et 200 millions de portes logiques dans la même puce. D'après les prévisions de l'association d'industrie de semi-conducteur et d'ITRS, les mémoires embarquées continueront de dominer la surface des systèmes monopuces dans les années qui viennent, à peu près 94 % de la surface totale en 2014.

La conception à base de réutilisation d'IP mémoire est survenue pour réduire le fossé entre cette grande capacité d'intégration et la faible production de mémoire. Cette solution peut être idéale dans le cas d'une architecture homogène où tous les éléments ont les mêmes interfaces et utilisent les mêmes protocoles de communication, ce qui n'est pas le cas pour les systèmes monopuces. Pour rendre cette solution efficace, le concepteur doit consacrer beaucoup d'efforts pour la spécification et l'implémentation des interfaces logiciel-matériel. Vu la pression du temps de mise sur le marché (" time to market "), l'automatisation de la conception de ces interfaces d'adaptation est devenue cruciale.

La contribution de cette thèse concerne la définition d'une méthode systématique permettant la conception des interfaces logiciel-matériel spécifiques aux mémoires globales. Ces interfaces correspondent à des adaptateurs matériels flexibles connectant la mémoire au réseau de communication, et à des pilotes d'accès adaptant le logiciel de l'application aux processeurs cibles. Des expériences sur des applications de traitement d'images ont montré un gain de temps de conception important et ont prouvé la flexibilité de ces interfaces ainsi que leur faible surcoût en surface et en communication.

MOTS CLES

Systèmes monopuces, mémoire, architecture logiciel-matériel, interface logiciel-matériel, adaptateur matériel, pilote logiciel, conception à base de composants, génération automatique, bibliothèque générique.

ABSTRACT

Embedded memory is becoming a new paradigm allowing entire systems to be built on a single chip. In the near future, new developments in process technology will allow the integration of more than 100 Mbits of DRAM and 200 millions gates of logic onto the same chip. According to Semiconductor Industry Association and ITRS prevision, embedded memory will continue to dominate SoC content in the next several years, approaching 94% of the die area by year 2014.

Memory Reuse based design is emerging to close the gap between this steadily increasing capacity and the design productivity in terms of designed transistors per time unit. This solution can be ideal in the case of homogeneous architecture where all the components have the same interfaces and use the same communication protocols, which is not the case for system on chip. However, the integration of several memory IP into a system on a chip makes specification and implementation of hardware-software interfaces a dominant design problem. Indeed, memory interface design is still hand-made, it is time-consuming and it is error prone. For these raisons, the design automation of these memory interfaces becomes crucial.

The contribution of this thesis consists on systematic method of hardware-software interfaces design for global memory. These interfaces correspond to flexible hardware wrappers connecting the memory to the communication network, and software drivers adapting the application software to the target processors. Experiments on image processing applications confirmed a saving of significant design time and proved the flexibility as well as the weak communication and the area overhead.

KEY WORDS

System on chip, memory, Hardware / Software architecture, Hardware / Software interface, hardware wrapper, driver, component-based design, automatic generation, generic library.

ISBNE 2-84813-013-X (electronic format) **ISBN 2-84813-012-1 (paperback)**