Exploration et conception systématique d'architectures multiprocesseurs monopuces dédiées à des applications spécifiques = methods and tools for multiprocessor systems on chip, hardware/software co-designExploration and Systematic Design of Application-Specific Heterogeneous Multiprocessor SoC

Amer Baghdadi

▶ **To cite this version:**

Amer Baghdadi. Exploration et conception systématique d'architectures multiprocesseurs monopuces dédiées à des applications spécifiques = methods and tools for multiprocessor systems on chip, hardware/software co-designExploration and Systematic Design of Application-Specific Heterogeneous Multiprocessor SoC. Autre [cs.OH]. Institut National Polytechnique de Grenoble - INPG, 2002. Français. NNT : . tel-00002932

HAL Id: tel-00002932
https://theses.hal.science/tel-00002932

Submitted on 3 Jun 2003

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

N ° attribué par la bibliothèque
|__/__/__/__/__/__/__/__/__/__|

T H E S E

pour obtenir le grade de

DOCTEUR DE l'INPG

Spécialité : Microélectronique

préparée au laboratoire **TIMA** dans le cadre de
l'Ecole Doctorale Electronique Electrotechnique Automatique Télécommunications Signal

présentée et soutenue publiquement

par

Amer BAGHDADI

le 14 Mai 2002

Titre :

Exploration et conception systématique d'architectures multiprocesseurs monopuces dédiées à des applications spécifiques

Directeur de thèse : Ahmed-Amine JERRAYA

Co-directeur : Nacer-Eddine ZERGAINOH

JURY

M.	Pierre GENTIL	, Président
M.	Eric MARTIN	, Rapporteur
M.	Jean-Paul CALVEZ	, Rapporteur
M.	Ahmed-Amine JERRAYA	, Directeur de thèse
M.	Nacer-Eddine ZERGAINOH	, Co-directeur
M.	Marco CARILLI	, Examinateur

A mon père

A ma mère, mes frères et sœurs,
Aux personnes à qui je tiens,
A toi Aba Saleh

Remerciements

Je voudrais en premier lieu remercier profondément Monsieur Ahmed-Amine Jerraya, chargé de recherche au CNRS et responsable du groupe de Synthèse au niveau système (SLS) du laboratoire TIMA de m'avoir accueilli dans son groupe pour travailler sur un sujet si promoteur et si passionnant, m'avoir accordé beaucoup de son temps et avoir mis à disposition ses compétences et ses conseils très pertinentes afin de mener à bien mes travaux de thèse. Qu'il trouve toute l'expression de ma sincère reconnaissance.

De même, j'exprime ma plus grande gratitude à Monsieur Nacer-Eddine Zergainoh, maître de conférences à l'UJF-ISTG de m'avoir accordé beaucoup de son temps, merci Nacer pour tous les conseils, les discussions fructueuses et les efforts fournis. J'exprime ma profonde reconnaissance aussi pour les moments très amicaux, la confiance et les encouragements, je ne les oublierais pas.

Je remercie Monsieur Bernard Courtois, Directeur de Recherches au CNRS et Directeur du laboratoire TIMA, de m'avoir accueilli dans son laboratoire et Monsieur Pierre Gentil de m'avoir fait l'honneur de présider le jury de ma thèse.

Je remercie Monsieur Jean-Paul Calvez, professeur à l'école polytechnique de Nantes et Monsieur Eric Martin, professeur à l'Université de Bretagne Sud qui ont accepté d'être membres du Jury de cette thèse et d'assumer la tâche de rapporteur. Pour la participation au jury, je remercie Monsieur Marco Carilli et Monsieur Stéphane Curaba de STMicroelectronics.

J'exprime mes sincères remerciements à mes amis du bureau 121 pour les débats et discutions qui ont été très utiles et pour leur agréable compagnie : Fabiano, Pascal, Ayman, Ludovic, Moez, Morad, Lobna, Aimen et Wassim. Je remercie de même tous les amis et collègues du groupe SLS pour leur aides leur disponibilité et leur gentillesse : Sonja, Sungjoo, Frederic, Paul, Philippe, Imed, Damien, Lovic, Gabriela, Samy, Ferid, Wander, Yanick, Anouar, Arif et Andi.

Je remercie tous les membres de TIMA, CMP et CIME. Je tiens à remercier tout particulièrement : Kholdoun, Hubert, Alexandre, Isabelle et Isabelle, Corinne, Joëlle, Françoise, Chantal, Ahmed, Patricia, Lucie et Marie-Christine pour leur disponibilité et leur sympathie exceptionnelle. Mes sincères remerciements pour ceux qui m'ont aidé dans la correction des erreurs de français : Hubert, Xavier et Nacer.

Mes sincères et profondes reconnaissances à tous mes très chers amis et frères, avec vous la vie à Grenoble devient encore plus belle et plus enrichissante. Je ne vais pas citer les noms car je ne veux pas en manquer un. Un très grand merci pour votre soutien inestimable. Je ne l'oublierais jamais.

Enfin ma famille, mon père, ma mère, Aba AbdelMajid et Aba Ali, sans vous je n'aurais pas pu commencé cette thèse, c'est avec du chagrin que j'écris ces mots car mon père à qui je dois la reconnaissance et à qui reviendrait la joie n'est plus ici. Ma famille qui m'a toujours manifesté son soutien et sa confiance inestimables, à qui je suis redevable de tant et plus encore, et à qui je dédie avec une reconnaissance sans borne cette thèse.

Sommaire

Table des figures

Chapitre 1

Introduction generale

Sommaire

1.1 Contexte : multiprocesseur monopuce

Tous les acteurs dans le domaine des semi-conducteurs travaillent déjà sur des produits intégrant des processeurs multiples (CPU, DSP, ASIC, IP, etc.) et des réseaux de communication complexes (bus hiérarchiques, bus avec protocole TDMA, connexion point à point, structure en anneau et même des réseaux de communication par paquets !) dans des systèmes monopuces. Certains commencent même à parler de réseaux de composants monopuces (networked system on chip). Il s'agit d'assembler des composants standards pour en faire un système comme on le fait pour les cartes. Ainsi, le problème n'est plus tant de concevoir des composants efficaces mais surtout de les assembler efficacement afin d'obtenir une architecture qui respecte les contraintes de performance et de coût. Pour les composants, le problème devient plutôt de s'assurer de leur validité avant de les utiliser.

Le Tableau 1 présente quelques exemples de ces nouveaux produits multiprocesseurs monopuces (multiprocessor system on chip). Des composants spécifiques (DSP, MCU) sont utilisés pour implémenter les fonctionnalités spécifiques (traitement et/ou contrôle). De plus, pour offrir la largeur de bande exigée par la communication, plusieurs types de réseaux de

communication (bus hiérarchiques, réseaux de commutateurs, etc.) sont employés. Outre les composants standards (mémoires, processeurs), ces architectures incluent des blocs matériels (IP) dédiés à l'application et qui peuvent s'élever à plusieurs millions de portes logiques.

Tableau 1. Etat de l'art sur les produits multiprocesseurs monopuces

Domaine d'application	Traitement des données	Contrôle	mémoire embarquée	Réseau de comm.	Taille de la puce	Exemple typique
Terminaux télécom	1 DSP	1 MCU	> MB	Bus hiérarchiques	> M portes	VDSL (ST) [83]
Multimédia	Quelques DSP	1 MCU	>> MB	Réseau de commutateurs	< M portes	TRIMEDIA (Philips) [110]
Processeurs de réseau	Beaucoup de DSP	Quelques MCU	>> MB	Réseau	> M portes	IXP2800 (Intel) [52]
Processeurs de jeux	Quelques DSP	Quelques MCU	>> MB	Réseau hiérarchique	>> M portes	PlayStation 2 (Sony) [89]

Ces systèmes sont destinés à des marchés de masse. Ainsi, afin de réduire les coûts de production et d'améliorer les performances (fréquence d'horloge), ils ont tous été (ou seront) intégrés sur une seule puce. Il est prévu que ces systèmes deviennent les principaux vecteurs d'orientation de toute l'industrie des semi-conducteurs.

Cette évolution induit deux autres : évolution de la technologie de fabrication et évolution des techniques de conception (CAO). La technologie de fabrication continue à suivre la loi empirique de Moore (le niveau d'intégration des circuits double tous les 18 mois). Ceci est d'autant plus vrai que Intel a introduit le nouveau transistor TeraHertz [48]. Cependant la CAO reste à la traîne, ce qui fait que la lacune de productivité de la conception ne cesse d'augmenter. On ne traite pas 50 millions de transistors avec les méthodes que l'on utilisait pour concevoir une porte logique. Ainsi, dans l'édition 2001 du rapport de l'ITRS [50], le message principal annoncé est celui-ci : «*Le coût de la conception constitue la plus grande menace à l'avenir de l'industrie des semi-conducteurs*».

Il devient donc crucial de maîtriser la conception des systèmes monopuces complexes tout en respectant les contraintes de mise sur le marché et les objectifs de qualité.

Ainsi plusieurs approches de conception ont émergé ces dernières années pour faire face à ce défi. La conception conjointe matérielle/logicielle (ou Codesign) était l'une des plus explorées par les groupes de recherche. Il s'agit de méthodologies et d'outils de CAO qui permettent la synthèse automatique des parties matérielles, des parties logicielles, et de la communication en partant d'une spécification de haut niveau du système complet [63]. Malgré sa réputation, cette approche n'a pas trouvé un grand succès dans la pratique principalement à cause d'un problème de qualité. Ceci est dû aux hypothèses très restrictives sur l'architecture (ex. elle ne considère pas les architectures complexes incluant des réseaux de communication sophistiqués). Ainsi cette approche pourra être acceptable pour donner des estimations, mais pas pour réaliser des produits.

Depuis 1999, les travaux de recherches au sein du groupe SLS (System Level Synthesis) ont été orientés vers une nouvelle approche plus proche de l'architecture. Nous pensons qu'elle sera

plus appropriée que le codesign pour faire face au défi annoncé. Le travail de cette thèse s'inscrit dans ce cadre et sera développé dans la suite de ce document.

1.2 Approche de conception

La grande difficulté de l'architecture, lors de la conception de ces systèmes multiprocesseurs monopuces, est de maîtriser la complexité. Le point de départ est la spécification de l'application au niveau fonctionnel (plus les contraintes de réalisation), et le point d'arrivée est l'architecture RTL finale (matérielle/logicielle). Du fait que les outils classiques de CAO permettent un passage quasi automatique d'un modèle RTL à la puce, le défi actuel est le passage aisé d'un modèle fonctionnel au modèle RTL et cela sous de fortes contraintes de qualité et de temps de conception.

En effet maîtriser la complexité lors de la conception des systèmes multiprocesseurs monopuces revient à maîtriser la complexité de l'architecture RTL et à maîtriser la complexité du flot de passage de l'application à l'architecture.

Concernant l'architecture, la Figure 1 montre une vue en couches des architectures multiprocesseurs monopuces (ou AMM[1]). Cette représentation en couches des AMM est nécessaire pour maîtriser leur complexité. Elle permet de séparer la conception des composants logiciels (application) et matériels (composants existants ou dédiés à l'application) des couches de communication.

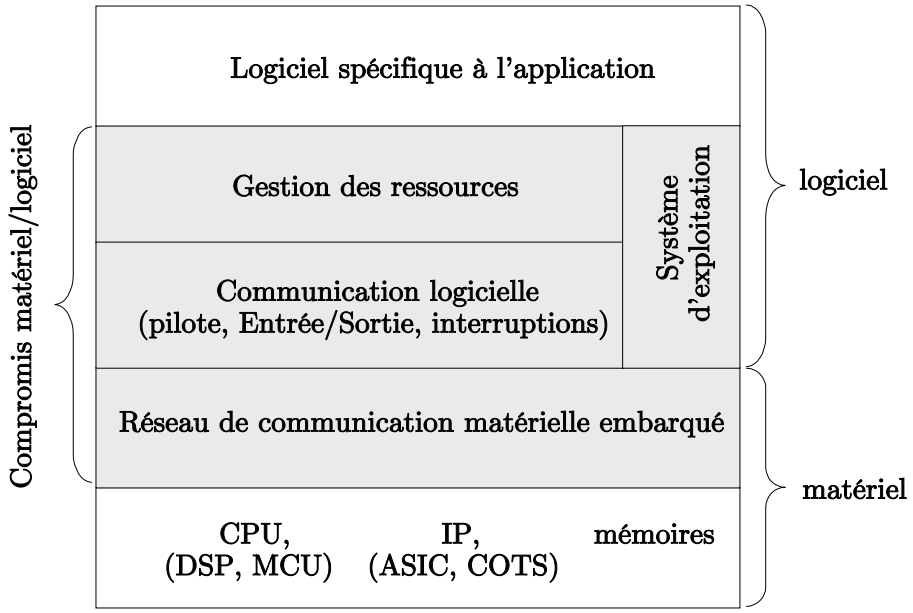


Figure 1. Une vue en couches des AMM

La partie matérielle est composée de deux couches :

- La couche basse contient les principaux composants utilisés par le système. Il s'agit de composants standards tels que des processeurs (DSP, MCU, IP, mémoires).

[1] L'abréviation AMM sera utilisée dans la suite de ce document pour designer les **a**rchitectures **m**ultiprocesseurs **m**onopuces

- La couche de communication matérielle embarquée sur la puce. Il s'agit des dispositifs nécessaires à l'interaction entre les composants. Cette couche peut contenir un réseau de communication complexe allant du simple pont (bridge) entre deux processeurs au réseau de communication par paquets. Bien que le réseau de communication lui-même soit composé d'éléments standards, il est souvent nécessaire d'ajouter des couches d'adaptation entre le réseau de communication et les composants de la première couche.

Le logiciel embarqué est aussi découpé en couches :

- La couche basse contient les pilotes d'entrées/sorties et d'autres contrôleurs de bas niveau permettant de contrôler le matériel. Le code correspondant à cette couche est intimement lié au matériel. Cette couche permet d'isoler le matériel du reste du logiciel.
- Une couche de gestion de ressources permet d'adapter l'application à l'architecture. Cette couche fournit les fonctions utilitaires nécessaires à l'application qu'il faut aussi personnaliser pour l'architecture considérée pour des raisons d'efficacité. Cette couche permet d'isoler l'application de l'architecture. Elle est généralement appelée système d'exploitation ou OS (de l'anglais Operating System).
- Le code de l'application.

Concernant le flot, il faut définir un flot de conception allant d'une spécification système de l'application à la réalisation au niveau RTL. Dans un flot complet deux étapes doivent être traitées : l'étape d'exploration d'architectures et l'étape d'implémentation de l'architecture choisie. La Figure 2 montre une représentation simplifiée d'un tel flot.

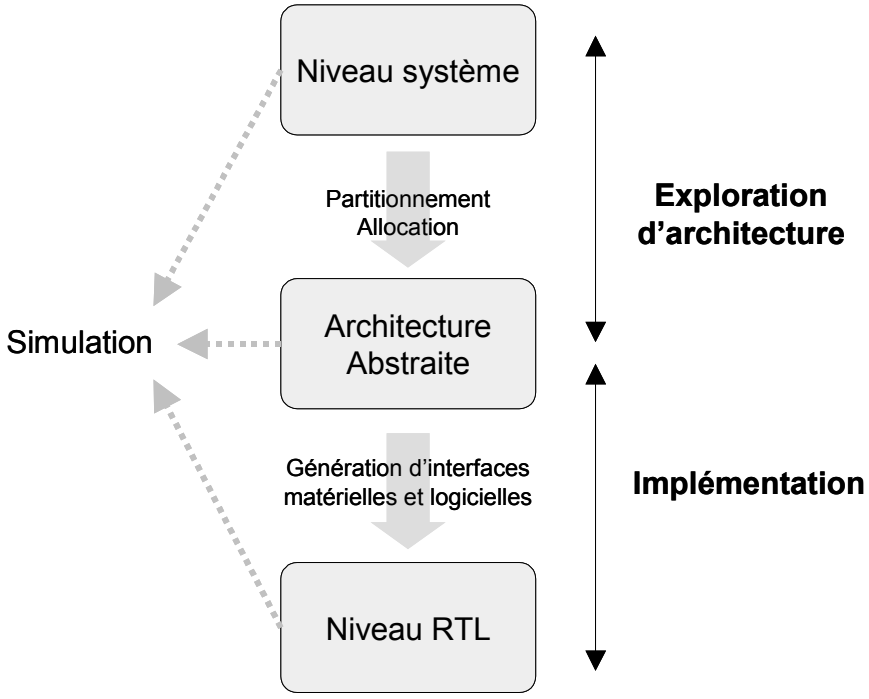


Figure 2. Représentation simplifiée du flot de conception complet

L'application est décrite au niveau système (par des langages tels que SDL, SystemC ...) comme des modules communicants. La première étape du flot consiste à trouver le meilleur partitionnement logiciel/matériel, l'allocation des processeurs et le choix des protocoles et réseaux de communication. L'application est ainsi raffinée à un niveau intermédiaire qui

contient les paramètres de l'architecture choisie (qui est dédiée à cette application spécifique). A partir de ce niveau, un flot systématique d'implémentation permet la génération de l'architecture RTL. Ceci comporte essentiellement le raffinement de la communication qui génère les interfaces matérielles/logicielles connectant les différents composants au réseau de communication. En parallèle, il est observé que plus la validation intervient tôt dans le flot de conception d'un système, plus une erreur peut être corrigée rapidement [11]. Afin de réduire les coûts de conception et d'économiser les efforts inutiles, il est important de vérifier la fonctionnalité du système à tous les niveaux durant le processus de conception.

L'objectif du groupe SLS est de définir une approche complète couvrant tous les aspects de ce flot. Cette approche doit être systématique pour permettre le développement d'outils de CAO. La contribution de cette thèse couvre certains aspects de ce flot que nous détaillons dans la section suivante. D'autres travaux se déroulent en parallèle, dans le groupe SLS, pour définir un environnement complet.

1.3 Contributions

Cette thèse s'est passée en chevauchement entre la fin des recherches sur un outil de codesign classique (MUSIC [57]) et le début des recherches sur une nouvelle approche. Ainsi nous avons pu profiter de l'outil de codesign pour définir une méthode d'estimation de performance qui permet une exploration efficace de l'espace des solutions architecturales (*première contribution*). La faiblesse de l'outil de codesign pour la partie implémentation du flot (hypothèses restrictives sur l'architecture) a fait que nous avons orienté nos recherches vers une meilleure approche. Une vaste étude sur les AMM a alors été entreprise. L'analyse de ces architectures a permis de proposer un modèle architectural multiprocesseur efficace (*deuxième contribution*). Ensuite, basé sur ce modèle, un flot de conception systématique pour la partie implémentation de la Figure 2 a été développé (*troisième contribution*). Ce travail concerne la partie architecture matérielle (d'autres travaux traitaient la partie logicielle). Le modèle architectural associé au flot constitue une approche systématique pour la conception des AMM dédiées à des applications spécifiques et par conséquent il permet de trouver une réponse aux défis annoncés plus haut. Plusieurs applications industrielles ont été réalisées en suivant cette approche (*quatrième contribution*). Ces expérimentations ont permis de valider l'approche, de montrer son efficacité et aussi de bien la définir et de l'améliorer.

En parallèle aux travaux présentés dans cette thèse, d'autres travaux se sont déroulés pour compléter les autres parties du flot de conception complet, tels que la partie logicielle (OS), les optimisations mémoire (matérielle et logicielle), l'approche de validation (par co-simulation), ainsi que le développement des outils de génération automatique basés sur notre approche systématique.

1.4 Plan du mémoire

La suite de ce mémoire est organisée de la façon suivante :

Le chapitre 2 présente l'étude et l'analyse effectuées sur les systèmes multiprocesseurs monopuces. Il montre d'abord pourquoi nous nous sommes intéressés aux AMM dédiées à des applications spécifiques. Ensuite il présente une étude détaillée de l'état de l'art sur ces architectures et sur leurs flots de conception, à travers les travaux de recherche (académique et industrielle) sur ce sujet. Enfin, il introduit l'approche de conception proposée. Cette approche

qui est basée sur trois éléments principaux : un modèle architectural générique, une méthodologie d'exploration d'architectures au niveau système et un flot systématique d'implémentation, sera développée dans les trois chapitres suivants (un élément par chapitre).

Ainsi le chapitre 3 présente un modèle architectural multiprocesseur générique. Il permet la conception d'AMM dédiées à des applications spécifiques et ceci pour un très large domaine d'applications. Son intérêt ainsi que sa structure sont détaillés.

Le chapitre 4 développe l'approche d'exploration d'architecture. Une méthode d'estimation de performance au niveau système est détaillée. Cette méthode est basée sur l'outil de codesign MUSIC (développé avant 1999 au sein du groupe SLS). Son efficacité sera montrée à travers une application significative.

Le chapitre 5 présente le flot systématique pour la partie implémentation de la Figure 2. Ce flot est basé sur le modèle architectural proposé au chapitre 3. Les différentes étapes du flot sont illustrées via un exemple d'application. L'efficacité de l'approche est ensuite démontrée et analysée à travers la réalisation de trois applications industrielles.

Le chapitre 6 présente les conclusions et les perspectives correspondant aux travaux menés dans le cadre de cette thèse.

Chapitre 2

SYSTEMES MULTIPROCESSEURS MONOPUCES : PROBLEMES POSES PAR LEUR CONCEPTION

Sommaire

Les applications embarquées appartiennent à un domaine en très forte expansion. Néanmoins elles sont de plus en plus soumises à des fortes contraintes fonctionnelles (telle que puissance de calcul et de communication) et non fonctionnelles (tels que temps de mise sur le marché, coût, surface et consommation). D'un autre coté, le progrès technologique permettra très prochainement une capacité d'intégration de centaines de millions de transistors sur une seule puce. Ce progrès technologique détermine ce qui est possible; et c'est l'architecture associée à un flot de conception efficace qui traduit le potentiel de la technologie en performance et capacité. Concernant l'architecture, les trois manières dont un grand volume de ressources améliore la performance sont le parallélisme, la localité, et la spécialisation. Le parallélisme implique multiprocesseur, la localité implique monopuce, et la spécialisation implique dédiée à une application spécifique. Quant au flot de conception, manipuler un grand volume de

ressources revient à remonter le niveau d'abstraction et proposer une méthodologie systématique pour le passage de ce niveau à une implémentation optimale.

Ce chapitre justifie le choix des AMM dédiées à des applications spécifiques. Ensuite il présente une étude détaillée de l'état de l'art sur les AMM et les flots de conception existants. Enfin, il présente l'approche de conception proposée.

2.1 Les AMM dédiées à des applications spécifiques

2.1.1 Les AMM (architectures multiprocesseurs monopuces)

2.1.1.1 Monopuce

Un système monopuce, appelé encore SoC (System-on-Chip) ou système sur puce, désigne l'intégration d'un système complet sur une seule pièce de silicium. Ce terme est devenu très populaire dans le milieu industriel malgré l'absence d'une définition standard [96]. Certains considèrent qu'un circuit complexe en fait automatiquement un SoC, mais cela inclurait probablement chaque circuit existant aujourd'hui. Une définition plus appropriée de système monopuce serait : *Un système complet sur une seule pièce de silicium, résultant de la cohabitation sur silicium de nombreuses fonctions déjà complexes en elles mêmes : processeurs, DSP, mémoires, bus, convertisseurs, blocs analogiques, etc. Il doit comporter au minimum une unité de traitement de logiciel (un CPU) et doit dépendre d'aucun (ou de très peu) de composants externes pour exécuter sa tâche. En conséquence, il nécessite à la fois du matériel et du logiciel.*

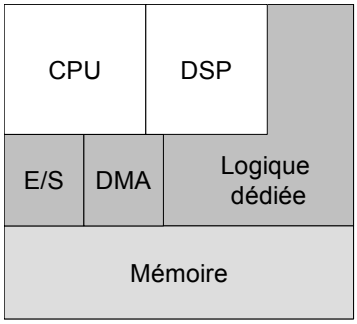


Figure 3. Un exemple de système monopuce (SoC)

La Figure 3 présente un exemple de système monopuce typique. Il se compose d'un cœur de processeur (CPU), d'un processeur de signal numérique (DSP), de la mémoire embarquée, et de quelques périphériques tels qu'un DMA et un contrôleur d'E/S. Le CPU peut exécuter plusieurs tâches via l'intégration d'un OS. Le DSP est habituellement responsable de décharger le CPU en faisant le calcul numérique sur les signaux de provenance du convertisseur A/N. Le système pourrait être construit exclusivement de composants existants, ou d'un combinaison de composants existants et de solutions faites sur mesure. Plus récemment, il y a eu beaucoup d'efforts pour implémenter des systèmes multiprocesseurs sur puce [40].

Il y a plusieurs raisons pour lesquelles la solution monopuce représente une manière attrayante pour implémenter un système. Les processus de fabrication (de plus en plus fins) d'aujourd'hui permettent de combiner la logique et la mémoire sur une seule puce, réduisant

ainsi le temps global des accès mémoire. Étant donné que le besoin en mémoire de l'application ne dépasse pas la taille de la mémoire embarquée sur la puce, la latence de mémoire sera réduite grâce à l'élimination du trafic de données entre des puces séparées. Puisqu'il n'y a plus aucun besoin d'accéder à la mémoire sur des puces externes, le nombre de broches peut également être réduit et l'utilisation de bus sur carte devient obsolète. Le coût de l'encapsulation représente environ 50% du coût global du processus de fabrication de puce [93][101]. Par rapport à un système-sur-carte ordinaire, un SoC emploie une seule puce réduisant le coût total d'encapsulation, et de ce fait, le coût total de fabrication. Ces caractéristiques aussi bien que la faible consommation et la courte durée de conception permettent une mise sur le marché rapide de produits plus économiques et plus performants.

2.1.1.2 Multiprocesseur

Avec le progrès technologique et la capacité d'intégration de centaines de millions de transistors sur une seule puce deux tendances architecturales ont émergé pour relever ce défi.

La première tendance s'est restreinte à l'utilisation d'architectures monoprocesseurs tout en améliorant considérablement les performances du CPU utilisé et l'utilisation de coprocesseurs. Un tel CPU se distingue par : une fréquence de fonctionnement très élevée, des structures matérielles spécialisées, un ensemble d'instructions sophistiquées, plusieurs niveaux de hiérarchie mémoire (plusieurs niveaux de caches), et des techniques spécialisées d'optimisation logicielle (nombre d'accès mémoire, taille, etc.). Dans cette direction on peut citer : PowerPC, Intel Pentium 4, ST100, et beaucoup d'autres processeurs de type VLIW ou superscalaire. La Figure 4 présente un exemple d'architecture conventionnelle µP/co-processeurs. Dans de telles architectures, la communication est basée sur le principe maître/esclaves : le CPU est le maître et les périphériques sont les esclaves. Les interfaces des co-processeurs sont généralement faites de registres transposés dans la mémoire du CPU et peuvent produire des interruptions au CPU. Ces communications se font généralement via un bus partagé (le bus mémoire du CPU). En termes de performance, de telles architectures –centrées autour d'un seul CPU– ont un inconvénient : la dégradation de performance engendrée par le fait que le processeur effectue la communication aussi bien que le calcul.

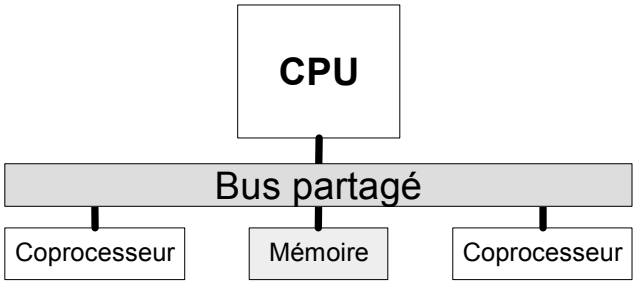


Figure 4. Un exemple d'architecture monoprocesseur (µP/co-processeurs)

La deuxième tendance s'est tournée vers les architectures multiprocesseurs (multimaîtres). Ces architectures qui étaient réservées jusqu'à maintenant aux machines de calculs scientifiques (tel que CM [16], CRAY [99], etc.) ne le sont plus avec le progrès de la technologie. Ainsi, l'idée de ceux qui ont choisi d'adopter cette tendance est la suivante : *En regardant plus étroitement les avancements en technologie et architectures sous-jacentes, il s'avère qu'il peut être de plus*

en plus difficile d'avoir des architectures monoprocesseurs assez rapides alors que les architectures multiprocesseurs sur puce deviennent déjà réalisables.

Cette thèse traite des architectures multiprocesseurs. Les principales raisons pour lesquelles les ***multiprocesseurs monopuces*** deviennent plus attrayants sont :

Le parallélisme : Les transistors supplémentaires disponibles sur la puce sont utilisés par les concepteurs principalement pour extraire plus de parallélisme à partir des programmes afin d'effectuer plus de travail par cycle d'horloge. La majorité des transistors sont employés pour construire les processeurs superscalaires. Ces processeurs visent à exploiter une quantité plus élevée de parallélisme au niveau instruction. Malheureusement, comme les instructions sont, en général, fortement interdépendantes, les processeurs qui emploient cette technique voient leurs rendements diminuer malgré l'augmentation quadratique de la logique necessaire au traitement de multiples instructions par cycle d'horloge. Un AMM évite cette limitation en employant principalement un type complètement différent de parallélisme : le parallélisme au niveau tâches. Ce type de parallélisme est obtenu en executant simultanément des séquences d'instructions complètement séparées sur chacun des processeurs.

Les retards causés par les interconnexions : Comme les portes CMOS deviennent plus rapides et les puces deviennent physiquement plus grandes, le retard causé par les interconnexions entre les portes devient plus significatif. Ainsi, les fils, dans les années prochaines, pourront seulement acheminer les signaux sur une petite partie de la puce durant chaque cycle d'horloge. Ceci diminue dramatiquement la performance dans le cas de large puce monoprocesseur. Cependant, une puce multiprocesseur peut être conçu de sorte que chacun de ses petits processeurs occupe un secteur relativement petit réduisant, ainsi, au minimum la longueur de ses fils et simplifiant la conception des chemins critiques. Seulement les fils, connectant les processeurs, les plus rarement employés et donc les moins critiques peuvent être longs.

Le temps de conception : Il est déjà difficile de concevoir des processeurs. Le nombre croissant de transistors, les méthodes de plus en plus complexes d'extraction de parallélisme au niveau instruction et le problème des retards causés par les interconnexions rendront ceci encore plus difficile. Un AMM, cependant, permet de réduire le temps de conception car il permet l'instanciation de plusieurs composants pré-validés sur une même puce. En outre, ces composants pré-validés peuvent être re-utilisés dans plusieurs applications de différentes tailles (en changeant le nombre de composants). Seulement la logique d'interconnexion entre composants n'est pas entièrement reproduite.

Puisqu'un AMM traite tous ces problèmes potentiels d'une façon directe et extensible, pourquoi ne sont-ils pas déjà répandus ? Une raison est que les densités d'intégration viennent juste d'atteindre les niveaux où ces problèmes deviennent assez significatifs pour considérer un changement de paradigme dans la conception des systèmes. Une autre raison, cependant, est que les problèmes de synchronisation et de communication entre les différents processeurs surviennent. Ceci rappelle encore une fois l'importance de l'architecture de communication dans les systèmes multiprocesseurs.

2.1.2 Les AMM à usage général

2.1.2.1 Construction de l'architecture

Les AMM à usage général (appelées aussi «universelles») sont construites indépendamment de l'application. Pour cette raison, elles se caractérisent par une topologie régulière et souvent homogène. Elles sont construites dans le but d'être réutilisées dans une large gamme de produits. En principe, les systèmes universels sont fortement flexibles et sont, en effet, réutilisables à travers presque toutes les applications à performance réduite. A titre d'exemple dans le monde de monoprocesseurs : un processeur RISC à usage général peut être re-programmé pour une grande variété de tâches.

Il existe de même les architectures domaine-spécifiques qui ciblent un domaine d'applications restreint (traitement d'image, xDSL, réseaux, téléphonie mobile, PDA ...) mais pas une application spécifique. Souvent les architectures universelles et les architectures dédiées à un domaine sont incapables de répondre aux besoins de calcul embarqué rencontrés dans les nouvelles applications. A titre d'exemple : les processeurs RISC, superscalaires, VLIW et DSP ne s'étendent pas efficacement au-delà de leurs limites architecturales respectives. Ils ont des limites concernant la fréquence de l'horloge et la capacité pour exploiter le parallélisme qui les rendent chers et non pratiques aux niveaux de performance les plus élevés.

Les architectures universelles se basent généralement sur les processeurs universels pour leurs puissance de calcul.

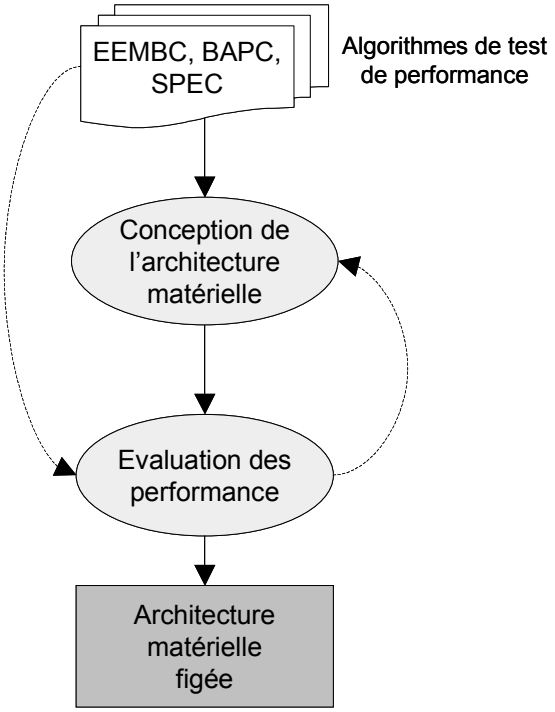


Figure 5. Construction des architectures universelles

Lors de la construction d'un système universel, le but des architectes est de réaliser une performance maximale pour un grand nombre d'applications. Comme ils ne ciblent pas une seule application spécifique, des algorithmes de test de performance fournis par des organisations spécialisées sont utilisés pour évaluer les performances des architectures conçues

(Figure 5). Parmi ces organisations on peut citer EEMBC (Embedded Microprocessor Benchmark Consortium), SPEC (Standard Performance Evaluation Corporation), et le BAPC (Business Application Performance Computing) [115].

Selon leurs expertises, leur savoir faire et les performances à atteindre, les architectes choisissent les valeurs des différents paramètres de leur architecture. Ces paramètres sont les types de processeurs utilisés, leur nombre, l'architecture de mémoire, les niveaux de caches et les techniques de cohérence utilisés, les réseaux de communication, les techniques et protocoles de communication et de synchronisation. Aux quels s'ajoute le modèle de programmation, sous forme de bibliothèque d'APIs et manuels de programmation.

Pour bien illustrer cette classe d'AMM à usage général, nous avons décidé de détailler un exemple typique : l'UMS (Universal Micro System) de Cradle Technologies, Inc. Il s'agit d'une architecture multiprocesseur qui peut contenir des douzaines de processeurs rapides qui s'exécutent en parallèle. L'architecture UMS (Figure 6) est un système multiprocesseur monopuce avec des E/S programmables pour se connecter aux dispositifs externes. Elle est composée de nœuds de processeurs reliés à l'aide de bus à large bande passante. Ces nœuds de processeurs, appelés Quads, communiquent avec les DRAMs externes et les interfaces E/S moyennant, respectivement, un contrôleur de DRAM et un système d'E/S entièrement programmable.

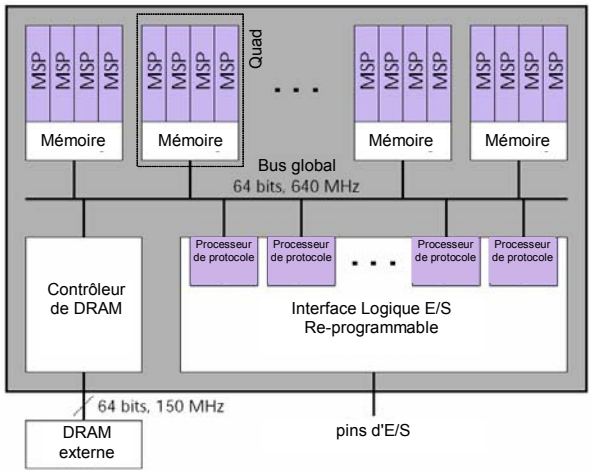


Figure 6. L'AMM universelle de l'UMS de Cradle Technologies

Chaque Quad comporte quatre MSP (de l'anglais Multi-Stream Processor), mémoires de programme et de données, une unité de DMA programmable et une interface au bus global. Le MSP est l'unité du traitement pour les applications. Selon la performance exigée, une application emploie un ou plusieurs MSP. Chaque MSP se compose d'un élément de traitement (PE) et de deux processeurs de signaux numériques (DSEs). Les PEs (~80 MIPS) sont des processeurs RISC (de 32 bits) universels, alors que les DSEs (~324 MIPS) sont des calculateurs très puissants. Le Quad inclut les mémoires de programme et celles de données. Chaque mémoire peut être configurée en tant qu'une combinaison aléatoire de mémoire locale et de mémoire cache. Les PEs utilisent la mémoire cache d'instruction comme source de leur flot d'instructions. PEs et DSEs emploient la mémoire de données en tant que mémoire locale pour des calculs. Les PEs utilisent l'unité de DMA pour transférer des données entre la mémoire locale de données et la SDRAM en parallèle (comme tâche de fond).

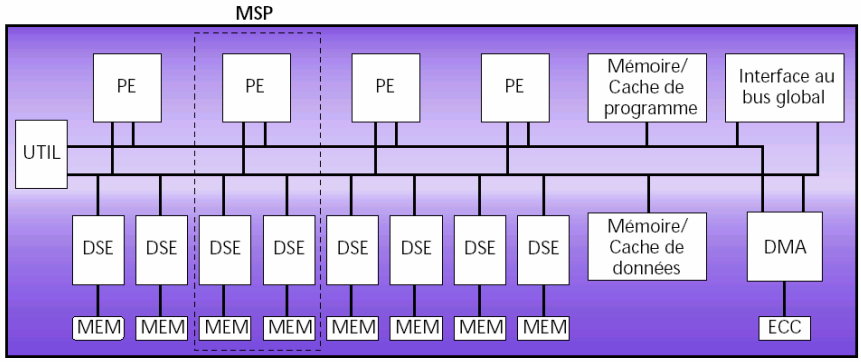


Figure 7. Architecture du Quad

Le système programmable d'E/S est un bloc matériel programmable à usage général qui permet d'implémenter en logiciel la plupart des périphériques d'E/S (tels que le PCI, le 1394 et les interfaces de SCSI). D'ailleurs, il est assez flexible pour que les utilisateurs y implémentent des interfaces pour des blocs matériels préconçus (IPs). L'implémentation de périphériques d'E/S en logiciel offre plusieurs avantages par rapport aux interfaces matérielles. Parmi ces avantages :

- Délai réduit de mise sur le marché : E/S implémenté en logiciel sur un matériel existant.
- Prise en charge rapide des changements dans la conception : un changement de logiciel, pas de changement de matériel.
- Facilité d'ajout de nouvelles fonctionnalités et de nouveaux périphériques : seulement ajout de logiciel.
- Facilité de maintenance : corriger les bugs avec une mise à niveau du logiciel.

Le bus global est rapide. A une fréquence de 640 MHz, il peut transférer des données à un taux de :

(0,64 GHz)*(8 octets/mot)*(80% d'efficacité pour 4 transferts d'octet) = 4,096 GOctets/sec.

A cette vitesse, le bus est tellement rapide qu'il est transparent. Ceci signifie que la vitesse du système n'est pas déterminée par celle du bus mais par celle des composants sur le bus. Si une partie significative de la bande passante du bus n'est pas utilisée, le retard d'arbitrage pour le bus reste faible. Dans le cas du bus global de l'UMS la latence et la bande passante sont déterminées par l'accès à un composant cible occupé de vitesse moyenne. Dans ce cas, il faut considérer le composant émetteur et le taux d'utilisation du composant cible afin de déterminer les latences et la répartition de la bande passante.

2.1.2.2 Utilisation de l'architecture

En effet dans cette approche les étapes de conception matérielle et logicielle sont bien séparées. On commence par l'étape de conception matérielle en utilisant des algorithmes de test de performance (Figure 5), ensuite l'architecture résultante est utilisée pour toutes les applications. Ainsi, son utilisation se réduit à l'étape de conception logicielle.

Ici l'architecture est considérée comme un composant qui fournit une puissance de calcul et de communication. Donc lorsque le concepteur désire utiliser une AMM universelle (déjà

conçue) pour implémenter son application, sa mission consistera à distribuer les différentes tâches de l'application sur les composants de l'architecture de façon optimale. C'est-à-dire utiliser au mieux les capacités de calcul et de communication fournies par cette architecture (Figure 8). Des outils d'aides à la conception sont en général fournis avec l'architecture : compilateurs, débuggeurs, simulateurs, etc.

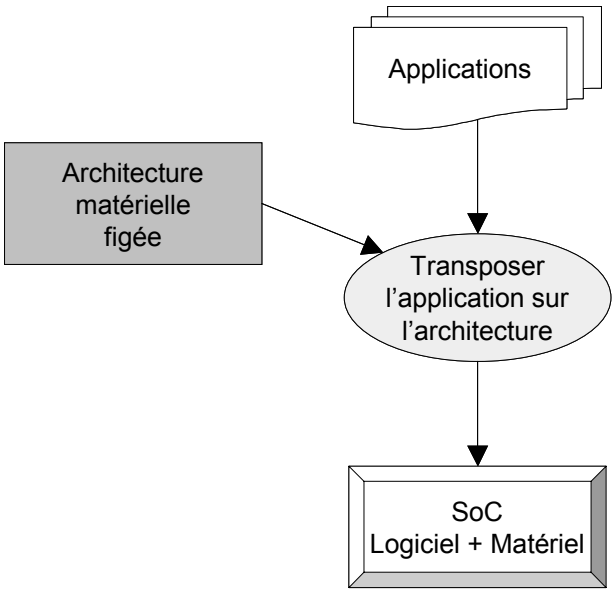


Figure 8. Utilisation des architectures classiques

Les hautes performances obtenues par les architectures multiprocesseurs sont dues à l'exploitation de trois formes de parallélisme : parallélisme fonctionnel, parallélisme par pipeline et parallélisme de données.

On parle de parallélisme fonctionnel lorsque deux tâches (ou plus) exécutent différentes fonctions, et les fonctions n'interagissent pas (ou très peu) entre elles. Un exemple est le traitement des canaux audio et vidéo dans un caméscope numérique. Les canaux audio et vidéo sont fonctionnellement indépendants. Puisqu'ils n'interagissent pas entre eux, ils peuvent fonctionner en parallèle. Le parallélisme fonctionnel est commun parce que c'est un résultat naturel de la conception de système. Quand on conçoit un système, on divise les tâches dans des fonctions. La majorité des tâches et des fonctions sont fonctionnellement indépendantes et sont donc des candidates naturelles pour une exécution parallèle.

On parle de parallélisme de pipeline lorsque la sortie d'un module alimente l'entrée d'un autre. Chaque module travaille sur une partie d'une tâche et passe ses résultats partiels au module suivant pour continuer leur traitement. Chaque module travaille sur sa partie du problème parallèlement aux autres modules travaillant eux-mêmes sur leurs parties du problème. Un exemple est la chaîne de montage, où une station met les roues sur la voiture tandis que la station suivante attache les portes.

On parle de parallélisme de données lorsqu'on a plusieurs blocs indépendants de données, où chacun est opéré par un processeur, et chaque processeur exécute le même algorithme de traitement pour opérer ses données.

Pour atteindre une performance maximale, le parallélisme doit être analysé dès le début du processus de conception système. La spécification du système doit être vérifiée contre les trois formes de parallélisme mentionnées ci-dessus. Selon l'architecture matérielle, le concepteur système décide la distribution des tâches sur les différents processeurs de l'architecture et l'allocation de bande passante sur le réseau de communication. Le concepteur peut être amené à regrouper plusieurs petites tâches sur un seul processeur ou à repartir une tâche complexe sur plusieurs processeurs. En tout cas, il est contraint par le nombre de processeurs et par le réseau de communication fournis par l'architecture matérielle utilisée.

2.1.2.3 Remarques sur les performances

Adapter une application quelconque sur une architecture matérielle universelle mène souvent à une solution sous-optimale en terme de performance et de coût. Puisque les architectures multiprocesseurs universelles sont conçues sans connaissance de l'application, elles fournissent une architecture de communication uniforme parmi les processeurs. Des réseaux d'interconnexions universels et des mémoires partagées sont généralement utilisés. En effet, le matériel de communication est ou bien «sur-dimensionné», en autorisant des transferts de données de haut volume qui ne se produisent jamais, ou bien «sous-dimensionné» et incapable d'accommoder les transferts de données requis.

On retrouve exactement les mêmes remarques de performance à propos des processeurs embarqués. Ces processeurs, qui sont en général identiques et non spécialisés, sont reliés d'une façon symétrique et sont souvent en sur-utilisation ou sous-utilisation.

2.1.3 Les AMM dédiées à des applications spécifiques

2.1.3.1 Construction et utilisation de l'architecture

L'approche pour concevoir une architecture dédiée à une application spécifique est complètement différente de celle d'une architecture universelle. En effet, dans le cas d'une architecture dédiée à une application spécifique, l'architecture matérielle et l'application logicielle interagissent mutuellement (Il est d'ailleurs à noter qu'il est difficile de séparer cette section en deux –construction et utilisation– comme cela a été naturel de le faire dans le premier cas !).

L'idée des architectures dédiées consiste à tailler les composants et le réseau de communication en fonction de la structure de l'application à concevoir. Après l'étape d'analyse de parallélisme et des différentes contraintes de l'application, le concepteur utilise son expérience et sa connaissance de l'application pour écrire une architecture abstraite. Il s'agit d'une spécification en forme de modules communicants via des canaux de communication abstraite. Ces modules représentent les différents processeurs (matériels ou/et logiciels) de son architecture finale. Les canaux de communication abstraits représentent le réseau de communication. Cette modélisation est devenue réalisable de nos jours grâce à l'émergence de nouveaux langages de modélisation système tel que SystemC 2.0 [106]. Avec de tels langages, non seulement la structure du système peut être modélisée mais aussi le comportement de chaque module (ou canal de communication) peut lui être attaché, et bien évidement une validation par simulation peut être effectuée.

L'analyse de contraintes de parallélisme effectuée sur l'application aide le concepteur à effectuer ces choix architecturaux tels que : le type de processeur exécutant chaque module (matériel ou logiciel, et si logiciel : le type de CPU qui sera utilisé), le type de réseau de communication utilisé pour implémenter les canaux de communication entre les processeurs. Ces choix peuvent être rapportés dans la spécification du système qui devient l'architecture abstraite de l'application. Cette architecture abstraite contient l'architecture matérielle multiprocesseur finale qui est bien dédiée à l'application.

