# Exploration d'architectures et allocation/affectation mémoire dans les systèmes multiprocesseurs mono puce = Architectures exploration and memory allocation/assignment in multiprocessor SoC

Samy Meftali

**UNIVERSITE JOSEPH FOURIER – GRENOBLE 1**
**SCIENCES & GEOGRAPHIE**

N° attribué par la bibliothèque

# THESE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITE JOSEPH FOURIER

**Discipline : Informatique**
**Option : Systèmes et communication**

Présentée et soutenue publiquement

Par

Samy MEFTALI

Le 06 Septembre 2002

**Titre**

# Exploration d'architectures et allocation/ affectation mémoire dans les systèmes multiprocesseurs monopuce

**Directeur de thèse : Ahmed A. Jerraya**

JURY

| | |
|---|---|
| Anne Mignotte | Présidente |
| El Mostapha Aboulhamid | Rapporteur |
| Michel Auguin | Rapporteur |
| Ahmed Jerraya | Directeur de thèse |
| Frédéric Rousseau | Co-Directeur de thèse |

Thèse préparée au sein du laboratoire Techniques de l'Informatique et de la Micro-électronique pour l'Architecture des ordinateurs - TIMA

# AVANT-PROPOS

Le travail présenté dans cette thèse a été réalisé au sein du groupe System Level Synthesis (SLS) du laboratoire : Techniques de l'Informatique et de la Microélectronique pour l'Architecture des ordinateurs (TIMA) de Grenoble. Je remercie Monsieur Bernard Courtois, Directeur de recherche au CNRS et Directeur du laboratoire TIMA de m'avoir accueilli et donné les moyens pour effectuer mon travail de recherche.

Je tiens tout particulièrement à exprimer ma profonde gratitude à Monsieur Frédéric Rousseau, maître de conférences à l'université Joseph Fourier de Grenoble, pour son encadrement et son suivi. Ses conseils, son soutien, ses encouragements permanents tout au. long de ces années de recherche, ses qualités humaines, son dynamisme et sa disponibilité ont joué un rôle déterminant dans ce travail. Qu'il trouve ici, l'expression de ma profonde reconnaissance.

Je remercie très vivement Monsieur Ahmed Amine Jerraya, Directeur de recherche au CNRS, pour la confiance qu'il m'a témoigné, de m'avoir accueilli dans son groupe, et pour avoir supervisé mes travaux. Je tiens à lui exprimer ma gratitude pour ses conseils, ses orientations et ses remarques pertinentes.

Tous mes remerciements à Madame Anne Mignotte Professeur à l'université de Lyon d'avoir accepté de présider le jury de ma soutenance. Merci à Monsieur El Mostapha Aboulhamid, Professeur à l'université de Montréal et Monsieur Michel Auguin Directeur de recherche CNRS à Sophia, rapporteurs de cette thèse.

Je remercie ma tendre épouse Paméla pour son soutien et ses encouragements dans les instants les plus difficiles. Merci pour son aide, sa lecture de ce manuscrit et pour avoir supporté mes humeurs durant cette période dense en activités. Infiniment merci.

Merci à tous mes collègues du laboratoire et particulièrement ceux du groupe SLS qui de près ou de loin m'ont aidé, à la réalisation de ce travail. Qu'ils trouvent ici l'expression de ma reconnaissance pour leur témoignage de sympathie et d'amitié. Ce fut, en effet, un plaisir de partager quotidiennement de si bons moment avec eux tout le long de ces trois années. Grand

merci à Sonja pour sa gentillesse et sa bonne humeur. Merci à (dans l'ordre alphabétique) : A. Bagdadi, W. Césario, A. Dziri, L. Gauthier, F. Gharsalli, D. Lyonnard, G. Nicolescu, Y. Paviot, A. Rezzag, L. Tombour, N. Zergainoh et bien d'autres.

Merci à Patricia pour son aide, et à ma fille Cécilia qui par sa présence m'a donné la force d'aller jusqu'au bout de ce travail.

## Résumé

**L**es dernières années ont connu une grande évolution dans la technologie de fabrication des circuits intégrés. Ces derniers sont de plus en plus complexes. Ils intègrent des parties dites logicielles (processeurs + programmes) et des parties matérielles dédiées ou spécifiques de calcul ou de mémorisation.

De nombreuses applications dans les domaines du multimédia et des télécommunications sont apparues. Elles nécessitent l'intégration de mémoires de différents types et tailles dans ces modèles d'architectures multiprocesseurs. Dans ces applications embarquées, les performances du système sont étroitement liées à celles de la partie mémoire. Celle ci occupe plus de 90% de la surface du système, et la consommation en énergie ainsi que les performances temporelles du système sont essentiellement dues au stockage et à l'échange de données entre les différents composants.

Avec cette présence croissante de la mémoire dans les systèmes monopuce, on note de nos jours l'absence d'une méthodologie systématique et optimisée pour la conception de tels systèmes avec une architecture mémoire spécifique.