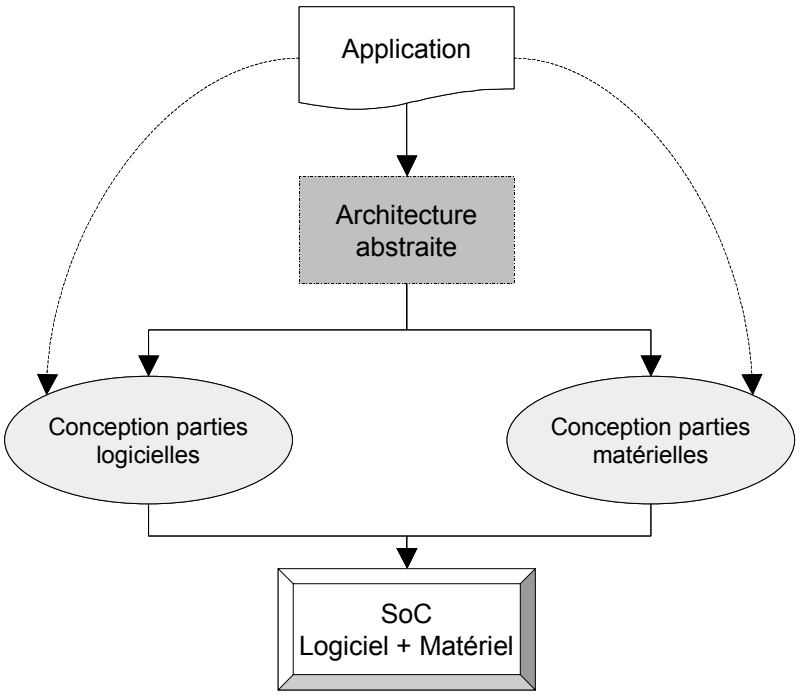


Figure 9. Construction et utilisation des architectures dédiées à des applications spécifiques

De cette architecture, et en utilisant des outils d'aide à la conception, l'architecture matérielle ainsi que le code applicatif sont générés (Figure 9). La génération de l'architecture matérielle consiste en l'instanciation[2] de cœurs de CPU, d'IPs, de blocs matériels, de mémoires, de bus et réseaux de communication, ainsi que la génération d'interfaces et de ponts qui permettent de connecter tous ces composants ensemble. La génération du code applicatif consiste en l'instanciation de pilotes de périphériques et de services de système d'exploitation, requises par les tâches du CPU, ceci pour tous les CPUs figurant dans l'architecture abstraite. Ensuite, les pilotes, les services de système d'exploitation ainsi que le code des tâches sont compilés tous ensemble pour générer le code applicatif à télécharger dans les différents CPUs.

On voit bien dans cette approche comment l'architecture abstraite est un mélange d'architecture et d'application. Il s'agit de restructurer l'application pour qu'elle ressemble à l'architecture. Dans cette approche, le CPU exécute une ou plusieurs tâches bien fixées et communique avec son environnement via des canaux bien adaptés à la fonction exécutée.

[2] Nous utilisons dans la suite ce mot de l'anglais par abus de langage pour designer l'appel d'un objet apparentant à une classe.

2.1.3.2 Remarques sur les performances

Bien évidemment une architecture dédiée à une application spécifique réalise la performance optimale, et ceci en terme de performance et de coût. Il n'y ni sur-utilisation ni sous-utilisation, il y a le juste nécessaire. En plus, la possibilité de choisir un partitionnement logiciel/matériel adapté montre une très grande flexibilité et permet de construire une architecture matérielle réalisant le juste performance requise au coût minimal.

2.1.4 Comparaison des AMM à usage général et ceux dédiées

2.1.4.1 Réseaux de communication

Le fait que les architectures universelles ne sont pas dédiées à une seule application oblige l'utilisation d'un réseau de communication régulier permettant d'adapter les besoins de communication à un grand nombre d'applications de différentes tailles. Donc un seul réseau de communication sera utilisé pour toutes les applications. Cela cause une flexibilité médiocre et mènera forcement à une sous-utilisation ou à une sur-utilistaion de ce réseau selon la taille et le besoin de communication exigé par l'application. La situation est différente dans le cas des architectures dédiées car dans ce cas le réseau de communication est bien spécialisé pour accommoder le besoin exact de communication requis par l'application. Un réseau de communication hétérogène peut être utilisé et il sera différent d'une application à une autre. Cela prouve une très grande flexibilité/performance dans le cas des architectures dédiées.

2.1.4.2 Processeurs à usage général

Les architectures universelles intègrent surtout des processeurs à usage général. Rares sont celles qui contiennent des processeurs dédiés à certaines applications (ASIPs) ou des blocs matériels spécialisés (décodeur Viterbi, décrypteur ...). Ces processeurs universels sont extensibles du fait que beaucoup d'applications peuvent y être implémentées, tandis que les performances obtenues peuvent être bien inférieures à celles obtenues avec des processeurs dédiés. Ces processeurs dédiés ainsi que les blocs matériels spécialisés caractérisent les architectures dédiées à des applications spécifiques. Le partitionnement matériel/logiciel de l'application et l'allocation de processeurs hétérogènes dédiés à des applications spécifiques permet à ces architectures d'atteindre des performances hors de portée des architectures universelles.

2.1.4.3 Logiciel applicatif

Concernant le logiciel applicatif, une grande différence existe entre les deux catégories d'architectures en question. Dans le cas des architectures universelles, il s'agit souvent de paralléliser une spécification séquentielle à l'aide du compilateur fourni avec l'architecture. Extraire du parallélisme (parallélisme de tâches, ou TLP) d'une application séquentielle de manière automatique (à l'aide d'un compilateur) donne des résultats très médiocres. Souvent il faudra re-écrire l'application de façon à coller au mieux à l'architecture. Ce problème n'existe pas dans les architectures dédiées à des applications spécifiques. Dans ces architectures le concepteur part d'une spécification déjà parallèle, il manipule son application en la restructurant en forme d'architecture. Donc le logiciel est déjà parallèle et distribué sur les différents processeurs de l'architecture. Dans ce cas, les compilateurs des processeurs individuels sont utilisés, et il n'y a pas besoin de compilateur dédié à l'architecture (ce qui représente une

forte économie). Concernant le système d'exploitation, souvent un seul système d'exploitation distribué est utilisé par les architectures universelles. Dans les architectures dédiées à des applications spécifiques, toutes les techniques sont valables, mais la plus utilisée est l'inclusion de systèmes d'exploitation locaux et bien ajustés aux besoins des tâches locales à chaque processeur (si plusieurs tâches y sont exécutées).

2.1.4.4 Domaine d'application

Du fait de leur universalité, les architectures universelles sont supposées avoir un large domaine d'application. Mais ce domaine d'application reste quand même limité à cause de l'utilisation de processeurs et réseau de communication homogènes. Souvent ces architectures sont dédiées à un domaine spécifique (traitement d'image, xDSL, réseaux, téléphonie mobile, PDAs ...). D'autre part, une architecture dédiée à une application spécifique est, comme son nom l'indique, dédiée à cette unique application. Cependant, les outils pour les construire sont capables de traiter une très grande variété d'applications de différentes tailles.

2.1.4.5 Validation

Dans le cas des architectures universelles deux validations sont effectuées. La première est purement fonctionnelle et la deuxième est directement sur la puce. La grande distance entre ces deux niveaux de validation augmente la difficulté de déboguer l'application. Or déboguer un system multiprocesseur sur puce est une mission très difficile. Ce problème est allégé dans le cas des architectures dédiées à des applications spécifiques grâce à l'utilisation d'une stratégie de validation multi-niveaux. Dans ce cas des cosimulations à plusieurs niveaux intermédiaires sont effectuées ce qui permet un déboguage aisé de l'application [85].

2.1.4.6 Performance

Transposer une application sur une architecture à usage général donne souvent lieu à une solution sous optimale. Ceci est dû à la grande variation dans les besoins de calcul et communication d'applications différentes. Rappelons que l'architecture universelle est généralement dotée d'une certaine puissance de calcul et de communication généralement répartie de façon homogène sur les différents processeurs qui la constituent. Ainsi, souvent cette architecture fixe sera ou bien sur-dimensionnée ou sous-dimensionnée par rapport à l'application qui y sera transposée. Ajoutons que l'implémentation directe d'une application au niveau RTL rend la tâche d'optimisation très difficile. L'architecture dédiée à une application spécifique est exactement sur mesure par rapport à l'application et donne forcement les meilleures performances.

2.2 État de l'art sur les travaux existants

Cette section est divisée en deux parties. La première partie est consacrée aux architectures rencontrées dans le monde des systèmes multiprocesseurs monopuces. Cette étude permet de mieux comprendre ce type d'architectures et d'extraire les avantages des différents travaux existants. Ainsi d'en tirer profit lors de la proposition de notre propre modèle architectural. La deuxième partie de cette section présente les flots de conception récents dans le domaine des AMM dédiées à des applications spécifiques. Les différents aspects des travaux présentés seront détaillés. **L'état de l'art sur les méthodes et les techniques d'estimation de performances est présenté dans le chapitre 4 (section 4.1.2).**

2.2.1 État de l'art sur les architectures

Les SoCs gagnent de plus en plus de popularité parce qu'ils se présentent comme «la» solution capable de fournir un système de haute performance à prix réduit. Parmi les produits SoC déjà existants, on peut citer les suivants : Pnx8500 de Philips (utilisé pour *set-top box*, contient un processeur universel, un DSP, deux coprocesseurs matériels de contrôle, et une mémoire partagée), Emotion Engine (cœur de la PlayStation-2, contient deux DSPs commandés par un processeur universel, plusieurs coprocesseurs spécialisés, une RAM partagée externe, et un bus hiérarchique), MxP (processeur de réseau, composé d'un processeur universel qui commande quatre processeurs RISC et des unités d'E/S), C-5 (processeur de réseau, contient 16 CPUs et plusieurs coprocesseurs matériels interconnectés par trois bus).

En effet, toutes les architectures qui seront présentées ci-après semblent en apparence différentes. Cependant une analyse détaillée de toutes ces architectures donne un résultat intéressant : elles sont toutes constituées de composants (processeurs, coprocesseurs, mémoires, etc.) interconnectés via un réseau de communication. Ainsi on peut les représenter toutes par le modèle générique de la Figure 10.

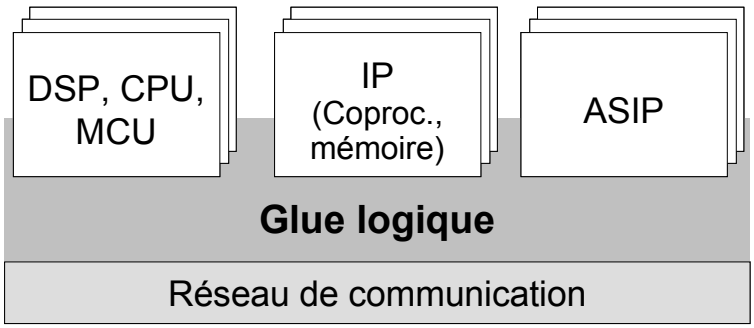


Figure 10. Le modèle générique représentant les AMM

Dans le reste de cette section nous avons tenté un classement de ces travaux selon le schéma de communication utilisé par leur architectures cibles.

2.2.1.1 Utilisation d'un coprocesseur de communication universel programmable

Dans cette classe, le point clé caractérisant l'architecture de communication est l'utilisation d'un coprocesseur universel de communication. Parmi les travaux les plus pertinents dans ce domaine on peut citer les travaux effectués à l'université de Stanford sur l'architecture multiprocesseur FLASH qui utilise un coprocesseur universel programmable. Ces travaux appuient le concept d'utilisation d'un coprocesseur de communication et permettent d'analyser les effets de spécialisation et de généricité.

2.2.1.1.1 Le système multiprocesseur FLASH de l'université de Stanford

Un problème important des machines DSM (Distributed Shared Memory) à cohérence de cache est leur surcoût élevé en matériel, alors que la critique majeure des actuelles machines à passage de message est leur surcoût élevé en logiciel. Ainsi l'intégration efficace, le support de la mémoire partagée à cohérence de cache ainsi que le passage de message, le tout pour un surcoût matériel/logiciel très bas, sont les objectifs principaux du multiprocesseur FLASH (Flexible

Architecture for SHared memory). L'efficacité implique peu de surcoût et une haute performance.

Le multiprocesseur FLASH [64] est un multiprocesseur à usage général qui intègre efficacement le support de la mémoire partagée à cohérence de cache et du passage de message à haute performance, tout en réduisant au minimum les surcoûts matériels et logiciels. Chaque nœud dans FLASH (Figure 11) contient un microprocesseur, une partie de la mémoire globale de la machine, un port vers le réseau d'interconnexion, une interface d'E/S et un contrôleur de nœud spécifique appelé MAGIC (Memory And General Interconnect Controller).

MAGIC traite toutes les communications, dans le nœud et entre les nœuds, en utilisant des chemins de données spécialisés pour le transfert de données efficace et un processeur programmable optimisé pour exécuter des opérations de protocole. L'utilisation de processeur de protocole rend FLASH très flexible (il peut supporter une grande variété de mécanismes de communication) et simplifie la conception et l'implémentation.

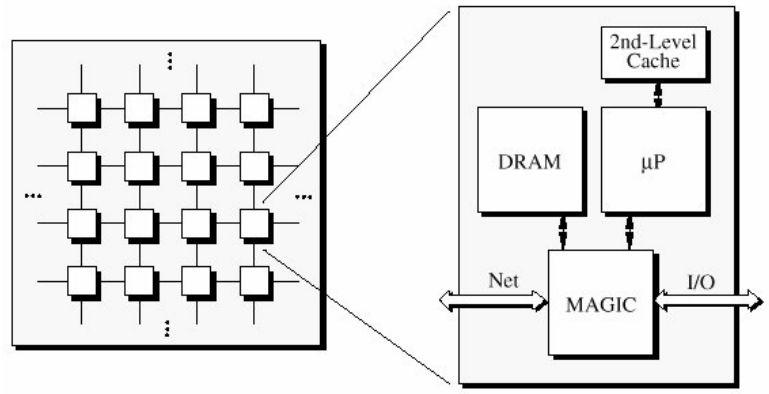


Figure 11. Le modèle architectural du système multiprocesseur FLASH [64]

MAGIC (Figure 12) est conçu pour offrir la flexibilité et la haute performance. D'abord, MAGIC comprend un processeur programmable de protocole pour la flexibilité. Ensuite, l'emplacement central de MAGIC dans le nœud assure qu'il voit tous les processeurs, réseau et transactions d'E/S, ce qui lui permet de contrôler toutes les ressources du nœud et de supporter une variété de protocoles. Troisièmement, pour éviter de limiter la structure du nœud à un protocole spécifique et pour s'adapter à des protocoles de besoins variables en mémoire, le code et les données du protocole résident dans une partie réservée de la mémoire centrale du nœud. Cependant, pour fournir un accès à vitesse élevée aux code et données des protocoles fréquemment utilisés, MAGIC contient des caches d'instruction et de données. Enfin, MAGIC sépare la logique de transfert de données de la logique de manipulation d'état du protocole. La logique câblée de transfert de données assure une basse latence et une bande passante élevée en supportant des transferts de données en pipeline. Le processeur de protocole exploite une table de répartition de matériel pour aider à entretenir les services rapidement, et un pipeline pour réduire l'occupation du processeur de protocole. Ces séparation et spécialisation de la logique de transfert de données et de contrôle assurent que MAGIC ne devient pas un goulet d'étranglement de latence et de bande passante. MAGIC forme le cœur du nœud, en intégrant le contrôleur de mémoire, le contrôleur d'E/S, l'interface de réseau et un processeur programmable de protocole. Le PP (Protocol Processor) est un processeur superscalaire de 64-

bit avec ordonnancement statique. Il partage la mémoire avec le processeur principal et il a ses propres caches d'instruction et de données.

En plus du passage de message, MAGIC doit communiquer avec le réseau, le processeur et le sous-système d'E/S. Les trois unités d'interface –PI, NI, et I/O (Figure 12)– implémentent ces interfaces en isolant le reste du système des détails d'interface. Cet isolement est un autre facteur de flexibilité de MAGIC puisqu'il limite la quantité de modifications matérielles exigées pour adapter MAGIC à d'autres systèmes.

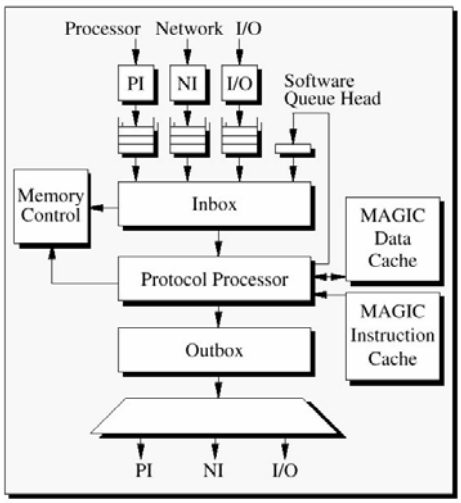


Figure 12. Architecture du co-processeur de communication MAGIC [64]

En effet ce système montre les avantages énormes de l'utilisation d'un coprocesseur de communication puissant et efficace. Cependant, puisque FLASH est un multiprocesseur universel, il doit pouvoir pour supporter différents protocoles de communication, ce qui conduit à l'utilisation d'un processeur universel pour la communication (MAGIC). Dans notre cas, comme des architectures dédiées à des applications spécifiques sont visées, une telle flexibilité n'est pas exigée (le coprocesseur contiendra les contrôleurs de communication nécessaires et suffisants pour cette application spécifique).

2.2.1.2 Les architectures de communication centrées autour d'un bus système

Plusieurs travaux ont convergé vers l'utilisation d'un bus standard (ou propriétaire) autour duquel l'architecture se construit. Parmi ces travaux on peut citer les travaux d'IBM (CoreConnect). La plupart des processeurs et composants de cette compagnie sont directement adaptés pour être intégrés autour du bus spécifique proposé.

2.2.1.2.1 Les travaux d'IBM (CoreConnect)

CoreConnect d'IBM [46] est une architecture de bus embarqué disponible sous un accord de licence libre (sans honoraires) de IBM. Les composants qui sont connectés au bus doivent être conformes à la norme Blue Logic d'IBM. Son architecture hiérarchique fournit trois bus synchrones (Figure 13).

- Processor Local Bus (PLB).
- On-Chip Peripheral Bus (OPB).

- Device Control Register (DCR) Bus.

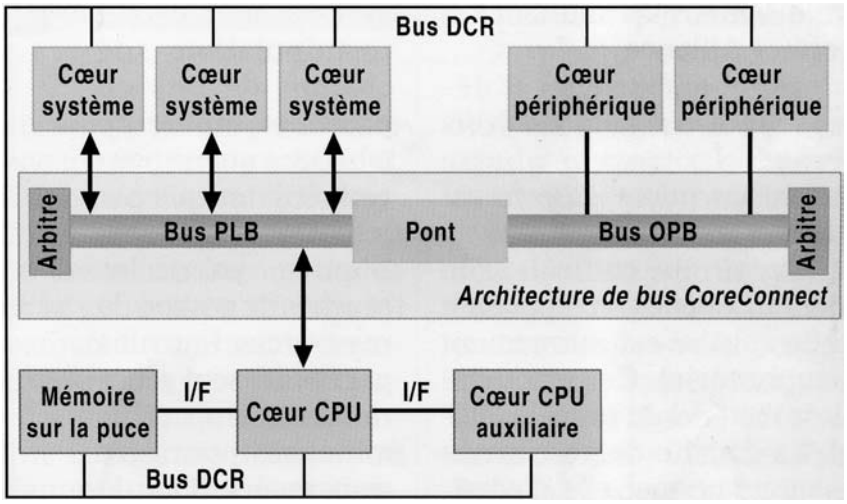


Figure 13. Le modèle architectural de CoreConnect [46]

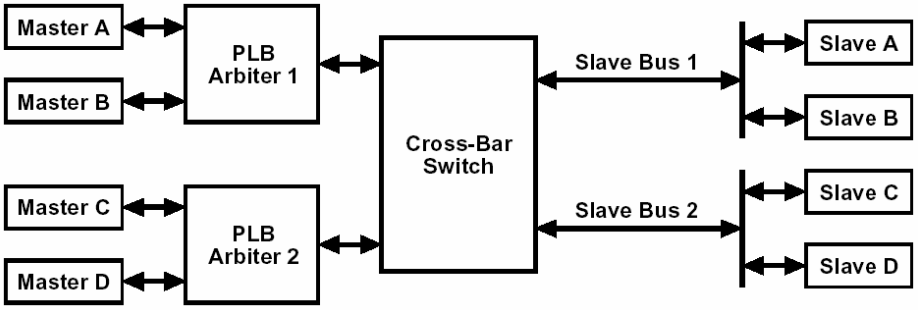


Figure 14. Utilisation de plusieurs bus PLB à l'aide du CBS (PLB Crossbar switch) [46]

L'architecture du PLB est très semblable à celle de l'AHB d'AMBA avec des bus séparés de lecture/écriture de données, multi-maître, pipeline, "split-transactions" et les transferts en rafale. Son but est de fournir une performance élevée, une faible latence, et une flexibilité de conception en connectant des composants de haut débit, tels que CPU, interfaces externes de mémoire et contrôleur DMA. La largeur de données est 32 ou 64 bits, extensible à 128 et 256 bits. Le PLB et le OPB ont des structures et signaux différents, ainsi les composants IP connectés aux bus ont des interfaces différentes. Les lectures/écritures concurrentes permettent une utilisation maximale du bus à deux transferts de données par cycle d'horloge. La latence maximale contrôlable est supportée par l'architecture en employant des timers de latence comme maîtres. Les composants du PLB peuvent augmenter le débit du bus en employant de longs transferts en mode rafale. Quand la bande passante sur un seul bus PLB dépasse les limites de sa capacité, une possibilité est de placer les maîtres de haut débit et leurs esclaves sur des bus séparés. Un composant IP –PLB Cross-Bar Switch (CBS)– peut être utilisé dans ce but comme représenté sur la . Le CBS permet des transferts de données simultanés multiples sur les deux bus PLB et il emploie les priorités pour traiter des demandes multiples sur un port esclave commun. Une demande prioritaire interrompt une transaction à faible priorité en cours d'exécution. Ainsi CoreConnect permet la conception de systèmes multiprocesseurs.

L'architecture CoreConnect est employée dans le contrôleur incorporé par PowerPC 405GP pour connecter un cœur PowerPC 405, un pont de PCI, et un contrôleur de SDRAM.

2.2.1.3 Les architectures de communications utilisant des réseaux commutés

Beaucoup de travaux se sont concentrés sur la proposition d'un nouveau réseau de communication mieux adapté aux AMM. Les réseaux proposés sont tous attractifs et sont bien adaptés pour certaines application. Ainsi il sera nécessaire de supporter leur intégration si une couverture d'un large domaine d'application est visée.

2.2.1.3.1 Le modèle PROPHID

Le modèle d'architecture de PROPHID [70][69]constitue une application remarquable de l'utilisation de la commutation de circuits pour la communication sur puce. PROPHID utilise un commutateur T-S-T (Time-Space-Time) (Figure 15) pour monter en cascade le traitement de flux de données vidéo.

Cependant, l'inconvénient de la commutation de circuits est le manque de réactivité contre les changements rapide de communications. Par exemple, l'interconnexion de PROPHID ne peut pas dynamiquement augmenter l'attribution de bande passante pour supporter les transmissions en rafale dans un flot binaire de type MPEG. Elle est d'autant moins convenable aux systèmes multi-maîtres avec un trafic aléatoire. Pour cette raison, une technique plus performante a émergé, connue sous le nom de commutation de paquets.

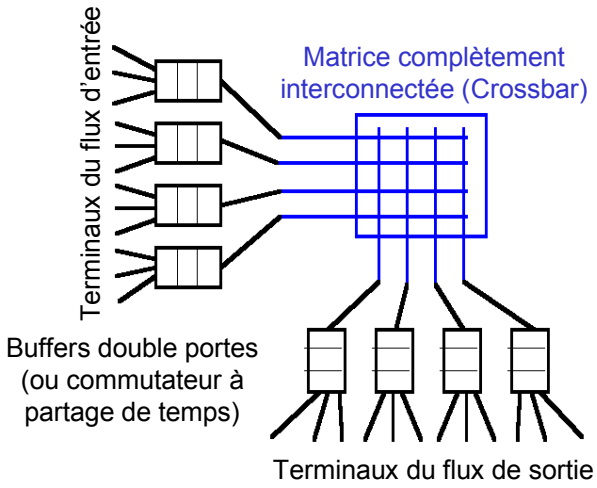


Figure 15. Le modèle architectural de PROPHID [70]

2.2.1.3.2 Le réseau SPIN de l'université Pierre et Marie Curie

Un réseau de commutation de paquets déplace les données d'un nœud à l'autre dans des petits morceaux formatés appelés paquets. Puisque les décisions de routage sont réparties sur les routeurs, le réseau peut rester très réactif même pour des paquets de tailles importantes. Malgré les nombreuses étapes de routage, une faible latence peut être maintenue si les routeurs expédient l'en-tête des paquets ASAP, sans attendre la queue (wormhole routing). Dans les travaux effectués à l'université Pierre et Marie Curie [39], les auteurs présentent un réseau à

commutation de paquets, intégré, programmable, et extensible appelé SPIN (Scalable, Programmable, Integrated Network) pour les interconnexions sur puce.

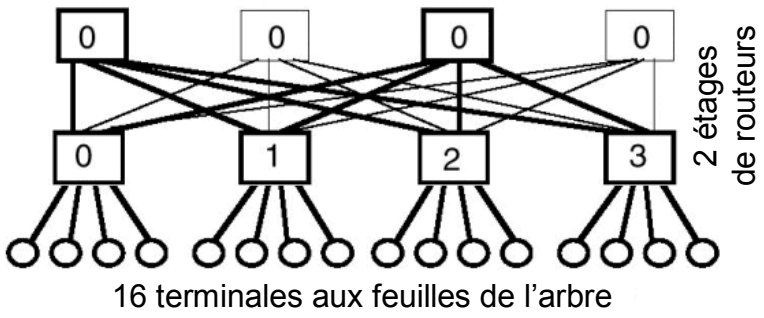


Figure 16. Réseau de type Fat-Tree [39]

Le lien élémentaire est un lien parallèle établi de deux chemins de données unidirectionnels de 32 bits. En effet, la topologie de réseau influence la complexité des décisions distribuées de routage. Ainsi, la topologie de réseau employée est le Fat-Tree (Figure 16), car il a été prouvé formellement par Leiserson [71] que cette topologie est la plus efficace pour des réalisations VLSI. La taille d'un paquet peut être quelconque, voire infinie. En ce qui concerne le protocole d'accès au réseau, nativement les paquets implémentent le modèle de communication à passage de message. Des messages peuvent être employés pour établir des protocoles émulant d'autres modèles comme flot de données ou espace d'adressage.

Concernant l'implémentation matériel (c.-à-d. les enveloppes) de ces protocoles (à travers le réseau de commutation de paquets), les auteurs n'ont présenté aucune approche automatique pour la conception de ces enveloppes. Des enveloppes compatibles VCI pour la communication par espace d'adressage ont été conçues et sont activement étudiées. Une critique bien connue contre leur approche est la complexité des concepts de réseau de commutation. Un inconvénient des réseaux de commutation de paquets est qu'ils présentent un retard intrinsèque arbitraire.

2.2.1.3.3 Le réseau µNetwork de Sonics

Silicon Backplane [102] est un système de communication pour SoC, fortement configurable, développé par Sonics, qui fait partie de l'architecture d'intégration de Sonics (SonicsIA). L'architecture se compose d'une paire de protocoles (propriété de la société), d'une spécification ouverte d'une interface pour composant IP. Le protocole OCP (Open Core Protocol) est une interface point à point qui fournit un ensemble standard de signaux de données, de contrôle, et de test permettant aux cœurs du système de communiquer entre eux. Silicon Backplane diffère de plusieurs des interconnexions conventionnelles de SoC en utilisant seulement un simple bus sur lequel tous les composants sont attachés par l'intermédiaire d'agents de Silicon Backplane. Un agent adapte l'OCP aux protocoles de Silicon Backplane. La communication externe (à la puce) est fournie par le Backplane Multichip. La montre l'architecture de Silicon Backplane. La largeur du bus de données est configurable à 32, 64 ou 128 bits. La bande passante est de 50MB à 4GB/s et le bus est entièrement pipeliné.

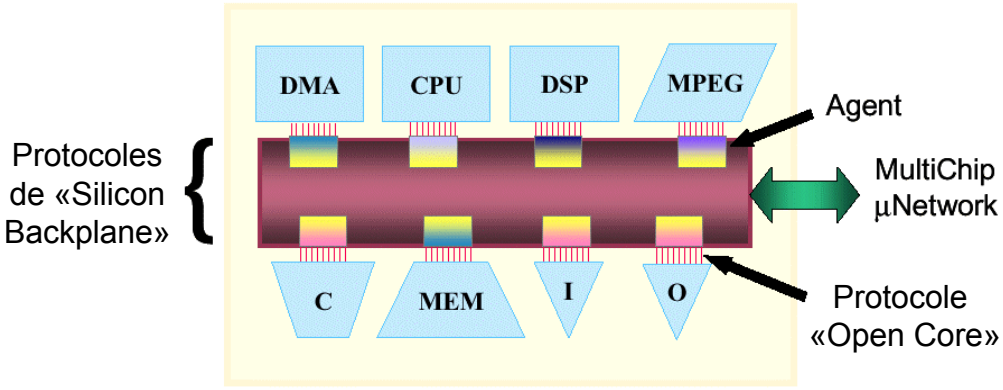


Figure 17. Le modèle architectural du µNetwork de Sonics [102]

Pour garantir une certaine bande passante pour un composant avec des contraintes de temps réel, le bus emploie le TDMA pour l'allocation de bande passante par composant. L'allocation de bande passante par composant peut être fixée à une certaine valeur au moment de la conception ou être laisser reconfigurable et pouvoir ainsi être fixée durant l'exécution du système. Le nombre de composants IP qui peuvent être attachés au bus n'a pas de limite architecturale. Les agents de Silicon Backplane permettent de contrôler la latence par composant et permettent la co-existence de composants IP rapides et lents sans dégradation de la performance et tout en garantissant les contraintes de temps réel.

2.2.1.3.4 Le modèle multiprocesseur de NUMAchine

Le type d'interconnexion est une interconnexion hiérarchique basée sur le modèle en anneau. NUMAchine [38] se compose d'un certain nombre de stations reliées par une hiérarchie à deux niveaux composée d'anneaux locaux et d'un anneau central (Figure 18).

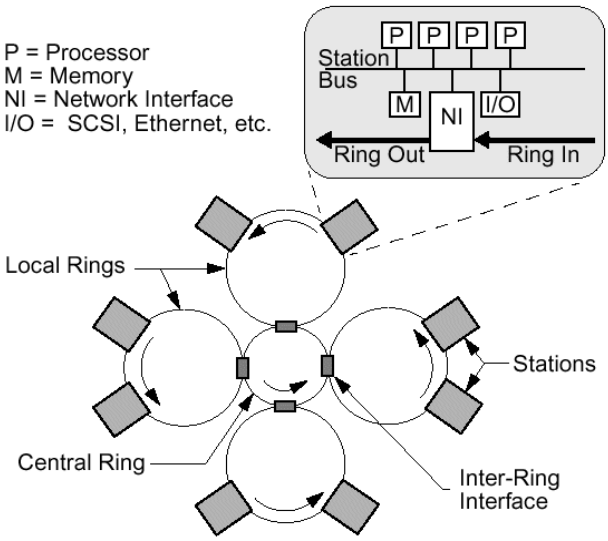


Figure 18. Le modèle architectural de NUMAchine [38]

Chaque station dans NUMAchine se compose d'un bus connectant un certain nombre de processeurs, un module de mémoire, un module d'E/S, et une interface de réseau qui la

connecte à un anneau local. La mémoire physique est distribuée parmi les stations de telle façon que chaque adresse mémoire a une station d'accueil fixe. Les accès mémoire locaux à une station génèrent seulement une latence d'interconnexion à un bus, tandis que les accès mémoire à distance génèrent une latence additionnelle d'interconnexion à un anneau pour accéder à des adresses dont l'emplacement physique se trouve sur une autre station. L'interface de réseau contient également une cache de réseau (NC) qui réduit la latence moyenne pour les accès de données à distance.

2.2.1.3.5 Processeur à réseau : l'architecture Octagon

La motivation pour ce travail vient d'un besoin d'architectures de communication sur puce à haute performance afin d'aider à fournir la capacité de traitement exigée par les versions les plus récentes des routeurs OC-768 0[61]. Les architectures traditionnelles d'interconnexion telles que le bus partagé et les crossbars auront des difficultés à répondre à ces besoins de débits tout en maintenant des coûts raisonnables [61].

L'unité de base de Octagon se compose de huit nœuds et de douze liens bidirectionnels connectés comme représenté sur la Figure 19.

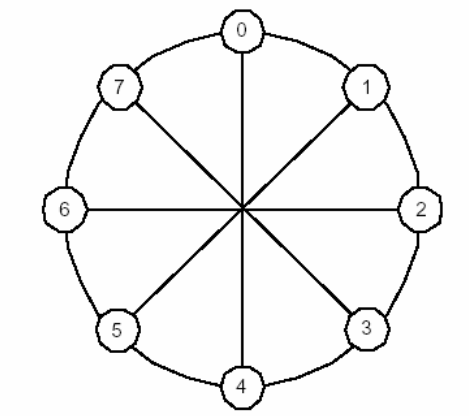


Figure 19. Configuration basic de l'architecture Octagon [61]

L'architecture Octagon a les propriétés suivantes : a) La communication entre n'importe quelle paire de nœuds peut être exécutée en passant au maximum par deux routeurs, b) Débit total plus élevé qu'un bus partagé ou un crossbar, c) Algorithme simple de routage de plus court chemin, d) Moins de câblage comparé à celui d'une interconnexion crossbar. Cette vue fournit une deuxième option d'extensibilité pour intégrer un nombre plus grand de composants sur la puce. Octagon peut fonctionner selon les deux modes de commutation : commutation de paquets ou commutation de circuit.

2.2.2 Etat de l'art sur les méthodologies de conception

Concevoir un système multiprocesseur monopuce signifie de nos jours un important travail manuel et une grande expertise pour choisir l'architecture, concevoir les interfaces matérielles, écrire des modules de gestion de périphériques et/ou configurer les systèmes d'exploitation commerciaux. La tâche la plus difficile est de faire fonctionner ensemble tous ces éléments qui sont taillés sur mesure aux besoins de l'application, sachant que le circuit résultant est toujours unique et imprévisible. L'organisation VSIA [114] essaye de réaliser une méthode basée sur

l'assemblage automatique (*plug-and-play*) de composants préconçus (IPs) [73]. Un grand nombre de méthodologies de conception travaillent dan cette direction en employant des composants et des interfaces standard de hardware/software [102], [98], [15]. Les pénalités de performance de la solution architecturale choisie sont acceptées en raison de la nécessité de répondre à des pressions toujours croissantes de temps de mise sur le marché. L'outil MCSE [81][13] présente une bonne méthodologie pour la conception des systèmes électroniques matériels/logiciels partant du plus haut niveau d'abstraction (contraintes fonctionnelles et non-fonctionnelles). Elle proposent une approche descendante pour la spécification, ascendante pour l'implémentation, basée sur plusieurs niveaux d'abstraction et couvrant un grand nombre de contraintes du système. L'architecture cible se compose d'une partie matérielle, d'une partie logicielle et d'une partie mixte qui est traitée par une approche de codesign proposée par l'outil MCSE.

Plusieurs travaux ont récemment émergé proposant d'accélérer le flot de conception avec de plus en plus d'automatisation à travers de nouveaux genres d'outils d'EDA capables de traiter la conception de systèmes multiprocesseurs monopuces. Ces outils se concentrent sur l'automatisation du raffinement de la communication (pour des tâches distribuées s'exécutant sur différents processeurs embarqués) et sur la réutilisation de blocs préconçus (IPs) avec la génération automatique d'interface [87][19][33][37].

Parmi les travaux les plus avancés proposant des méthodologies efficaces pour la conception d'AMM dédiées à des applications spécifiques, nous avons choisi d'en détailler deux : les travaux de Cadence (VCC) et ceux de l'IMEC (Coware).

2.2.2.1 Les travaux de Cadence (VCC)

Cadence a récemment lancé l'environnement VCC [103][112] (Virtual Component Co-design) pour l'exploration et la conception de systèmes électroniques partant du niveau d'abstraction système. L'outil d'exploration permet déjà d'analyser la performance d'un partitionnement matériel/logiciel au niveau système (cf. chapitre 4). Des travaux en cours tentent de trouver une solution pour l'implémentation. Cependant nous n'avons pas pu obtenir une présentation précise de leur modèle architectural ni de l'approche qui sera suivie.

Les composants technologiques principaux de Cadence VCC pour les ponts vers l'implémentation sont :

- Le raffinement de la communication qui facilite le raffinement des descriptions abstraites d'interface vers le niveau physique.
- La synthèse d'interfaces de communication qui génère la logique de communication qui connecte l'implémentation des composants virtuels matériels et logiciels.
- L'exportation de netlist matériel (Top-level) entièrement raffiné –en VHDL ou en Verilog– à partir d'un diagramme architectural abstrait.
- L'exportation de logiciel après assemblage et génération de code C tenant compte des effets de communications dédiés au RTOS.
- L'exportation de test à la Co-vérification M/L aide à la vérification de l'implémentation M/L de bas niveau du système en utilisant la représentation des flots de données à haut niveau via la description du système par VCC en tant que flots de messages (*tokens*).

2.2.2.2 Les travaux de l'IMEC (Coware)

Dans [113], une méthode pour générer des architectures multiprocesseurs pour les systèmes embarqués est présentée. Cette méthode traite la partie implémentation du flot de conception (Figure 2) et ne présente pas de solution pour la partie exploration d'architecture. L'architecture physique est abstraite comme étant une interconnexion entre des unités de traitement (Processor Component Units) via des canaux de communication point à point soit unidirectionnels soit bidirectionnels (Figure 20). Dans ce modèle, les composants de traitement peuvent communiquer directement par l'intermédiaire d'opérations explicites d'envoi et de réception ("*send*" et "*receive*") sur un canal spécifique, ou indirectement par l'intermédiaire d'un composant de mémoire partagée. Sur la , ceci est illustré en assignant un composant de mémoire qui peut être consulté par le DSP, le processeur RISC, ainsi que le co-processeur matériel. Cette configuration permettra aux trois unités de traitements de communiquer entre elles par l'intermédiaire de la mémoire partagée. Tous ces canaux sont implémentés au niveau RTL par un protocole commun (Figure 21) appelé le protocole d'attente synchronisée (ou *synchronous wait protocol*).

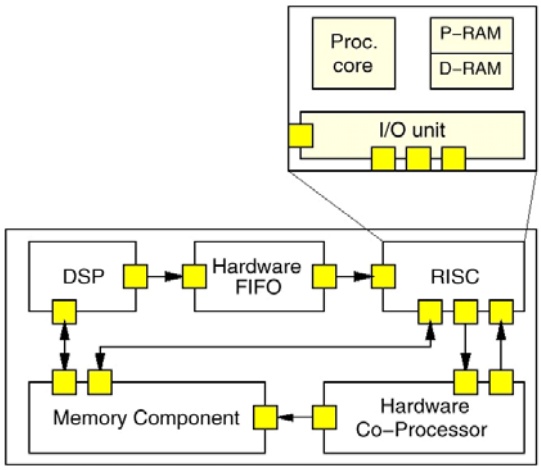


Figure 20. Architecture abstraite typique selon le modèle de l'IMEC (CoWare) [113]

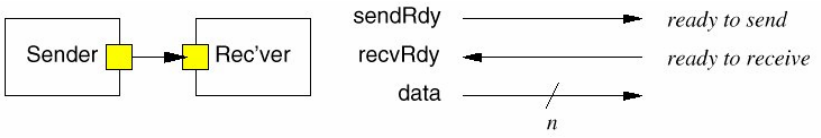


Figure 21. Le protocole d'attente synchronisée (*synchronous wait protocol*) [113]

L'approche de l'IMEC pour incorporer des processeurs de logiciel (CPU) dans une architecture embarquée spécifique, est basée sur la construction d'un «modèle d'architecture» paramétré autour du cœur du processeur. Ce modèle d'architecture admet trois composantes principales (Figure 22) : le coeur du processeur lui-même, une structure de mémoire interne pour les instructions du programme et les données, et une unité d'E/S qui implémente l'interface matériel de communication avec l'environnement externe. Ces composants sont interconnectés par l'intermédiaire du bus local du processeur. Du point de vue de l'utilisateur,

ce modèle d'architecture peut être adapté aux besoins de l'application pour fournir un nombre spécifique de canaux physiques qui acceptent des directions et des largeurs de données définies par l'utilisateur. En fait, c'est l'unité d'E/S dans le modèle d'architecture qui implémente ces canaux physiques pour l'interconnexion avec d'autres composants. Les ports d'E/S sur la Figure 22 sont connectés aux canaux de communication, qui sont eux-mêmes connectés à d'autres composants de traitement. Ces ports implémentent le contrôle du canal en utilisant le protocole d'attente synchronisée. L'entrée/sortie de données du cœur du processeur se fait par l'une des deux méthodes suivantes : E/S transposée dans la mémoire ou via des instructions programmées d'E/S.

En ce qui concerne la génération de l'unité d'E/S, elle est réalisée par leurs outils de synthèse d'interface qui peuvent également automatiser la conception du matériel nécessaire à l'adaptation de protocoles pour la communication avec un protocole spécifique de bus de processeur [75].

Côté logiciel, pour faciliter la communication externe, ils génèrent automatiquement une bibliothèque sur mesure faite de fonctions (APIs) en C et C++ comportant des opérations d'envoi et de réception. Cette bibliothèque est automatiquement générée selon le nombre de canaux que le processeur doit supporter, leurs directions, et les types de données qui sont véhiculées. La bibliothèque peut être considérée comme un noyau de communication dédié à l'application. Du point de vue du programmeur, le programme d'application communique avec le monde extérieur par l'intermédiaire d'appels de fonctions d'envoi et de réception appropriées.

Dans le cas d'un module matériel de traitement, ils produisent automatiquement pour l'utilisateur une enveloppe qui implémente essentiellement une couche de communication autour du matériel que l'utilisateur fournira plus tard. Ils produisent alors automatiquement un bloc VHDL spécifique qui réalise les canaux de communication selon le protocole d'attente synchronisée.

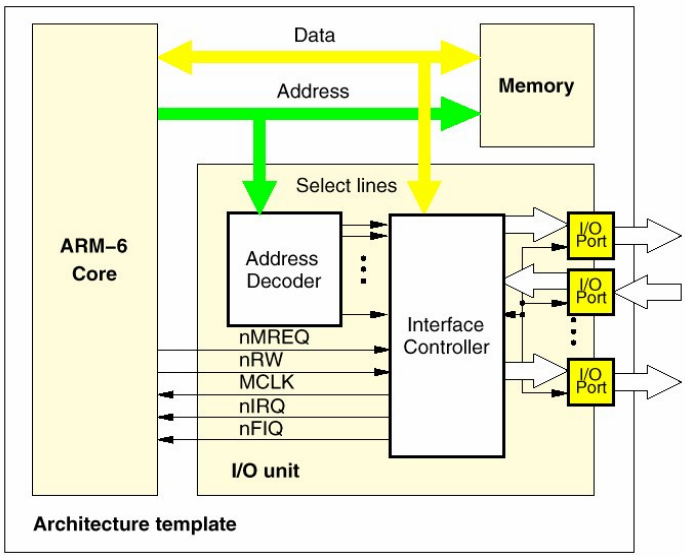


Figure 22. Le modèle architectural pour un processeur de logiciel [113]

Malgré ses avantages, le grand inconvénient de cette approche est la limitation du protocole de communication exclusivement au protocole d'attente synchronisée. Ainsi le réseau de

communication est limité au réseau point à point. En plus l'efficacité et le surcoût de l'unité d'E/S restent inconnus. Enfin ils n'ont pas considéré le compromis M/L pour l'architecture de communication.

Nous pensons que les architectures et les méthodologies existantes manquent d'aspects génériques et abordent ainsi seulement un champ restreint d'applications. La plupart de ces travaux limite les types de composants utilisés et/ou le réseau de communication à peu de modèles spécifiques et/ou propriétaires conçus pour être connectés ensemble. A notre connaissance, il n'y a pas une seule approche permettant la conception systématique d'AMM dédiées à des applications spécifiques avec l'intégration de tout type de composant et réseau de communication. Notre but étant de définir une telle approche, la section suivante présente sommairement l'approche proposée qui sera ensuite détaillée dans le reste du document.

2.3 Approche de conception proposée

L'approche de conception proposée est basée sur trois éléments : un modèle multiprocesseur générique, une méthodologie d'exploration d'architectures et un flot systématique d'implémentation. Ces trois éléments seront développés en détail dans les trois chapitres qui suivent (un par chapitre).

Maîtriser la complexité des AMM nécessite la définition d'un modèle générique sur lequel le flot de conception sera basé. Ce modèle doit permettre la conception d'AMM dédiées à des applications spécifiques. Il doit également supporter l'intégration des éléments architecturaux existants (cœurs de processeurs, blocs dédiés et réseaux de communication).

La partie exploration d'architectures est aussi primordiale dans un environnement de conception complet. Ceci est d'avantage vrai dans le contexte d'architectures multiprocesseurs où l'espace des solutions architecturales est énorme.

Enfin, un flot systématique d'implémentation partant d'un niveau d'abstraction plus élevé que le RTL est le seul espoir pour réduire le temps de conception. Il doit permettre un raffinement automatique de l'application vers une AMM, au niveau RTL, dédiée à cette application.

La Figure 23 présente le flot de conception proposé. Dans l'étape d'exploration d'architecture, en partant de la description de l'application au niveau système, et par l'assistance d'un outil d'estimation de performance, l'espace de solution architecturale est exploré pour trouver l'architecture système optimale pour l'application spécifique. Dans un flot parfait, les contraintes de temps d'exécution, de surface, de consommation, et de coût (s'ils existent) doivent être prises en compte durant cette étape d'exploration d'architecture. Le résultat de cette étape est de fixer les paramètres architecturaux (optimaux) dédiés à l'application. Ces paramètres sont utilisés dans la seconde étape –qui est l'étape d'implémentation– pour produire l'architecture RTL. Cette étape comporte trois types d'actions : la conception des composants logiciels, la conception des composants matériels et la conception du réseau de communication permettant d'intégrer les composants de base. Cette étape est réalisée de façon systématique basée sur l'instanciation et la configuration de composants dans une bibliothèque.

Ce flot considère trois niveaux d'abstraction : le niveau système, le niveau macro-architecture et le niveau micro-architecture. Au niveau système le circuit est spécifié au niveau

des transactions entre les éléments de calcul : un ensemble de modules hiérarchiques et de processus communiquant par des protocoles de communication de haut niveau par l'intermédiaire de canaux abstraits. Un canal peut cacher des protocoles de communication de haut niveau et des primitives de communication qui manipulent des types de données abstraits. A ce niveau, les modules peuvent être décrits en utilisant différents langages et/ou en utilisant différents niveaux d'abstraction. La co-simulation est utilisée pour valider la fonctionnalité du système. Ce modèle peut être utilisé dans l'étape de conception de système pour explorer l'espace des solutions architecturales et fixer les grandes lignes de l'architecture.

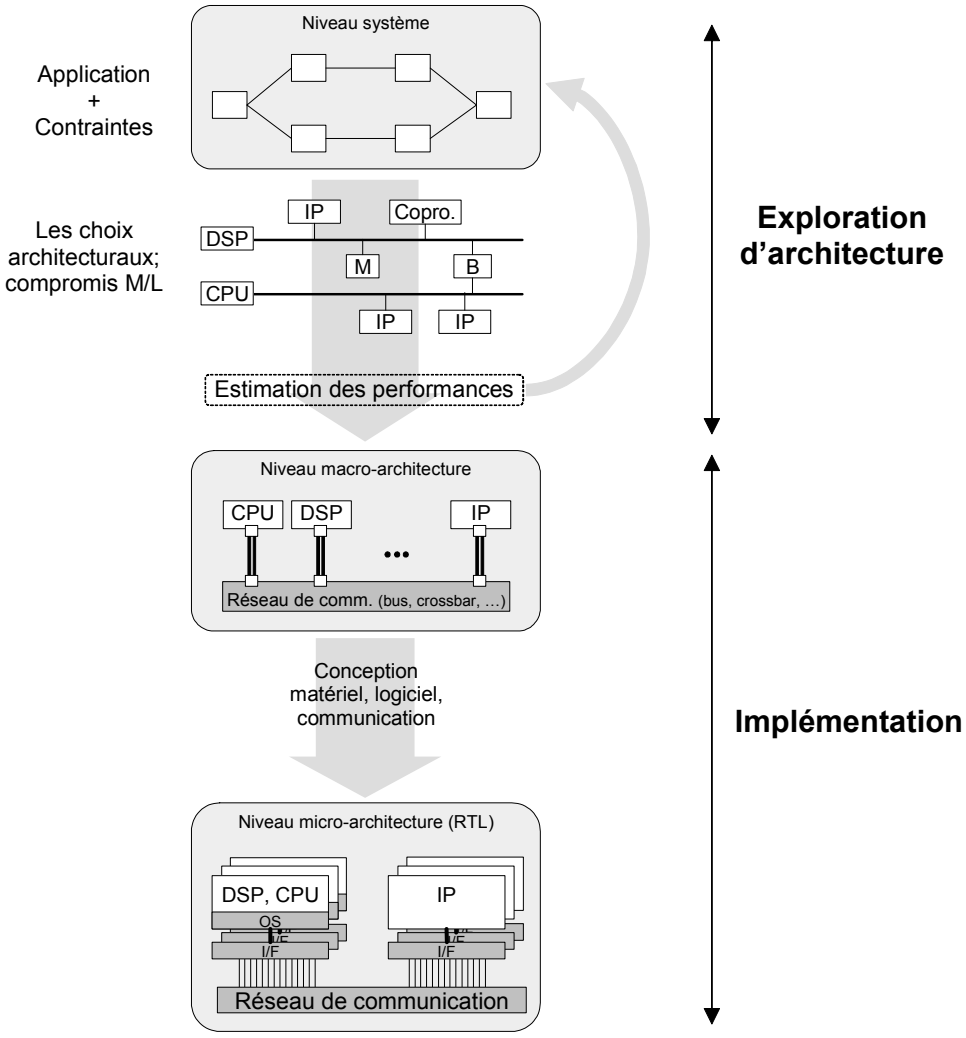


Figure 23. Flot du système au RTL

Le deuxième niveau d'abstraction représente une architecture abstraite qu'on appelle macro-architecture, ce modèle est composé d'un ensemble de modules liés ensemble par des fils logiques. Chaque module représente un processeur dans l'architecture finale. Ceci peut être un processeur logiciel (par exemple : un DSP ou un microcontrôleur exécutant le logiciel), un processeur matériel (un composant spécifique) ou un IP existant (mémoire globale, périphérique, contrôleur de bus, etc.). Les fils logiques sont des canaux abstraits qui transfèrent des types de données fixes (par exemple : nombre entier, virgule flottant) et peuvent cacher des protocoles de bas niveau (par exemple : poignée de main, transfert en mode rafale). Les différents modules peuvent être décrits en utilisant un ou plusieurs langages. La co-simulation

peut être utilisée pour valider ce découpage architectural. Cette macro-architecture est utilisée pour réaliser les composants matériels, les composants logiciels et l'intégration des différents composants.

Le troisième niveau d'abstraction est le niveau RTL (ou micro-architecture). Le système à ce niveau contient tous les détails de la communication entre les composants. Les couches de communication logicielles peuvent comporter un système d'exploitation spécifique (OS). Les couches de communication matérielles comportent les bus et autres dispositifs permettant d'acheminer les informations entre les composants. Les blocs matériels sont raffinés au niveau du cycle d'horloge. Finalement, les blocs existants (souvent appelés IP de l'anglais Intellectual Property) sont enveloppés dans des interfaces afin de les adapter aux bus et réseaux de communication utilisés. La connexion entre les différents blocs matériels est faite par les fils physiques qui implémentent les protocoles choisis. Les composants logiciels communiquent entre eux et avec l'extérieur via des appels système à l'OS.

L'objectif du groupe SLS est de définir une approche complète couvrant tous les aspects de ce flot. Cette approche doit être systématique et par conséquence permettre le développement d'outils CAO. Les trois points qui seront développés dans le reste de ce document sont : la définition d'un modèle architecturale générique (chapitre 3), une méthodologie d'exploration d'architectures au niveau système (chapitre 4) et un flot systématique d'implémentation (chapitre 5).

Chapitre 3

Modele architectural multiprocesseur modulaire, flexible et extensible

Sommaire

Le chapitre précédent montre l'intérêt des AMM dédiées à des applications spécifiques. Ainsi l'architecture doit correspondre juste aux besoins de l'application. Ces besoins varient largement selon le domaine d'application. Ce qui amène à utiliser, selon l'application, divers composants architecturaux (processeurs hétérogènes, blocs matériels dédiés, DSP spécialisés, le réseau de communication Octagon, le réseau SPIN, le µNetwork de Sonics, etc.).

Ce chapitre définit un modèle architectural multiprocesseur générique qui permet d'accommoder tous les types de composants architecturaux de calcul et de communication. Ce modèle est :

- modulaire pour maîtriser la complexité,
- flexible pour s'adapter rapidement aux changements imposés par l'environnement,
- extensible pour traiter une large classe d'applications de différentes tailles
- capable de permettre une conception systématique pour réduire le temps de mise sur le marché.

3.1 Définition du modèle architectural

3.1.1 Schéma d'organisation

L'analyse effectuée sur l'état de l'art (2.2.1) a permis de définir une structure commune à toutes les AMM (Figure 10). Vu les motivations présentées ci-dessus, on a défini le modèle architectural générique représenté par la Figure 24. Dans ce modèle multiprocesseur, les composants de traitement sont dissociés du réseau de communication via des interfaces de communication. Ces interfaces adaptent les composants de traitement au réseau de communication et jouent le rôle de coprocesseurs de communication.

Concernant le logiciel (qui dépasse la portée de cette thèse), une couche de système d'exploitation est définie [35] pour accommoder la communication entre le logiciel applicatif et le matériel. Le contrôle est distribué sur les différents composants qui constituent l'architecture multiprocesseur.

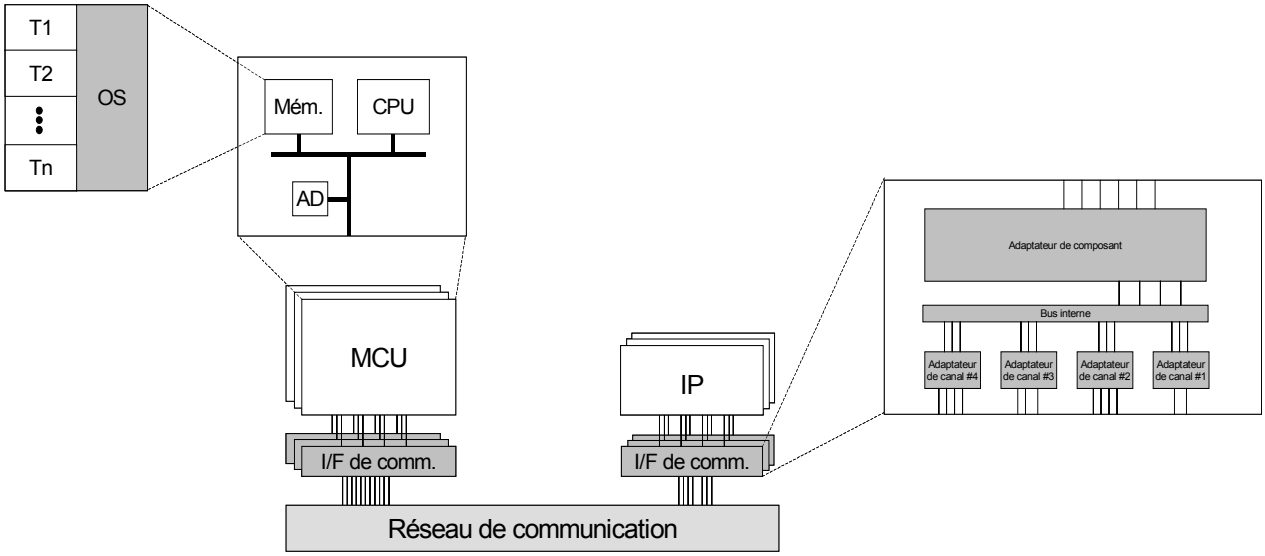


Figure 24. Le modèle architectural générique propose

3.1.2 Composants de base

3.1.2.1 Composants de traitement logiciel

Il n'y a pas de limitations concernant le type des processeurs. Cette catégorie inclut les microprocesseurs disponibles sur le marché, les microcontrôleurs, les DSPs et les processeurs dédiés à des applications spécifiques (ASIPs). Les attributs demandés par ce modèle d'architecture pour cette catégorie de composants peuvent être vérifiés par la plupart d'entre eux. Le processeur devrait pouvoir communiquer par un bus de mémoire externe (synchrone ou asynchrone). Il devrait également pouvoir manipuler des interruptions externes. Les ports additionnels d'entrée-sortie et les périphériques internes peuvent être utiles mais pas nécessaires. Nous supposons que ce sont les cœurs des processeurs qui sont utilisés dans le cas de systèmes monopuces. Nous utiliserons le terme CPU dans ce document, par abus de langage, pour designer le cœur d'un microprocesseur, d'un microcontrôleur, d'un DSP ou d'un ASIP.

3.1.2.2 Composants matériels spécifiques

Le modèle architectural proposé permet l'intégration de blocs matériels préconçus (IPs). Les composants matériels sont nécessaires dans certaines applications pour des contraintes de performance (codage, cryptage, modems, etc.). On fait recours de plus en plus dans les systèmes actuels aux blocs IPs du fait des contraintes de temps de conception. Ces IPs sont fournis avec des interfaces standards (OCP, VC et FI de VSIA, etc.) ou propriétaires.

3.1.2.3 Mémoires

Les mémoires sont intégrées en tant que blocs préconçus. Ainsi pour employer un bloc de mémoire seuls ses modes d'accès et de synchronisation sont exigés. Un contrôleur de mémoire avec un décodeur d'adresse est ajouté pour connecter ce mémoire au bus du processeur (mémoire locale de programme et de données) ou au réseau de communication (mémoire partagée). Des travaux en cours dans le groupe SLS traitent les mémoires partagées et les méthodes d'optimisation de mémoire [82].

3.1.2.4 Réseau de communication

Aucune limitation n'existe concernant le type du réseau de communication. Notre objectif est de supporter tout type de réseau de communication : point-à-point, AMBA, CoreConnect, Octagon, Sonics, SPIN, bus hiérarchique ainsi que tout autre type de communication. Ceci est nécessaire pour pouvoir supporter une très large classe d'application. C'est une voie qui n'a pas encore été totalement explorée.

3.1.3 Interfaces de communication

Un point clé de ce modèle d'architecture est l'utilisation d'un modèle générique d'interface pour adapter les composantes au réseau de communication sur puce. Ces interfaces agissent en tant que coprocesseurs de communication, permettant de dissocier les CPUs et IPs du réseau de communication. Ces interfaces de communication peuvent être très simples ou très sophistiquées selon le composant à adapter et le réseau de communication. En fait l'utilisation de telles interfaces dissocie naturellement le composant du réseau de communication. Chaque interface de communication agit en tant que coprocesseur pour le CPU ou l'IP correspondant. Il peut également être vu comme un pont entre le bus interne du CPU (ou IP) et le réseau de

communication. Aussi, les protocoles de communication qui peuvent être employés n'étant pas limités à un modèle spécifique, notre interface de communication peut supporter des protocoles divers et complexes de communication. Ces interfaces peuvent être conçues par assemblage systématique de peu de cellules de base. Nous avons besoin d'un adaptateur de composant pour chaque type de CPU ou IP et d'un adaptateur de canal pour chaque type de protocole utilisé.

Chaque interface de communication se compose de trois parties (Figure 25) : (1) adaptateur de composant, (2) adaptateurs de canaux de communication et (3) bus interne reliant ces deux derniers. Cette interface agit comme un pont (bridge) entre le composant et le réseau de communication.

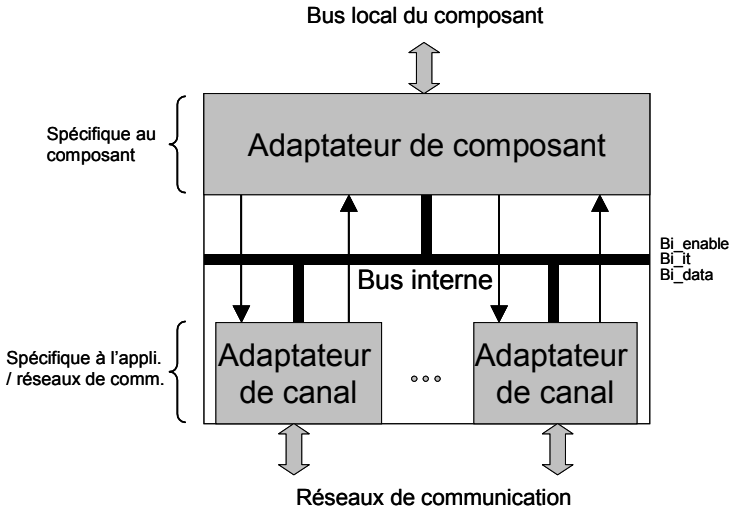


Figure 25. Interface de communication

3.1.3.1 Adaptateur de composant

Il est spécifique au composant, adaptant son bus pour le connecter au reste du système. À chaque type de CPU (ou IP) correspond un adaptateur. Cet adaptateur est mis dans une bibliothèque (bibliothèque d'interfaces de communication) pour être réutilisé. Dans le cas d'un CPU, l'adaptateur se connecte au bus du CPU. Il adapte le comportement des accès de lecture/écriture et permet l'envoi d'interruptions au CPU. La Figure 26 montre un exemple d'une interface de communication d'un ARM7 avec quatre contrôleurs de canaux. Pour construire l'adaptateur de ce CPU il faut étudier la spécification de son bus (différents modes d'exécution, signaux, interruptions, etc.) ainsi que les chronogrammes d'accès lecture/écriture. De cette étude nous avons conçu un bloc VHDL RTL contenant un FSM de contrôle d'accès lecture/écriture, un contrôleur d'interruption, et un décodeur d'adresse. Ce bloc adapte le bus du ARM7 au bus interne (3.1.3.3). Il est mis dans une bibliothèque d'interface de communication pour sa réutilisation. De la même façon on peut rajouter à la bibliothèque un nouvel adaptateur pour chaque nouveau composant. L'adaptateur de composant est conçu une fois pour toutes.

3.1.3.2 Adaptateur de canal de communication

Cette partie sera connectée directement au réseau de communication. L'adaptateur de canal de communication dépend du réseau de communication et du protocole utilisé dans

l'application. A chaque type de protocole et à chaque topologie de réseau correspond un adaptateur de canal. Le nombre d'adaptateurs de canaux instanciés dans une interface de communication dépend de l'application (de cette façon l'architecture résultat est taillée sur mesure pour l'application). Par exemple, dans la Figure 26, l'adaptateur de canal AC1 est un adaptateur de sortie correspondant à un protocole poignée de main avec FIFO et le réseau de communication est de type point-à-point. Cet adaptateur a été décrit en VHDL RTL sous forme d'un bloc paramétrable (taille du FIFO) et il a été ajouté à la bibliothèque d'interface de communication. Ce bloc sera instancié et configuré spécifiquement à l'application pour chaque nouvelle application qui en aura besoin. Si l'adaptateur n'existe pas dans la bibliothèque, il faudra le construire (de façon paramétrable) et l'ajouter à la bibliothèque. La complexité de l'adaptateur dépend de la complexité du protocole et du réseau de communication, et il n'y a pas de restrictions sur le type de ces deux éléments.

3.1.3.3 Bus interne

La troisième partie est un simple bus générique se composant d'un bus de données de largeur variable et des signaux de commande (Figure 26). Les signaux de commande sont les signaux de validation (*enable*) des adaptateur de canaux (un seul adaptateur à la fois peut écrire sur le bus de donnée), et de signaux de demande d'interruptions. Ce bus permet la modularité de l'interface en dissociant les deux parties : adaptateur de composant et adaptateurs de canaux de communication.

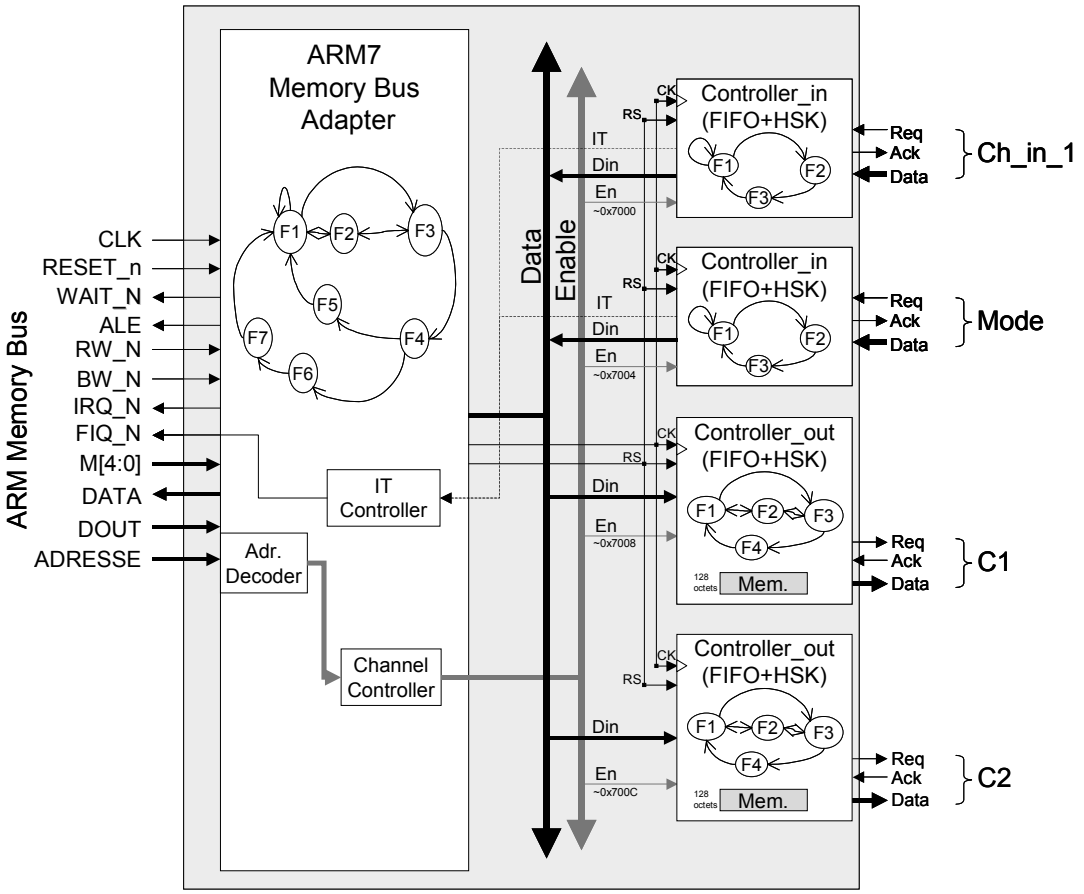


Figure 26. Exemple d'interface de communication

3.1.4 Partie logicielle

La partie logicielle sort du cadre de cette thèse. Une autre thèse en parallèle est consacrée à cette partie [35]. Pour chaque CPU de l'architecture, une couche de système d'exploitation est conçue pour assurer l'exécution multitâche et pour accommoder la communication entre ces tâches et la communication externe (entre processeurs). L'idée est de laisser le logiciel applicatif intact sans modification. Ceci est assuré en utilisant une bibliothèque d'appels système (APIs) lors de la création du logiciel applicatif. Ainsi trois couches de système d'exploitation sont conçues (Figure 27) : une couche instanciant les codes des APIs utilisés dans les tâches, une couche assurant les services requis de système d'exploitation (ordonnanceur (*schedular*), routines d'interruptions, routines d'entrée/sortie, etc.), et une couche de pilotes de matériel (*Drivers*) assurant la communication directe avec le matériel des interfaces de communication.

Notons que cette couche de système d'exploitation peut aussi être conçue de façon dédiée à l'application et en parfaite concordance avec l'interface de communication. Ainsi un compromis matériel/logiciel peut être effectué (par exemple un FIFO et son contrôleur peuvent être matériel, logiciel, ou un compromis matériel/logiciel).

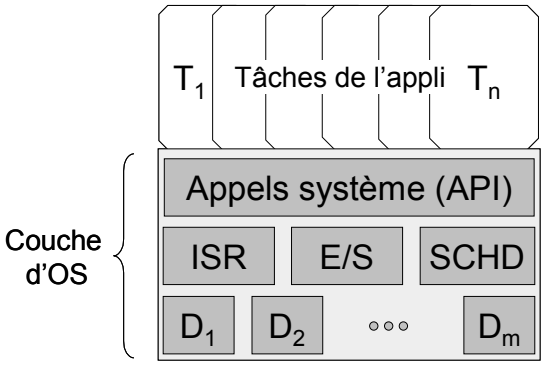


Figure 27. Partie logicielle

3.2 Abstraction du modèle architectural

3.2.1 Séparation entre comportement et communication

Lors de la conception de système multiprocesseur sur puce, comme des processeurs hétérogènes multiples et divers protocoles de communication sont utilisés, l'intégration de système devient une étape très importante dans la conception de système. Le concept fondamental est de permettre que le processus d'intégration de système, c.-à-d. le raffinement de la communication, soit fait indépendamment du raffinement comportemental. Pour faire cela, le comportement et la communication sont séparés dans la spécification du système. Ainsi la communication dans le système peut être raffinée indépendamment de la partie comportementale du système.

Le raffinement de la communication se compose (1) de la conception d'interface de communication et (2) de la conception de réseau de communication.

3.2.2 Niveaux d'abstraction utilisés

Comme le but est d'arriver à concevoir une architecture dédiée à l'application partant d'une spécification de haut niveau de cette application, quatre niveaux d'abstraction différents ont été définis [84] : le niveau service, le niveau fonctionnel, le niveau macro-architecture et le niveau micro-architecture (RTL). Le Tableau 2 compare les caractéristiques de ces différents niveaux, et la Figure 28 donne un exemple pour chaque niveau.

Tableau 2. Les niveaux d'abstraction définis

Niveau d'abstraction	Service	Fonctionnel	Macro-architecture	Micro-architecture
Comportement	Objets concurrents	Transactions partiellement ordonnées	Pas de calcul	Cycles
Communications	Requêtes / Services	Passage de messages	Transmission de données / événements	Valeurs de bits
Exemples de langages	CORBA/UML	SDL, SystemC	VHDL, SystemC	VHDL, SystemC

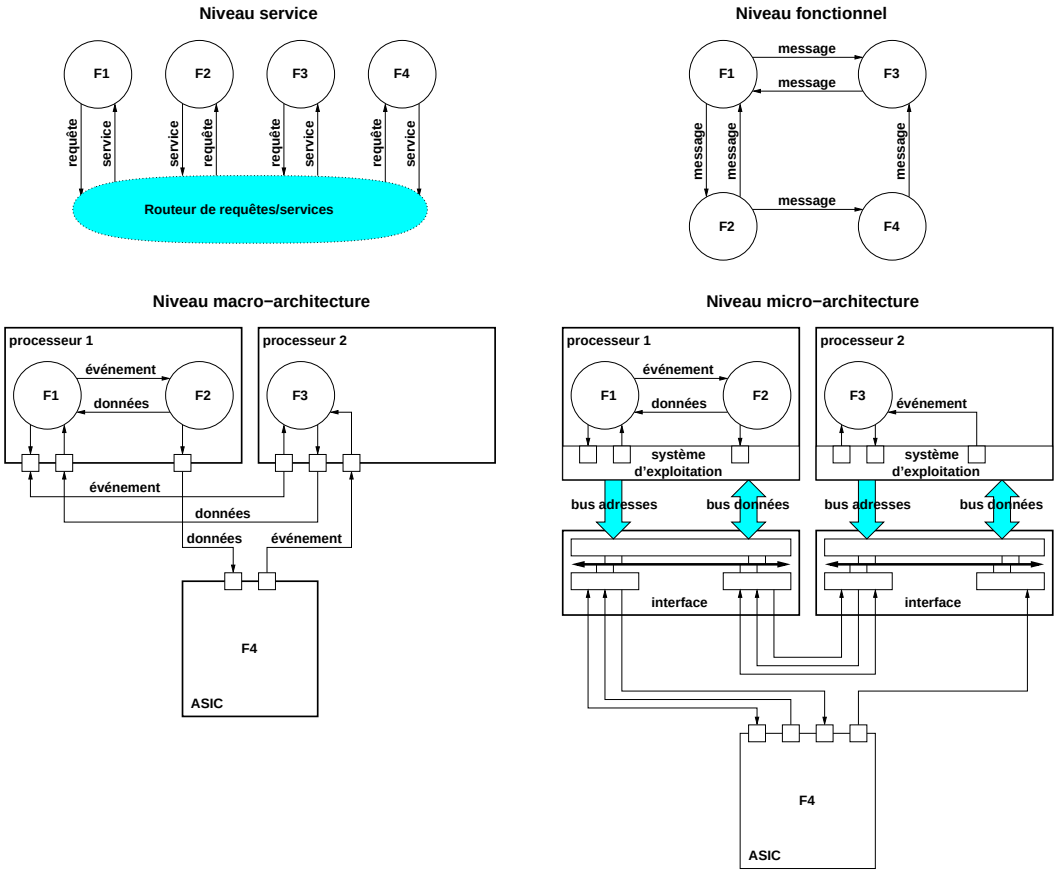


Figure 28. Exemple de description pour chaque niveau d'abstraction

Au niveau service, les objets peuvent interagir anonymement par le biais de requêtes et de services. Seules les fonctionnalités sont définies. Au niveau fonctionnel, les connexions entre les objets sont définies. Au niveau macro-architecture, les éléments d'architecture sont définis : les processeurs, coprocesseurs, réseau de communication, etc. Au niveau micro-architecture, tous

les détails de réalisation sont définis, comme les interfaces de communications des processeurs et les systèmes d'exploitation.

Dans l'exemple de la Figure 28, une application est présentée aux quatre niveaux d'abstraction. Cette application est composée de quatre fonctions (ou tâches) F1, F2, F3 et F4. Au niveau service elles communiquent anonymement grâce au routeur de requêtes. Au niveau fonctionnel, les connexions entre les tâches sont explicitées, par exemple F1 envoie et reçoit des messages avec F2 et F3 mais F4 n'envoie des messages qu'à F3, et n'en reçoit que de F2. Au niveau macro-architecture, l'architecture globale a été décidée : F1 et F2 sont sur le processeur 1, F3 est sur le processeur 2 tandis que F4 est un ASIC. Finalement, au niveau micro-architecture, les détails locaux sont aussi explicités : les processeurs disposent de leurs interfaces de communications matérielles et logicielles (systèmes d'exploitation). Les ports des processeurs et de l'ASIC sont réels et non plus ceux de la communication entre les tâches.

Le niveau service n'est pour l'instant pas traité par le flot. Le flot actuel commence par le niveau fonctionnel pour arriver jusqu'au niveau micro-architecture. Ce dernier est souvent appelé niveau transfert de registres ou RTL (Register Transfer Level). Le flot d'implémentation sera présenté au chapitre 5.

3.3 Définition de plateforme architecturale

3.3.1 Introduction

Nous allons voir dans le chapitre suivant sur l'exploration d'architectures que l'espace de solution architecturale est énorme si on considère tous les réseaux de communication, protocoles, et processeurs existant dans la nature et supportés par notre modèle architectural lors de la conception d'une nouvelle application. On peut résumer ces paramètres en quatre groupes :

Paramètres des composants de traitement

- Type des composants (ARM7, 68000, PowerPC, DCT, décodeur Viterbi, MPEG4, etc.)
- Nombre des instances pour chaque type
- Taille de la mémoire locale (RAM, ROM), et plan de mémoire

Paramètres du réseau de communication

- Type du réseau de communication
- Interconnexions entre les composants

Paramètres des interfaces de communication

- Nombre d'interconnexions du composant avec le réseau de communication
- Type du protocole pour chaque canal de communication
- Paramètres de chaque protocole choisi (type des données, taille du FIFO, adresses et interruptions allouées, etc.)

Paramètres des mémoires partagées

- Taille de la mémoire
- Schéma d'arbitrage et plan de la mémoire

Avec tous ces paramètres, l'espace de solutions est immense (chapitre 4). Aussi nous proposons de réduire cette espace en vue de permettre une exploration efficace. Le terme

plateforme architecturale sera utilisé dans ce document pour designer un sous-ensemble de composants de base constituant l'architecture. Ce regroupement orientera le concepteur qui désire implémenter son application selon le domaine d'application (téléphonie mobile, multimédia, réseau, etc.), ce qui réduit l'espace de solution architecturale à explorer. Ainsi les paramètres d'une plateforme architecturale se limitent à :

Paramètres des composants de traitement

- Nombre des instances pour chaque type (les types sont préfixés)
- Taille de la mémoire locale (RAM, ROM), et plan de mémoire

Paramètres du réseau de communication

- Interconnexions entre les composants (le type est préfixé)

Paramètres des interfaces de communication

- Nombre d'interconnexions du composant avec le réseau de communication
- Type du protocole pour chaque canal de communication
- Paramètres de chaque protocole choisi (type des données, taille du FIFO, adresses et interruptions allouées, etc.)

Paramètres des mémoires partagées

- Taille de la mémoire
- Schéma d'arbitrage et plan de la mémoire

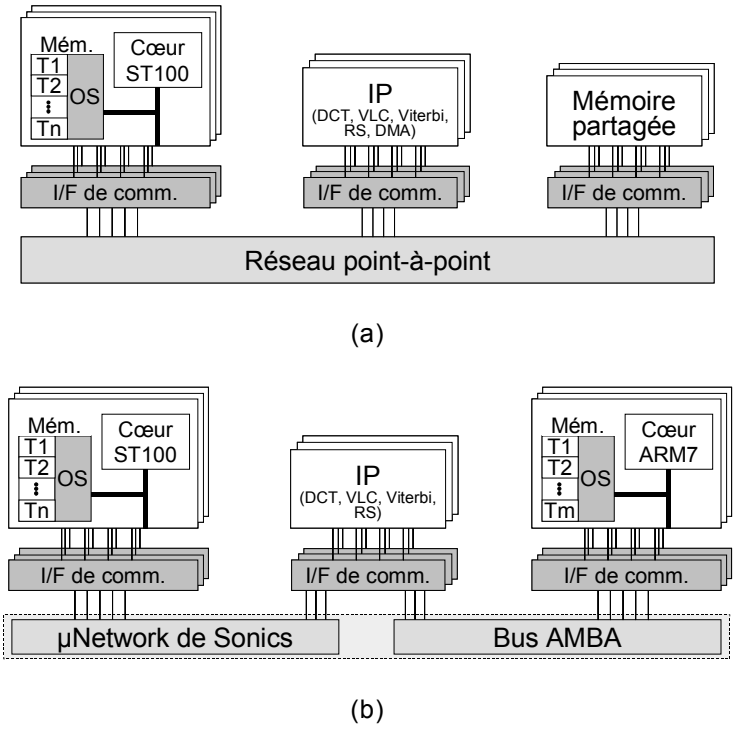


Figure 29. Exemples de plates-formes architecturales

Ainsi les types des composants et le type de réseau de communication sont préfixés (selon le domaine d'application par exemple) ce qui réduit considérablement l'espace de solutions et aide le concepteur dans ses choix architecturaux. La Figure 29 montre deux exemples de plateforme

architecturale issues de notre modèle architectural. Par exemple, dépendant du domaine d'application, le concepteur choisit la plateforme architecturale de la Figure 29(a) qui comporte un réseau de communication point-à-point, ou celle de la Figure 29(b) qui comporte un réseau de communication de haut débit et un bus partagé.

3.3.2 Plateforme architecturale utilisée dans les exemples d'application

Durant cette thèse nous avons pu construire une petite bibliothèque de composants et de réseau de communication, petite du fait de la contrainte de temps et de disponibilité des composants. Ainsi dans les applications que nous avons conçues, nous avons utilisé la plateforme de la Figure 30. Avec cette plateforme et les analyses effectuées sur les applications conçues nous avons pu montrer l'efficacité de notre modèle architectural et de notre approche.

C'est un exemple typique de plateforme architecturale issue de notre modèle architectural, faite de N processeurs (processeurs ARM7 et M68000). Le réseau de communication est un réseau point à point.

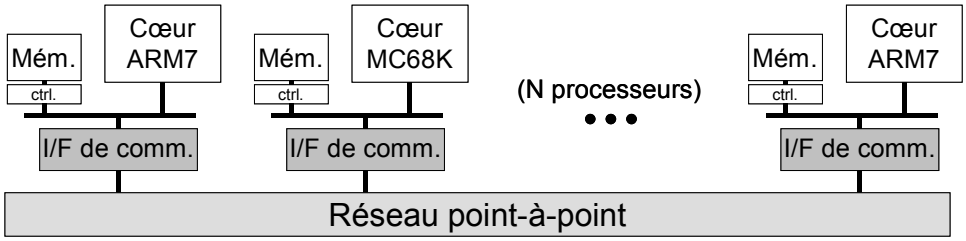


Figure 30. Plateforme multiprocesseur utilisée dans nos applications.

Le choix des processeurs a été basé sur la disponibilité. Ils étaient les deux seuls processeurs auxquels nous avions accès quand nous avons commencé ce projet.

Les paramètres de cette plateforme architecturale qui peuvent être configurés par le concepteur sont le nombre de CPUs, les capacités des mémoires pour chaque processeur, les ports d'entrée-sortie pour chaque processeur et les interconnexions entre les processeurs, les protocoles de communications et les raccordements externes (périphériques). Ces paramètres montrent l'extensibilité de la plateforme et permettent la conception d'architectures dédiées à des applications de différentes tailles. L'interface de communication dépend des attributs du processeur correspondant et des paramètres dédiés à l'application à concevoir (nombre de canaux, protocoles).

3.4 Analyse de l'efficacité du modèle proposé

3.4.1 Modularité

La modularité est obtenue grâce à l'interface de communication qui permet la séparation entre le comportement des modules et la communication entre ces modules. Cette qualité du modèle architectural permet la conception séparée des différents modules voire l'importation de modules externes préconçus (IPs). Elle devient indispensable pour maîtriser la complexité des systèmes actuels.

3.4.2 Flexibilité

La flexibilité est définie comme la facilité de s'adapter à de nouvelles conditions imposées par le concepteur, l'utilisateur ou la technologie. Elle est obtenue grâce à l'utilisation de modèles génériques et grâce à la modularité et à la façon systématique d'assembler l'architecture.

3.4.3 Extensibilité

L'extensibilité, qui est définie comme la capacité d'incrémenter le nombre de composants. Elle est également due à la modularité et à la stratégie d'assemblage systématique. Elle permet d'adapter le modèle proposé d'architecture à des applications de différentes tailles et complexités. Les seules restrictions ici sont l'extensibilité du réseau de communication choisi et la technologie.

3.4.4 Automatisation

La modularité et la stratégie d'assemblage des composants du modèle propose permet l'automatisation de cet assemblage. C'est à dire l'automatisation de l'interconnexion des composants, enveloppes, réseau de communication. Nous avons montré en 3.2 comment abstraire notre modèle architectural. Cette abstraction permet de même une automatisation du raffinement vers l'architecture RTL finale et cette automatisation devient indispensable pour répondre à la forte contrainte de temps de mise sur le marché.

3.4.5 Validation

Un lien vers la validation a été réalisé [121] en se basant sur ce modèle. Les auteurs [121] ont montré comment en utilisant ce modèle ils arrivent à générer de façon automatique un modèle exécutable pour la cosimulation de l'architecture, et ceci à différents niveaux d'abstraction.

3.4.6 Performance

La qualité du design obtenu avec notre modèle dépendra de la qualité des bibliothèques de composants utilisés. Il permettra des performances équivalentes à celles obtenues avec une conception manuelle taillée sur mesure car la flexibilité des interfaces de communication permet toutes les configurations et c'est à la charge du concepteur de prendre les choix architecturaux optimaux. Il n'y aura pas de surcoût en surface ni en temps d'exécution.

3.5 Conclusion

Dans ce chapitre un modèle multiprocesseur flexible, modulaire, et extensible a été présenté. Ce modèle permet de couvrir un très large domaine d'application. Il est basé sur le concept de séparation entre comportement et communication et permet surtout une conception systématique d'AMM dédiées à des applications spécifiques basée sur l'assemblage de composants de bibliothèques. Une définition de plateformes architecturales a été également présentée. Ces plateformes architecturales sont basées sur le modèle architectural proposé, et dédiées chacune à une classe d'applications. Leur définition permet de diminuer l'espace de solutions architecturales à explorer. Enfin, une analyse de l'efficacité de ce modèle a été présentée en conclusion.

Chapitre 4

Exploration d'architectures[3]

Sommaire

Ce chapitre traite la première partie du flot (Figure 23) qui est l'exploration de l'espace des solutions architecturales. Le but est d'assister le concepteur dans le choix de l'architecture système qui satisfait d'une façon optimale les contraints de l'application (fonctionnelles et non fonctionnelles). Ainsi, nous proposons une nouvelle méthodologie d'estimation de performance au niveau système permettant une exploration rapide de l'espace d'architectures. Cette méthodologie est basée sur un outil de codesign (MUSIC). Elle exploite les deux niveau d'abstraction système et RTL, et se base sur un modèle hybride qui combine les deux approches

[3] Le travail présenté dans ce chapitre a été réalisé en collaboration avec Wander Cesario et fait partie des thèses des deux auteurs.

d'analyse : statique et dynamique, afin d'obtenir un bon compromis entre la rapidité et la précision. L'avantage est que l'estimation de performance peut être faite à une vitesse à peu près identique à celle de la simulation fonctionnelle du système. Par conséquent un grand nombre de solutions architecturales peut être exploré dès le début du processus de conception. L'efficacité de la méthodologie proposée est illustrée par un exemple d'application significatif. Les résultats expérimentaux montrent les grands avantages de cette méthodologie.

4.1 Introduction

4.1.1 Motivations et objectifs

L'exploration de l'espace de solutions architecturales est une composante primordiale dans un flot de conception complet de systèmes multiprocesseurs. Le problème à résoudre dans le flot de conception des systèmes complexes consiste à trouver la meilleure architecture du système incluant le découpage fonctionnel du système, la détermination des protocoles de communication, la topologie du réseau de processeurs et le placement/ordonnancement. Pour un système de n processus, le nombre de partitions différentes que l'on peut construire est donné par l'équation (4-1).

$$U_n = \sum_{q=1}^{n} \sum_{i=0}^{q} \frac{(-1)^{q-i}.(i)^n}{(q-i)!.(i)!} \tag{4-1}$$

Si maintenant nous supposons que nous disposons de p types de processeurs différents (matériels/logiciels), le nombre d'architectures possibles sera donné par l'équation (4-2).

$$V_n^p = \sum_{q=1}^{n} p^q \sum_{i=0}^{q} \frac{(-1)^{q-i}.(i)^n}{(q-i)!.(i)!} = \sum_{i=1}^{n} i^n \frac{p^i}{i!} \sum_{j=0}^{n-i} \frac{(-p)^j}{j!} \tag{4-2}$$

La Figure 31 montre la variation du nombre d'architectures possibles quand le nombre de processus et le nombre de types de processeurs varie de 1 a 10. A titre d'exemple, pour un modèle composé de 10 processus et une technologie comportant 3 types de processeurs (deux types de microprocesseurs pour le logiciel et la réalisation matérielle) l'espace des solutions contient $4{,}872.10^7$ éléments. Nous remarquons que ce nombre s'accroît exponentiellement avec n et p. Si nous considérons encore que nous disposons de c protocoles de communication, le nombre d'architectures sera encore plus énorme. En plus, le temps de synthèse et de cosimulation au niveau RTL d'une architecture peut prendre quelques jours. Donc, nous ne pouvons pas nous permettre de synthétiser et de cosimuler, au niveau RTL, chaque architecture pour évaluer sa performance. Ces chiffres ont été la base de notre motivation pour les travaux présentés dans ce chapitre.

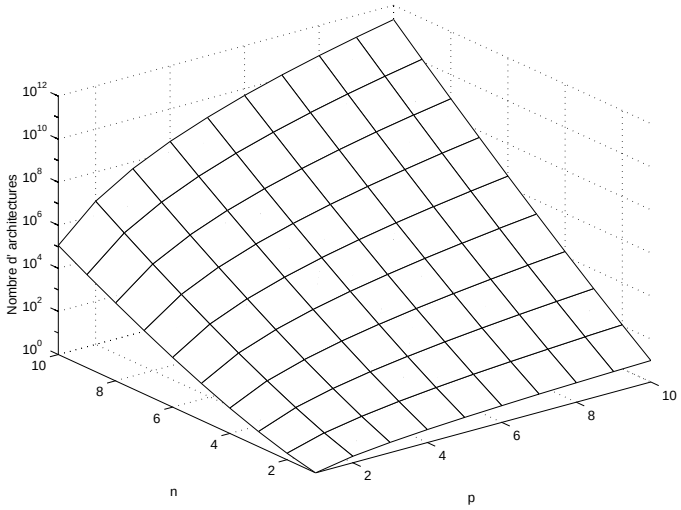


Figure 31. Le nombre d'architectures possibles

D'ici vient le besoin d'un outil d'estimation de performances suffisamment rapide pour effectuer cette tâche d'exploration d'architectures. Un outil d'estimation au niveau système qui soit assez précis sera un très bon candidat pour répondre à nos besoins de rapidité et de précision. La combinaison d'un tel outil avec un outil de codesign et un outil de cosimulation constituera un environnement parfait pour la conception conjointe matérielle/logicielle de systèmes complexes. L'objectif est, donc, de trouver une méthode d'estimation de performance qui soit rapide et précise et qui s'intègre facilement dans un flot de codesign pour assister le concepteur dans son choix architectural.

4.1.2 L'état de l'art concernant l'estimation de performance

Les méthodes d'estimations que l'on trouve dans la littérature peuvent être classées dans trois catégories: statiques, dynamiques et mixtes :

- **Dynamique :** les mesures de performance d'une architecture est le résultat de l'exécution d'un modèle (exemple : simulation).
- **Statique :** l'estimation de performance d'une architecture est le résultat d'une analyse statique d'une spécification (exemple : analyse de chemins dans une spécification de flot de contrôle).
- **Mixte dynamique/statique :** c'est l'utilisation de quelques éléments des deux approches précédentes pour l'analyse de performance d'une architecture.

Les approches dynamiques sont en général très précises. Leur inconvénient majeur est le temps nécessaire pour l'obtention du modèle à simuler (synthèse, génération, compilation), ainsi que le temps de la simulation. Ce qui les rend en pratique inutilisables dans le contexte particulier de l'exploration où le nombre de modèles à analyser est énorme. D'un autre coté, les approches statiques sont certes très rapides (pas de génération de modèles à simuler, ni de simulation), mais les tâches de modélisation et d'estimation sont complexes à cause de la distance qui sépare les concepts de spécification de l'implémentation.

Les critères considérés lors de la conception des systèmes électroniques sont en général le temps d'exécution, la surface et la consommation. La majorité de la littérature limite les

méthodes proposées à un de ces critères. De plus seules des solutions partielles traitant une partie d'un système matériel/logiciel complet sont proposées. Ainsi plusieurs études se concentrent sur une analyse fonctionnelle de l'énergie consommée dans un microprocesseur [8][90], un DSP [67]ou un FPGA [32]. Des modèles précis de consommation des caches [59], des RAM [49] et du logiciel [60][74] sont présentés. GAUT_W [80] propose des modules d'estimation, au niveau comportemental, de la consommation [28] et des interconnexions [54]. Cet outil permet l'exploration d'architectures au niveau comportemental et, ainsi, il permet la synthèse de circuits VLSI optimisés en surface et en consommation dans le cas des applications de télécommunications. Il propose également une approche mixte dynamique/statique pour l'estimation à un haut niveau de la consommation d'un algorithme transposé sur un cœur de DSP [67]. Enfin plusieurs travaux existent concernant l'estimation du temps d'exécution [12][26][118][79][31][103].

Dans le cadre de cette thèse, seul le critère de temps d'exécution est considéré. Ainsi le terme estimation de performance sera utilisé dans le reste de ce document pour designer l'estimation du temps d'exécution. D'autre part peu de travaux visant l'analyse de performance en vue de l'exploration d'architectures pour la conception conjointe matérielle/logicielle existent dans la littérature. Cependant, beaucoup de travaux ont été réalisés pour résoudre des problèmes séparés comme l'estimation du temps d'exécution du logiciel, l'analyse de performance des circuits ASIC ou encore l'architecture de systèmes complexes. Nous pouvons classer les travaux existants dans ce domaine dans deux catégories selon la complexité de l'architecture cible : les travaux visant des architectures cibles monoprocesseur et les travaux visant des architectures cibles multiprocesseurs.

4.1.2.1 Les travaux visant des architectures cibles monoprocesseur

Dans cette catégorie on peut citer PMOSS [26], COSYMA [118][42], et LYCOS [79]. L'architecture cible est monoprocesseur (une seule unité de contrôle). Il n'y a donc pas de difficultés liées au parallélisme par rapport aux architectures multiprocesseurs. Cependant, les analyses de performance des parties logicielles et matérielles sont réalisées conjointement.

PMOSS [26] se contente de calculer l'accélération due au co-processeur (partie matérielle), sur la performance globale du système. Pour cela, il utilise, pour le logiciel, des analyses statiques (calcul du temps d'exécution basé sur le code assembleur) et dynamiques (profilage). Pour le matériel, il utilise des analyses statiques (calcule du temps d'exécution basé sur la description de la machine de contrôle). Et pour les communications, des analyses dynamiques (profilage), sont utilisées.

COSYMA [118][42] calcule des métriques séparées pour le logiciel, le matériel et la communication. Ensuite ces métriques sont combinées dans des équations particulières pour procéder à une partition basée sur une méthode de recuit simulé (pour *simulated annealing*). Des mesures de temps dans le pire cas sont calculées pour les implémentations logicielles et matérielles en utilisant plusieurs variantes de techniques d'analyse de chemins. Le temps de communication est estimé pour un modèle particulier (mémoire partagée).

LYCOS [79] procède à des estimations de performance en utilisant des techniques de profilage et d'estimations de temps d'exécution à bas niveau pour le matériel, le logiciel et la communication.

Malgré leur performance, ces méthodes ne permettent pas de traiter des architectures complexes pouvant contenir plus qu'un seul processeur.

4.1.2.2 Les travaux visant des architectures cibles multiprocesseurs

Dans cette catégorie nous trouvons SpecSyn [31], VCC (ancien Polis) [103] et la méthode créée par Yen et al [119]. L'architecture cible est multiprocesseur complexe.

SpecSyn [31] admet des architectures avec un nombre quelconque de microprocesseurs et de co-processeurs. L'approche utilisée pour l'estimation de performance est mixte statique/dynamique. Elle est faite en deux étapes :

- Pre-estimation : elle est réalisée avant la phase d'exploration d'architectures. Un profilage de la description du système est réalisé pour obtenir des temps d'exécution pour différents niveaux (processus, bloc de base, communication).
- Estimation en ligne : elle est faite durant la phase d'exploration d'architectures. Les résultats obtenus durant la phase de pre-estimation sont utilisés dans des expressions complexes pour le calcul de la performance globale du système.

Le problème d'une telle approche est son incapacité à capturer les changements dynamiques du comportement temporel durant la phase d'exploration d'architectures. Car durant cette phase, des méthodes statiques sont utilisées (le temps global est la somme des temps d'exécution partiels des différentes ressources d'exécution). Par exemple, cette méthode n'est pas capable d'estimer le temps d'attente d'un processus pour qu'un autre finisse son exécution. Le passage sur un tel comportement dynamique peut introduire une grande imprécision sur les résultats de l'estimation.

VCC [103] est capable de surmonter le problème mentionné ci-dessus (capture du comportement dynamique), en combinant une simulation de haut niveau avec des estimations de bas niveau (approche statique/dynamique). Cette approche est similaire à notre approche, mais l'estimation du logiciel manque toujours de précisons dû à l'utilisation d'un modèle générique de processeur. En plus, le niveau d'abstraction du langage de spécification à l'entrée de VCC est plus bas que dans le cas de notre approche.

Yen et al [119], attaquent le problème d'un point de vue générique. Ils analysent, au niveau système, l'interaction entre les différents processus en donnant le meilleur et le pire délai pour chacun d'entre eux. Ensuite, en partant d'un graphe acyclique représentant les dépendances de données entre les processus, et à l'aide d'informations sur le partitionnement (la distribution sur les unités de traitement), ils calculent le temps d'exécution, dans le pire cas, pour le système entier. Cette méthode est précise et capable de prendre en compte les délais de communications. Malheureusement, elle est limitée à des applications pour lesquelles il est suffisant de connaître les délais dans le pire cas. De plus, les processus doivent être représentables par graphes acycliques.

L'estimation de performance peut être faite sur plusieurs niveaux d'abstraction. En effet, dans un environnement de conception conjointe matérielle/logicielle, nous partons d'un niveau d'abstraction système pour arriver au niveau d'implémentation RTL. Au niveau RTL, l'analyse de performance se caractérise par une grande précision, mais elle consomme beaucoup de temps. En remontant dans les niveaux d'abstraction, le temps de l'analyse de performance diminue, mais la précision diminue également. Aussi, avec l'écart important entre les deux niveaux

d'abstraction : système et RTL, l'analyse de performance au niveau système peut devenir très imprécise voir inexploitable.

4.1.3 Contribution

Afin d'exploiter les avantages de toutes les approches, nous proposons de combiner les deux niveaux d'abstractions (niveau système et niveau implémentation), tout en utilisant un modèle hybride statique/dynamique. Notre objectif est d'obtenir une méthodologie d'estimation de performance au niveau système à la fois rapide et précise. L'idée sous-jacente est d'utiliser, d'une part, l'approche dynamique pour l'évaluation des tâches qui ne peuvent être déterminées qu'à l'exécution (événements dépendants des données). D'autre part, utiliser l'approche statique (modélisation et estimation), pour l'évaluation du reste du système.

Une nouvelle méthode pour l'estimation du temps d'exécution d'une application à partir de sa spécification au niveau système a été développée. Cette méthode nous permet d'effectuer l'exploration de l'espace d'architectures au niveau système. Elle s'intègre bien dans le flot du système de codesign MUSIC (Figure 32). Dans ce flot nous partons d'une spécification du système en SDL (SDL_1) et nous générons une architecture matérielle/logicielle. Notre méthode d'estimation/exploration est faite en deux étapes :

1. **La génération de la bibliothèque de performance :** le processus de conception conjointe doit être exécuté deux fois pour produire les deux réalisations de référence : tout en logiciel et tout en matériel. Le précalcul des délais d'exécution partiels en fonction de la technologie est réalisé à partir de ces deux réalisations de référence. Ces données sont stockées dans une bibliothèque de performance ;
2. **L'exécution du modèle de performance :** le partitionnement logiciel/matériel et les choix des canaux de transmission sont employés pour créer un modèle de performance exécutable. Les informations de délai de la bibliothèque de performance sont annotées dans ce modèle selon les choix réalisés pendant le partitionnement logiciel/matériel ;

L'estimation de performance pour une architecture donnée consiste à enrichir la spécification du système en SDL par des annotations temporelles extraites d'un passage de cette spécification dans l'outil de codesign MUSIC. La nouvelle spécification annotée (SDL_2), et l'outil de simulation d'ObjectGEODE (de **Verilog**) nous permettent l'exploration de l'espace des architectures d'une façon très rapide et avec une grande précision. La méthode d'estimation/exploration que nous proposons est limitée au critère de temps d'exécution. Cependant, elle peut facilement s'étendre à d'autres critères de performance.

Le reste de ce chapitre est organisé en quatre sections. Dans la section 4.2, nous allons présenter l'environnement de codesign dans lequel nous avons développé et validé notre méthodologie d'estimation/exploration. Cette méthodologie sera détaillée dans la section 4.3. La validation de cette méthodologie par un exemple d'application sera présentée dans la section 4.4. Une analyse appropriée des résultats sera également présentée dans cette dernière section.

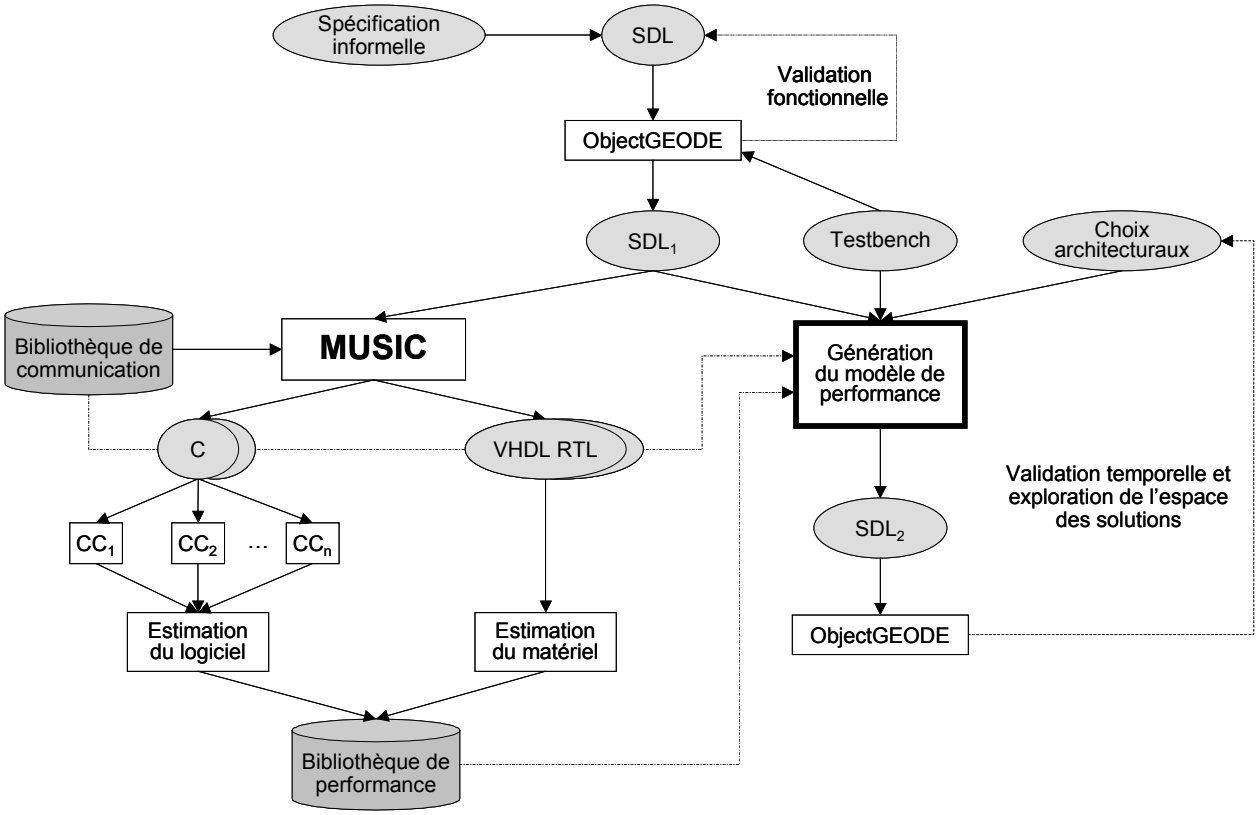


Figure 32. Le flot d'estimation de performance basé sur un outil de codesign (MUSIC)

4.2 Méthodologie de codesign basée sur SDL

Dans cette section nous présentons l'environnement de conception dans lequel nous avons développé et validé notre méthodologie d'estimation/exploration. Nous exposerons tout d'abord l'outil de codesign MUSIC. Ensuite, nous allons donner une brève présentation du langage SDL. En fin, nous présentons brièvement l'environnement d'ObjectGEODE, et nous décortiquons le module d'analyse de performance introduit récemment au simulateur SDL.

4.2.1 L'outil de codesign MUSIC

MUSIC est un outil de conception conjointe matérielle/logicielle qui a été développé au sien du groupe SLS, puis transférer à la compagnie Arexsys [4] (commercialisé sous le nom ArchiMate). Il utilise une approche multi-langages pour la conception et la validation de systèmes hétérogènes. La Figure 33 illustre le flot général de la conception conjointe matérielle/logicielle. Nous pouvons le résumer en trois grandes étapes :

- **Modélisation du système** : Il s'agit de spécifier la fonctionnalité désirée du système. Le système est décrit dans le langage de spécification au niveau système SDL. Cette spécification est validée en utilisant le simulateur *geodesim* d'ObjectGEODE. Le résultat de cette étape est la génération d'une spécification fonctionnelle, libre de tous les détails de réalisation.
- **Choix et synthèse de l'architecture** : Il s'agit d'explorer les alternatives de conception pour identifier celles qui satisfont au mieux les contraintes du système. Cette étape réalise la transposition des fonctions du système sur des processeurs (logiciels et matériels), interconnectés. La synthèse de la communication consiste en le choix des

protocoles de communication et la traduction des primitives de communication à un niveau de détail accepté par les outils de synthèse. C'est dans cette étape qu'interviennent les principaux choix architecturaux. Et c'est pour assister le concepteur dans la réalisation de cette étape que nous avons développé notre méthodologie d'estimation/exploration d'architectures.

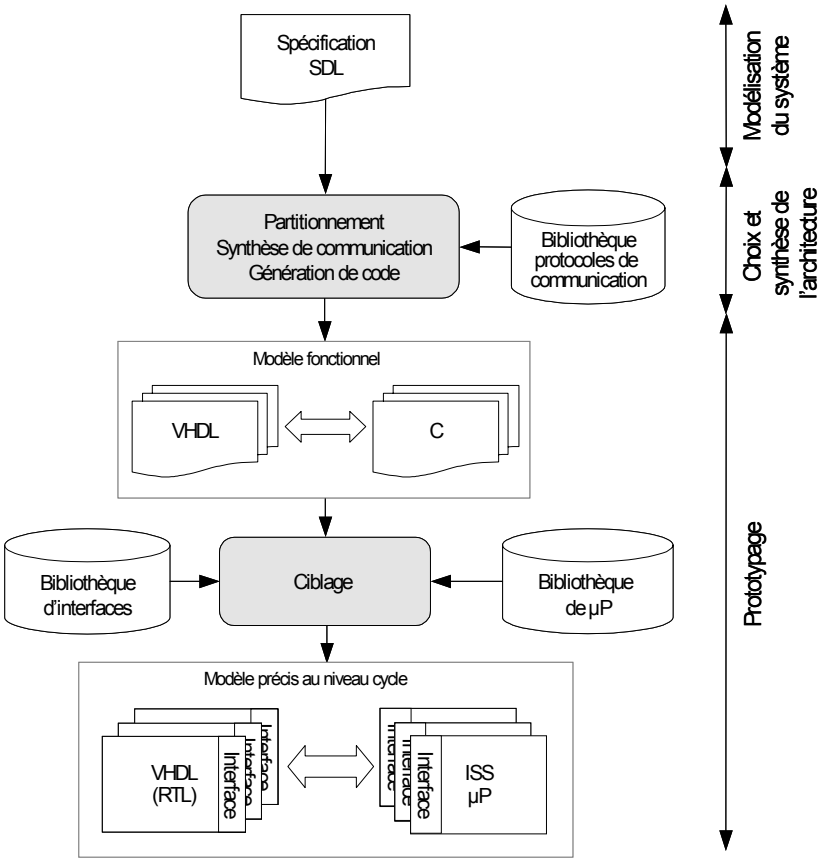


Figure 33. Flot de conception conjointe matérielle/logicielle de MUSIC

- **Prototypage** : C'est la génération du prototype physique final. Cette étape est divisée en étapes intermédiaires dans lesquelles des prototypes virtuels sont générés. Un premier prototype virtuel consiste dans la génération de code C pour les modules logiciels et du code VHDL pour les modules matériels et la génération des interfaces de communication. La fonctionnalité de ce prototype est validée par l'outil de cosimulation MCI [44]. Ensuite, un prototype virtuel à un niveau d'abstraction très bas est généré. Il s'agit de générer du code VHDL RTL pour les parties matérielles, et de compiler les modules C pour obtenir du code binaire ciblé sur les microprocesseurs (ou microcontrôleurs) choisis dans l'étape précédente. Ce dernier prototype contient les détails nécessaires sur la performance du système. Avec l'outil MCI, qui intègre des simulateurs de VHDL (VSS, Leapfrog...), et des simulateurs d'instructions d'assembleurs (ISS), nous validons le prototype obtenu. Cette validation est précise au niveau cycle. Par conséquent, le prototype physique est prêt à être fabriqué.

Les modèles de spécification, à l'entrée du flot de conception, sont traduits dans un format intermédiaire appelé SOLAR [55]. SOLAR est basé sur des machines d'états finis étendues qui communiquent à travers des appels de procédures à distance. Ce format, qui est utilisé pendant

tout le reste des étapes de conception, préserve l'indépendance de ces dernières face aux modèles de spécification. MUSIC est doté d'une interface graphique qui facilite l'interaction avec le concepteur. Il laisse au concepteur la prise de décisions, tout en y assistant efficacement. Toutes les transformations sont exécutées automatiquement par MUSIC. La flexibilité du processus de synthèse est assurée par l'interaction continue avec l'utilisateur et par l'utilisation d'une bibliothèque de modèles de communication très riche. En fin, la possibilité de valider le système tout au long du flot de conception, grâce à l'outil de cosimulation MCI, est un atout très précieux.

4.2.2 Le langage SDL

Le langage SDL (pour *Specification and Description Language*) [27], est dédié à la modélisation et à la simulation des systèmes temps réel distribués pour les télécommunications. Il est standardisé par l'ITU [51]. Un système décrit en SDL est composé d'un ensemble de processus concurrents communicants à travers des signaux. Le langage SDL supporte les différents concepts permettant la description des systèmes (la structure, le comportement et la communication).

4.2.2.1 La structure du langage SDL

La structure statique d'un système décrit en SDL est hiérarchique. L'entité la plus haute de la hiérarchie est appelée « *système* ». Une instance de système contient un ensemble d'instances de « *blocs* ». Une instance de bloc peut contenir d'autres instances de blocs ou un ensemble d'instances de « *processus* » en utilisant une hiérarchie de blocs. Un bloc peut contenir d'autres blocs ou bien un ensemble de processus. Les différents processus d'un même bloc sont connectés entre eux et jusqu'à la frontière du bloc par des « *routes* ». Les blocs sont connectés entre eux par des « *canaux* ». Les routes et les canaux sont des vecteurs de « *signaux* ». Les signaux échangés par les processus suivent un chemin composé de routes et de canaux. SDL offre, également, des structures dynamiques tel que la création et la destruction de processus, ou l'adressage dynamique. La Figure 34 illustre la structure d'un système SDL.

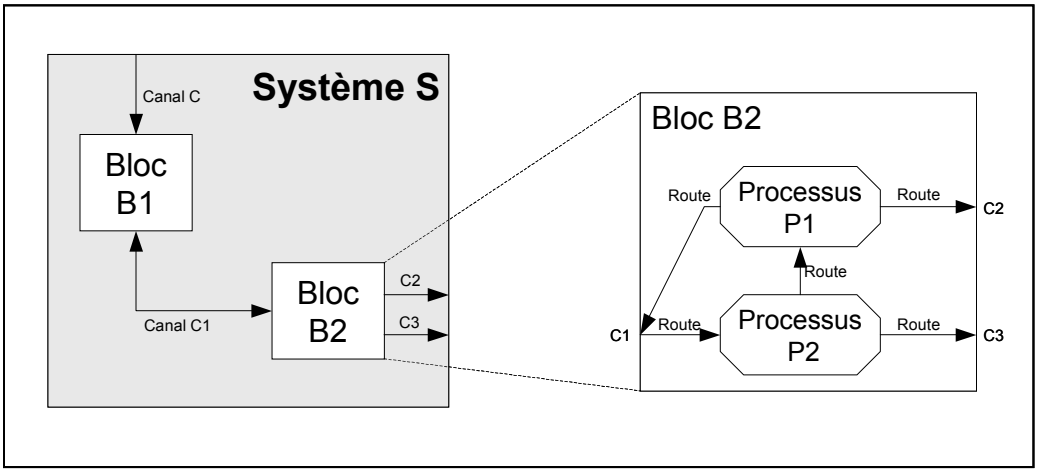


Figure 34. La structure d'un système SDL : blocs, processus, routes, canaux et signaux

4.2.2.2 Le comportement du système

Le comportement d'un système est représenté par la combinaison des comportements de l'ensemble de processus autonomes et concurrents du système. Un processus est décrit par un automate d'états finis qui communique avec les autres processus, à travers des signaux, de manière asynchrone (voir Figure 35). Chaque processus est composé d'un ensemble d'états et de transitions. Il possède une « *file d'attente* » en entrée, de type FIFO (*First-In-First-Out*) de taille infinie, dans la quelle les signaux sont stockés à leur arrivée. L'arrivée d'un signal attendu dans la file d'attente détermine et valide la transition à exécuter. Le signal qui a initié la transition est retiré de la file d'attente et le processus peut alors exécuter un ensemble d'opérations telles que la manipulation de variables, des appels de procédures ou émission de signaux.

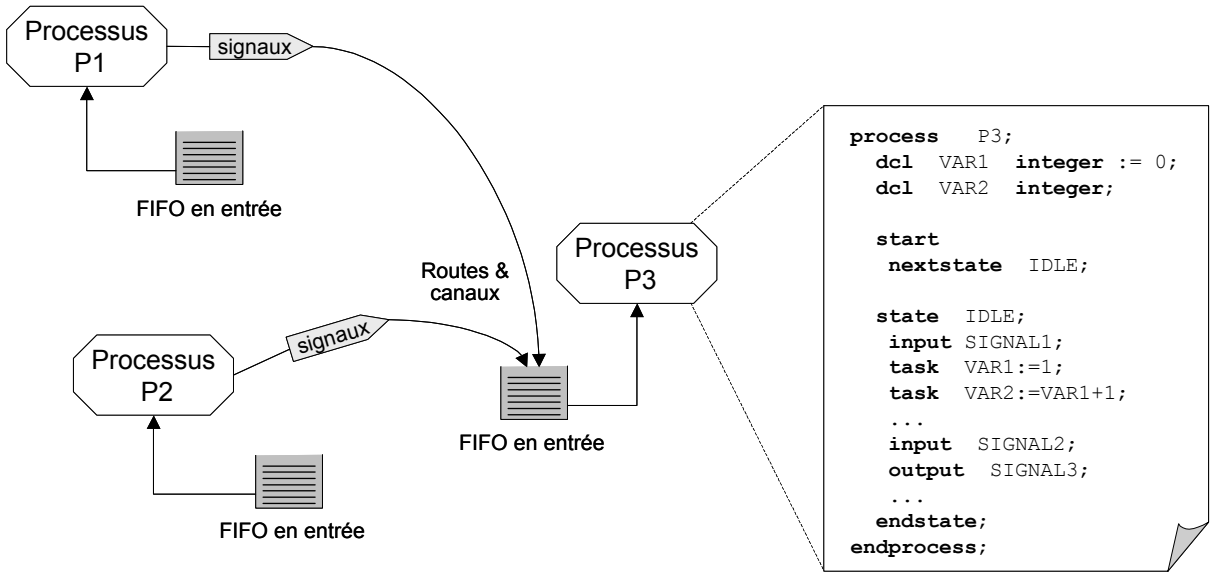


Figure 35. Le comportement d'un système SDL

4.2.2.3 La communication inter-processus

La communication inter-processus est basée sur le modèle de « *passage de messages* ». Les signaux sont l'unique moyen de synchronisation inter-processus. Un signal transporte toujours implicitement l'adresse du processus émetteur, l'adresse du destinataire si elle est spécifiée explicitement ainsi qu'un ensemble éventuel de paramètres. Deux autres concepts sont utilisés comme support de transfert des signaux : les routes et les canaux. Les routes assurent la connexion des processus du même bloc et se terminent à la frontière du bloc. Les canaux connectent les blocs entre eux. Tous les deux peuvent être unidirectionnels ou bidirectionnels. Pour réaliser une communication entre deux processus situés dans des blocs différents, les signaux doivent emprunter les canaux et les routes. Un canal peut être connecté à plusieurs routes, alors qu'une route ne peut être connectée qu'à un seul canal. Une communication à travers une route s'effectue en un temps nul, alors qu'à travers un canal, elle s'effectue en un temps non déterministe. Les délais et l'ordre d'arrivée des signaux utilisant deux chemins différents ne peuvent pas être prédits.

4.2.3 L'estimation de performance à partir du modèle SDL

Grâce au module d'analyse de performance qui a était introduit dans le simulateur d'ObjectGEODE, il est devenu possible d'utiliser ce simulateur pour l'évaluation de performance d'un système décrit en SDL. Ici, nous introduisons l'environnement utilisé pour le développement de modules en SDL, ObjectGEODE de **Verilog**. Nous allons particulièrement décortiquer le module d'analyse de performance.

4.2.3.1 L'environnement ObjectGEODE

L'environnement d'ObjectGEODE résulte de la combinaison de l'environnement GEODE pour SDL et de l'environnement LOV pour OMT (pour *Object Modeling Technique*) [53], tous les deux en provenance de la société **Verilog**. ObjectGEODE est composé des outils suivants :

- Un environnement SDL, incluant un éditeur graphique, un simulateur et un générateur de code.
- Un environnement OMT, incluant un générateur de squelette C++.
- Un éditeur MSC (*Message Sequence Chart*, International Telecommunication Union (ITU) recommandation Z.120).
- Un outil de suivi de projet, permettant de diviser une description OMT en plusieurs fichiers.

4.2.3.2 L'environnement SDL

L'éditeur fournit une interface graphique de développement de systèmes en SDL. Le simulateur s'appuie sur la sémantique du langage SDL (ITU recommandation Z.100 et Z105). Cependant, quelques différences existent lorsque les recommandations Z.100 (ou Z.105) sont ambiguës, et lorsque l'interprétation par le simulateur implique l'introduction de concepts ou de vocabulaires qui ne sont pas décrites dans la recommandation Z.100. Exemples de telles différences :

- La concurrence est simulée par l'exécution entrelacée des différents processus.
- Les types de données sont interprétés selon le modèle du langage Pascal à la place du modèle algébrique proposé dans les recommandations Z.100 et Z.105.
- Les extensions SDL.

Le simulateur offre les options suivantes :

- La trace d'une simulation peut être conservée pour être exécutée de nouveau.
- La commande « *undo* » permet de revenir en arrière dans l'exécution.
- Les extensions SDL :
  - → Les modes de communications (rendez-vous, diffusion et diffusion sélective).
  - → Le module d'analyse de performance.
  - → La possibilité d'intégrer de code externe sous la forme de types et fonctions en C (types de données abstraites).

4.2.3.3 Le module d'analyse de performance

Le langage SDL (recommandation Z.100 et Z105) ne contient pas de notions de temps. Aussi, les canaux de communication sont basés sur un protocole FIFO de taille infinie. Donc, le langage, tel que définit par les standards, n'est pas adapté à l'estimation de performance ni à la modélisation d'architectures. Cependant, un module d'analyse de performance a été développé comme un nouveau composant du simulateur d'ObjectGEODE. Il s'agit de nouvelles extensions SDL qui ne sont pas reconnues que par le simulateur d'ObjectGEODE, et qui permet à ce dernier d'exécuter des commandes spécifiques pour l'évaluation de performance et la modélisation d'architectures.

4.2.3.3.1 L'introduction de la performance dans le langage SDL

Pour que l'introduction de la performance dans le langage SDL n'affecte pas les outils déjà existants (par exemple : l'éditeur SDL), les extensions SDL ne doivent pas modifier la syntaxe standard de SDL. Ceci est réalisable grâce aux directives (COMMENT phrases), qui sont reconnues uniquement par le simulateur d'ObjectGEODE. La forme générique de ces extensions est la suivante :

```
<special comment> ::=   COMMENT  ' [ <free comment prefix> ]
                        <directive>
                        [ <free comment suffix> ] '
<directive>       ::=   # <directive> [ ( <directive parameters> ) ]
```

Un des points les plus forts de cette approche est que ces directives peuvent être attachées aux actions (TASK) à l'intérieur d'une transition SDL. Cela nous permet de réaliser des estimations relativement précises (à bas niveau) par rapport aux résultats obtenus avec un outil comme OPNET (outil commercial développé par **MIL3**) [5]. Dans ce dernier, les caractéristiques de performance sont attachées à des entités de haut niveau, par exemple : les blocs en SDL.

Du point de vue sémantique, les nouvelles extensions de SDL ont un sens beaucoup plus puissant que les « *timers* » qui existent déjà en SDL. Ces derniers donnent l'accès à une horloge de référence globale pour mesurer le temps actuel (global). En plus, ils ne permettent pas de modéliser le partage de ressources d'exécution entre les différentes entités SDL. Un tel partage est devenu modélisable grâce à la directive NODE. Dans la section suivante nous présentons les nouvelles directives NODE, PRIORITY et DELAY.

4.2.3.3.2 Les nouvelles directives pour l'analyse de performance

La directive NODE permet d'identifier les ressources d'exécution d'un modèle, que l'on appelle « *nœuds* ». Un nœud peut être associé aux systèmes, types de système, blocs, types de bloc, processus, types de processus. Tous les processus à l'intérieur d'un nœud partagent les mêmes ressources d'exécutions. Les processus qui ne sont pas à l'intérieur d'un nœud (explicitement), sont considérés comme des nœuds (par défaut). La syntaxe de cette directive est la suivante :

```
<node directive> ::= #node
```

La directive PRIORITY permet d'assigner un ordre de priorité à chaque processus SDL, à l'intérieur d'un même nœud, pour ordonnancer leur exécution. Le choix d'une transition entre toutes les transitions valides de toutes les instances de processus à l'intérieur d'un nœud (i.e. la

stratégie d'ordonnancement) suit une distribution aléatoire uniforme. La directive PRIORITY peut être associée à un processus ou à un type de processus, à l'intérieur d'un nœud pour changer ce choix aléatoire. La syntaxe de cette directive est la suivante :

```
<priority directive> ::= #priority (<integer constant expression>)
```

La Figure 36 (SDL en format textuel), montre comment on peut utiliser les directives NODE et PRIORITY. A la simulation, le processus P1 commence toujours le premier car il a une priorité supérieure à celle du processus P2 (par défaut, c'est la priorité la plus petite, zéro). Le processus P2 peut commencer l'exécution seulement quand le processus P1 s'arrête.

```
SYSTEM sys;
  /*BLOCK B1 : P1 et P2*/
  BLOCK B1   COMMENT '#node';
     PROCESS P1 COMMENT '#priority (1)';
    ...                                          Processus 1
     ENDPROCESS;
     PORCESS P2;
    ...                                          Processus 2
     ENDPROCESS;
  ENDBLOCK;
ENDSYSTEM;
```

Figure 36. Les directives NODE et PRIORITY

La directive DELAY est associée aux actions (TASK) en SDL, pour spécifier le temps d'exécution de l'action. Par défaut, le temps d'exécution d'une action est nul. Lors de la simulation, l'arrivée à une action annotée par cette directive cause le blocage du nœud contenant cette action pendant un temps spécifié comme paramètre de la directive. Après ce retard de temps (temps global d'exécution progresse toujours), l'action est exécutée. La syntaxe de cette directive est la suivante :

```
<delay directive> ::= #delay ( <duration expression> )
        | #delay ( <duration expression 1>, <duration expression 2>
              [ , <distribution> ] )
        | #delay ( <duration expression>,  *  [ , <distribution> ] )
<distribution> ::= <distribution NAME> [ ( <constant  expression  list> ) ]
```

La durée de l'action est spécifiée comme paramètre :

- *delay (<duration expression>)* : la durée de l'action est spécifiée par *<duration expression>* ;
- *delay (<duration expression 1>, <duration expression 2>)* : la durée est choisie aléatoirement entre *<duration expression1>* et *<duration expression2>* ;
- *delay (<duration expression>, *)* : la durée est choisie aléatoirement entre *<duration expression>* et une valeur pseudo infinie, définie par une variable d'environnement du simulateur. Cette valeur doit être choisie soigneusement selon les contraintes temporelles définies dans le modèle. Le choix aléatoire de la durée est pris selon la *<distribution>* spécifiée. Les distributions possibles sont les suivantes :

  → Distribution uniforme (par défaut) ;

  → Distribution exponentielle ;

  → Distribution normale.

Les expressions de délai dans une directive DELAY peuvent être actives (i.e. contenir des variables). Donc les contraintes sur la durée d'une action peuvent varier selon les variables manipulées dans d'autres actions. Par exemple, la durée d'une instruction « *output* » peut varier selon les valeurs des paramètres du signal envoyé.

Il y a quelques restrictions dans l'utilisation de la directive DELAY en SDL. La spécification des délais avec une distribution aléatoire empêche l'utilisation de quelques caractéristiques du simulateur, telles que les conditions « *filter* », « *undo/redo* », « *replay* » d'un scénario et les simulations exhaustives.

4.3 Nouvelle méthodologie d'estimation et d'exploration

Le principe de notre méthodologie est de combiner les deux niveaux d'abstractions (niveau système et niveau implémentation), tout en utilisant une approche hybride statique/dynamique. Le but est d'obtenir un compromis entre la rapidité d'exécution et la précision, les deux points de conflit dans une méthodologie d'estimation de performance. La rapidité est assurée par l'utilisation d'une simulation au niveau système. La précision est obtenue aussi par la simulation au niveau système (prise en compte des comportements dynamiques), mais aussi par des informations de bas niveau sur quelques implémentations du système. Dans la suite, nous allons donner une vue d'ensemble sur le cadre dans lequel nous avons appliqué notre méthodologie, ensuite, notre flot d'estimation/exploration sera présenté et les différentes étapes du flot seront détaillées.

4.3.1 Le flot d'estimation/exploration

L'idéal pour avoir une très grande précision, serait d'utiliser la simulation pour chaque événement dépendant de l'exécution, et ceci au niveau d'implémentation. Malheureusement, cette approche n'est pas faisable dans notre contexte, où nous cherchons à explorer largement l'espace d'architectures multiprocesseurs. Donc, nous avons décidé d'utiliser une simulation au niveau système, beaucoup plus rapide. Avec une telle simulation, nous pouvons estimer les comportements dynamiques dus à l'interaction entre les différents processus, indépendamment de la complexité de l'architecture. Pourtant cette simulation de haut niveau ne contient pas d'information sur la performance de l'implémentation. Ici vient l'importance de l'approche statique et des implémentations de très bas niveau. La réalisation et l'analyse de quelques implémentations nous permettront de récupérer des résultats partiels de performance. Ces résultats seront utilisés pour instrumenter la spécification de niveau système à simuler. Avec la nouvelle spécification instrumentée nous pouvons prédire la performance de toutes les architectures réalisables (indépendamment de leur complexité), avec une précision et une rapidité permettant une exploration efficace de l'espace des solutions.

Nous avons développé cette méthodologie d'estimation/exploration dans le cadre de notre environnement de conception MUSIC. Le langage de spécification au niveau système est SDL. La simulation au niveau système est réalisée par le simulateur ObjectGEODE. Le point de départ dans notre flot d'estimation/exploration est la description du système en SDL. La Figure 37 montre les caractéristiques principales de ce flot.

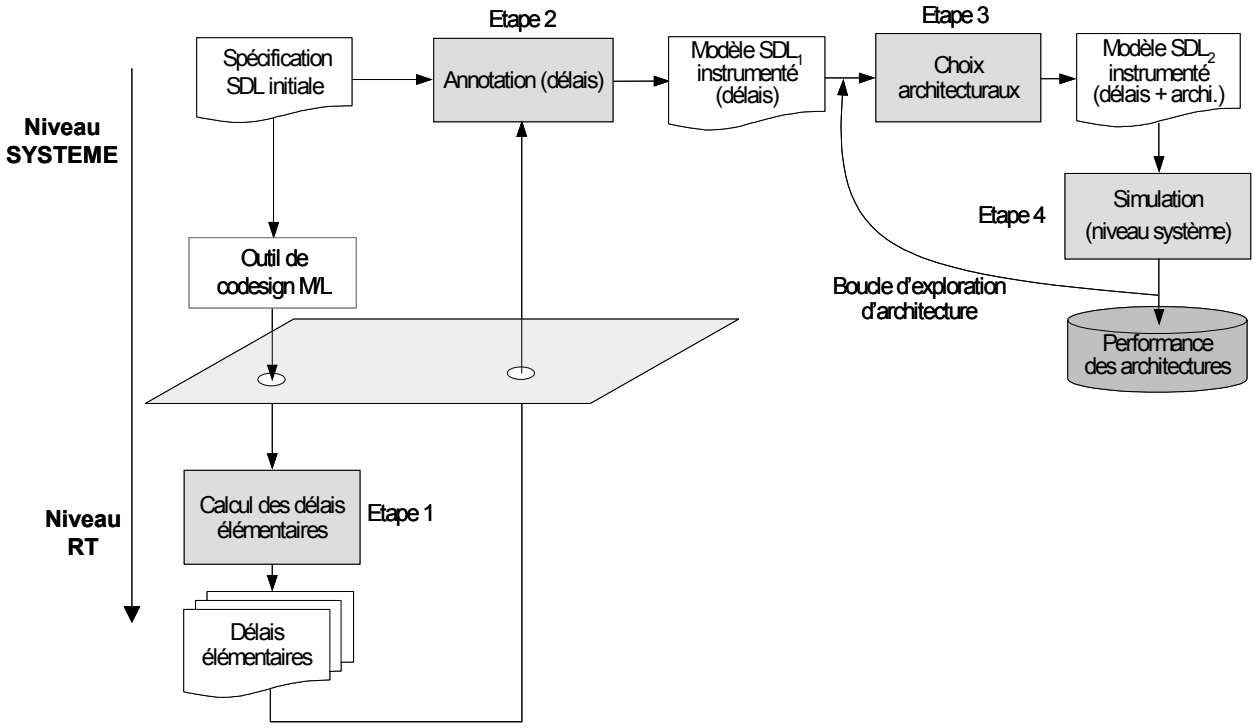


Figure 37. Le flot global d'estimation/exploration

Nous pouvons facilement identifier les quatre étapes suivantes : calculs des délais élémentaires, annotation, choix architectural et simulation. Nous remarquons qu'il s'agit bien d'une approche hybride où nous utilisons deux niveaux d'abstractions très distants : le niveau système et le niveau RT (ou implémentation). La boucle d'exploration d'architectures se situe au niveau système, d'où la rapidité de cette phase. L'aspect statique de notre méthodologie se manifeste dans les étapes de calcul des délais élémentaires et d'annotation. L'aspect dynamique se matérialise par la simulation et les choix architecturaux. Les quatre étapes que nous avons identifiées dans ce flot ne sont pas de complexité égale. Dans la suite de cette section, nous allons développer ces étapes en détail.

4.3.2 L'estimation des délais élémentaires

C'est la première étape et la plus consommatrice de temps. Cependant, c'est aussi un des points les plus forts de notre méthodologie. Elle est basée sur la propriété du simulateur ObjectGEODE de pouvoir attacher des directives DELAY à des actions (TASKs) à l'intérieur d'une transition SDL. Cela nous permet de réaliser des estimations relativement précises (à bas niveau).

En analysant les techniques utilisées par MUSIC pour la génération de code C/VHDL, et en étudiant le comportement de l'exécution de l'implémentation physique (ordonnancement, communication), et le modèle d'exécution du simulateur ObjectGEODE, nous arrivons à obtenir les résultats clés suivantes :

1. Les actions décrites en SDL peuvent être identifiées, relativement facilement, dans le code généré.
2. Nous pouvons regrouper une séquence contiguë d'actions qui ne contiennent ni d'instructions de contrôle (branchements et décision), ni d'instructions de

communication, pour lui attribuer un seul délai global, sans modifier le comportement de l'exécution. Les séquences d'actions qui suivent la définition ci-dessus, sont appelées «*blocs de base*». Ce résultat facilite l'estimation des délais élémentaires (leur nombre diminue considérablement), tout en gardant un degré de fiabilité très élevé.

3. Les communications doivent être estimées à part. Une action de communication sera éventuellement modélisée par une séquence de délais pour se rapprocher du modèle d'exécution de l'implémentation physique. Par exemple : modéliser l'ordonnancement des communications entre différents processus qui s'exécutent sur le même microprocesseur.

Il s'agit d'identifier les blocs de base dans la description du système (SDL), pour ensuite calculer le délai de chacun d'entre eux et ceci pour toute implémentation envisageable. Les délais des communications sont aussi estimés pour tous les protocoles de communications possibles. Les résultats de cette étape constituent une base de donnée suffisante pour la suite des étapes du flot, et notamment, la boucle d'exploration d'architectures. Les délais élémentaires calculés sont stockés en nombre de cycles. Cela nous permet de mettre en paramètre, la fréquence d'horloge de chaque processeur. Donc, nous gagnions plus de souplesse et de réutilisabilité.

4.3.2.1 Les blocs de base

Ici nous allons présenter le flot que nous utilisons pour l'identification et le calcul du temps d'exécution des blocs de base pour toutes les implémentations envisageables. Dans la littérature, très souvent, on définit le bloc de base comme étant une séquence (maximale) d'instructions contiguës qui ne contiennent pas d'instructions de contrôle (branchements et décisions). Dans notre cas, un bloc de base ne contient pas non plus des instructions de communications. Dans un contexte de conception conjointe matérielle/logicielle, chaque processus peut avoir deux options d'implémentation :

- **Implémentation(s) logicielle(s) :** Le comportement du processus, spécifié en SDL, sera traduit en code C. La performance de l'implémentation logicielle dépend du microprocesseur et du compilateur qui seront utilisés. Donc pour chaque couple (microprocesseur, compilateur), nous générons le code assembleur correspondant (niveau registre). L'analyse statique (ou dynamique) de ce code nous permettra d'analyser sa performance. Le nombre d'implémentations logicielles (d'un processus) possibles dépend du nombre de microprocesseurs et de compilateurs disponibles.
- **Implémentation matérielle :** Le comportement du processus, spécifié en SDL, sera synthétisé en un code VHDL RTL. L'analyse statique (ou dynamique) de ce code est suffisante pour extraire les informations de performance. En effet, le code VHDL est généré par MUSIC, et donc il a une certaine forme spécifique, qui nous permet facilement d'analyser sa performance.

La première chose à faire est d'identifier les blocs de base dans la description SDL. La correspondance entre les actions en SDL et les instructions du code généré par MUSIC est très facile. Pour pouvoir identifier les blocs de base dans tous les niveaux d'abstraction (assembleur compris), d'une façon facile et automatisable, nous avons choisi d'ajouter des annotations de référence au début et à la fin de chaque bloc de base, dans la description SDL. Ces marques

seront retrouvées dans le code généré (assembleur pour le logiciel, et VHDL RTL pour le matériel). Les marques choisies sont des simples affectations de variable. Une affectation est ajoutée au début et à la fin de chaque bloc de base. La valeur affectée indique s'il s'agit d'un début/fin d'un bloc de base, elle indique aussi son numéro. La Figure 38 montre un exemple qui illustre l'identification des blocs de base en SDL.

```
BLOCK PC;
  .../*déclaration et connexions des routes*/
    PROCESS host(1,1);
       .../*déclaration des types et variables*/

      dcl bb integer; /*déclaration de variable pour le marquage
                        de début-fin des blocs de base*/
      START ;
        TASK bb:=65280; /*début du BB1 0xff00*/
        TASK ind1:=0,
        memoire(0):=10,    memoire(1):=0,
        memoire(2):=0,     memoire(3):=0,                Bloc de base 1
        memoire(4):=10,    memoire(5):=9,
        memoire(6):=0,     memoire(7):=15;
        TASK bb:=65296; /*fin du BB1 0xff10*/
        NEXTSTATE DistCtrl;
      STATE DistCtrl;
        INPUT pid_pc_values via pc_pid.1;
        TASK bb:=65281; /*début du BB2 0xff01*/
        TASK ind := 0;                                  Bloc de base 2
        TASK bb:=65297; /*fin du BB2 0xff11*/
        loop1:
          DECISION ind1 >= 14;
            ( false ):
              DECISION (ind1=7) and (ind > 0);
                ( true ):
                  NEXTSTATE DistCtrl;
                ( false ):
              ENDDECISION;
            ( true ):
              STOP ;
          ENDDECISION;
          DECISION ind1<=7;
            ( true ):
              TASK bb:=65282; /*début du BB3 0xff02*/
              TASK ind:=ind1;                           Bloc de base 3
              TASK bb:=65298; /*fin du BB3 0xff12*/
            ( false ):
              TASK bb:=65283; /*début du BB4 0xff03*/
              TASK ind:=ind1-7;                         Bloc de base 4
              TASK bb:=65299; /*fin du BB4 0xff13*/
          ENDDECISION;
          TASK bb:=65284; /*début du BB5 0xff04*/
          TASK ind1:=ind1+1,temp:=memoire(ind);         Bloc de base 5
          TASK bb:=65300; /*fin du BB5 0xff14*/
          OUTPUT pc_pid_value (temp) via pc_pid.1;
        JOIN loop1;
      ENDSTATE;
    ENDPROCESS;
ENDBLOCK;
```

Figure 38. L'identification des blocs de base dans une description SDL

Le flot de calcul des délais des blocs de base est présenté sur la Figure 39. En effet, après l'identification des blocs de base en SDL, on procède à deux implémentations en utilisant MUSIC. La première est complètement logicielle. La seconde est complètement matérielle. L'implémentation logicielle se devise en plusieurs implémentations selon le nombre de microprocesseurs et de compilateurs disponibles. Les canaux de communications ne sont pas implémentés (pour simplifier l'analyse des blocs de base).

Finalement, les codes générés (assembleurs et VHDL RTL) sont inspectés pour identifier les blocs de base et estimer la performance de chacun d'entre eux. Donc, pour chaque bloc de base

(SDL), nous allons obtenir plusieurs délais. Un délai correspond à l'implémentation matérielle, et le reste correspond aux différentes implémentations logicielles. Actuellement, les mesures de performance en ce qui concerne la réalisation sont effectuées manuellement. L'automatisation de cette étape ne représente aucune difficulté théorique.

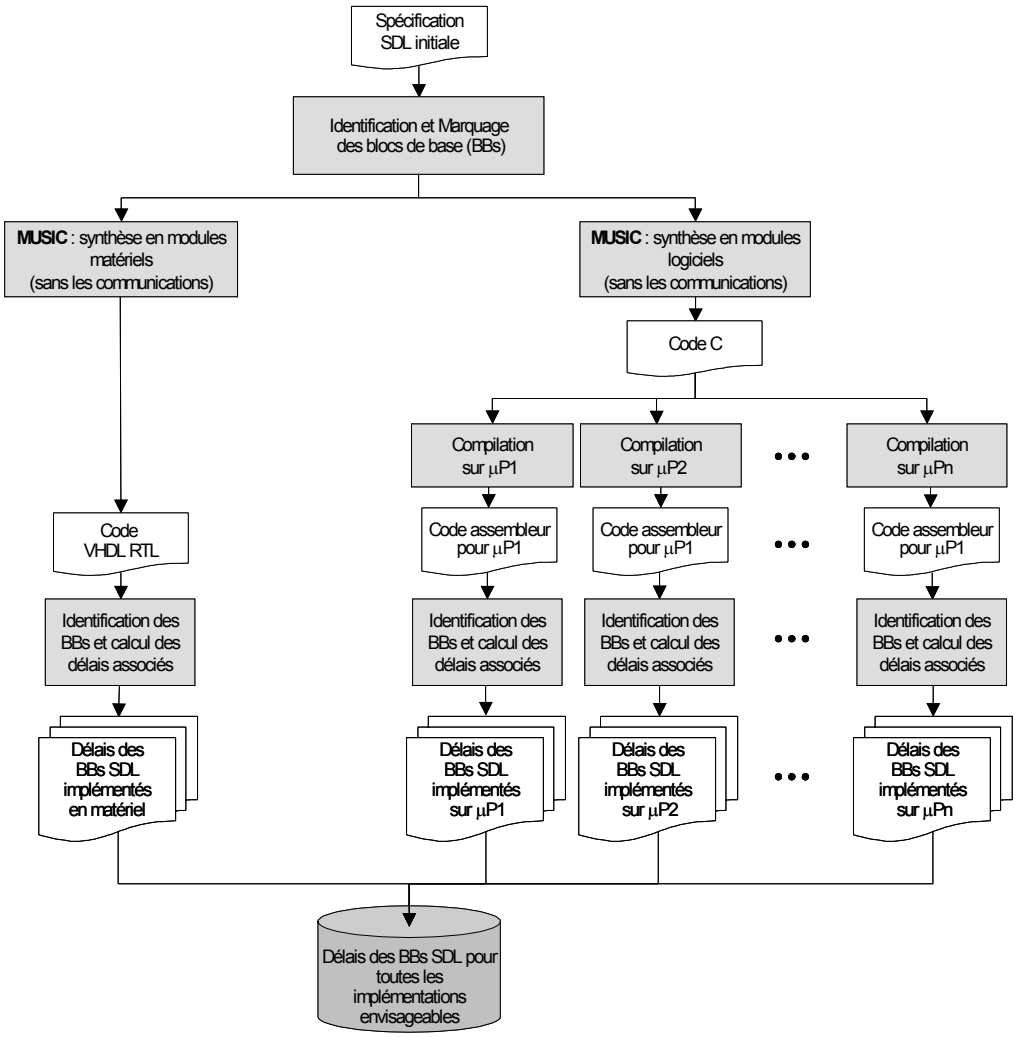


Figure 39. Le flot de calcul des délais des blocs de base

Nous avons envisagé deux grandes difficultés dans la réalisation de cette étape. La première est de déterminer comment un bloc de base est implémenté. Si la distance (d'abstraction) entre la spécification du bloc de base (en SDL) et son implémentation est très grande, et si beaucoup d'optimisations ont étés réalisées entre ces deux niveaux, il sera difficile de retrouver comment le bloc de base était implémenté. La deuxième difficulté est comment, à partir de l'implémentation (listage assembleur ou code VHDL) d'un bloc de base, calculer son temps d'exécution avec précision. Car, même si au niveau système, un bloc de base ne contient pas de branchement, c'est très probable que ce ne sera plus le cas au niveau implémentation (par exemple : les branchements créés par la dépendance de donnée). Ce comportement dynamique sera mesuré statiquement durant cette étape d'estimation, d'où l'existence de quelques imprécisions inévitables.

Le modèle de synthèse de MUSIC, à l'état actuel, nous aide à alléger ces difficultés car il ne procède pas à des optimisations durant la génération du code C/VHDL. Donc, il est

relativement facile de faire la correspondance entre le niveau système et le niveau C/VHDL. Ensuite, si on utilise les compilateurs (pour le C) avec des options d'optimisation allégées, et en utilisant les marques de début/fin de bloc de base, on peut facilement identifier les blocs de base au niveau RT (assembleur/VHDL RTL). La Figure 40 illustre cette correspondance des blocs de base entre les niveaux SDL et assembleur. En conséquence, nous arrivons à surmonter la première difficulté. Par ailleurs les imprécisions de calculs des temps d'exécution peuvent être négligées sans nuire à la qualité de l'estimation.

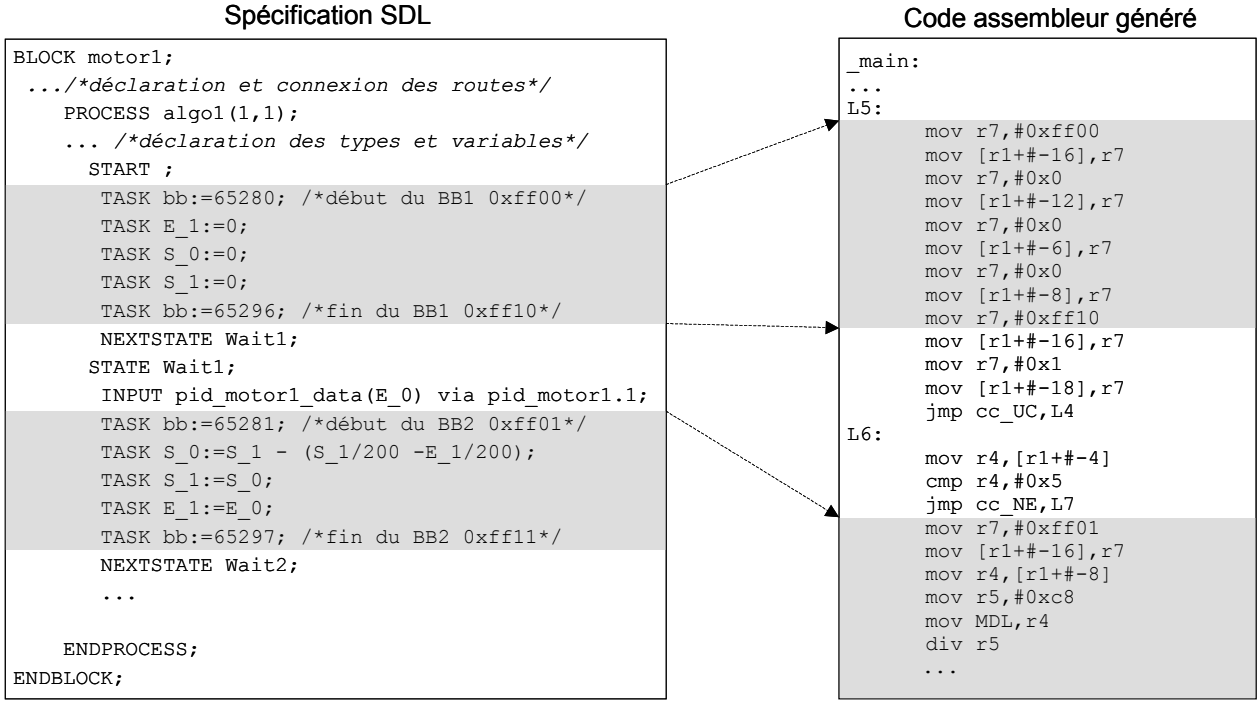


Figure 40. La correspondance des blocs de base aux niveaux SDL/assembleur

Pour l'estimation de l'implémentation matérielle, MUSIC génère du code VHDL RTL, et à ce niveau d'abstraction, le parallélisme d'opérations se restreint au niveau cycle. Donc le modèle architectural considéré est simple : chaque transition en VHDL RTL dure un cycle d'horloge. Cela facilite énormément cette tâche, car elle est réduite au calcul du nombre de transitions.

L'estimation de l'implémentation logicielle demande des procédures plus sophistiquées. Il faut d'abord utiliser les compilateurs avec des options d'optimisation allégées pour permettre aux annotations introduites en SDL d'apparaître dans le listage d'assembleur et faciliter l'identification des blocs de base. Ayant le code en assembleur, et les spécifications du microprocesseur, on peut calculer la performance d'une séquence d'instructions d'assembleur en ajoutant le nombre de cycles nécessaire pour chaque instruction. Notons que l'estimation des délais à partir du code assembleur dépend fortement de l'architecture du microprocesseur. Pour certains microprocesseurs, cette tâche est très simple car une instruction assembleur prend toujours un nombre bien précis de cycles (ex : 8051). Cependant pour d'autres, qui ont des architectures complexes, avec plusieurs niveaux de *pipeline* (ex : Pentium), c'est relativement difficile.

Une autre possibilité, pour ne pas restreindre les options d'optimisation des compilateurs, est d'isoler les blocs de base dans des procédures indépendantes, avant la compilation. Cela nous

permet de retrouver les blocs de base quelles que soient les options d'optimisation des compilateurs. Cependant, cette solution est bien plus difficile à mettre en œuvre.

Nous avons noté un autre problème durant l'investigation du code assembleur. A l'intérieur d'une séquence de code assembleur (correspondant à un bloc de base), nous pouvons trouver des appels de procédure de la bibliothèque du compilateur. A l'intérieur de ces procédures on peut trouver des branchements de dépendance de données, ce qui rendent l'estimation de leurs délais plus difficile. La solution adoptée est de pré-calculer un délai moyen pour chacune de ces procédures.

4.3.2.2 Les communications

L'estimation de la communication est traitée à part. En effet, la durée d'une communication peut être divisée en trois parties :

1. **Interface :** C'est le délai nécessaire à l'exécution des instructions des procédures de communication (sans les temps d'attente s'ils existent) à l'intérieur du processeur (matériel/logiciel). En effet c'est la partie fixe du protocole de communication, qui est la même pour toutes les instances de communications du même protocole. Elle dépend seulement du type de l'implémentation (logicielle/matérielle), elle ne dépend ni des données ni de l'environnement.
2. **Transmission de données :** C'est la partie de la communication qui dépend de la taille des données. Ce délai est complémentaire au délai précédent. Il faut noter que le temps nécessaire pour qu'une donnée traverse les fils physiques de communication (s'ils existent) est pratiquement nul, car nous sommes dans le cadre de circuits embarqués (ou même *system-on-a-chip*).
3. **Attente active :** Il contient tous les délais de blocage du processus tout au long de la communication (ex : le rendez-vous). C'est une partie très variable qui dépend de l'exécution dynamique du système.

Nous remarquons de cette analyse de la communication qu'une grande partie du délai de communication ne dépend pas des données. Aussi, les protocoles de communications dans l'environnement de MUSIC sont décrits dans une bibliothèque à part avec le langage intermédiaire SOLAR. Donc nous avons décidé de créer une bibliothèque d'estimation de la communication.

En effet, l'attente active est prise en compte par le simulateur d'ObjectGEODE. Par exemple pour un protocole rendez-vous, il existe un attribut *rendez-vous* que l'on peut ajouter lors de la déclaration d'une route en SDL, cet attribut provoque le blocage du processus lors de l'initialisation d'une communication (à travers cette route) jusqu'à l'arrivée de l'autre partie (rendez-vous), tout en avançant le temps global (attente active). Il existe d'autres attributs qui nous permettent de modéliser la plus part des schémas de communications.

Par conséquent, la bibliothèque d'estimation de la communication contiendra les délais d'interface et de transmissions de données, mentionnés ci-dessus. Le flot de cette étape est similaire à cela du calcul du temps d'exécution des blocs de base. Le flot est appliqué pour chaque protocole de communication. Cette procédure est décrite sur la Figure 41.

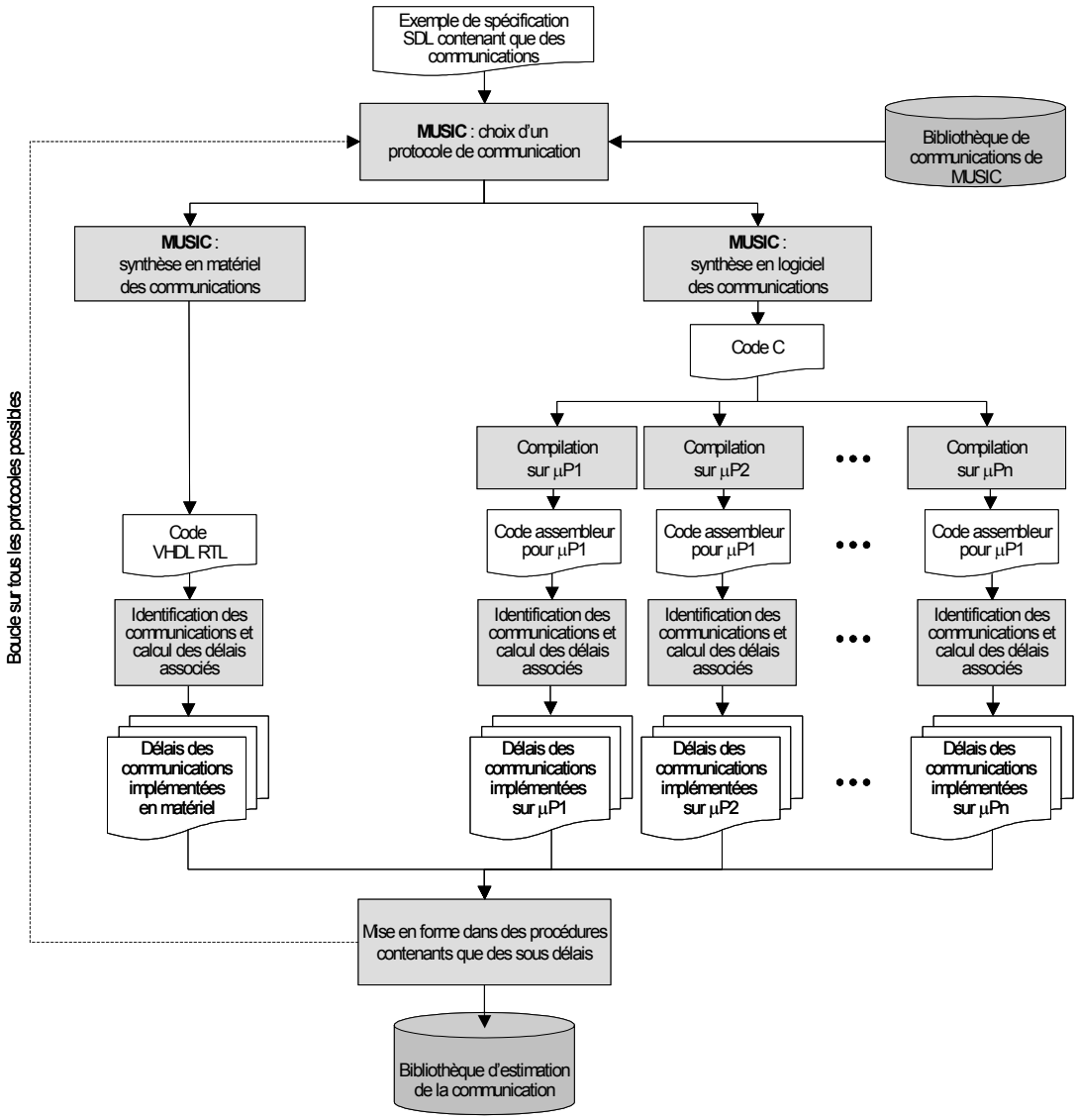


Figure 41. Le flot de création de la bibliothèque d'estimation de la communication

En effet, il s'agit de créer une description SDL qui ne contient que des primitives de communications. Cette description sera ensuite synthétisée par MUSIC pour chaque protocole de communication de la bibliothèque de MUSIC. Et de la même façon que pour les blocs de base, nous générons toutes les implémentations possibles (matérielle/logicielle). La prochaine étape est l'analyse du code généré et le calcul des délais (comme pour les blocs de base). L'avantage de la génération de cette bibliothèque est qu'elle sera réutilisable par toutes les applications ultérieures.

Les délais de communication (pour un protocole spécifique) seront différents si les deux processus communicants sont sur le même processeur ou sur deux processeurs différents (le délai de l'interface n'est pas le même). Donc il faut implémenter les deux cas (par MUSIC) et calculer les deux types de délais. Pour certains protocoles de communications, les délais calculés ci-dessus peuvent contenir des expressions dépendant de la taille des données pour exprimer le délai de transmissions de données. Ces expressions ne posent pas de problèmes car elles sont supportées pas le module de performance du simulateur d'ObjectGEODE. La façon dont on a représenté les délais dans la bibliothèque est expliquée dans la section suivante.

4.3.3 L'annotation de la spécification SDL

Il s'agit d'instrumenter la spécification SDL avec les délais élémentaires obtenus dans l'étape précédente en utilisant les extensions de SDL introduites par ObjectGEODE. Nous pouvons aussi diviser cette phase en deux parties : annotation des blocs de base et annotations des communications. A la sortie de cette étape, nous allons obtenir une spécification SDL prête à être utilisée dans la boucle d'exploration d'architectures.

4.3.3.1 L'annotation des blocs de base

Pour chaque bloc de base en SDL, nous avons obtenu plusieurs délais correspondants aux différentes implémentations possibles. Ces délais seront introduits dans la spécification SDL grâce à la directive DELAY. En effet, nous déclarons tous les délais trouvés, au début de la spécification SDL comme des constantes. Une action (TASK) est rajoutée au début de chaque bloc de base, pour lui attacher la directive DELAY avec la bonne valeur (selon l'implémentation choisie). Quand le simulateur arrive à cette action, le processus se bloque pendant le délai précisé, tout en avançant le temps global. Ensuite le bloc de base est exécuté avec un temps nul. Donc tout se passe comme si le bloc de base était exécuté avec le délai précisé (délai de l'exécution de son implémentation). La Figure 42 illustre un code SDL où les blocs de base ont été annotés.

```
BLOCK PC;
  ...
    PROCESS host(1,1);
       ...

          DECISION ind1<=7;
            ( true ):
              TASK  'delay'  COMMENT '#delay (PC_delay3+0)';
              TASK bb:=65282;
              TASK ind:=ind1;                                  Bloc de base 3
              TASK bb:=65298;
            ( false ):
              TASK  'delay'  COMMENT '#delay (PC_delay4+0)';
              TASK bb:=65283;
              TASK ind:=ind1-7;                                Bloc de base 4
              TASK bb:=65299;
          ENDDECISION;
          TASK  'delay'  COMMENT '#delay (PC_delay5+0)';
          TASK bb:=65284;
          TASK ind1:=ind1+1,temp:=memoire(ind);                Bloc de base 5
          TASK bb:=65300;
          OUTPUT pc_pid_value (temp) via pc_pid.1;
        JOIN loop1;
      ENDSTATE;
    ENDPROCESS;
ENDBLOCK;
```

Figure 42. L'annotation des blocs de base

4.3.3.2 L'annotation des communications

L'annotation de la communication est plus compliquée. Chaque protocole demande une réflexion différente pour pouvoir la modéliser avec les atouts disponibles. Par exemple, pour le protocole rendez-vous, nous devons ajouter l'attribut *rendez-vous* à la route correspondante. Avec cela, les deux actions *output* et *input* deviennent bloquantes. Ensuite, il y a deux possibilités :

1. Si les processus communicants sont sur deux processeurs différents, et donc s'exécutent en parallèle, le délai de communication peut être approché par l'équation (4-3).

$$D_{Comm\_P} = D_{attente\ active} + Moyenne(D_{in}, D_{out}) \qquad (4\text{-}3)$$

où : $D_{in} = D_{(interface + tarns.\ données)\ input}$;
$D_{out} = D_{(interface + tarns.\ données)\ output}$.

Donc pour estimer cette communication, il suffit de rajouter le délai *Moyenne(D_{in},D_{out})* après les actions *output* et *input* dans les deux processus communicants. Donc, à la simulation, après une période d'attente active (éventuellement), la communication se fait avec un temps nul, ensuite les processus sont bloqués simultanément pendant la durée *Moyenne(D_{in},D_{out})* (le temps global avance toujours).

2. Si les deux processus communicants sont sur le même processeur, et donc s'exécutent en série, le délai de communication peut être approché par l'équation (4-4).

$$D_{Comm\_S} = D_{attente\ active} + D_{out} + D_{in} \qquad (4\text{-}4)$$

Donc pour modéliser ce délai, il suffit de rajouter le délai D_{out} après l'action *output*, et le délai D_{in} après l'action *input*, dans les deux processus communicants. Comme les deux processus s'exécutent sur le même processeur, le délai total de la communication sera la somme des délais partiels. Ceci donne une modélisation assez précise et fiable pour ce cas.

Dans le cas où plusieurs processus s'exécuteraient sur un seul processeur, la version actuelle de MUSIC génère un code en utilisant une technique de synthèse de logiciel très particulière. Les flots d'exécution (*threads* en anglais) sont fusionnés par entrelacement. Même les lignes de codes correspondants à la communication sont entrelacées (éventuellement), avec du code d'autres processus. Pour pouvoir modéliser ce schéma d'ordonnancement, il faut diviser les délais D_{in} et D_{out} en unités plus petites qui représentent les délais des blocs de base du protocole de communication. Pour cela nous avons décidé de remplacer D_{in} et D_{out} par des appels de procédures. Ces procédures contiennent simplement une suite de directives DELAY représentant la suite des délais des blocs de base du protocole de communication correspondant. Avec cela, nous pouvons à la simulation suivre le même modèle d'entrelacement implémenté par MUSIC ou même n'importe quel autre modèle d'ordonnancement des processus.

Cette stratégie pour modéliser le protocole rendez-vous est tellement générale et abstraite que l'on peut utiliser pour n'importe quel autre protocole de communication.

4.3.4 Les choix architecturaux

Cette étape introduit les choix architecturaux dans la spécification système pour indiquer, au niveau système, la technologie choisie pour la réalisation des composants. Les choix architecturaux peuvent se résumer en trois : la définition des types et des nombres de microprocesseurs et de co-processeurs, l'attribution des processus aux processeurs et le choix du protocole et du type d'implémentation pour chaque canal de communication.

Trois types d'annotation sont effectués sur la spécification SDL, obtenue de l'étape précédente, pour obtenir une spécification SDL modélisant une certaine architecture. La simulation de cette spécification donne la performance de l'architecture choisie. Cette possibilité de modéliser, au niveau système, le comportement temporel d'une architecture à partir d'une spécification SDL est due aux extensions introduites dans le simulateur d'ObjectGEODE. Les trois modifications sont développées ci-dessous.

4.3.4.1 Le partitionnement du système

Cette tâche modélise le partage des mêmes ressources par plusieurs processus. Ceci est faisable grâce à la directive NODE introduite par le simulateur d'ObjectGEODE. Cette directive, qui peut être associée aux déclarations SYSTEM et BLOCK en SDL, cause l'exécution sur un même processeur des processus contenus dans cette hiérarchie.

Donc, le partitionnement consiste à rassembler les processus, qui vont s'exécuter sur le même processeur, dans des BLOCKs en ajoutant la directive NODE à ces derniers. Cette procédure nécessite une restructuration du réseau de communication. Cette restructuration est facile à réaliser, la Figure 43 montre un exemple d'une spécification SDL qui illustre cette tâche.

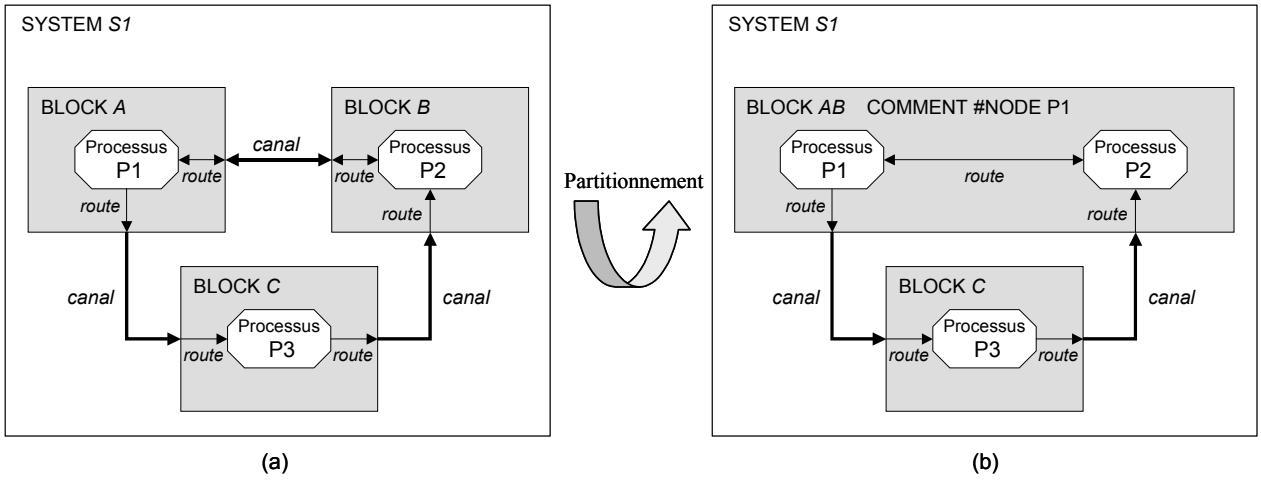


Figure 43. Le partitionnement du système et son influence sur le réseau de communication

Dans cette figure, les flèches entre les blocs sont des canaux de communication et les autres sont des routes (voir Figure 43a). Nous remarquons dans cette figure la simplification de la communication entre les deux blocs **A** et **B** après leur fusion. Elle devient une communication interne à travers d'une route (voir Figure 43b). La fusion des deux blocs **A** et **B** est très facile à faire en pratique. A part cela, il y a très peu de modifications à faire. Il s'agit de changer légèrement la définition des canaux de communications connectés du nouveau bloc **AB** au bloc **C**.

4.3.4.2 L'attribution des processeurs logiciels/matériels

Il s'agit de choisir, pour chaque nœud, le type du processeur sur lequel les processus du nœud seront exécutés. Le processeur peut être matériel (ASIC), ou un certain microprocesseur.

Comme pour chaque bloc de base nous avons calculé plusieurs délais correspondant aux plusieurs processeurs, dont nous disposons, cette étape d'attribution des processeurs est réalisée

simplement par l'annotation du bloc de base par le délai correspondant à son implémentation sur le processeur choisi. Bien évidement, tous les blocs de base qui se trouve à l'intérieur d'un nœud doivent être implémentés sur le même processeur.

En effet, cette étape est complémentaire à celle présentée dans la section 4.3.3.1.

4.3.4.3 Le choix de la communication

Dans cette tâche nous décidons, pour chaque canal de communication, le protocole de communication qui sera utilisé. Ce choix est un paramètre qui influence grandement la performance d'une architecture.

Nous avons vu dans la section 4.3.3.2, comment l'annotation de la communication est faite. Le choix d'un certain protocole de communication revient à ajouter des attributs aux routes connectées aux canaux de communication et de sélectionner les bonnes procédures, de la bibliothèque d'estimation de la communication, à appeler lors des actions *input* et *output*.

Il faut noter que le choix de ces dernières procédures dépend aussi du choix des deux processeurs sur les quels les deux processus communicants vont être exécutés. Car, dans la bibliothèque d'estimation de la communication, à chaque type de protocole correspond plusieurs procédures (délais), selon le type du processeur implémentant ce protocole. La Figure 44 illustre un modèle SDL instrumenté avec des attributs de communication.

Bibliothèque d'estimation de la communication

```
package RDV_PROCEDURES;
...
procedure  pin_int_sst10_e_rdv;
  start;
     task 'delay' comment '#delay ((28+input_char_st10+10)*p_sst10)';
     task 'delay' comment '#delay ((28+input_char_st10+output_char_st10)*p_sst10)';
     task 'delay' comment '#delay ((28+input_int_st10+output_char_st10)*p_sst10)';
     task 'delay' comment '#delay ((28+input_char_st10+output_char_st10)*p_sst10)';
     return;
endprocedure pin_int_sst10_e_rdv;
...
endpackage RDV_PROCEDURES;
...
```

Délai pour recevoir un *int* avec le protocole rendez-vous implémenté sur ST10

```
use RDV_PROCEDURES;
SYSTEM motor;

  ...
  procedure motor1_iPIDMOTOR1_int;
    start;CALL pin_int_sst10_e_rdv;return;
  endprocedure motor1_iPIDMOTOR1_int;
  ...
  BLOCK motor1;
    ...
    PROCESS algo1(1,1);
      ...
      STATE Wait1;
       INPUT pid_motor1_data(E_0) via pid_motor1.1;
       CALL motor1_iPIDMOTOR1_int;
       TASK 'delay' COMMENT '#delay (motor1_delay2+0)';
       TASK bb:=65281;
       TASK S_0:=S_1 - (S_1/200 -E_1/200);
       TASK S_1:=S_0;
       TASK E_1:=E_0;
       TASK bb:=65297;
       NEXTSTATE Wait2;
       ...
    ENDPROCESS;
  ENDBLOCK;
  ...

ENDSYSTEM;
```

Choix du protocole de communication

Annotation de la communication

Figure 44. Le choix et l'annotation de la communication

4.3.5 Simulation

Il s'agit d'exécuter la description SDL obtenue à la sortie de l'étape précédente, et qui contient toutes les informations de performance et de l'architecture modélisée. Ceci est réalisé grâce au simulateur *geodesim* d'ObjectGEODE. Le résultat de la simulation indique la performance de l'architecture.

En effet, il faut d'abord compiler la spécification SDL avec l'outil *gsmcompil* d'ObjectGEODE. Le résultat de la compilation est exécuté par le simulateur *geodesim*. Dans un fichier d'initialisation (*system.startup*), nous pouvons tracer les variables et les E/S, nous pouvons également préciser une condition d'arrêt de la simulation. Il faut choisir la même condition d'arrêt pour toutes les architectures qu'on va simuler pour les comparer (on peut aussi mettre plusieurs points d'arrêt à l'intérieur du même système pour l'analyse détaillée des différentes phases de l'exécution de l'application). A la fin de chaque simulation (ou à l'arrivée à un point d'arrêt), *geodesim* affiche le temps de l'exécution (cumuls des délais partiels),qui représente la performance de l'architecture simulée.

Dans la section 4.2.3.3.2, nous avons mentionné que l'ordonnancement des processus à l'intérieur d'un nœud (un processeur), suit une distribution aléatoire uniforme. Et que la directive PRIORITY peut être associée à un processus ou à un type de processus, à l'intérieur d'un nœud pour changer ce choix aléatoire. Si le comportement de cette directive ne peut pas répondre à une certaine stratégie d'ordonnancement, nous pouvons indiquer au simulateur *geodesim* un *scénario* particulier à suivre lors de l'exécution. Ce *scénario*, qui est dépendant de l'application, contiendra les informations nécessaires pour modéliser notre stratégie d'ordonnancement.

La simulation avec *geodesim* (niveau système), prend un temps très court, de l'ordre de la seconde. C'est l'avantage principal de notre méthodologie. Nous pouvons donc parcourir la boucle d'exploration d'architectures (voir Figure 37) plusieurs fois dans un temps raisonnable.

4.4 Analyse expérimentale de la méthodologie et résultats

Pour valider notre méthodologie, nous l'avons appliqué sur deux exemples réels : (1) un système d'asservissement de la vitesse de deux moteurs et (2) un système d'interface réseau ATM. Pour chacun nous avons estimé la performance de plusieurs architectures du système. Ces mêmes architectures ont été synthétisées et cosimulées dans l'environnement de MUSIC. La comparaison et l'analyse des résultats nous ont permis de démontrer la puissance de notre méthodologie. Aussi, nous avons relevé la capacité et les limitations de cette méthodologie.

Dans la section 4.4.1 nous présentons le premier exemple (contrôleur d'un bras de robot). Les architectures choisies et l'application de notre méthode sont présentées dans la section 4.4.2. Ensuite, la synthèse et les résultats de la cosimulation de ces architectures sont faites dans la section 4.4.3. Finalement, les résultats sont analysés dans la section 4.4.4.

Les détails concernant le deuxième exemple (interface réseau ATM) sont présentés dans le journal *Technique et Science Informatiques* (voir Publicstion n ° 11).

4.4.1 L'application : contrôleur d'un bras de robot

Il s'agit de concevoir un système d'asservissement d'un bras de robot. Dans la version complète de cet exemple, le bras du robot est constitué de 8 moteurs. Nous allons en considérer

seulement deux pour améliorer la clarté de l'exemple. Ce système ajuste la variation de vitesse de chaque moteur de façon à ce que le mouvement du bras soit souple et que les contraintes physiques d'accélération et de freinage soient respectées.

Le système est modélisé en SDL (voir Figure 45). Il s'agit de quatre processus *Host, Pid, Moteur1* et *Moteur2*. Le processus *Host* (à l'intérieur du bloc PC) envoie les consignes de vitesse et les paramètres de contrôle au *Pid,* qui à son tour, prend la commande des deux moteurs. Le processus *Pid* (Proportionnel, Intégral, Dérivation), contrôle la vitesse des deux moteurs selon certaines équations. A chaque cycle, le processus *Pid* calcule et envoie une nouvelle valeur à un des deux moteurs, après la lecture de sa vitesse instantanée. Le contrôle des deux moteurs se fait en série. Le moteur est modélisé par son équation au dérivé.

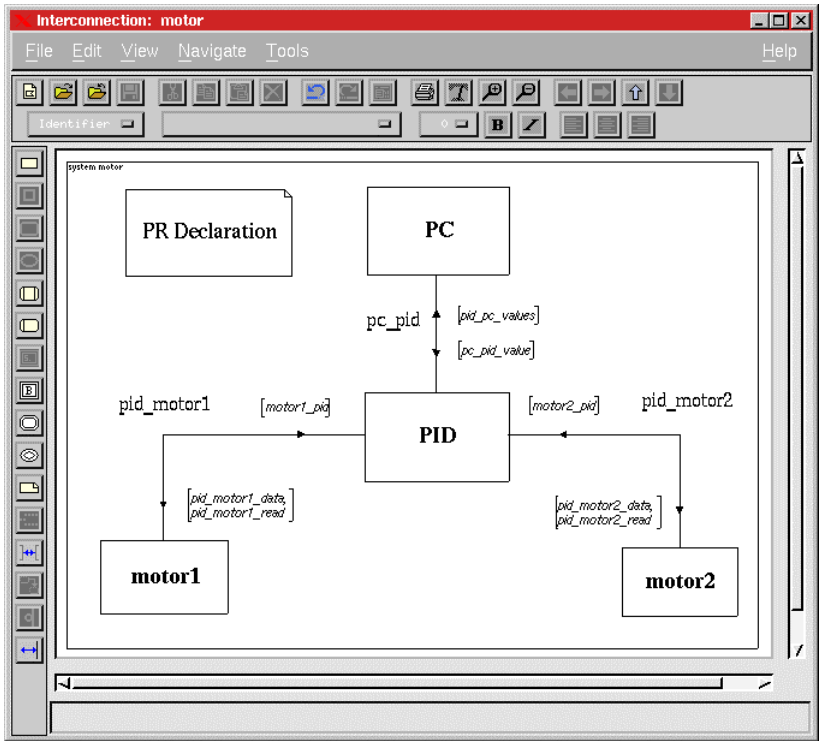


Figure 45. La modélisation du système de contrôle en SDL

Nous avons décidé de concevoir le système entier en modules logiciels et matériels. Nous disposions lors de l'achèvement de cette expérimentation de deux chaînes de développement pour deux microcontrôleurs : ST10 de STMicroelectronics et 8051, avec un seul compilateur pour chacun ; en plus de l'option matérielle (ASIC). Si on ne considère aucune contrainte d'implémentation, et si on ne considère pas le choix de la communication, le nombre d'architectures possibles est donné par l'équation (4-2). Dans ce cas, **p** est égal à 3 et **n** est égal à 4, donc le nombre d'architectures possibles V_4^3 est 309. Nous remarquons que c'est un nombre énorme pour un exemple aussi simple.

4.4.2 L'application de la méthode d'estimation/exploration

Nous allons appliquer les différentes étapes de notre méthode, comme elles ont été présentées à la section 4.3.

4.4.2.1 L'estimation des délais élémentaires

Les délais élémentaires sont de deux types : pour les blocs de base et pour la communication.

Nous analysons la description SDL du système et nous identifions tous les blocs de base en ajoutant des marques de début/fin de bloc. En utilisant MUSIC, nous procédons à une première implémentation complètement matérielle, sans la synthèse de la communication. Nous obtenons quatre fichiers en VHDL RTL (correspondant aux 4 processus du système). Nous retrouvons facilement les marques de blocs de base. Le temps d'exécution (en nombre de cycles), d'un bloc de base est égal au nombre de transitions entre les deux marques qui lui bornent. On calcule ce temps pour tous les blocs de base. Ensuite, nous procédons à une deuxième implémentation complètement logicielle, sans la synthèse de la communication. Nous obtenons quatre fichiers en C (correspondant aux 4 processus du système). Nous compilons (sans optimisation), ces fichiers avec le compilateur de ST10 et le compilateur de 8051. Nous retrouvons, relativement facilement, les marques des blocs de base dans les fichiers assembleurs résultants. En utilisant les spécifications (*data sheets*) des deux processeurs, nous calculons le temps d'exécution (en nombre de cycles d'horloge), de chaque bloc de base sur chaque processeur. Nous avons ainsi obtenu trois délais différents pour chaque bloc de base en SDL.

Nous avons du créer la bibliothèque d'estimation de la communication. Pour cela, nous avons décrit un système très simple en SDL qui ne contient que quelques processus qui communiquent. Pour simplifier l'exemple, nous avons considéré un seul protocole de communication de la bibliothèque de MUSIC : le rendez-vous. Donc, comme dans le cas des blocs de base, nous procédons aux deux implémentations logicielle/matérielle par MUSIC, mais en synthétisant la communication et en choisissant le protocole rendez-vous. Les fichiers C et VHDL obtenues sont traités de la même manière. Le début et la fin de la communication sont reconnus facilement due à la simplicité du système décrit en SDL. Cependant, nous avons pris beaucoup de soin lors du calcul du temps d'exécution de la communication, car comme nous l'avons expliqué dans la section 4.3.2.2, à chaque action *input* et *output* doit correspondre une procédure contenant des sous délais indivisibles (blocs de base), selon l'implémentation. Donc pour chaque action *input* et chaque action *output* nous avons construit trois procédures (trois délais différents). Ces procédures dépendent aussi de la taille de données transmises. Ils sont regroupés pour construire une sous bibliothèque d'estimation de la communication.

4.4.2.2 L'annotation

Nous déclarons, au début de la spécification SDL, tous les délais des blocs de base que nous avons calculé dans l'étape précédente. Aussi, nous déclarons la bibliothèque d'estimation de la communication. Nous annotons aussi les blocs de base et les actions *input* et *output* par des directives DELAY et des appels de procédures (de la bibliothèque), comme expliqué dans la section 4.3.3. Ces dernières annotations seront configurées avec les bonnes valeurs lors du choix d'architecture (étape suivante).

4.4.2.3 Le choix de l'architecture

Concernant la tâche du choix de la communication, nous avons décidé de choisir un seul type de communication : le protocole rendez-vous. Donc l'attribution des bonnes valeurs aux appels de procédures, ajoutés (à la spécification SDL) dans l'étape précédente, dépend seulement des deux autres tâches de cette étape.

Le partitionnement du système et l'attribution des processeurs logiciels/matériels caractérisent l'architecture que l'on veut réaliser. Nous avons vu qu'il y a un nombre très grand d'architectures possibles. Nous avons décidé d'en choisir quelques-unes parmi les plus pertinentes. Ainsi, nous avons choisi de faire les architectures présentées dans le Tableau 3.

Tableau 3. Les architectures choisies

Architecture	Host	PID	Moteur1	Moteur2
A1	8051	Matériel	ST10	
A2	ST10	Matériel	ST10	
A3	ST10	Matériel	ST10	ST10
A4	Matériel	Matériel	ST10	ST10
A5	Matériel	Matériel	Matériel	Matériel

Nous avons choisi les fréquences d'horloges suivantes : 2MHz pour ST10, 12MHz pour 8051 et 1MHz pour la partie matérielle (ASIC).

4.4.2.4 La simulation

Pour chaque architecture, nous compilons (avec *gsmcompil*), la spécification SDL résultante de l'étape précédente. Un fichier d'initialisation de la simulation (*.startup*), est créé. Pour les résultats présentés dans cette section, la simulation a été arrêtée quand la vitesse du moteur 1 arrive à une valeur stable (environ 10 unités), ainsi le temps de simulation correspond à l'intervalle représenté sur la Figure 46.

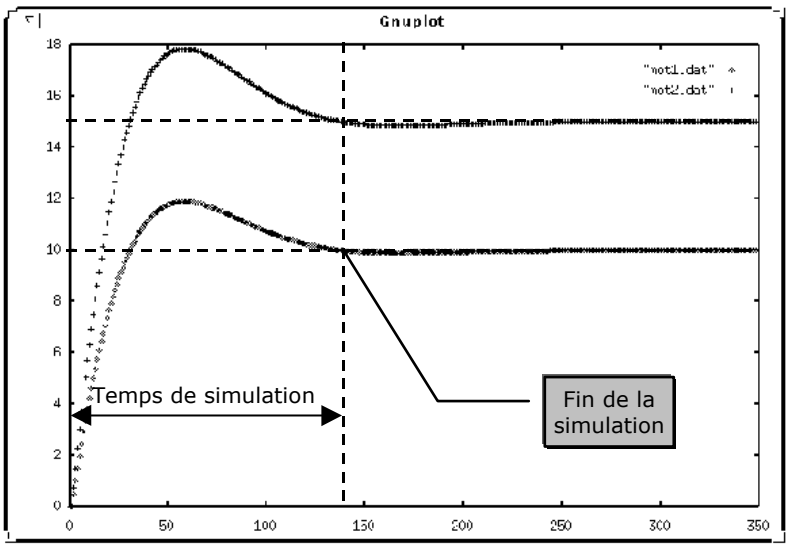


Figure 46. Les résultats de la cosimulation et le temps de simulation

Nous avons procédé à la simulation de toutes les architectures, avec *geodesim*. Le Tableau 4 montre les résultats obtenus. La performance est mesurée en µs.

Tableau 4. La performance des architectures pour la simulation avec geodesim

Architecture	Estimation SDL (µs)
A1	1866
A2	1873
A3	1871
A4	1779
A5	21

4.4.3 La synthèse et cosimulation avec MUSIC

Pour valider les résultats obtenus par notre méthode d'estimation, il a fallu les comparer avec les résultats réels. Grâce à l'outil de cosimulation MCI [44], nous pouvons obtenir des résultats de performance précis au niveau cycle. Pour les deux microcontrôleurs ST10 et 8051, nous disposons du simulateur d'architecture ISS que nous avons adapté à MCI. Pour l'ASIC (VHDL), nous utilisons le simulateur VSS de Synopsys, qui lui aussi était adapté à MCI. A partir de la spécification SDL originale du système, nous procédons à la synthèse des architectures choisies (voir Tableau 3), en utilisant MUSIC. Les résultats de la cosimulation sont présentés dans le Tableau 5.

Tableau 5. La performance des architectures pour la cosimulation avec MCI

Architecture	Cosimulation RTL (µs)
A1	2977
A2	2986
A3	1841
A4	1747
A5	22

4.4.4 L'analyse des résultats

Dans cette section, nous allons analyser les résultats que nous avons obtenus. Ces résultats nous permettent d'effectuer des analyses non seulement qualitatives, mais aussi quantitatives. Nous démontrons la validité de notre méthodologie. Ensuite, nous présentons ses avantages et sa grande capacité. Les difficultés, ainsi que quelques limitations seront développées en dernier.

4.4.4.1 Validité de la méthode

Nous dressons un tableau récapitulatif des résultats de performance obtenus (voir Tableau 6). Nous remarquons pour les trois dernières architectures, la précision est excellente, l'erreur d'estimation est, en moyenne, de l'ordre de 3%.

Tableau 6. Les performances des architectures et la comparaison des résultats

Architecture	Host	PID	Moteur1	Moteur2	Estimation SDL (µs)	Cosimulation RTL (µs)	Erreur d'estimation
A1	8051	Matériel	ST10		1866	2977	-37%
A2	ST10	Matériel	ST10		1873	2986	-37%
A3	ST10	Matériel	ST10	ST10	1871	1841	2%
A4	Matériel	Matériel	ST10	ST10	1779	1747	2%
A5	Matériel	Matériel	Matériel	Matériel	21	22	-5%

Pour les deux premières architectures, nous avons une erreur d'estimation assez grande (38%). Cette erreur nous a incité à regarder les traces de l'exécution en détail. Nous remarquons que cette erreur arrive lorsque *Moteur1* et *Moteur2* partagent le même processeur. L'ordonnancement de ces deux processus suit une distribution aléatoire uniforme dans le modèle d'exécution de *geodesim*. Cependant, l'ordonnancement dans le code généré par MUSIC se matérialise par l'entrelacement des deux machines d'états des deux processus. En particulier, les protocoles de communications sont entrelacés. C'est à dire, lorsque *Pid* est entrain de communiquer avec *Moteur1*, chaque transition dans le protocole de communication du coté *Moteur1*, sera suivi par une transition du *Moteur2*. Il se trouve dans notre cas, où *Moteur1* est identique à *Moteur2*, que la transition exécutée dans le *Moteur2* est la première partie d'une communication. Il s'agit de la première transition d'une demande de communication (*handshake*). Cette transition consomme un temps non négligeable du temps de la communication (presque 20%). En plus, comme elle sera exécutée un nombre de fois égale au nombre de transitions dans le protocole de communication, elle peut augmenter le temps d'une communication de presque 30 à 50 %. Il faut se rappeler que cette analyse est faite dans le cadre d'un protocole rendez-vous.

L'utilisation d'un *scénario* prédéfini pour la simulation avec *geodesim*, permet de modéliser l'entrelacement utilisé par MUSIC. Cependant, il ne permet pas de modéliser ce schéma précis d'entrelacement d'une communication avec une autre qui n'est pas établie. Pour résoudre ce problème, une première solution est de modifier la spécification SDL en ajoutant quelques actions pour modéliser ce *scénario*. Dans la pratique, cette solution est très difficile à mettre en œuvre d'une façon automatisé. En plus, ce n'est pas acceptable d'introduire des modifications lourdes dans la spécification SDL originale, on risque de changer sa fonctionnalité. La deuxième solution est analytique. Il s'agit de trouver des équations analytiques fiables qui corrigent cette erreur. Cette solution sera plus facile à mettre en pratique. Pour cet exemple, nous avons changé les délais des communications pour prendre en compte le résidu expliqué ci-dessus. Cette solution est spécifique pour cette application. Nous avons obtenu les résultats montrés dans le Tableau 7.

Tableau 7. Les performances après la correction des résultats de la simulation SDL

Architecture	Host	PID	Moteur1	Moteur2	Estimation SDL (µs)	Cosimulation RTL (µs)	Erreur d'estimation
A1	8051	Matériel	ST10		3149	2977	6%
A2	ST10	Matériel	ST10		3156	2986	6%

Nous remarquons que nous obtenons un excellent résultat. Cependant, si la solution apportée était faite plus soigneusement, nous arrivons à une erreur d'estimation encore plus petite. Maintenant nous pouvons assurer la validité de notre méthodologie. Cette nouvelle méthode d'estimation/exploration donne des résultats parfaitement fiables et très précis.

Cette méthode a été de même appliquée à un système d'interface réseau ATM (voir Publication n ° 11 pour les détails). Dans cette expérimentation des situations plus compliquées ont été testés (par exemple : exécuter le code de l'application avec un micro-noyau temps réel (RTOS)). La gestion des tâches dans ce micro-noyau est exécutée de façon classique avec 3 niveaux de priorité. A chaque niveau est associée une file gérée en « round robin ». Les résultats des mesures obtenus par l'approche d'exploration sont quasiment identiques à ceux obtenus au niveau cycle. Les taux d'erreurs sur les cinq architectures du système ATM varient entre 1 % et 3 %. Ces résultats démontrent l'efficacité de la technique de la simulation que nous avons élaborée dans l'approche. En effet, cette technique permet au concepteur de choisir lui-même le modèle d'exécution dans le simulateur SDL. L'approche supporte trois algorithmes d'ordonnancement : Round Robin, par priorité et FIFO. Chaque processus peut choisir son mode d'ordonnancement grâce à la directive « COMMENT #SCHEDULER ». Le temps de commutation de contexte et le quantum de temps (time slice) sont paramétrables grâce aux directives « COMMENT #SWITCHDELAY » et « COMMENT #TIMESLICE ».

4.4.4.2 Capacité de la méthode

Notre méthode d'estimation/exploration réalise de très bons taux de précision et de rapidité. Nous avons vu que l'erreur d'estimation est de l'ordre de 4%. L'analyse des résultats nous permet de conclure que cette méthode est parfaitement fiable. En plus, le temps d'exploration d'architectures par cette méthode est incomparable avec celui nécessaire à l'implémentation et la cosimulation du même nombre d'architectures (surtout si ce dernier est grand). En fait, la cosimulation au niveau cycle, par MCI, prend actuellement un temps énorme. Le Tableau 8, montre la différence de temps entre la cosimulation et la simulation SDL.

Paramètres	Simulation SDL	Cosimulation RTL
Pas de simulation	Transition	Cycle d'horloge
Nombre de pas simulés	3300	2 451 000
Nombre de pas / sec	330	23
Temps de simulation	10 s	30h 22min 39s

Tableau 8. La comparaison entre les temps de simulation

Nous pouvons facilement noter dans cette figure, la capacité de notre méthode. Nous avons vu aussi que le nombre d'architectures à explorer augmente exponentiellement avec la complexité de l'application et la diversité des éléments d'architectures dont nous disposons. Ce qui rend l'utilisation de notre méthodologie d'estimation/exploration encore plus avantageuse. La durée totale du flot d'estimation/exploration de notre méthode est très adaptée au besoin de rapidité demandé par la conception conjointe matérielle/logicielle.

Aussi il faut noter que notre modèle d'estimation de la communication est très général, et peut être appliqué sur tout type de communication. En plus, l'idée de construire une bibliothèque d'estimation de la communication est très avantageuse. Cette bibliothèque peut être réutilisée pour toutes les autres applications, ce qui facilite beaucoup l'application de notre méthode. En fin, la plus part des étapes du flot d'estimation/exploration peuvent être automatisées facilement.

4.4.4.3 Les limitations

Comme toute nouvelle méthodologie, il y a toujours quelques difficultés et quelques limitations. Nous allons en citer celles que nous avons pu soulever :

1. Il faut disposer pour chaque microprocesseur une chaîne de développement contenant particulièrement un compilateur C qui génère du code assembleur. Ce type de compilateur n'est pas toujours disponible, surtout pour des microprocesseurs spécialisés.
2. Pour des microprocesseurs complexes (pipeline, parallélisme interne, etc), il est difficile de calculer, avec précision, le temps d'exécution statiquement, i.e. à partir du code assembleur.
3. L'identification des blocs de base dans le code assembleur, très facilité avec l'ajout des marques de début/fin, reste parfois difficile due à l'introduction, par le compilateur, d'appels de procédures internes de sa bibliothèque.
4. Pour pouvoir identifier les blocs de base dans le code assembleur, nous sommes obligés de ne pas utiliser des options d'optimisation qui risquent de brouiller le code. En particulier, les marques de début/fin de blocs de base risquent de disparaître. Cette contrainte ne nous permet pas d'estimer le code optimal.

5. Le modèle architectural des parties matérielles est très simple. Nous disposons actuellement d'un seul modèle d'exécution pour le matériel, chaque transition dans la machine d'états finis s'exécute en un seul cycle.

6. La difficulté de modéliser en SDL quelques schémas spéciaux d'ordonnancement de processus. Cette difficulté a été illustrée par un exemple dans la section 4.4.4.1.

7. Il faut ajouter aussi les limitations du simulateur d'ObjectGEODE (peu de modèles de communication et quelques *bugs*).

Toutes ces difficultés sont superficielles, elles ne constituent pas une défaillance dans notre méthodologie.

4.5 Conclusion

Dans ce chapitre, une nouvelle méthodologie d'estimation de performance au niveau système est présentée. Cette méthodologie permet l'exploration efficace de l'espace des solutions avant la conception. Nous avons développé et validé cette méthodologie dans l'environnement de codesign MUSIC pour des spécifications en SDL. La combinaison d'un tel outil avec un outil de conception systématique constituera un environnement parfait pour la conception de systèmes multiprocesseurs complexes.

L'avantage d'employer l'exploration de l'espace des solutions au niveau système a été mis en évidence par le fait que seulement quelques minutes sont nécessaires pour tester de nouvelles architectures avec notre méthodologie, tandis qu'il fallait plusieurs jours pour tester les mêmes architectures au niveau RTL.

Dans ce chapitre, nous avons présenté une nouvelle méthode d'estimation du temps d'exécution de systèmes multiprocesseurs. Cependant, cette méthodologie peut être appliquée pour l'estimation d'autres paramètres comme la surface et la consommation.

Chapitre 5

Conception systematique de l'architecture

Sommaire

Ce chapitre décrit le processus de conception systématique de l'architecture à travers plusieurs exemples. Ces travaux ont permis la mise au point d'une démarche systématique. Tous les exemples présentés dans ce chapitre résultent de conception manuelle. L'automatisation du flot fait actuellement l'objet de plusieurs thèses.

Dans la première partie de ce chapitre nous présentons la partie implémentation systématique de notre flot (Figure 23). Les détails du flot sont illustrés concrètement à travers un exemple. La deuxième partie de ce chapitre est consacrée à la présentation des applications conçues durant cette thèse. Ces applications nous ont permis de valider notre approche de conception d'AMM dédiées à des applications spécifiques et d'analyser son efficacité ainsi que l'efficacité du modèle architectural proposé.

5.1 Flot systématique de conception

5.1.1 Présentation du flot de conception

Le flot d'exploration d'architectures (chapitre 4) part d'un modèle de l'application au niveau système (processus parallèles communicants) et, en considérant notre modèle architectural, produit les paramètres architecturaux détaillés de l'architecture finale. Ces paramètres comportent : le partitionnement logiciel/matériel, l'allocation de processeurs (CPUs, IPs), du réseau de communication, des protocoles de communication et des mémoires ainsi que tous les paramètres de configuration liés à ces composants et leurs interconnexions. Notons que ces paramètres sont dédiés à l'application, ainsi l'architecture finale sera dédiée à l'application.

En entrée du flot de conception nous rassemblons ces paramètres au sein de deux entités : une architecture abstraite et une table d'allocation. Nous appelons cette étape dans le flot (Figure 47) : extraction des paramètres. L'architecture abstraite constitue le squelette de l'architecture finale. C'est la description de l'architecture au niveau macro-architecture (3.2.2). La table d'allocation contient tous les paramètres architecturaux relatifs à chaque composant de l'architecture (protocoles, adresses mémoire, niveaux d'interruption réservés à chaque CPU, etc.).

La deuxième étape dans le flot (Figure 47) est l'étape de conception de l'architecture. Elle fait appel à la table d'allocation pour raffiner l'architecture abstraite et produire l'architecture détaillée. L'action principale est la conception des interfaces de communication des différents processeurs de l'architecture. Celle-ci est réalisé en instanciant les composants de base constituant l'interface à partir de la bibliothèque d'interface de communication (3.1.3). Cette bibliothèque contient des blocs VHDL RTL génériques. Il suffit d'instancier les composants requis, de les configurer et de les connecter pour construire une interface de communication dédiée à l'application. Les cœurs de processeurs, le réseau de communication, les mémoires, ainsi que les interfaces de communication conçues sont donc instanciés et connectés ensemble (selon la *netlist* représentée par l'architecture abstraite) pour former ainsi l'architecture finale au niveau micro-architecture (RTL).

L'étape d'adaptation du logiciel (Figure 47) produit les images binaires qui s'exécuteront sur chaque CPU de l'architecture. Ici le logiciel applicatif assigné à chaque CPU doit être ciblé pour ce CPU. Cette étape consiste à remplacer les appels systèmes (APIs) rencontrés dans le logiciel applicatif par les codes de bas niveau (pilotes de matériel). Un système d'exploitation (3.1.4) peut aussi être développé, si nécessaire, pour chaque CPU. Cette couche d'adaptation est ensuite compilée et liée avec le logiciel applicatif (qui reste inchangé en raison de l'utilisation d'APIs). La table d'allocation qui a été employée dans l'étape de conception d'architecture est également indispensable dans cette partie du flot car elle précise les plages mémoires réservées, les protocoles utilisés, l'allocation d'adresses et les interruptions réservées. Le résultat de cette

étape est l'obtention des codes binaires qui doivent être chargés sur la mémoire de chaque CPU. L'opération d'adaptation du logiciel est faite séparément pour chaque CPU.

L'étape finale dans le flot est la validation de l'architecture conçue au niveau RTL (avant la synthèse de la puce). Pour cette validation, nous avons besoin d'un prototype exécutable, précis au cycle près, qui puisse simuler l'application. A cet effet nous avons employé une approche de cosimulation basée sur SystemC [36] où les CPUs sont remplacés par leur simulateur de jeu d'instruction (ISSs) précis au cycle près et par des modèles fonctionnels de bus (BFM). Toutes les autres parties de l'architecture sont modélisées en VHDL ou/et SystemC au niveau RTL et exécutées par leurs simulateurs (le simulateur VHDL utilisé est Synopsys VSS [104]).

Après validation de l'AMM dédiée à l'application, un flot classique de conception de circuit intégré est suffisant pour obtenir la puce finale.

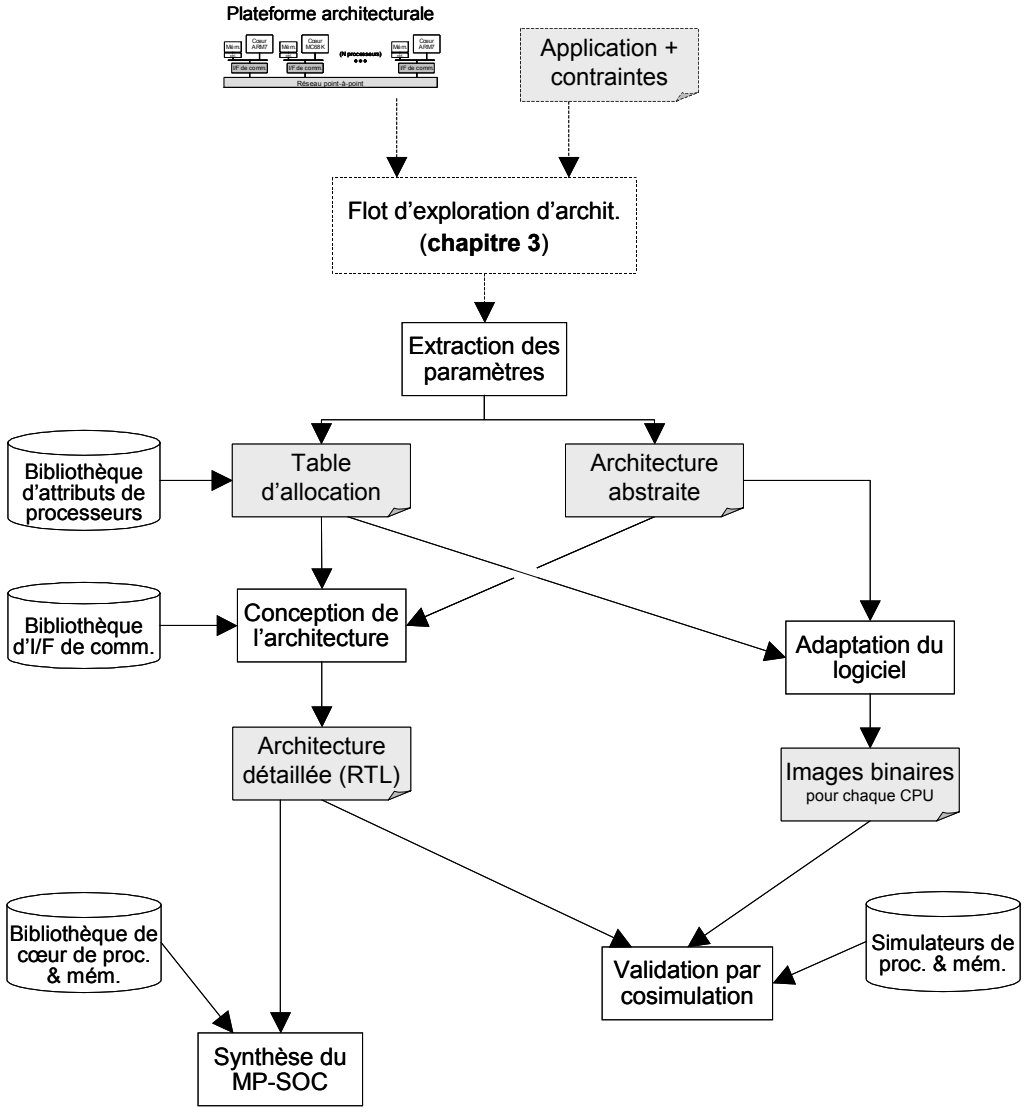


Figure 47. Le flot de conception systématique proposé

Dans cette section nous avons donné un aperçu rapide de notre flot de conception systématique. Ce flot est présenté plus en détail dans la section suivante. Nous avons choisi d'illustrer les détails de façon concrète à travers un exemple d'application que nous avions

réalisé. Les différentes étapes du flot seront présentées via leur application à la conception d'un système industriel : *un commutateur de cheminement de paquets* (Packet Routing Switch) [47].

5.2 Détails du flot à travers un exemple

5.2.1 Présentation de l'exemple : commutateur de cheminement de paquets (CCP)

Les commutateurs de cheminement de paquets (CCP) constituent une solution puissante pour les systèmes à commutation de cellules (cell-switching systems) [47]. La version que nous présentons ici est composée de deux contrôleurs d'entrée et de deux contrôleurs de sortie. Chacun des contrôleurs traite un canal de communication. Le transfert de donnée se fait par paquets de 128 octets. Les liens de communication entre les contrôleurs d'entrée et les contrôleurs de sortie sont configurés par un signal externe (Mode) qui détermine s'ils sont de type direct ou commuté. La Figure 48 montre le schéma fonctionnel du CCP.

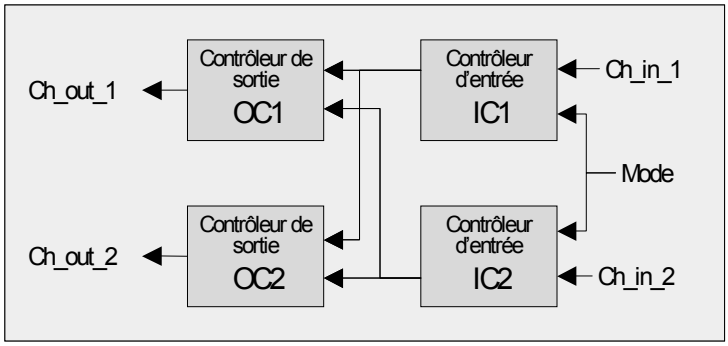


Figure 48. Schéma fonctionnel du CCP

5.2.2 Aperçu de la partie exploration d'architecture

A partir de ces spécifications, nous pouvons noter que nous avons 4 modules communicants, 4 canaux internes de communication et 5 liens externes. La largeur d'une trame d'entrée est de 128 octets. L'application a été décrite en SystemC au niveau fonctionnel. A partir de ce niveau et en prenant en considération notre modèle architectural, une exploration d'architectures va nous permettre de déterminer la meilleure architecture possible (chapitre 4). De nombreuses solutions se présentent (type du réseau de communication, type des processeurs, nombre de processeurs utilisés, mémoire partagée, etc.). En choisissant une plateforme architecturale telle que celle de la Figure 30 l'espace de solutions est réduit substantiellement. Le réseau de communication est fixé de type point-à-point, mais va-t-on implémenter les quatre modules sur un/deux/trois/quatre processeurs (ARM7, 68000)? Quels protocoles de communication allons nous utiliser? La mise à profit d'une méthode d'estimation de performances telle que celle que nous avons proposée au chapitre 4 répondra à toutes ces questions.

Dans les applications que nous avons réalisées, nous avons effectué cette étape d'analyse manuellement. Ainsi, pour cette application, nous avons opté pour une architecture à 4 processeurs (2 CPU d'ARM7 et 2 CPU de 68000). Un protocole du type *poignée de main*, avec des FIFOs de tailles adéquates, a été retenu.

5.2.3 Flot systématique de conception de l'architecture

Une fois les choix architecturaux effectués, examinons le flot d'implémentation.

5.2.3.1 Extraction des paramètres

Dans cette étape nous établissons l'architecture abstraite au niveau macro-architecture (Figure 49) et la table d'allocation (Tableau 9). L'architecture abstraite (Figure 49) présente le squelette de l'architecture finale. Les modules sont connectés par des canaux logiques (abstraits) qui cachent un protocole physique. Quelques paramètres architecturaux sont annotés (types des processeurs, réseau de communication, protocoles).

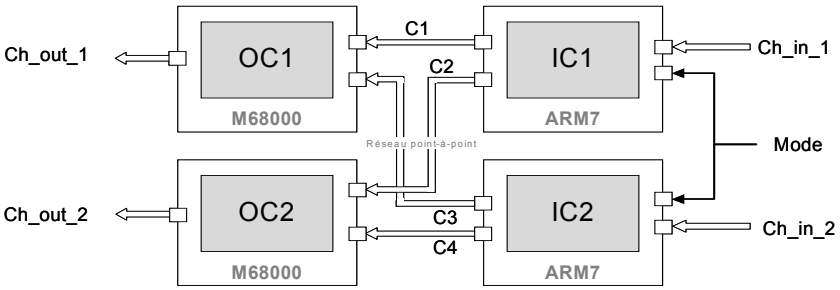


Figure 49. Architecture abstraite : implémentation à 4 processeurs du CCP

Tableau 9. Table d'allocation : implémentation à 4 processeurs du CCP

Module	CPU	Taille mémoire	Canaux de communication	Protocole de comm.	Taille buffer	IT	Adresse
IC1	ARM7 40 MHz	ROM: 10KB RAM: 20KB	Ch_in_1	FIFO + HSK	0	IRQ*	0x7000
			Mode	FIFO + HSK	0	IRQ*	0x7004
			C1 (IC1⇒OC1)	FIFO + HSK	128 octets	–	0x7008
			C2 (IC1⇒OC2)	FIFO + HSK	128 octets	–	0x700C
IC2	ARM7 40 MHz	ROM: 10KB RAM: 20KB	Ch_in_2	FIFO + HSK	0	IRQ*	0x7000
			Mode	FIFO + HSK	0	IRQ*	0x7004
			C3 (IC2⇒OC1)	FIFO + HSK	128 octets	–	0x7008
			C4 (IC2⇒OC2)	FIFO + HSK	128 octets	–	0x700C
OC1	M68000 20 MHz	ROM: 20KB RAM: 20KB	C1 (IC1⇒OC1)	FIFO + HSK	0	Niveau 5	0x9000
			C3 (IC2⇒OC1)	FIFO + HSK	0	Niveau 6	0x9002
			Ch_out_1	FIFO + HSK	128 octets	–	0x9004
OC2	M68000 20 MHz	ROM: 20KB RAM: 20KB	C4 (IC2⇒OC2)	FIFO + HSK	0	Niveau 5	0x9000
			C2 (IC1⇒OC2)	FIFO + HSK	0	Niveau 6	0x9002
			Ch_out_2	FIFO + HSK	128 octets	–	0x9004
Env.	VHDL 100 MHz	–	Ch_in_1	FIFO+HSK	–	–	–
			Mode	FIFO+HSK			
			Ch_in_2	FIFO+HSK			
			Ch_out_1	FIFO+HSK			
			Ch_out_2	FIFO+HSK			

* L'adresse du contrôleur de canal de communication demandant l'interruption est délivrée au CPU par l'interface de communication lorsqu'un accès en lecture à l'adresse fixe **0x7100** est effectué.

La table d'allocation (Tableau 9) détaille toutes les valeurs choisies pour les paramètres dédiés à l'application. Dans cette table, chaque ligne contient les paramètres dédiés à un module (processeur) de l'architecture. Par exemple, dans la première ligne nous voyons que le module IC1 sera implémenté sur un processeur ARM7 fonctionnant avec une fréquence d'horloge de 40 MHz. La capacité de la mémoire locale pour ce module a été choisie approximativement (après une première pré-compilation du logiciel qui devra y être exécuté). Il y a 4 canaux de communication en périphérie de ce module, et le protocole de communication utilisé pour tous ces canaux est un protocole de type poignée de main (*HSK*) avec FIFO. Les trois dernières colonnes dépendent du protocole de communication choisi. Nous pouvons voir que dans le cas de ce protocole de communication, la FIFO est placée près du port de sortie et sa taille est égale à celle d'une trame (128 octets). Pour les canaux d'entrée (Ch_in_1 et *mode*), la communication est réalisée à l'aide d'interruptions. Cela permet au processeur d'effectuer d'autres tâches en parallèle. Nous utilisons ici l'interruption IRQ du processeur ARM7. Dans la dernière colonne de la table, nous avons réservé des adresses globales pour les contrôleurs de canaux de communication (voir la section suivante).

Cette table sera utilisée par la suite lors de la configuration des interfaces de communication et de l'adaptation du logiciel (Figure 47).

5.2.3.2 Conception de l'architecture

Comme l'architecture contient 4 CPUs, 4 interfaces de communication doivent être conçues. Les interfaces de la Figure 25 sont modélisées en VHDL RTL sous la forme de composants génériques qui vont être spécifiés en fonction de l'application. Ils sont stockés dans une bibliothèque d'interface de communication. Ainsi, pour chacune des 4 interfaces de communication nous analysons les paramètres du Tableau 9, nous instancions et configurons les bonnes entités VHDL génériques, et nous connectons ces entités afin d'obtenir l'interface spécifique. Par exemple, pour l'interface de communication du premier module (IC1), l'adaptateur de bus du ARM7 a été instancié, ainsi que 4 contrôleurs de canaux de communication (2 contrôleurs d'entrée et 2 contrôleurs de sortie). Les contrôleurs choisis correspondent au protocole de poignée de main avec FIFO côté émetteur. La taille de la FIFO est configurée à 128 octets (Tableau 9). Les adresses mémoire réservées à ces contrôleurs sont également configurées d'après le Tableau 9. Elles servent à configurer le décodeur d'adresses de l'interface de communication correspondante. Le bus interne générique est également configuré (largeur du bus de données : 8 bits, 4 signaux de validations et 2 signaux d'interruptions). L'adaptateur de bus du ARM7 ainsi que les quatre contrôleurs de canaux sont interconnectés à l'aide du bus interne.

Le résultat de cette étape est un composant décrit en VHDL RTL qui représente l'interface de communication spécifique au premier module (IC1). La Figure 50 présente l'architecture de cette interface. Notons que les contrôleurs de communication d'entrée utilisent des interruptions pour communiquer des données au CPU. Ainsi le contrôleur d'interruptions doit être configuré pour utiliser l'interruption IRQ de l'ARM7 comme cela est mentionné dans le Tableau 9. Comme une seule interruption matérielle est utilisée (IRQ), et qu'il existe plusieurs sources d'interruption (les 2 contrôleurs de canaux en entrée), une technique de vectorisation des interruptions va devoir être utilisée.

Ainsi, quand le CPU est stimulé par une interruption IRQ, la routine d'interruption associée effectue une lecture à une adresse fixe (pour l'ARM7 nous avons choisi l'adresse 0x7100, cf. Tableau 9). Quand l'interface reçoit un ordre de lecture à cette adresse, elle délivre au CPU l'adresse du contrôleur de canal ayant initié l'interruption en question. Une seconde lecture permet de lire les données transmises par le contrôleur de canal. Notons que cela nécessite l'écriture de routines d'interruption dédiées à l'application du coté du logiciel (cf. étape d'adaptation du logiciel).

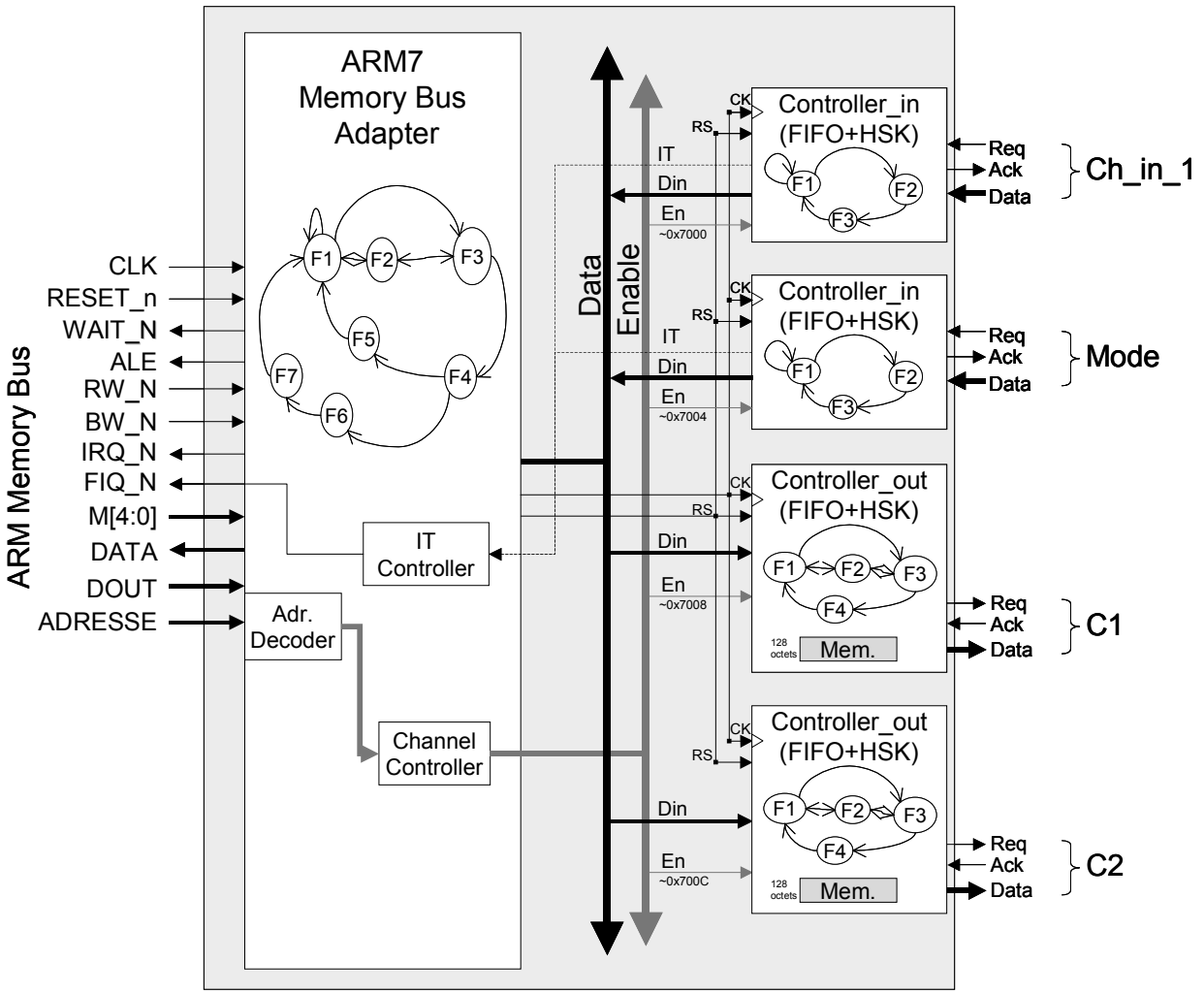


Figure 50. Interface de communication du module IC1

De façon analogue, nous avons construit les 3 autres interfaces de communication. Comme ces interfaces sont des composants VHDL, nous avons choisi de les assembler avec le réseau de communication (réseau point-à-point) dans un seul bloc VHDL.

Faisons de nouveau appel au Tableau 9 et à la Figure 49 pour déterminer les connexions entre les 4 interfaces et leurs liens externes avec l'environnement.

Dans cet exemple, l'environnement est un banc de test simple qui envoie et reçoit des paquets de données. Il a été décrit en langage VHDL et utilise là encore le protocole poignée de main.

5.2.3.3 Adaptation du logiciel

Il s'agit ici de cibler le logiciel applicatif qui va être exécuté par chaque CPU. Nous n'avons pas utilisé de système d'exploitation car cette couche était en cours de développement à l'époque de cette réalisation. Ainsi des programmes simples de test seront suffisants pour valider l'architecture matérielle (qui est l'objet essentiel de cette thèse).

Les programmes de test incorporent des fonctions de lecture/écriture externes (communications). Une opération de sortie correspond à l'écriture des données correspondantes à la bonne adresse. Par exemple, dans le module IC1, pour transférer le paquet reçu au module OC1, le paquet est écrit (octet par octet) à l'adresse physique 0x7008. L'interface de communication stocke ces données et prend en charge leur transfert au module OC1. Les opérations d'entrée sont déclenchées par l'arrivée d'une interruption; des routines d'interruption adaptées sont donc nécessaires. Par exemple, pour le vecteur d'IRQ du module IC1, l'ARM7 commence par effectuer un accès en lecture à l'adresse 0x7100. En accédant à cette adresse, l'interface de communication livre l'adresse du contrôleur de canal de communication qui demande cette interruption (0x7000 ou 0x7004 dans cet exemple), et ensuite les données sont lues à cette adresse. Notez que cette manière de vectoriser les interruptions est très utile quand le nombre de contrôleurs de communication (d'entrée) est plus grand que les niveaux d'interruption disponibles du CPU.

Dans cet exemple, le paquet de données est reçu octet par octet.

Les programmes de test ont été compilés et liés avec le code *boot* propre à chaque CPU (pour initialiser la RAM/ROM, la pile, les vecteurs d'interruptions et les modes d'exécution). Nous possédons donc à ce stade les codes binaires qui doivent être chargés dans la mémoire de chaque processeur.

5.2.3.4 Validation de l'architecture

Afin de valider l'architecture produite, nous avons mis en œuvre une méthode de cosimulation basée sur SystemC [36][100]. Dans cette approche les CPUs sont remplacés par leur ISS, précis au cycle près, enrichi par un model fonctionnel de bus (BFM). Nous disposons déjà de deux simulateurs de processeurs précis au cycle près, un pour l'ARM7 (basé sur ARMulator) et l'autre pour le M68000. En outre, avec ces simulateurs, les mémoires locales sont modélisées en logiciel comme faisant partie de l'ISS, et l'accès à ces mémoires est précis au cycle près également. Les interfaces et le réseau de communication ainsi que l'environnement externe (le banc de test) ont été modélisés en VHDL RTL (5.2.3.2). La partie VHDL est exécutée à l'aide d'un simulateur de VHDL (VSS de Synopsys dans notre cas). La construction manuelle de cet environnement de cosimulation nécessite l'écriture d'une enveloppe en SystemC pour chaque simulateur, ainsi qu'un module *Top* instanciant, connectant et lançant les 5 simulateurs. Le bus de cosimulation (*router*) assure l'exécution parallèle, la synchronisation, et l'intercommunication entre les différents simulateurs (en utilisant une mémoire partagée et des moniteurs) et garantit une exécution cohérente de l'ensemble du système [36]. Lors du lancement de la cosimulation, les codes binaires sont chargés automatiquement dans les ISS correspondants et le bloc VHDL compilé est fourni au simulateur VHDL.

Nous avons donc construit l'environnement de cosimulation qui se compose de 2 ISS ARM7, de 2 ISS M68000 et d'un VSS. La Figure 51 représente l'environnement de cosimulation obtenu.

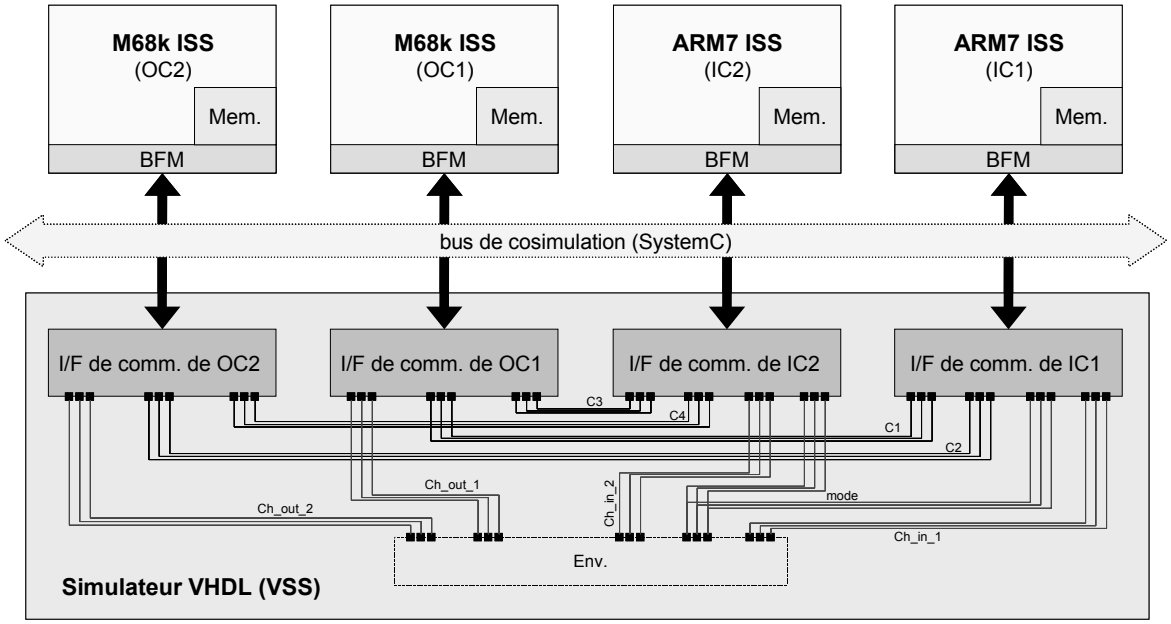


Figure 51. L'architecture de cosimulation du commutateur de paquets: implémentation à 4 processeurs

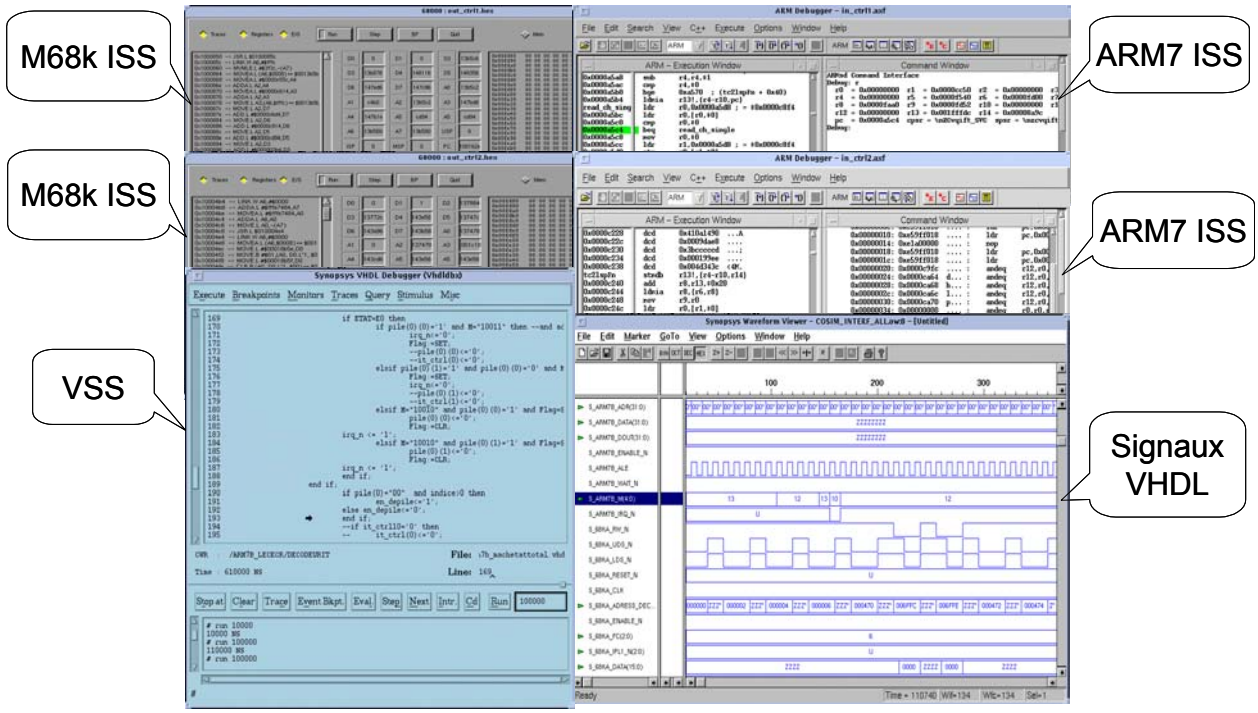


Figure 52. Copie d'écran durant la cosimulation du commutateur de paquets: implémentation à 4 processeurs

La Figure 52 montre une copie d'écran prise durant la cosimulation. Chacun des 5 simulateurs permet une exécution pas à pas du code logiciel ou matériel ce qui facilite grandement la mise au point de l'application (déboguage). Le tracé des signaux VHDL est possible et très utile pour la validation des résultats.

L'architecture (matériel/logiciel) RTL multiprocesseur dédiée à cette application a ainsi pu être validée. Si tous les cœurs des composants utilisés sont disponibles, un flot classique de conception de circuit intégré permet d'obtenir la puce finale.

La suite de ce chapitre est consacrée à la présentation des expérimentations menées durant cette thèse ayant permis de valider notre approche. L'analyse de son efficacité et l'évaluation des résultats seront effectuées au fur et à mesure de la présentation de chaque expérimentation.

La manière la plus concrète de valider une méthodologie est de la submerger d'exemples d'horizons divers. En effet, les différentes applications conçues nous ont incontestablement permis de prouver la faisabilité de notre méthodologie, d'évaluer ses performances et de contribuer à son amélioration. Il est aussi indéniable que ces exemples ont permis aux autres membres du groupe travaillant sur l'automatisation et le développement des outils de les faire considérablement évoluer et de faire apparaître des éléments pour les évolutions possibles.

Trois exemples sont présentés. Dans le premier nous montrons l'extensibilité et la flexibilité du modèle architectural proposé. Dans le deuxième exemple nous montrons comment notre approche de conception systématique permet de diminuer fortement le temps de conception. Enfin, dans le troisième exemple, nous mettons en évidence un cas très probant de l'utilité de notre approche. Nous y illustrons par ailleurs la possibilité d'intégrer des blocs préconçus (IPs).

Les deux seuls processeurs qui étaient disponibles dans notre bibliothèque lors de l'achèvement de ces expérimentations sont l'ARM7 de la société ARM et le M68000 de Motorola ; c'est pourquoi ce sont les seuls utilisés pour les trois applications. Nous avons de plus utilisé un réseau de communication point à point pour des raisons de performance et de disponibilité.

5.3 Commutateur de cheminement de paquets

5.3.1 But de l'expérimentation

Une première implémentation de cette application a été montrée dans la section précédente lors de la présentation du flot de conception. En effet deux implémentations ont été réalisées pour le commutateur de cheminement de paquets (CCP). Le but était de montrer la faisabilité et l'extensibilité de notre approche pour des architectures de différentes tailles. De plus c'est l'implémentation concrète de cette application qui nous a éclairé sur les éléments et les paramètres importants du flot de conception.

5.3.2 Spécification de l'application

La spécification de cette application a été présentée ci-dessus (5.2.1).

5.3.3 Solution architecturale à 4 processeurs

Partant de cette spécification nous avons décidé de réaliser deux implémentations différentes de cette application. Les choix architecturaux pour chaque implémentation n'ont pas été effectués à l'aide d'un outil d'estimation de performance tel que celui présenté au chapitre 4, mais de façon manuelle, en étudiant cette spécification, le modèle architectural ainsi que les contraintes. La première implémentation constitue une architecture à quatre processeurs ;

chaque module du système est implémenté sur un processeur. Cette implémentation a été exposée dans la section 5.2.1.

Pour évaluer l'efficacité et l'extensibilité de notre modèle architectural nous avons réalisé une autre implémentation comportant seulement deux processeurs. Cette implémentation est présentée dans la section suivante.

5.3.4 Solution architecturale à 2 processeurs

Une implémentation à 2 processeurs du CCP a également été réalisée en suivant le même flot de conception. Voici les différentes étapes qui nous ont permis d'aboutir à cette implémentation.

5.3.4.1 Extractions des paramètres

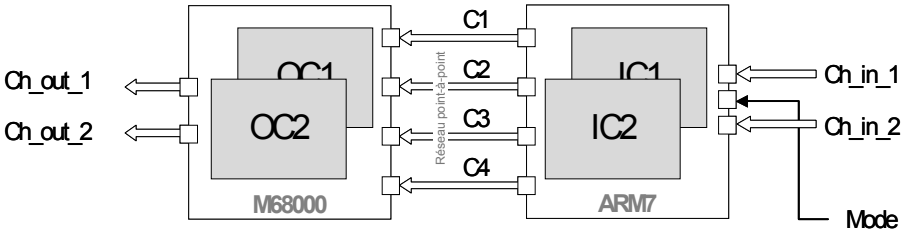


Figure 53. Architecture abstraite : implémentation à 2 processeurs du CCP.

Seuls deux CPUs sont employés ici : les modules IC1 et IC2 sont implémentés sur un cœur ARM7, les deux autres (OC1 et OC2) sur un cœur M68000. Nous avons gardé le même nombre de canaux de communication (pas de groupement/multiplexage) et les mêmes protocoles que dans l'implémentation à 4 processeurs afin de limiter les changements au nombre de processeurs utilisé et de porter notre comparaison sur ce seul paramètre.

Nous avons ainsi construit la table d'allocation (Tableau 10) et l'architecture abstraite (Figure 53) correspondant à cette nouvelle implémentation de l'application.

Tableau 10. Table d'allocation : implémentation à 2 processeurs du CCP

Module	CPU	Taille mémoire	Canal de communication	Protocole de comm.	Taille buffer	IT	Adr.
IC (IC1&IC2)	ARM7 40 MHz	ROM: 20KB RAM: 40KB	Ch_in_1	FIFO + HSK	0	IRQ*	0x7000
			Ch_in_2	FIFO + HSK	0	IRQ*	0x7004
			Mode	FIFO + HSK	0	IRQ*	0x7008
			C1 (IC⇒OC)	FIFO + HSK	128 octets	–	0x700C
			C2 (IC⇒OC)	FIFO + HSK	128 octets	–	0x7010
			C3 (IC⇒OC)	FIFO + HSK	128 octets	–	0x7014
			C4 (IC⇒OC)	FIFO + HSK	128 octets	–	0x7018
OC (OC1&OC2)	M68000 20 MHz	ROM: 40KB RAM: 40KB	C1 (IC⇒OC)	FIFO + HSK	0	Niveau 3	0x9000
			C2 (IC⇒OC)	FIFO + HSK	0	Niveau 4	0x9002
			C3 (IC⇒OC)	FIFO + HSK	0	Niveau 5	0x9004

			C4 (IC⇒OC)	FIFO + HSK	0	Niveau 6	0x9006
			Ch_out_1	FIFO + HSK	128 octets	–	0x9008
			Ch_out_2	FIFO + HSK	128 octets	–	0x900A
Env.	VHDL 100 MHz	–	Ch_in_1	FIFO+HSK	–	–	–
			Mode	FIFO+HSK			
			Ch_in_2	FIFO+HSK			
			Ch_out_1	FIFO+HSK			
			Ch_out_2	FIFO+HSK			

* L'adresse du contrôleur de canal de communication demandant l'interruption est délivrée au CPU par l'interface de communication lorsqu'un accès en lecture à l'adresse fixe **0x7100** est effectué.

5.3.4.2 Conception de l'architecture

Dans la mesure où cette nouvelle architecture ne contient que deux processeurs, seules deux interfaces de communication vont être nécessaires. Ces interfaces de communication diffèrent de celles développées dans l'architecture précédente (à 4 processeurs) par le nombre de contrôleurs de communication qu'elles intègrent. Ainsi lors de la conception de celle associée à l'ARM7, 7 contrôleurs de communication ont été instanciés (3 contrôleurs d'entrée et 4 contrôleurs de sortie). Pour la construction des deux interfaces nous avons effectué le même travail que celui détaillé dans la section 5.2.3.2. Ces interfaces sont ici encore assemblées avec le réseau de communication (réseau point à point) pour former un seul bloc VHDL. Le bloc modélisant l'environnement est le même que celui conçu lors de la première implémentation. Enfin, le Tableau 10 nous permet de déterminer les connexions entre les 2 interfaces et leurs liens externes avec l'environnement. La Figure 54 montre l'architecture finale obtenue.

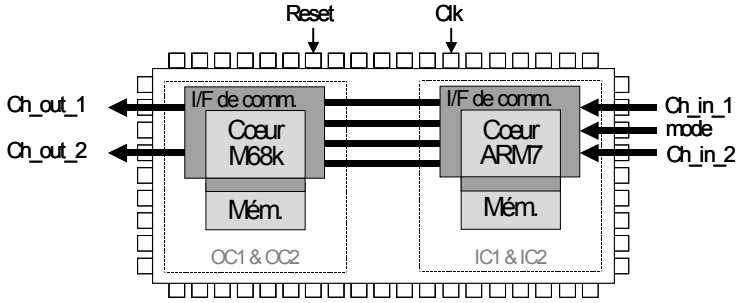


Figure 54. Architecture finale : implémentation à 2 processeurs du CCP

5.3.4.3 Adaptation du logiciel

Les programmes de test ont étés adaptés (la table d'allocation ayant changé, les adresses et les interruptions ne sont plus les mêmes) et nous avons suivi les mêmes étapes de compilation et d'édition de liens qu'en 5.2.3.3.

5.3.4.4 Validation par cosimulation

Tout le système a été, là encore, validé par cosimulation (voir 5.2.3.4). Le nouvel environnement de cosimulation que nous avons construit contient un ISS ARM7, un ISS M68000 ainsi qu'un VSS (simulateur VHDL de Synopsys). Cette nouvelle architecture spécifique a donc été cosimulée, ce qui nous a permis de valider le système. Nous présentons

dans la section suivante les résultats obtenus et nous les comparons avec ceux obtenus avec l'architecture à 4 processeurs.

5.3.5 Analyse et comparaison

Pour ce qui est de la surface et afin de comparer les deux architectures, nous avons synthétisé les blocs VHDL correspondant aux interfaces de communication pour les deux architectures. Les résultats de synthèse des deux architectures sont rassemblés dans le Tableau 11.

Nous remarquons que dans les deux cas la plus grande partie de la surface de la puce est occupée par les cœurs des CPUs et les mémoires. La logique des interfaces de communication s'élève seulement à 5156 portes pour l'architecture à 4 processeurs et à 3376 portes pour l'architecture à 2 processeurs. Dans les deux cas ceci représente moins de 5% de la surface totale de la puce. La taille de la mémoire utilisée pour la communication demeure la même pour les deux architectures car nous avons conservé le même réseau de communication dans les deux implémentations.

Ainsi la différence significative vient de la surface occupée par les cœurs des CPUs intégrés (2 au lieu de 4).

En ce qui concerne le temps d'exécution, comme cela était prévisible, la cosimulation a mis en évidence un débit en sortie deux fois inférieur dans le cas de l'architecture à 2 processeurs.

Tableau 11. Résultats de synthèse des deux architectures implémentant le commutateur de paquets

Architecture	Cœurs de CPU	Interfaces de communication
Architecture à 4 processeurs	2 Cœurs ARM7 + 2 Cœurs M68000	5156 portes + 6 FIFOs de 128 octets
Architecture à 2 processeurs	1 Cœur ARM7 + 1 Cœur M68000	3376 portes + 6 FIFOs de 128 octets

Cet exemple est très intéressant car il permet de mettre en lumière l'extensibilité de notre modèle architectural, ce dernier pouvant en effet s'adapter à des applications de différentes tailles. Cette extensibilité est obtenue grâce à la flexibilité de l'interface de communication proposée et à la modularité de notre approche.

5.4 Système téléphonique cellulaire : IS-95 CDMA

5.4.1 But de l'expérimentation

Après la démonstration de la faisabilité et de l'extensibilité de notre approche par l'exemple du CCP, nous avons décidé d'analyser la durée du cycle de conception. Pour cela, nous avons choisi une nouvelle application : un système téléphonique cellulaire de type IS-95 CDMA. Le temps nécessaire à chaque étape du flot de conception a donc été mesuré. Cette expérience a prouvé qu'une architecture multiprocesseur peut aisément être conçue en une semaine environ, à condition bien sûr que tous les composants de la plateforme d'architecture soient disponibles (bibliothèque complète).

5.4.2 Spécification de l'application

Dans un système téléphonique cellulaire de type IS-95 CDMA [121], la station mobile contient deux modems CDMA de bande de base (Tx et Rx), un encodeur de voix (ENC) de type QCELP (Qualcomm Code Excited Linear Prediction), un décodeur de voix (DEC) du même type, et un processeur d'appel (CAP). La Figure 55 montre le schéma fonctionnel de la partie numérique du système. Les sous-modules du modem CDMA Tx (respectivement Rx) exécutent les fonctions de modulation (respectivement démodulation) ainsi que le codage (respectivement décodage) convolutionnels, l'interlaçage des blocs, la modulation orthogonale et la génération d'indicatif de pseudo-bruit (pseudo-noise ; PN).

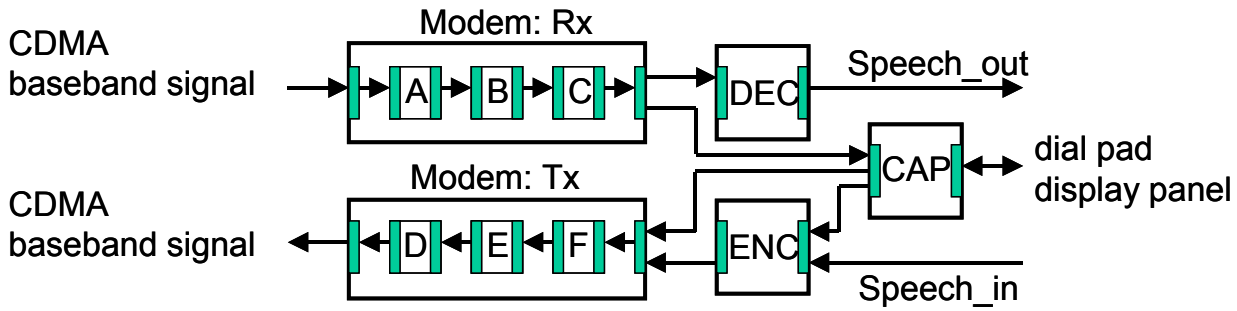


Figure 55. Schéma fonctionnel de la station mobile IS-95 CDMA

Les données codées de la voix de l'interlocuteur sont envoyées par la station de base, reçues par le récepteur du modem CDMA (module Rx) et décodées par la partie décodage du vocodeur (DEC).

Dans ce système, la voix (68 Kbps) de l'utilisateur du terminal mobile est codée (bloc ENC) pour 4 types de débit (9,6 Kbps, 4,8 Kbps, 2,4 Kbps, et 1,2 Kbps). Puis les données codées sont envoyées à la station de base par l'émetteur du modem CDMA (module Tx). La station de base et la station mobile communiquent entre elles sur la base de trames (20 ms par trame).

Dans le canal de transmission de trafic (speech_in), la taille de la trame d'entrée est 160 octets, la taille de la trame codée est 44 octets et la taille de la trame transmise est 1536 octets.

5.4.3 Solution architecturale à 4 processeurs

A partir de la spécification initiale du système en C++ nous avons bâti une spécification structurelle en SystemC. Les parties que nous avons décidé de raffiner jusqu'au niveau RTL sont les deux modems CDMA (Tx et Rx) et le vocodeur (ENC et DEC). Les autres parties du système (l'interface utilisateur (CAP) et le modèle de la station de base) restent modélisées en SystemC.

Partant de ce choix, nous avons suivi les 4 étapes de conception de notre flot : extraction des paramètres, conception de l'architecture, adaptation du logiciel et validation par cosimulation.

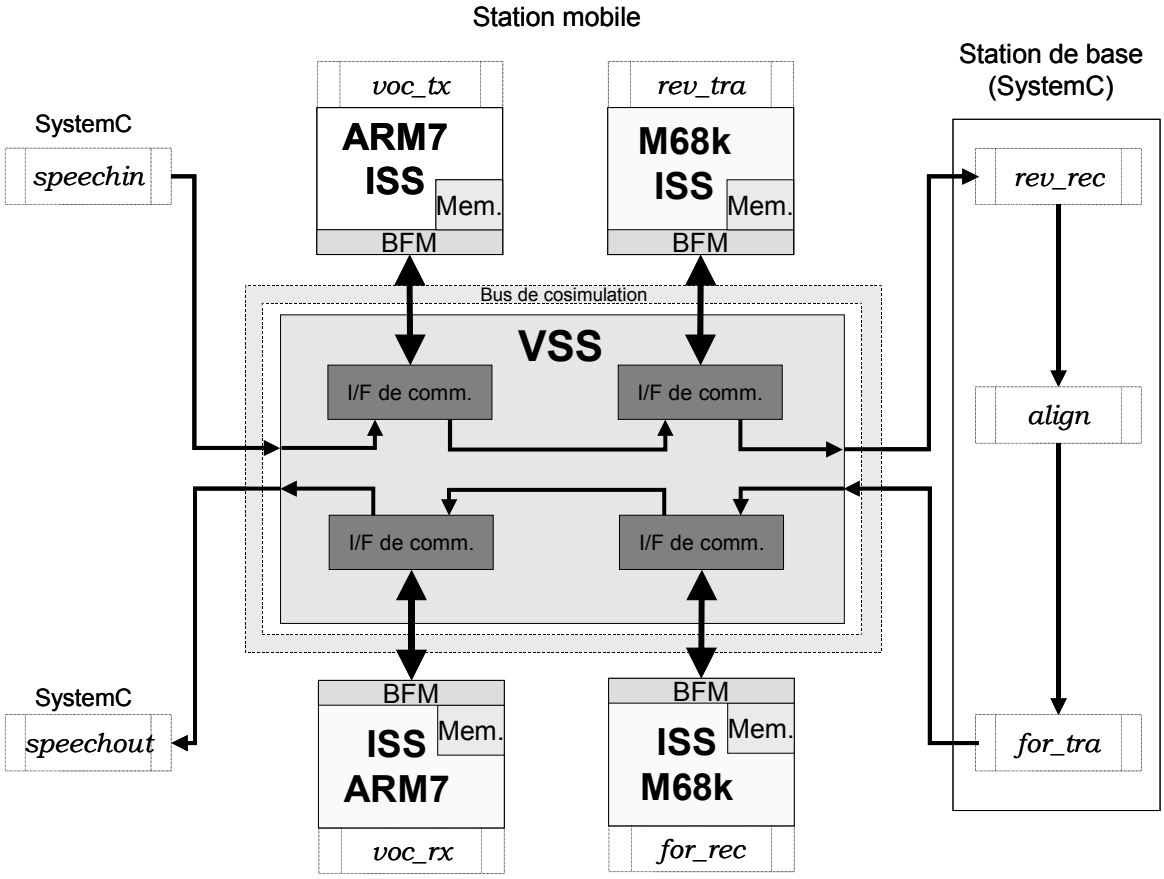


Figure 56. L'architecture de cosimulation à 4 processeurs du IS-95 CDMA

En analysant l'application et les besoins élevés en taux de calcul et de communication pour ce genre d'application nous avons choisi d'utiliser 4 processeurs. Deux processeurs ARM7 implémentent le vocodeur (ENC et DEC) et deux processeurs M68000 implémentent les modems CDMA (Tx et Rx). Ces processeurs sont connectés par l'intermédiaire d'un réseau de communication point à point. Dans un premier temps nous avons mis au point la table d'allocation et la description abstraite de l'architecture. Ensuite les 4 interfaces de communication ont été construites à partir d'éléments de la bibliothèque d'interfaces de communication. Si l'on se réfère à celle du CCP, l'architecture de communication ici est légèrement différente. En particulier le nombre et la taille des FIFOs des contrôleurs de communication sont différents. Ainsi nous avons reconstruit les 4 nouvelles interfaces de communication.

Par ailleurs, comme une version en C++ de l'application était disponible, nous avons pu extraire les 4 modules logiciels destinés à fonctionner sur les CPUs et nous avons généré les codes binaires correspondants (étape d'adaptation de logiciel). Rappelons que l'étape d'adaptation de logiciel inclut seulement l'insertion d'appels système dans un code logiciel existant.

Enfin nous avons validé l'architecture conçue par cosimulation. Le Figure 56 montre l'environnement de cosimulation mis au point pour valider l'application sur son architecture spécifique. Comme évoqué plus haut les modules externes (la station de base et l'entrée/sortie de l'utilisateur) ont été modélisés en SystemC.

5.4.4 Analyse de la durée de conception

Le Tableau 12 donne le temps passé sur chaque étape du flot de conception de la Figure 47 durant cette expérimentation. On peut noter que le temps nécessaire pour la génération manuelle dépend du nombre de processeurs (4 dans cet exemple) et qu'il constitue une fonction linéaire avec une pente de 8 heures/processeur.

Tableau 12. Délais nécessaires pour réaliser une implémentation du IS-95 CDMA selon notre approche

Étape	Délai (avec codage manuel)
Extraction des paramètres (table d'alloc. + archi. abstraite)	~ 2 hr x 4
Conception de l'architecture	~ 2 hr x 4 + 2 hr
Adaptation du logiciel	~ 4 hr x 4
Construction de l'env. de cosimulation	~ 2 hr x 4
Total	**~ 42 hr**

En fait, la génération manuelle prend non seulement beaucoup de temps mais elle est également très fastidieuse car il est toujours très difficile de traiter des applications complexes sans l'assistance d'outils de CAO. Il est important aussi de noter que nous supposons, lors de la génération manuelle, que le concepteur a une bonne connaissance des kits d'outils associés à chaque processeur ainsi que de l'application. Autrement le temps requis pour acquérir cette connaissance doit être ajouté. Cependant dans une génération automatique cette connaissance ne sera plus exigée.

5.5 Modem VDSL

5.5.1 But de l'expérimentation

Cette expérimentation a été menée en collaboration avec une équipe de la société STMicroelectronics. Elle porte sur un modem VDSL. Quatre aspects ont motivé la mise en œuvre de cette expérimentation : (1) montrer l'apport de la facilité de mettre plusieurs processeurs logiciels sur la même puce, (2) montrer comment notre modèle architectural supporte l'intégration de blocs préconçus (IPs), (3) utiliser cette application pour valider les outils de génération automatique développés par d'autres membres du groupe et qui étaient en phase finale de développement, (4) utiliser cette application pour une présentation publique (ex. University Booth at DAC 2001).

Dans cette section la première partie tente une introduction rapide aux techniques xDSL. La deuxième présente le système sur lequel porte l'expérience. Enfin la dernière décrit l'expérience menée et en tire un certain nombre de conclusions.

5.5.2 Introduction aux techniques xDSL (x Digital Subscriber Line)

La jonction entre abonné et central téléphonique est constituée de fils de cuivre dont les possibilités en terme de transport de signaux sont sous-utilisées car le réseau téléphonique a d'abord été conçu pour véhiculer la voix. La bande passante utilisée par les équipements de communication classiques est de l'ordre 3.3 kHz. Or les caractéristiques physiques des lignes de cuivre autorisent en réalité la transmission de signaux pouvant atteindre des fréquences de

l'ordre de 1 MHz. En modifiant les modems il est donc possible d'optimiser l'utilisation de ces lignes. En fonction de la distance séparant l'abonné de son central téléphonique, les paires de cuivre peuvent supporter des débits allant de 1.5 Mbits/s à 10 Mbits/s. Les technologies qui permettent d'exploiter ces possibilités sont appelées «xDSL» et sont toutes dérivées de la technologie DSL utilisée dans le cadre de liaisons numériques RNIS (le type de codage utilisé est le même).

La technologie xDSL se décompose en quatre groupes : ADSL, HDSL, SDSL et VDSL. A chacun de ces sous-groupes correspond une utilisation et des caractéristiques particulières :

- Actuellement l'ADSL (Asymmetric Digital Subscriber Line) est la technologie la plus au point et elle est commercialement prête.
- Le VDSL (Very high Data rate digital Subscriber Line) est une technologie voisine, permettant des débits plus élevés encore (jusqu'à 58 Mbps). Alcatel a dévoilé cette technologie au salon Télécom 99 en décembre 1999
- Les autres groupes correspondent à des spécifications en cours de normalisation et voisines de VDSL.

La technologie ADSL (Asymmetric Digital Subscriber Line) ou ligne asymétrique numérique utilise les fréquences supérieures à celles qui sont affectées au transport de la voix pour transmettre les données numériques. La traditionnelle modulation du courant électrique est remplacée par un procédé de transmission numérique DMT (Discrete Multitone) qui segmente la bande passante admissible du réseau téléphonique classique (qui est de 1,2 MHz) en sections de 4 kHz. L'utilisation de l'ASDL nécessite dans un premier temps une augmentation du spectre des fréquences reconnues pour le porter à 1,1 MHz. La bande de fréquence est découpée en 256 bandes indépendantes de 4 kHz chacune. Le travail d'un modem ADSL consiste à additionner ces canaux pour atteindre le débit maximum.

Il s'agit d'une technologie asymétrique : le débit des données émises est plus faible que celui des données reçues. De plus le débit, dans les deux cas, varie avec la distance à parcourir. La liaison se trouvant entre l'abonné et le central est divisée en trois canaux de transmission :

- Le haut de la bande (1MHz) est réservé au canal descendant unidirectionnel (central – abonné) à débit élevé (8 Mbits/s au maximum). L'efficacité de cette technique dépend des caractéristiques de la ligne (notamment sa longueur) ; c'est pourquoi seulement 50 % de la population française peut espérer obtenir une liaison à 8 Mbits/s alors que le reste devra se contenter d'une liaison à 4 Mbits/s.
- En milieu de bande (entre 300 et 700 kHz) se trouve un canal bidirectionnel à débit moyen (entre 640 et 800 kbits/s) utilisé pour émettre les données.
- Le troisième canal est réservé soit à la téléphonie analogique classique (entre 0 et 4 kHz) soit au RNIS (entre 0 et 80 kHz). Avec cette technologie l'abonné peut téléphoner et se connecter à l'Internet en même temps sur une seule prise téléphonique classique. A l'extrémité de la ligne de cuivre, les fréquences vocales sont acheminées vers le réseau téléphonique classique tandis que les données sont dirigées vers le réseau Internet.

La technologie VDSL emploie des procédés voisins de codage et propose une simplification du protocole. Cependant cette technique, encore en développement, n'est pas finalisée. En effet de nombreuses entreprises proposent leur propre version du modem VDSL.

5.5.3 Présentation du modem VDSL expérimental

STMicroelectronics, en partenariat avec Telia, a proposé une version du modem VDSL appelé Zipper-DMT. Cette version utilise le codage DMT (Discret Multi-Tone) qui découpe la bande de fréquence initiale en sous-bandes de transmission simultanée. Le Zipper découpe la bande de fréquence en 2048 sous-bandes qui peuvent être allouées dynamiquement et par logiciel à différents utilisateurs. Une particularité du modem VDSL est de calculer le débit optimal en fonction de la qualité de la ligne (longueur, état, etc.). Un prototype du modem utilisant cette technique a été développé et implémenté sur plusieurs cartes. Le schéma fonctionnel de ce prototype est présenté Figure 57. C'est la partie grisée de ce modem dont nous allons refaire la conception sous forme d'une AMM.

Cette architecture contient un circuit ASIC assurant la mise en forme des signaux en émission et en réception, deux FPGA implémentant les fonctions de traitement des signaux émis et reçus, un DSP implémentant des fonctions de traitement de signal et qui permet aussi, avec un stockage périodique de données transmises, de mesurer le débit maximal supporté par la ligne, et un microcontrôleur chargé de contrôler, configurer et synchroniser la chaîne de transmission en fonction de l'état de la ligne. Ce même microcontrôleur assure l'interfacage avec le PC hôte. Pour gérer les nombreuses tâches du microcontrôleur, un système d'exploitation commercial avait été intégré.

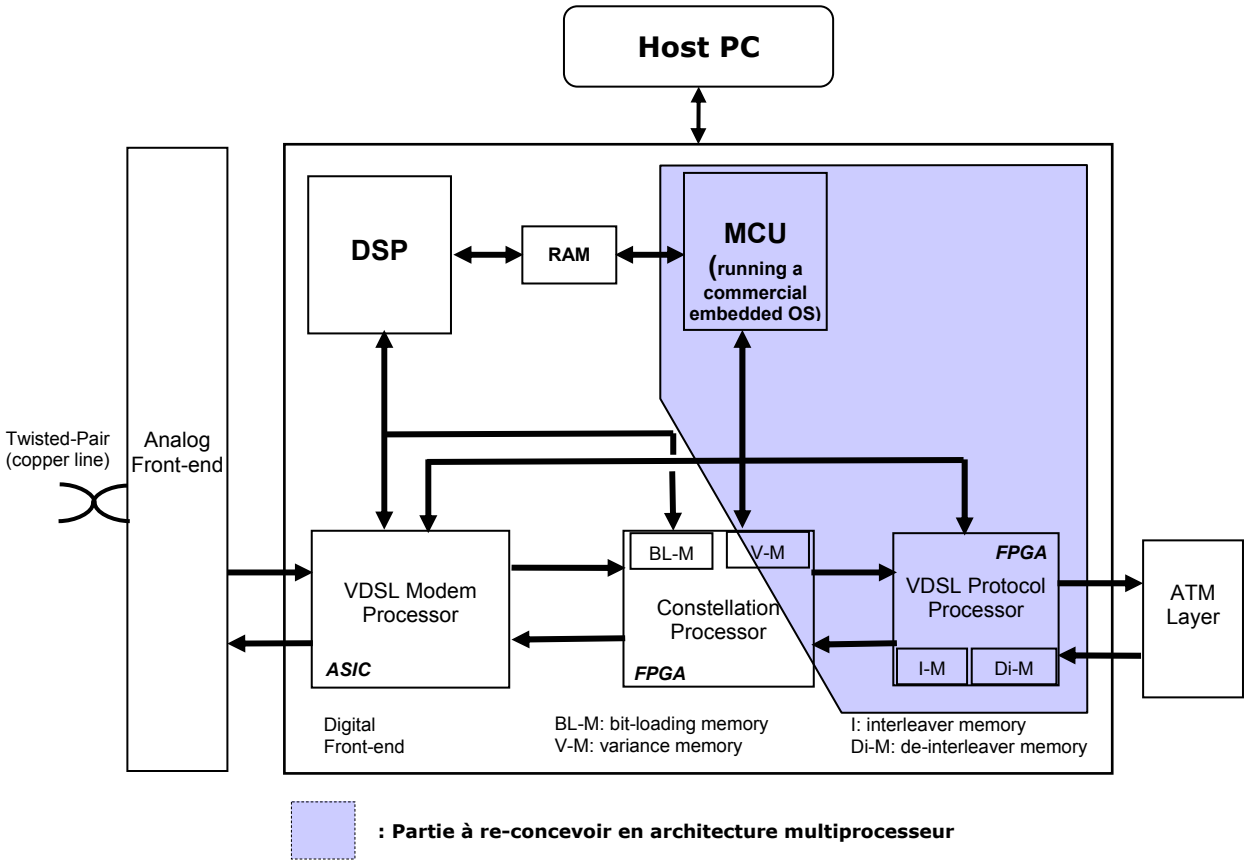


Figure 57. Schéma fonctionnel du prototype développé par STMicroelecronics du modem VDSL. En gris la partie à reconcevoir.

Nous avons donc décidé de reconstruire la partie grisée de la Figure 57 en proposant une architecture multiprocesseur. Une telle architecture sera plus flexible et permettra une

adaptation rapide au changement de norme (très probable pour ce genre d'application). L'autre raison qui nous a poussé à proposer une architecture multiprocesseur est de décharger le microcontrôleur d'une partie de ses tâches de contrôle.

5.5.4 Spécification de l'application

La spécification du modem VDSL que nous avons obtenue de STMicroelectronics comprenait une description de l'architecture, une description informelle des différentes tâches logicielles, et une implémentation en C++ du bloc TX_Framer. Le TX_Framer constitue la chaîne de traitement numérique du flot de données (en émission). A partir de ces données nous avons pu écrire une spécification «allégée» en SystemC de la partie à re-concevoir. La Figure 58 montre le schéma fonctionnel du sous-système VDSL (la partie à re-concevoir) modélisé en SystemC. Ainsi le module M1 est le bloc IP du TX_Framer. Le Module M2 regroupe plusieurs sous-modules qui sont les différentes tâches exécutées par le microcontrôleur de la Figure 57. Enfin un environnement communicant avec ce système a été aussi modélisé en SystemC (modules ATM, PMD et PC Host).

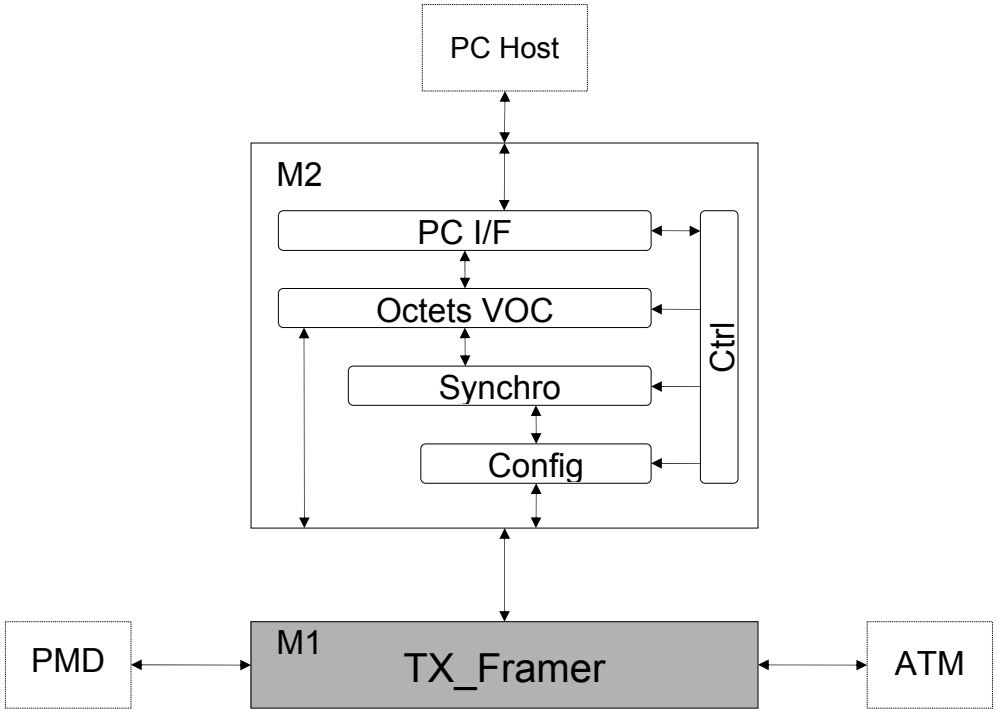


Figure 58. Schéma fonctionnel du sous-système VDSL modélisé en SystemC.

Il faut mentionner ici que le modèle RTL du TX_Framer n'était pas disponible. Nous avons donc développé ce modèle à partir des spécifications dont nous disposions. Ce modèle a été développé en SystemC RTL. La Figure 59 donne l'architecture de l'IP développé. Le code des six sous-blocs de traitement numérique du signal (Stuffer, Scrambler, Reed Solomon, Interleaver, Merger, Synchro) est écrit en SystemC RTL.

La communication entre le module M2 et ce bloc IP se fait par écriture/lecture de registres, à l'exception de l'insertion des octets VOC (cf. Figure 58) qui se fait elle par un protocole de poignée de main avec FIFO de taille 72 octets.

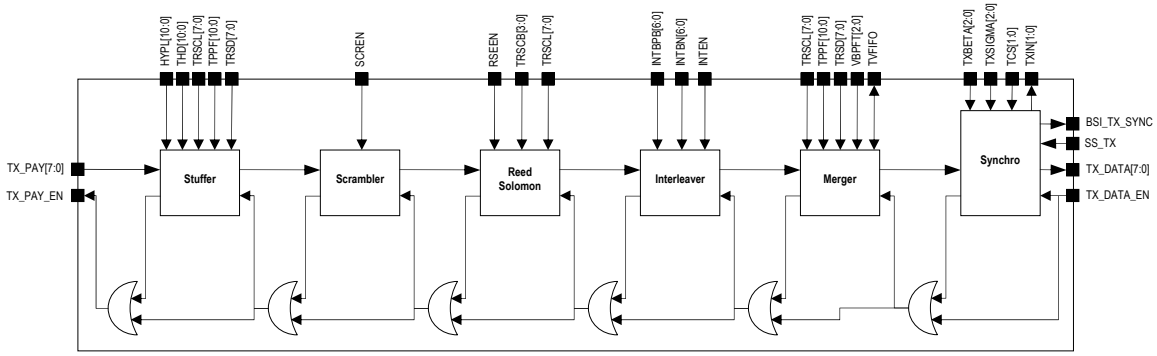


Figure 59. L'architecture de l'IP (TX_Framer)

5.5.5 Solution architecturale multiprocesseur

Avant de commencer à concevoir l'architecture il faut en fixer les paramètres, c'est-à-dire décider du partitionnement, choisir les protocoles de communication et fixer les autres différents paramètres (chapitre 4). Dans l'architecture proposée par ST le module M1 est implémenté sur un microcontrôleur Hitachi SH-3. En étudiant les contraintes aussi que les objectifs énoncés aux sections 5.5.1 et 5.5.3, nous avons opté pour l'architecture multiprocesseur décrite Figure 60. Notons que le nombre de tâches associées au module M2, dans cette version «allégée», est presque trois fois plus réduit que le nombre réel. Ceci accroît la nécessité de les distribuer sur plusieurs processeurs pour atteindre les performances et la flexibilité requises par la norme VDSL.

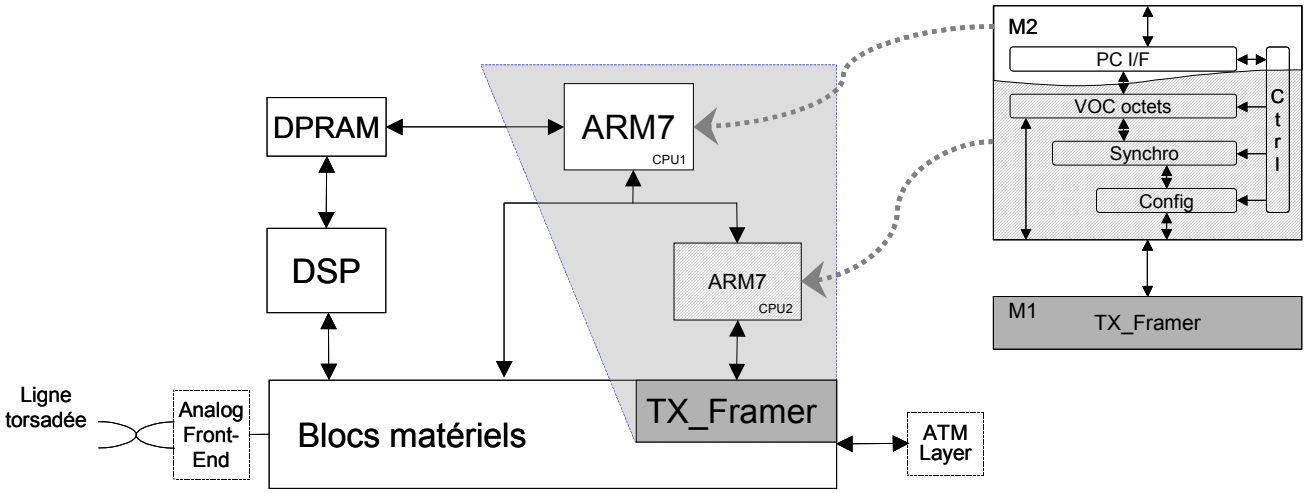


Figure 60. Schéma fonctionnel modifié du modem VDSL.

5.5.5.1 Extraction des paramètres

Suite à l'étape de partitionnement présentée ci-dessus, l'architecture cible comportera deux processeurs ARM7 et un bloc matériel spécifique (IP). En analysant ce partitionnement et en utilisant notre modèle architectural nous avons obtenu l'architecture abstraite de la Figure 61.

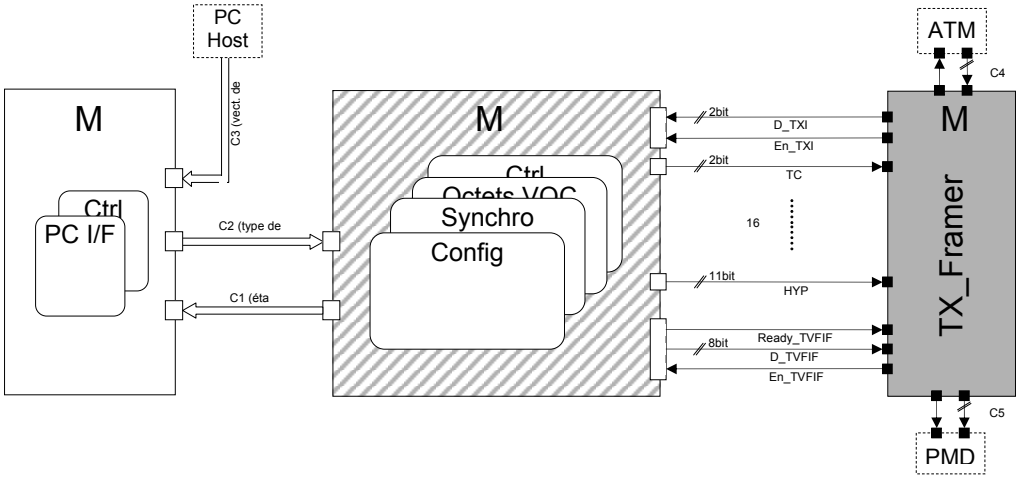


Figure 61. Architecture abstraite du sous-système VDSL.

En utilisant un réseau de communication point à point, en respectant les protocoles de communication préfixés du module M1 (l'IP), et en choisissant des protocoles de communication adaptés pour les canaux abstraits entre les modules M2 et M3, nous obtenons la table d'allocation ci-dessous.

Tableau 13. Table d'allocation : implémentation du sous-système VDSL.

Module	CPU	Taille mémoire	Canal de communication	Protocole de comm.	Taille buffer	IT	Adr.
M3	ARM7 40 MHz	ROM: 10KB RAM: 20KB	C1 (M2⇒M3)	FIFO + HSK	0	IRQ	0x7000
			C2 (M3⇒M2)	FIFO + HSK	5 octets	–	0x7004
			C3 (env⇒M3)	FIFO + HSK	0	IRQ	0x7008
M2	ARM7 40 MHz	ROM: 10KB RAM: 20KB	TXIN (M1⇒M2)	Reg_R	2 bits	–	0x7000
			TCS (M2⇒M1)	Reg_W	2 bits	–	0x7004
			TXBETA (M2⇒M1)	Reg_W	3 bits	–	0x7008
			TXSIGMA (M2⇒M1)	Reg_W	3 bits	–	0x700C
			VBPFT (M2⇒M1)	Reg_W	3 bits	–	0x7010
			INTEN (M2⇒M1)	Reg_W	1 bit	–	0x7014
			INTBN (M2⇒M1)	Reg_W	7 bits	–	0x7018
			INTBPB (M2⇒M1)	Reg_W	7 bits	–	0x701C
			TRSD (M2⇒M1)	Reg_W	8 bits	–	0x7020
			TPPF (M2⇒M1)	Reg_W	11 bits	–	0x7024
			TRSCL (M2⇒M1)	Reg_W	8 bits	–	0x7028
			TRSCB (M2⇒M1)	Reg_W	4 bits	–	0x702C
			RSEEN (M2⇒M1)	Reg_W	1 bit	–	0x7030
			SCREN (M2⇒M1)	Reg_W	1 bit	–	0x7034
			THD (M2⇒M1)	Reg_W	11 bits	–	0x7038
			HYPL (M2⇒M1)	Reg_W	11 bits	–	0x703C
			C_TVFIFO (M2⇒M1)	TVFIFO	72 octets	IRQ	0x7040
			C1 (M2⇒M3)	FIFO+HSK	1 octet	–	0x7044
			C2 (M3⇒M2)	FIFO+HSK	0	IRQ	0x7048
M1	IP VHDL RTL 200 MHz	–	TXIN (M1⇒M2)	Reg_W	0	–	–
			TCS (M2⇒M1)	Reg_R	0		
			TXBETA (M2⇒M1)	Reg_R	0		
			TXSIGMA (M2⇒M1)	Reg_R	0		
			VBPFT (M2⇒M1)	Reg_R	0		
			INTEN (M2⇒M1)	Reg_R	0		
			INTBN (M2⇒M1)	Reg_R	0		
			INTBPB (M2⇒M1)	Reg_R	0		
			TRSD (M2⇒M1)	Reg_R	0		
			TPPF (M2⇒M1)	Reg_R	0		
			TRSCL (M2⇒M1)	Reg_R	0		

			TRSCB (M2⇒M1)	Reg_R	0		
			RSEEN (M2⇒M1)	Reg_R	0		
			SCREN (M2⇒M1)	Reg_R	0		
			THD (M2⇒M1)	Reg_R	0		
			HYPL (M2⇒M1)	Reg_R	0		
			C_TVFIFO (M2⇒M1)	TVFIFO	0		
			C4 ((env⇒M1))	Reg_R	0		
			C5 (M1⇒env)	Reg_W	0		
Env	VHDL 200 MHz	–	C3 (env⇒M3)	FIFO+HSK	–	–	–
			C4 (env⇒M1)	Reg_W			
			C5 (M1⇒env)	Reg_R			

5.5.5.2 Conception de l'architecture

Trois interfaces de communication doivent être conçues pour adapter les trois processeurs (l'IP est un processeur matériel spécifique) au réseau de communication. Comme dans ce cas le réseau de communication est un réseau point à point, et comme le module M3 est au niveau RTL, il n'a donc pas besoin d'une interface d'adaptation. Une seule interface côté M2 suffira. Par ailleurs la construction de cette interface nécessite le développement de contrôleurs de communication adéquats (Tableau 13) lesquels n'existaient pas encore dans notre bibliothèque. Ainsi nous avons enrichi notre bibliothèque d'interfaces de communication par le développement de trois nouveaux contrôleurs de communication : FIFO dédiée à l'application VDSL, registre en écriture (registre de configuration), et registre en lecture (registre d'état) (Figure 62).

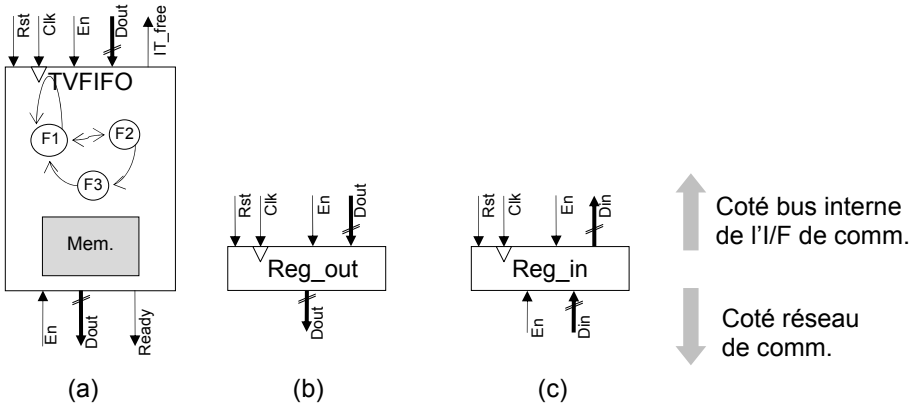
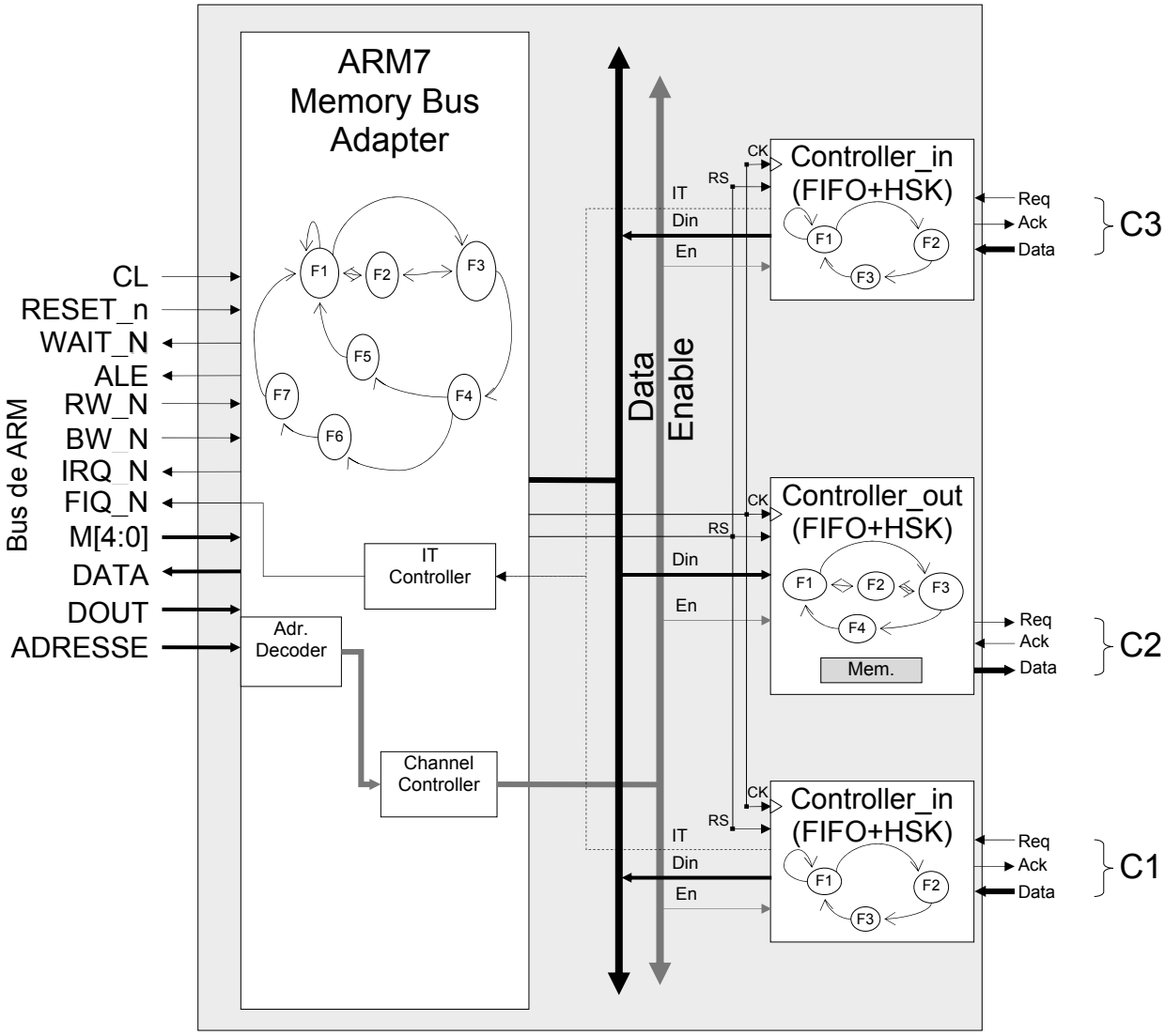


Figure 62. Les nouveaux contrôleurs de communication ajoutés à la bibliothèque d'interface : (a) FIFO dédiée à l'application VDSL (b) registre en écriture (c) registre en lecture.

Notons que l'utilisation d'une FIFO spécifique (par sa gestion, sa taille) au cahier de charge du VDSL rend l'architecture résultante encore plus dédiée à l'application (conformément à nos objectifs) et par conséquent encore plus optimale (en termes de surface, coût et performance).

Nous avons ainsi construit les deux interfaces de communication des modules M1 et M2 (Figure 63).


ARM7
Memory Bus
Adapter
F1
F2
F3
F4
F5
F6
F7
CL
RESET_n
WAIT_N
ALE
RW_N
BW_N
IRQ_N
FIQ_N
M[4:0]
DATA
DOUT
ADRESSE
Bus de ARM
IT
Controller
Adr.
Decoder
Channel
Controller
Data
Enable
CK
RS
IT
Din
En
Controller_in
(FIFO+HSK)
Controller_out
(FIFO+HSK)
Mem.
Req
Ack
Data
C3
C2
C1

(a)

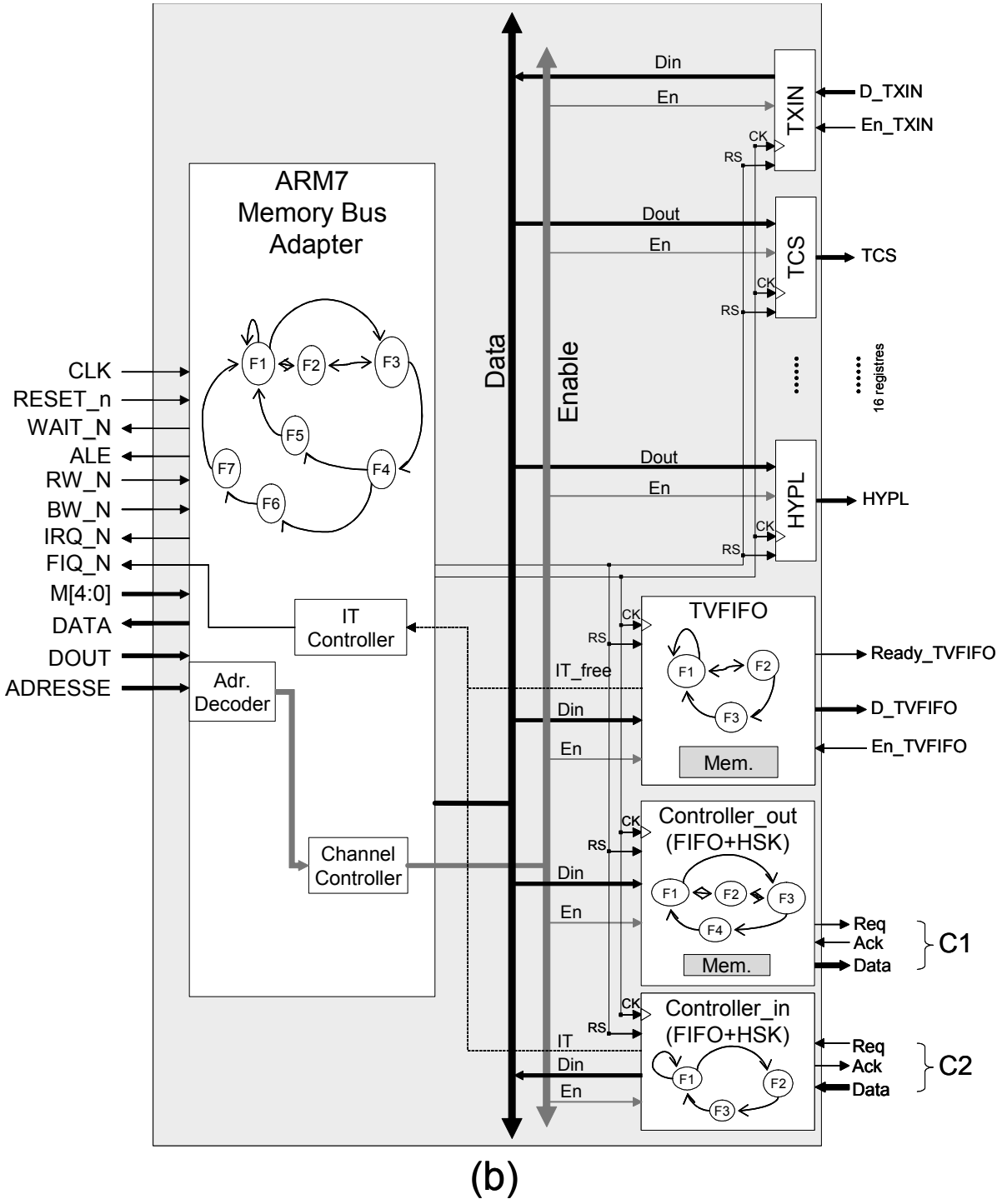


Figure 63. Implémentation du sous-système : (a) interface de communication du module M3 (b) interface de communication du module M2

5.5.5.3 Adaptation du logiciel

Lors de l'étape de spécification nous avons écrit les codes des tâches des modules M1 et M2 en SystemC à un niveau d'abstraction élevé (macro-architecture). Dans notre expérimentation, nous n'avons pas utilisé de système d'exploitation (notre but étant de valider l'architecture

matérielle). L'ordonnancement des tâches s'est fait par entrelacement de leurs codes (par «*switch case*»).

Donc nous avons écrit les vecteurs d'interruption correspondants et nous avons remplacé dans les codes des tâches les opérations d'E/S abstraites par des opérations au niveau RTL selon les valeurs de la table d'allocation (Tableau 13). Puis nous avons suivi les mêmes étapes de compilation et d'édition de liens qu'en 5.2.3.3 et avons obtenu les codes binaires correspondants, prêts à être exécutés sur les deux processeurs ARM7 de l'architecture.

5.5.5.4 Validation par cosimulation

Nous avons construit l'environnement de cosimulation contenant deux ISS ARM7 et le simulateur VHDL (VSS) (Figure 64). Tous les blocs matériels ont été regroupés (les deux interfaces de communications (Figure 63), le module M3 (l'IP) et les connexions entre ces différents blocs (réseau de communication point à point)) dans un seul bloc (VHDL RTL) qui est exécuté par le simulateur VSS.

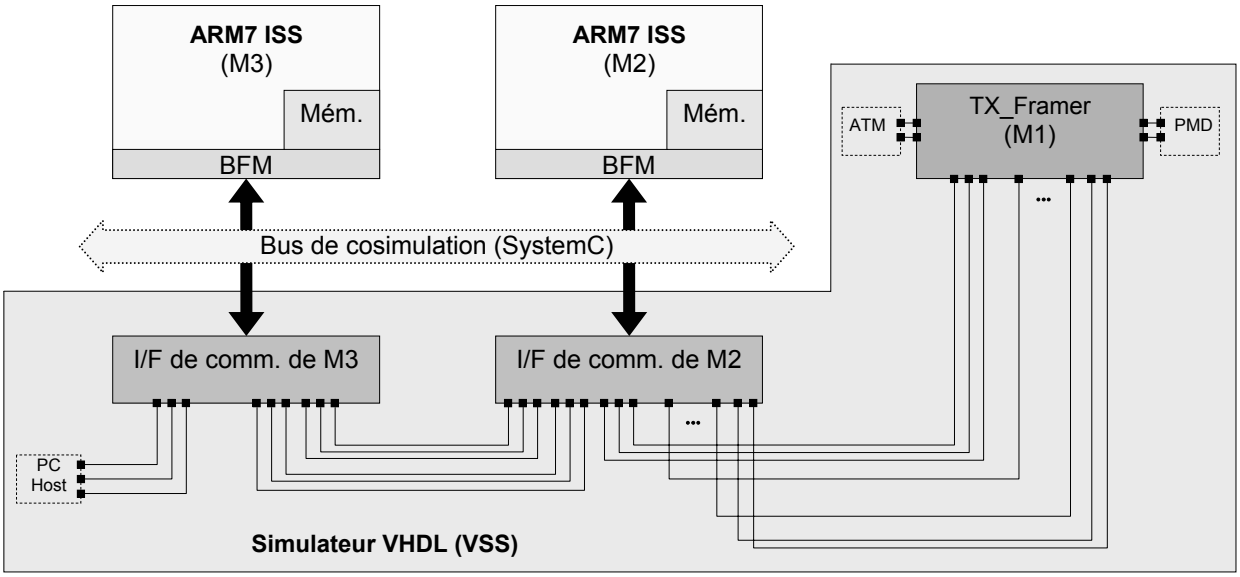


Figure 64. L'architecture de cosimulation du sous-système VDSL.

Après une étape de déboguage (erreurs dues au codage manuel), nous avons pu vérifier le bon fonctionnement du système par comparaison avec les vecteurs de tests fournis par la société ST. La Figure 65 présente les chronogrammes obtenus lors de la cosimulation RTL en parallèle avec ceux obtenus lors de la simulation fonctionnelle (au niveau macro-architecture). On rappelle que c'est la cosimulation RTL qui contient les informations significatives sur le temps d'exécution (précision au cycle près).

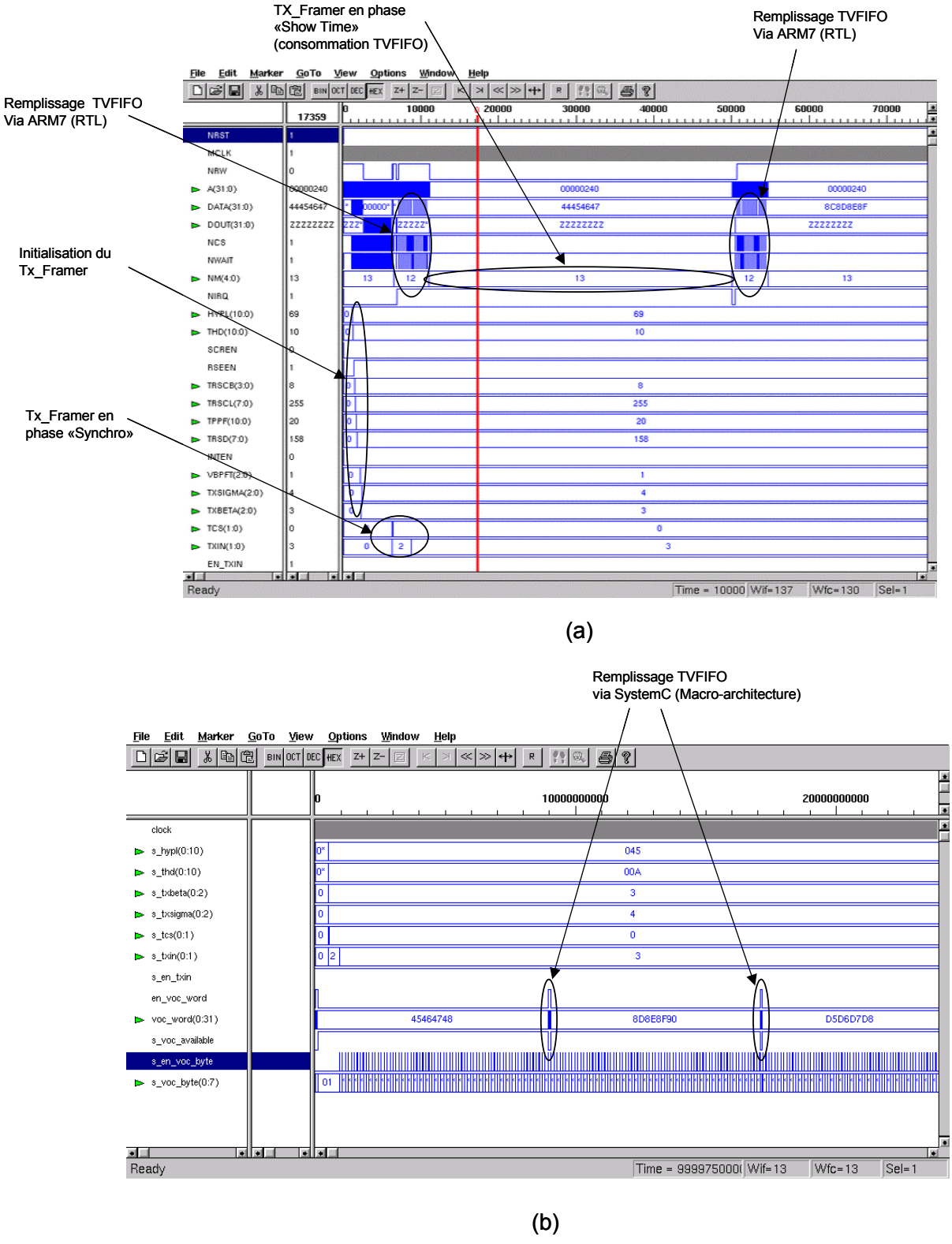


Figure 65. Chronogrammes issus de l'implémentation du sous-système VDSL : (a) cosimulation RTL, (b) simulation fonctionnelle

Ainsi nous avons mesuré les temps d'exécution (Tableau 14) et effectué la synthèse des deux interfaces de communication (Tableau 15).

Tableau 14. Implémentation du sous-système VDSL : temps d'exécution.

Opération	Temps d'exécution (cycles)
Lecture d'un registre	2
Écriture d'un registre	4
Remplissage de la TVFIFO (72 octets)	190
Latence de prise en compte d'une interruption	23
Consommation de la TVFIFO (1 octet/trame)	1619

Tableau 15. Implémentation du sous-système VDSL : synthèse des interfaces de communication.

I/F de communication	Nombre de portes	Surface (mm^2) (AMS CUP 0,6µ)	Délai du chemin critique (ns)
I/F de M2	3795	1,11	6,16
I/F de M3	3284	0,96	5,95
Nombre de lignes de code (VHDL RTL)			2168

Cette nouvelle architecture dédiée à l'application VDSL permet d'obtenir des performances optimales en terme de débit et surface. De plus l'ajout aisé de processeurs rend l'architecture résultante plus flexible (pour d'adapter aux éventuelles évolutions du norme VDSL).

Nous avons montré par cette application que notre modèle architectural ainsi que notre flot de conception supportent l'intégration de blocs préconçus (IPs). Il suffit d'enrichir la bibliothèque d'interfaces de communication avec les contrôleurs de communication adéquats.

5.6 Conclusion

Dans ce chapitre un flot d'implémentation systématique a été présenté. Il est basé sur le modèle architectural proposé et permet une conception systématique d'AMM dédiées à des applications spécifiques. Les différentes étapes du flot ont été détaillées. Ces étapes permettent de raffiner une architecture abstraite au niveau macro-architecture en une architecture RTL (micro-architecture) matérielle/logicielle. L'architecture est conçue de façon systématique en instanciant des composants de base dans une bibliothèque. Ce flot permet aussi de valider le système final en utilisant une approche de cosimulation aux différents niveaux d'abstraction. Nous avons présenté les trois expérimentations que nous avons effectuées durant cette thèse et qui nous ont permis de prouver la faisabilité de notre approche et d'analyser son efficacité.

Grâce à ces expérimentations nous avons pu évaluer l'efficacité de notre approche. Cette approche systématique permet une réduction considérable du coût de conception, ce qui représente un énorme gain en terme économique et une solution très prometteuse pour la diminution (voire la disparition) de l'écart entre le progrès de la technologie et celui de la productivité.

Chapitre 6

Conclusions et perspectives

Sommaire

6.1 Conclusions

Les prévisions stratégiques d'ITRS annoncent que 70% des ASIC comporteront au moins un CPU embarqué à partir de l'année 2005. Ainsi la plupart des ASICs seront des systèmes monopuces. Cette tendance semble non seulement se confirmer mais se renforcer : les systèmes monopuces contiendront des réseaux formés de plusieurs processeurs dans le cas d'applications telles que les terminaux mobiles, les set-top box, les processeurs de jeux et les processeurs de réseau. La complexité de ces applications ne cesse de croître tandis que les délais de mise sur le marché doivent diminuer sensiblement. Des AMM accompagnées de nouvelles méthodologies de conception adaptées semblent être incontournables pour relever ce défi. Ces méthodologies doivent proposer de nouvelles approches pour les deux partie d'un flot de conception complet : la partie exploration d'architectures et la partie conception de l'architecture choisie.

Ainsi nous avons présenté une analyse montrant l'importance des AMM dédiées à des applications spécifiques. Nous pensons que ce type d'architecture est la solution optimale pour répondre aux besoins actuels et futurs de complexité, performance, coût, et portabilité. L'analyse de l'état de l'art –académique et industriel– nous a éclairé sur la grande quantité et la grande variété d'architectures et d'éléments architecturaux existants dans ce domaine. Cependant les flots de conception existants ne présentent pas de solutions complètes et ne sont pas capables de supporter ces différents composants architecturaux.

Suite à cette étude nous avons constaté l'importance et le besoin d'un modèle architectural générique qui ne doit pas présenter de limitations pour l'intégration des différents réseaux de communication, processeurs et IPs dédiés à des domaines d'applications spécifiques. Ainsi nous avons proposé un modèle multiprocesseur flexible, modulaire et extensible. Ce modèle nous permet de couvrir un très large domaine d'applications. Il est basé sur le concept de séparation entre comportement et communication et permet surtout une conception systématique de l'architecture basée sur l'assemblage de composants de bibliothèques. Nous avons aussi présenté notre définition de plateformes architecturales, basées sur notre modèle, et dédiées à une classe d'applications. Cette notion permet de diminuer l'espace de solutions architecturales à explorer. Une analyse de l'efficacité de ce modèle est présentée en conclusion.

Concernant l'exploration d'architectures, une nouvelle approche pour une exploration efficace de l'espace de solutions architecturales a été proposée. Cet espace de solutions est énorme dans le cas des architectures multiprocesseurs hétérogènes. L'approche est basée sur une méthode d'estimation de performance au niveau système. Elle assure un compromis optimal entre rapidité et précision. Elle est basée sur un outil de codesign (MUSIC). Cet outil est utilisé pour obtenir une estimation du temps d'exécution (matériel et logiciel) de chaque bloc de base dans la spécification système (en SDL). Cette spécification est ensuite instrumentée par les informations sur le temps d'exécution. Une boucle d'exploration au niveau système permet (en prenant en considération une plateforme architecturale ou le modèle architectural complet) de trouver le meilleur partitionnement matériel/logiciel avec les composants architecturaux alloués. L'efficacité de cette approche a été validée par un exemple d'application industrielle.

Enfin un flot d'implémentation systématique a été présenté. Il est basé sur le modèle architectural proposé et permet une conception systématique d'AMM dédiées à des applications spécifiques. Les différentes étapes du flot ont été détaillées. Ces étapes permettent de raffiner une architecture abstraite au niveau macro-architecture en une architecture RTL (micro-architecture) matérielle/logicielle. L'architecture est conçue de façon systématique en instanciant des composants de base dans une bibliothèque. Ce flot permet aussi de valider le système final en utilisant une approche de cosimulation aux différents niveaux d'abstraction. Nous avons présenté les trois expérimentations que nous avons effectuées durant cette thèse et qui nous ont permis de prouver la faisabilité de notre approche et d'analyser son efficacité. Ces trois expérimentations différentes nous ont permis de comparer plusieurs aspects mettant en valeur le modèle architectural et la méthodologie de conception proposés. L'automatisation de ce flot est en cours et fait l'objet de plusieurs thèses au sein du groupe SLS.

Signalons enfin que notre travail rentre dans le cadre d'un grand projet de recherche. Ce projet vise à développer des outils et méthodologies innovatrices qui permettront aux concepteurs d'explorer et de concevoir les SoC de demain. Ainsi les autres aspects de cette recherche (tels que le ciblage logiciel et la conception de la couche OS, les mémoires partagées, la spécification multiniveaux et la cosimulation multiniveaux, la formalisation et l'automatisation de ces approches, ainsi que le développement des outils de génération automatique) font partie d'autres thèses s'effectuant en parallèle au sein du groupe SLS, car notre objectif est de construire un environnement complet. Ainsi, l'ensemble du travail présenté dans cette thèse constitue une des bases nécessaires au développement des architectures multiprocesseurs monopuces dédiées à des applications spécifiques.

6.2 Perspectives

Durant cette thèse, les circonstances nous ont amené à séparer les deux études exploration d'architectures d'une part et implémentation de l'architecture choisie d'autre part. Ces études ont mis en évidence la grande importance de ces deux composantes. Une continuité naturelle de ces travaux consiste à lier ces deux composantes (exploration et implémentation) pour construire un flot de conception complet. Cet axe de recherche est une des préoccupations principales du groupe SLS.

Concernant la méthode d'estimation de performance, il est important de poursuivre la recherche pour prendre en considération les autres contraintes telles que la surface, le coût et la consommation, et effectuer ainsi une exploration multi-objectifs.

Quant à la conception de l'architecture multiprocesseur monopuce, nous avons noté que le point central est l'architecture de communication. Par conséquent il est particulièrement intéressant d'étudier et d'analyser l'impact du choix de diverses architectures de communication sur les performances, et aussi de développer de nouvelles architectures de communication génériques de haute performance.

Un objectif ultime sera de trouver des «*algorithmes intelligents*» qui proposeraient au concepteur les architectures les mieux adaptées à son application et à ses contraintes. Il sera particulièrement intéressant d'entreprendre des recherches dans cette direction.

Glossaire

- **AMM** : Architecture Multiprocesseur Monopuce.
- **CCP** : Commutateur de Cheminement de Paquets
- **SoC** : System on a Chip (système sur une puce, ou système monopuce).
- **MCU** : Microcontroller Unit.
- **RTL** : Register Transfer Level (niveau transfert de registres).
- **ASIC** : Application Specific Integrated Circuit (Circuits Intégrés à Applications Spécifiques) Circuits intégrés conçus pour un usage particulier selon les exigences d'un client.
- **API** : Application Programming Interface, c'est l'ensemble des fonctions proposées par un logiciel pour permettre son utilisation par des programmes.
- **FIFO** : First In First Out, classe de protocole de communication qui assure que les premières données envoyées sont les premières données reçues.

- **DSP** : Digital Signal Processor (processeur de traitement du signal numérique).
- **ISS** : Instruction Set Simulator, simulateur de processeurs reproduisant l'exécution de leurs instructions.
- **OS** : Operating System (système d'exploitation).
- **CPU** : Central Processor Unit (unité de traitement logiciel).
- **SDL** : System Description Language (langage de description au niveau système).
- **IP** : Intellectual property (bloc réutilisable).

Références

[1] Ackland B. et al., "A Single-Chip 1.6 Billion 16-b MAC/s Multiprocessor DSP", Custom Integrated Circuits Conference, 1999.

[2] Albrecht T. W., Notbauer J., and Rohringer S., "HW/SW CoVerification Performance Estimation and Benchmark for a 24h Embedded RISC Core Design", Proc. Design Automation Conf., June 1998.

[3] AMBA specification (REV 2.0) ARM Limited, 13 may 1999. http://www.arm.com

[4] Arexsys, http://www.arexsys.com.

[5] Bacquet P., "Translating SDL into OPNET: From formal validation to performance evaluation", in Proc. of OPNETWORK Conference, Mil3, mai 1997.

[6] Bainbridge W., "Asynchronous Macrocell Interconnect Using Marble", Proceedings of the Fourth International Symposium on Advanced Research in Asynchronous Circuits and Systems, 1998, pp: 122-132

[7] Balarin F., Sentovich E., Chiodo M., Giusto P., Hsieh H., Tabbara B., Jurecska A., Lavagno L., Passerone C., Suzuki K., and Sangiovanni-Vincentelli A., "Hardware-Software Co-design of Embedded Systems – The POLIS Approach.", Kluwer Academic Publishers, 1997.

[8] Benini L., Bruni D., Chinosi M., Silvano C., Zaccaria V., and Zafalon R., "A Power Modeling and Estimation Framework for VLIW-based Embedded Systems", in Proc. Int. Workshop on Power And Timing Modelling, Optimization and Simulation PATMOS, Sept. 2001.

[9] Berekovic M., Heistermann D., and Pirsch P., "A Core Generator for Fully Synthesizable and Highly Parameterizable RISC-cores for System-on-chip", Design Signal Processing Systems, 1998. SIPS 98. 1998 IEEE Workshop on , 1998 , pp: 561-568

[10] Bergamaschi R. A. and Lee W. R., "Designing Systems-on-Chip Using Cores", Proc. Design Automation Conf., June 2000.

[11] Calvez J. P., Heller D., and Pasquier O., "System performance modeling and analysis with VHDL: benefits and limitations", In Proc. of VHDL forum for CAD in Europe Conference, 1995.

[12] Calvez J.P., "A System-Level Performance Model and Method". In "Current Issues in Electronic Modelling", Vol 6- Meta-Modelling. Performance and Information Modelling. Editors: J.M. Berge, O. Levia, J. Rouillard. Kluwer Academic Publishers, 1996, pp 57-102

[13] Calvez J.P., Pasquier O., and Heller D., "Hardware/Software System Design Based on the MCSE Methodology", In, "Current Issues in Electronic Modeling", Vol 9- System Design. Editors: J.M. Berge, O. Levia, J. Rouillard. Kluwer Academic Publishers, March 1997.

[14] Chang H., Cooke L., Hunt M., Martin G., McNelly A., and Todd L., "Surviving the SOC Revolution, A Guide to Platform-Based Design", Kluwer Academic Publishers, 1999, ISBN 0-7923-8679-5.

[15] Chou P., Ortega R., Hines K., Partridge K., and Borriello G., "IPChinook: an integrated IP-based design framework for distributed embedded systems", proceedings of DAC 1999.

[16] CM, "The CM-5 Connection Machine: A Scalable Supercomputer", W. Daniel Hillis and Lewis W. Tucker., Communications of the ACM, November 1993, Vol. 36, No. 11.

[17] CORBA Services, Object Management Group, Common Object Services Specification. Technical Rapport, OMG, July 1997.

[18] Coste P., Hessel F., Le Marrec Ph., Sugar Z., Romdhani M., Suescun R., Zergainoh N., and Jerraya A. A., "Multilanguage Design of Heterogeneous Systems", CODES Workshop on Hardware/Software Codesign, May 1999.

[19] Coware, Inc., "N2C", available at http://www.coware.com/cowareN2C.html.

[20] Culler D. E., Jaswinder P. S., and Gupta A., "Parallel Computer Architecture, A Hardware/software approach", Morgan Kaufmann Inc, San Francisco California, 1999, ISBN 1-55860-343-3.

[21] Daveau J., Marchioro G., Ben Ismail T., Jerraya A., "Protocol Selection and Interface Generation for Hw-Sw Codesign", IEEE Trans. on VLSI Systems, vol. 5, p. 136-144, 1997.

[22] Dawson W.K., and Dobinson R.W., "Buses and bus standards", Computer Standards & Interfaces 20, (1999), pp: 201-224

[23] De Kock E. A., Essink G., Smits W. J. M., Wolf P., Brunel J.-Y., Kruijtzer W.M., Lieverse P., Vissers K.A., "YAPI: Application Modeling for Signal Processing Systems", In the 37th Design Automation Conference, June 2000.

[24] Dick R. P. and CORDS N. Jha., "Hardware-Software Co-Synthesis of Reconfigurable Real-Time Distributed Embedded Systems", Proc. Int. Conf. on Computer Aided Design, November 1998.

[25] Eggers S.J., Emer J.S., Levy H.M., Lo J.L., Stamm R.L., and Tullsen D.M., "Simultaneous multi- threading: a platform for next-generation processors", IEEE Micro, Volume: 17 5 , Sept.-Oct. 1997

[26] Eikerling H., Hardt W., Gerlach J., and Rosenstiel W., "A Methodology for Rapid Analysis and Optimization of Embedded Systems", In Engineering of Computer Based Systems, Germany, mars 1996.

[27] Faergemand O. and Olsen A., "Introduction to SDL-92", Computer Networks and ISDN Systems, vol. 26, pp. 1143-1167, 1994.

[28] Gailhard S., Julien N., and Martin E., "Intégration de méthodes d optimisation faible consommation dans l'outil de synthèse architecturale", Gaut_W, AAA 98, Saclay, 29-30 Janvier 1998.

[29] Gajski D. D., Vahid F., Narayan S., and Gong J., "Specification and Design of Embedded Systems", Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1994.

[30] Gajski D. D., Zhu J., Dömer R., Gerstlauer A., and Zhao S., "SpecC: Specification Language and Methodology", Kluwer Academic Publishers., 2000.

[31] Gajski D., Vahid F., Narayan S., and Gong J., "System-Level Exploration with SpecSyn", In Proceedings of Design Automation Conference, pp. 812, 1998.

[32] Garcia D. A., "Consommation de puissance et optimisation dans les applications de traitement du signal à base de FPGA" , thèse ENST 2000, N ° 2000E026.

[33] Garg P., Shukla S. and R. K. Gupta, "Efficient Usage of Concurrency Models in an Object-Oriented Co-design Framework", Proc. of the Design, Automation and Test in Europe, Munich, Germany, 2001.

[34] Gasbarro J., "The Rambus memory system", International Workshop on, Memory Technology, Design and Testing, 1995, pp:94-96.

[35] Gauthier L., Yoo S., and Jerraya A. A., "Automatic Generation and Targeting of Application Specific Operating Systems and Embedded Systems Software", Proc. Design Automation and Test in Europe, Mar. 2001

[36] Gerin P., Yoo S., Nicolescu G., Jerraya A.A., "Scalable and Flexible Cosimulation of SoC Designs with Heterogeneous Multiprocessor Target Architectures", In the ASP-DAC, 2001.

[37] Gigascale Silicon Research Center, "Mescal: Modern Embedded Systems: Compilers, Architectures, and Languages", http://www.gigascale.org/mescal/index.html.

[38] Grbic A. and al., "Design and Implementation of the NUMAchine Multiprocessor", Proc. Design Automation Conf, June 1998.

[39] Guerrier P. and Greiner A., "A Generic Architecture for On-Chip Packet-Switched Interconnections", Proc. Design Automation and Test in Europe, 2000.

[40] Hammond L. and Olukotun K., "Considerations in the Design of Hydra: A Multiprocessor-on-a-Chip Microarchitecture", Stanford Technical Report CSL-TR-98-749, Stanford University, 1998.

[41] Harwood A., "Motorola's Peripheral Interface Standards", Embedded Processor Forum, May 1999 http://www.mot.com/SPS/MCORE/downloads/nal-epf.pdf

[42] Henkel J. and Ernst R., "A Path-Based Estimation Technique for Estimating Hardware Runtime in HW/SW-Cosynthesis", In Proceedings 8th IEEE International Symposium on System Level Synthesis, p. 116-121, Cannes, France, 1995.

[43] Hennessy L. J. and Patterson D. A., "Computer Architecture A Quantitative Approach", second edition, Morgan Kaufmann Inc, San Fransisco California, 1996, ISBN 55860-329-8.

[44] Hessel F., Le Marrec P., Valderrama C., Romdhani M., and Jerraya A., "MCI – Multilanguage Distributed Co-simulation Tool", DIPES 98, Paderborn, Germany, octobre 1998.

[45] Heuring V. P. and Jordan H. F., "Computer Systems Design and Architecture", Addison Wesley, California, 1997, ISBN 0-8053-4330-X.

[46] IBM, "The CoreConnect Bus Architecture", available at http://www.chips.ibm.com/product/coreconnect/docs/crcon_wp.pdf, 1999.

[47] IBM, Inc. "28.4G Packet Routing Switch", Networking Technology Datasheets, http://www.chips.ibm.com/techlib/products/commun/datasheets.html

[48] Intel, http://www.intel.com/pressroom/archive/releases/20011126tech.htm, Nov 2001.

[49] Itoh K., Sasaki K., and Nakagome Y., "Trends in LowPower RAM Circuit Technologies", Proceedings of the IEEE, vol.83, no.4, pp.524 543, April 1995.

[50] ITRS, International Technology Roadmap for Semiconductors, "Design", 2001 Edition.

[51] ITU-T, "Functional Specification and Description Language", Recommendation Z.100 - Z.104, March 1993.

[52] IXP2800, Intel, http://www.intel.com/design/network/products/npfamily/ixp2800.htm

[53] James R. and et al., "Object-Oriented Modeling and Design", Prentice Hall, 1991.

[54] Jego C., Casseau E., and Martin E., "Interconnect cost control during high-level synthesis", DCIS 2000, 15th Design of Circuits and Integrated Systems Conference, Montpellier, France, 21-24 November 2000.

[55] Jerraya A. A. and O'Brien K., "SOLAR an Intermediate Format For System Level Modeling and Synthesis", (Chapter) In "Hardware/Software Co-Design", Ed.Jerzy Rozenblit, Publ. IEEE, 1994.

[56] Jerraya A. A., "Specification and validation for heterogeneous multiprocessor soc", In Summer School on Application-Specific Multi-Processor SoC, July 2001.

[57] Jerraya A. A., and al., "Multilanguage Specification for System Design and Codesign", Chapter in "System-level Synthesis", NATO ASI 1998 edited by A. Jerraya and J. Mermet, Kluwer Academic Publishers, 1999..

[58] Kalavade A., "System Level Codesign of Mixed Hardware-Software Systems", PhD thesis, University of California, Berkeley, CA 94720, September 1995.

[59] Kamble M., and Ghose K., "Energy Efficiency of VLSI Caches: A Comparative Study", 10th International Conference on VLSI Design, January 1997.

[60] Kapoor B., "Low Power Memory Architecutres for Video Applications", Proceedings of the 8th Great lakes symposium on VLSI, pp. 2 7, 1998.

[61] Karim F., Nguyen A., Dey S., and Rao R., "On-Chip Communication Architecture for OC-768 Network Processors", Proc. Design Automation Conf, June 2001.

[62] Karkowski I. and Corporaal H., "Design Space Exploration Algorithm For Heterogeneous Multi-processor Embedded System Design", Proc. Design Automation Conf., June 1998.

[63] Klumar S., "The Codesign of Embedded Systems: A Unified Hardware Software Representation", Kluwer Academic Publishers, ISBN: 0792396367, nov 1995.

[64] Kuskin J., Ofelt D., Heinrich M., Heinlein J., Simoni R., Gharachorloo K., Chapin J., Nakahira D., Baxter J., Horowitz M., Gupta A., Rosenblum M., and Hennessy J., "The Stanford FLASH multiprocessor", Proceedings of Computer Architecture, 1994, pp: 302-313.

[65] Lahiri K., Raghunathan A., and Dey S., "Fast Performance Analysis of Bus-Based System- on-Chip Communication Architectures", IEEE/ACM International Conference on Computer-Aided De- sign, 1999 , pp: 566-572

[66] Lahiri K., Raghunathan A., Lakshminarayana G., and Dey S., "Communication Architecture Tuners: A Methodology for the Design of High-Performance Communication Architectures for System-on-Chips", Proc. Design Automation Conf., June 2000.

[67] Laurent J., Julien N., and Martin E. "High Level Power Estimation based on Functional Analysis for Embeded DSP", Conference IEEE/ACM IWLS lake Tahoe USA June 2001.

[68] Lee G., Quattlebaum B., Cho S., and Kinney L., "Design of a bus-based shared memory multiprocessor", DICE Microprocessors and Microsystems 22 (1999), pp: 403-411

[69] Leijten J. A. J. and al., "Stream Communication between Real-Time Tasks in a High-Performance Multiprocessor", Proc. Design Automation and Test in Europe, 1998.

[70] Leijten J., et al., "PROPHID: A Heterogeneous Multi-Processor Architecture for Multimedia", Proc. Int'l Conference on Computer Design, 1997.

[71] Leiserson C., "Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing", IEEE Transactions on Computers, vol. C-34, no. 10, pp. 892-901, October 1985.

[72] Lennard C. K., Schaumont P., De Jong G., Haverinen A., and Hardee P., "Standards for System-Level Design: Practical Reality or Solution in Search of a Question?", Proc. Design Automation and Test in Europe, Mar. 2000.

[73] Lennard C., Schaumont P., De Jong G., Haverinen A., and Hardee P., "Standards for System-Level Design: Practical Reality or Solution in Search of a Question?", Proc. of Design, Automation and Test in Europe Conference, Paris, France, March, 2000.

[74] Li Y. and Henkel J., "A Framework for Estimating and Minimizing Energy Dissipation of Embedded HW/SW Systems", Proceedings of DAC 1998, pp.188 193, 1998

[75] Lin B. and Vercauteren S., "Synthesis Of Concurrent System Interface Modules With Automatic Protocol Conversion Generation", In Proceedings of the IEEE International Conference on Computer-Aided Design, ICCAD 94, pages 101 – 108. San José, CA, November 1994.

[76] Lindh L. and Klevin T., "Scalable Architecture for Real-time Applications and Use of Bus Monitoring", Real-Time Computing Systems and Applications, 1999., pp: 208-211

[77] Lysecky R., Vahid F., and Givargis T., "Experiments with the Peripheral Virtual Component Interface", Proc. Int'l Symposium on System Synthesis, Sept. 2000

[78] Lysecky R., Vahid F., and Givargis T., "Techniques for reducing read latency of core bus wrapper", Proceedings of Design, Automation and Test in Europe 2000, pp:84-91

[79] Madsen J., Grode J., Knudsen P., Petersen M., and Haxthausen A., "LYCOS: the Lyngby Co-Synthesis System", Kluwer Journal for Design Automation for Embedded Systems, pp. 195-235,

vol. 2, n ° 2, mars 1997.

[80] Martin E., Gailhard S., Julien N., and Sentieys O., "Un environnement logiciel pour la synthèse de haut niveau d'applications DSP faible consommation", SEE, 20-21 Novembre 1997.

[81] MCSE Homepage, MCSE research group, IRESTE, Nantes University, Available on-line at http://www.ireste.fr/mcse, 2000.

[82] MEFTALI S., GHARSALLI F., ROUSSEAU F., and JERRAYA A. A., "An Optimal Memory Allocation for Application-Specific Multiprocessor System-on-Chip", ISSS 2001 Montreal, Canada, October 2001.

[83] Nava M.D. and Okvist G.S., "The Zipper prototype: A Complete and Flexible VDSL Multi-carrier Solution", ST Journal special issue xDSL, September 2001.

[84] Nicolescu G., Svarstad K., Meunier O., Cesário W., Gauthier L., Lyonnard D., Yoo S., Coste P., and Jerraya A.A., "Desiderata pour la spécification et la conception des systèmes électroniques" , TSI, 2001.

[85] Nicolescu G., Yoo S., and Jerraya A.A., "Mixed-Level Cosimulation for Fine Gradual Refinement of Communication in Soc Design", In Proc. Design Automation and Test in Europe, March 2001.

[86] O'Nils M. and Jantsch A., "Operating System Sensitive Device Driver Synthesis from Implementation Independent Protocol Specification", Proc. Design Automation and Test in Europe, Mar. 1999.

[87] Passerone R., Rowson J. A., and Sangiovanni-Vincentelli A., "Automatic Synthesis of Interfaces between Incompatible Protocols", Proc. of the Design Automation Conference, San Francisco, USA, 1998.

[88] Peterson W., "Application Note: WBAN003", Design Philosophy of the WISHBONE SoC Architecture September 7, 1999. http://www.silicore.net

[89] PlayStation 2, Haruyuki Tago, Toshiba, Japan, "CPU for PlayStaion2", invited talk, SASIMI 2000.

[90] Qu G., Kawabe N., Usami K., and Potkonjak M., "Function-Level Power Estimation Methodology for Microprocessors", in Proc. Design Automation Conf, June 2000.

[91] Ravindran G. and Stumm M., "Performance Comparison of Hierarchical Ring- and Mesh-connected", Multiprocessor Networks High-Performance Computer Architecture, 1997, Third Interna- tional Symposium on , 1997 , pp: 58-69

[92] Ray S. and Jiang H., "A reconfigurable bus structure for multiprocessors with bandwidth reuse", Journal of Systems Architecture 45, 1999.

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

[94] Rhodes D. L. and Wolf W., "Co-Synthesis of Heterogeneous Multiprocessor Systems Using Arbitrated Communication", Proc. Int'l Conf. on Computer Aided Design, Nov. 1999.

[95] Rincon A. M., Lee W., and Slattery M., IBM Microelectronics Corp., "The Changing Landscape of System-on-Chip Design", Custom Integrated Circuits, 1999. Proceedings of the IEEE 1999, pp: 83-90

[96] Ron W., "Is SoC really different?", EETIMES November 8, 1999. http://www.eetimes.com/story/OEG19991108S0009

[97] Roosseel G., Sonics Inc., "Decouple core for proper integration", EETIMES Jan 3, 2000. http://www.eetimes.com.story/OEG20000103S0048

[98] Savaton G., Casseau E., and Martin E., "Behavioral VHDL Styles and High-Level Synthesis for IPs", FDL 2000, Forum on Design Languages, HDL Workshop, 4-8 septembre 2000, pp. 107-115.

[99] Scott S. and Thorson G., "The Cray T3E Network: Adaptive Routing in a High Performance 3D Torus", HOT Interconnects IV, Stanford University, August 1996.

[100] Semeria L. and Ghosh A., "Methodology for Hardware/Software Co-verification in C/C++", In the Asia South Pacific Design Automation Conference, January 2000.

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

[102] Sonics, Inc. http://www.sonicsinc.com

[103] Suzuki K. and Sangiovanni-Vincentelli A., "Efficient Software Performance Estimation Methods for Hardware/Software Codesign", In Proceeding of Design Automation Conference, June 1996.

[104] Synopsys, Inc., "Synopsys VHDL System Simulator", http://www.synopsys.com/products/simulation/vss_cs.html

[105] Synopsys, Inc., "SystemC, Version 1.1", available at http://www.systemc.org/

[106] SystemC Design Language, available on-line at http://www.systemc.org, 2000.

[107] Takahashi M., Takano H., Kaneko E., and Suzuki S., "A shared-bus control mechanism and a cache coherence protocol for a high-performance on-chip multiprocessor", Proceedings of High-Performance Computer Architecture Symposium 1996, pp: 314-322.

[108] Tensilica homepage: http://www.tensilica.com

[109] Transmeta homepage: http://www.transmeta.com

[110] Trimedia, http://www.semiconductors.philips.com/trimedia/

[111] Tummala R. and Vijay K., "System on Chip or System on Package ?", IEEE Design & Test of Computers Volume: 16 2 , April-June 1999 , Page(s): 48 -56

[112] VCC, Cadence Inc., http://www.cadence.com/products/vcc.html

[113] Vercauteren S., Lin B., and De Man H., "Constructing Application-Specific Heterogeneous Embedded Architectures from Custom HW/SW Applications", Proc. Design Automation Conf., June 1996.

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

[115] Weiss A.R., "The standardization of embedded benchmarking: pitfalls and opportunities Computer Design", 1999. (ICCD '99). International Conference on, 1999, pp: 492-508

[116] Wingard D., Kurosawa A., "Integration Architecture for System-on-a-Chip", Design Custom Integrated Circuits Conference, 1998, Proceedings of the IEEE 1998 pp: 85-88

[117] Xilinx DSL Modems Glossary. http://www.xilinx.com/products/xaw/mdm/dsl/dsl_gls.htm

[118] Ye W., and Ernst R., "Worst Case Timing Estimation Based on Symbolic Execution", COBRA report '95, Institute of Computer Engineering, Technical University of Braunschweig, Germany, octobre 1995.

[119] Yen T.Y. and Wolf W., "Communication Synthesis for Distributed Embedded Systems", in Proceedings of 1995 IEEE International Conference on Computer-Aided Design.

[120] Yoo S., Lee J., Jung J., Rha K., Cho Y., and Choi K., "Fast Prototyping of an IS-95 CDMA Cellular Phone: a Case Study", In the 6th Conference of Asia Pacific Chip Design Language, October 1999.

[121] Yoo S., Nicolescu G., Lyonnard D., Baghdadi A., and Jerraya A., "A Generic Wrapper Architecture for Multi-Processor SoC Cosimulation and Design", Proceedings CODES 2001, Copenhagen, Denmark, April 2001.

Publications

1. Baghdadi A., Zergainoh N.-E., Cesário W., Roudier T., and Jerraya A.A., "*Design Space Exploration for Hardware/Software Codesign of Multiprocessor Systems*", Proceedings 11th IEEE International Workshop on Rapid System Prototyping, Paris, France, June 21-23, 2000.

2. Baghdadi A., Lyonnard D., Zergainoh N.-E., and Jerraya A.A., "*An Efficient Architecture Model for Systematic Design of Application-Specific Multiprocessor SoC*", Proceedings DATE 2001, March 2001, Munich, Germany. (**Nominated for best paper award**).

3. Baghdadi A., Zergainoh N.-E., Lyonnard D., and Jerraya A.A., "*Generic Architecture Platform for Multiprocessor System-on-Chip Design*", chapter in "*Architecture and Design of Distributed Embedded Systems*", Edited by B. Kleinjohann, Kluwer Academic Publishers, April 2001.

4. Zergainoh N.-E., Baghdadi A., Tambour L., Lyonnard D., Gauthier L., and Jerraya A.A., "Framework for System Design, Validation and Fast Prototyping of Multiprocessor SoCs", chapter in "Architecture and Design of Distributed Embedded Systems", Edited by B. Kleinjohann, Kluwer Academic Publishers, April 2001.

5. Yoo S., Nicolescu G., Lyonnard D., Baghdadi A., and Jerraya A.A., "*A Generic Wrapper Architecture for Multi-Processor SoC Cosimulation and Design*", Proceedings CODES 2001, Copenhagen, Denmark, April 2001.

6. Baghdadi A. "Systematic Design of Application-Specific Heterogeneous Multiprocessor SoC", SIGDA PhD Forum at DAC 2001, June 2001, Las Vegas, USA.

7. Baghdadi A., Lyonnard D., Gauthier L., Nicolescu G., Paviot Y., Cesário W., Yoo S., Zergainoh N.-E., and Jerraya A.A., "*Automatic Generation of Hardware/Software Communication Architecture for Multiprocessor SoC*", Demo in SIGDA University Booth at DAC 2001, June 2001, Las Vegas, USA.

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

9. Baghdadi A., Zergainoh N.-E., Cesário W., and Jerraya A.A., "*Combining a Performance Estimation Methodology with a Hardware/Software Codesign Flow Supporting Multiprocessor Systems*", submitted after revision to IEEE Transaction on Software Engineering, 2001.

10. Jerraya A.A., Baghdadi A., Cesário W., Gauthier L., Lyonnard D., Nicolescu G., Paviot Y., and Yoo S., "*Application-Specific Multiprocessor Systems-on-Chip*", SASIMI, 2001.

11. Baghdadi A., Zergainoh N.-E., Cesário W., Roudier T., and Jerraya A.A., "*Exploration de l'espace des solutions architecturales dans le codesign - Estimation de performances au niveau système*", Technique et Science Informatiques, January 2002.

12. Cesário W., Paviot Y., Baghdadi A., Gauthier L., Lyonnard D., Nicolescu G., Yoo S., Nava M.D., and Jerraya A.A., "HW/SW Interfaces Design of a VDSL Modem using Automatic Refinement of a Virtual Architecture Specification into a Multiprocessor SoC: a Case Study", Design Forum, DATE 2002, March 2002, Paris, France.

13. Baghdadi A., Zergainoh N.-E., Jerraya A.A., "*Exploration et conception systématique d'architectures multiprocesseurs monopuces dédiées à des applications spécifiques*", to appear, Colloque CAO de circuits intégrés et systèmes, May 2002, Paris, France.

14. Cesário W., Baghdadi A., Gauthier L., Lyonnard D., Nicolescu G., Paviot Y., Yoo S., Nava M.D. and Jerraya A.A., "*Component-Based Design Approach for Multicore SoCs*", to appear, Proceedings DAC 2002, June 2002, New Orleans, USA.

RESUME

Les applications embarquées actuelles imposent des contraintes de plus en plus sévères. La puissance sans cesse croissante de calcul et de communication implique l'utilisation d'architectures multiprocesseurs, la portabilité implique des architectures monopuces et la faible consommation et faible coût impliquent des architectures dédiées. Ajouté à cela, les méthodes de conception évoluent moins vite que les possibilités technologiques d'intégration. Ainsi, une approche systématique partant d'un niveau d'abstraction plus élevé que le RTL est nécessaire pour réduire le temps de mise sur le marché et maîtriser la complexité.

Le sujet de cette thèse porte sur la mise en œuvre d'une nouvelle approche de conception systématique d'architectures multiprocesseurs monopuces dédiées à des application spécifiques.

Ainsi, un modèle architectural multiprocesseur générique est proposé. Ce modèle est modulaire, flexible et extensible, permettant de couvrir un large domaine d'applications. Les composants de traitement sont dissociés du réseau de communication via des interfaces génériques de communication jouant le rôle de coprocesseurs.

Un flot de conception complet est constitué de deux étapes principales. La première étape est l'étape d'exploration d'architecture. Concernant cette étape, une méthode d'estimation de performance au niveau système est proposée. Cette méthode permet une exploration rapide de l'espace de solutions architecturales pour trouver l'architecture système optimale pour l'application à concevoir. Le but de cette étape est de fixer les paramètres architecturaux (optimaux) dédiés à l'application. Ces paramètres sont utilisés dans la seconde étape –qui est l'étape d'implémentation– pour produire l'architecture RTL. Cette étape comporte trois types d'actions : la conception des composants logiciels, la conception des composants matériels et la conception du réseau de communication permettant d'intégrer les composants de base. Cette étape est réalisée de façon systématique basée sur l'instanciation et la configuration de composants dans une bibliothèque.

L'approche proposée permet de réduire significativement le temps de mise sur le marché de systèmes multiprocesseurs monopuces complexes. Plusieurs applications industrielles ont été réalisées pour valider et évaluer les performances de cette approche.

MOTS-CLES

Architecture multiprocesseur, monopuce, conception systématique, conception conjointe matérielle/logicielle, architecture de communication sur puce, modèle architectural, exploration d'architecture, estimation de performance.

TITLE

Systematic Design and Exploration of Application-Specific Multiprocessor SoC.

ABSTRACT

Current embedded applications impose increasingly severe constraints. The constantly increasing computation and communication rates imply the use of multiprocessor architectures, portability implies single chip architectures, and low cost and power consumption imply application-specific architectures. Added to that, design methods progress less quickly than technological possibilities of integration. Thus, a systematic approach starting from a higher abstraction level than RTL is necessary to reduce the time-to-market and to control the complexity.

The subject of this thesis relates to the research on a new approach for the systematic design of application-specific multiprocessor SoC.

Thus, a generic multiprocessor architectural model is proposed. This model is modular, flexible, and scalable, making it possible to cover a large application field. The computation components are dissociated from the communication network by means of generic communication interfaces playing the role of coprocessors.

A complete design flow consists of two principal stages. The first stage is *architecture exploration*. Concerning this part, a performance estimation method at the system level is proposed. This method allows a fast design space exploration in order to find the best system architecture for the application to be designed. The goal of this stage is to fix the optimal architectural parameters specific to the application. These parameters are used in the second stage –which is *implementation*– to produce the RTL architecture. It comprises three actions: design of the software components, design of the hardware components, and design of the communication network allowing to integrate all of these components. This stage is carried out in a systematic way based on the instantiation and the configuration of basic components in libraries.

The proposed approach reduces significantly the design time of complex multiprocessor SoC. Several industrial applications were designed in order to validate and evaluate the performances of this approach.

INTITULE ET ADRESSE DU LABORATOIRE

Laboratoire TIMA, 46 avenue Félix Viallet, 38031 Grenoble Cedex, France.

ISBN 2-913329-87-X (version brochée) **ISBN 2-913329-88-8 (version électronique)**