Nous proposons dans cette thèse un flot de conception d'une architecture mémoire spécifique pour les systèmes monopuce. L'architecture mémoire est obtenue avec une méthode exacte basée sur un modèle de programmation linéaire en nombres entiers. Notre modèle permet d'obtenir une architecture mémoire distribuée partagée optimale pour l'application, minimisant le coût global des accès aux données partagées et le coût de la mémoire. On réalise ensuite automatiquement les transformations de l'architecture et du code de l'application en fonction de l'architecture mémoire choisie. Cette nouvelle spécification système (architecture + code applicatif) reste simulable.

La faisabilité et les performances de ce flot ont été testées sur l'application du VDSL.


**Mots clés :** flot de conception, système monopuce, architecture mémoire, allocation mémoire, transformation de code.

## Abstract

The last years saw a great evolution in the manufacture technology of the integrated circuits. Indeed they were marked by the appearance of heterogeneous systems on-chip. The latter are increasingly complex and integrate dedicated or specific material parts, such as the memories of various types, but also of the programmable parts as processors for example.

Many applications in fields such as the multi-media ones (audio and video) and the image processing handle very bulky and strongly dependent data, they consequently, require the integration of a great number of memories of various types and sizes in multi-task multiprocessor systems-on-chip. In many of these embedded applications, the area cost is for a large part dominated by the memories and a very large part of the power consumption is due to the data storage and transfer between the architecture parts.

To face such a complexity and to make it possible for the designer to satisfy the time-to-market constraints, a coherent and complete methodology of design of multi-task multiprocessor architectures with integrated shared memories is required.

In this thesis, we develop an automatic application-specific shared memory architecture design flow, starting from a parallel system level description of a given application.. We propose an exact method, which consists of an integer linear programming model to resolve the memory blocks allocation problem in multiprocessor on-chip architectures. The proposed model gives an exact and optimal solution for the fixed criteria (total access time to the shared data and the cost of the memory architecture). Taking into account the linear program's results, we perform automatically the application-code and architecture transformations corresponding to the chosen memory architecture, and generate a macro-architecture level description of the application.

The feasibility and the performances of this methodology were tested on a VDSL application.

**Key words :** design flow, system-on-chip, memory architecture, memory allocation, code-transformations.

# SOMMAIRE

## Chapitre I. Introduction



## Chapitre II. Les mémoires dans les systèmes monopuce

# Chapitre I

# INTRODUCTION

Dans ce chapitre d'introduction, le cadre de cette thèse est défini, puis les motivations et les objectifs de la thèse, qui sont essentiellement la définition d'un flot global et systématique de conception mémoire contenant un modèle d'allocation optimisée ainsi que son automatisation, sont présentés. La contribution de cette étude est ensuite brièvement exposée. Finalement le plan de ce mémoire est donné.

## *I.1. Contexte de la thèse*

Les dernières années ont connu une grande évolution dans la technologie de fabrication des circuits intégrés. En effet elles ont été marquées par l'apparition de systèmes hétérogènes monopuce (SoC, pour « System-on-Chip ») [Jer01a], [Jer01b]. Ces derniers sont de plus en plus complexes et intègrent des parties matérielles dédiées ou spécifiques, telles que les mémoires de différents types, mais aussi des parties programmables du type processeur par exemple.

De nombreuses applications dans des domaines tels que le multimédia (audio et vidéo) manipulent des données très volumineuses et par conséquent, nécessitent l'intégration d'un grand nombre de mémoires de différents types et tailles dans des modèles d'architectures multiprocesseurs multitâches.

En effet, la conception des systèmes modernes est influencée par les tendances technologiques et celles du marché. Ainsi, on voit émerger des puces de plus en plus complexes, (intégration des DRAMs avec la logique sur la même puce, systèmes avec 200 millions de portes…) et ce, sous des contraintes de délais de mise sur le marché de plus en plus courts.

Le Tableau I. 1 montre quelques exemples de systèmes monopuce très répandus dans le commerce de nos jours. Ces systèmes (jeux, processeurs réseau, ..) intègrent plusieurs processeurs et sont d'une grande complexité (supérieure à 1 million de portes). Ils disposent aussi de plusieurs Mbits de mémoire embarquée.

| Composants / Application | Processeurs | Mémoire embarquée | Logique spécifique | Exemple typique |
|---|---|---|---|---|
| **Terminal XDSL** | 1 MCU<br>1 DSP | > Mbits | > M portes | STEP 1,<br>VDSL (ST) |
| **Multimédia** | 1 MCU<br>plusieurs DSPs | >> Mbits | < M portes | TRIMEDIA<br>(Philips) |
| **Processeurs réseau** | plusieurs MCUs<br>plusieurs DSPs | >> Mbits | > M portes | IXPIZDE<br>INTEL |
| **Processeurs de jeux** | plusieurs MCUs<br>plusieurs DSPs | >> Mbits | >> M portes | Play station<br>(Sony) |

**Tableau I. 1.** Exemples de systèmes monopuce modernes

Pour faire face à une telle complexité et permettre aux concepteurs de satisfaire les contraintes du temps de mise sur le marché, une méthodologie cohérente et complète de conception de

systèmes multiprocesseurs est exigée. Les processeurs doivent supporter le multitâche et l'architecture de tels systèmes doit intégrer des architectures mémoire sophistiquées.

En l'absence d'une telle méthodologie le concepteur d'aujourd'hui n'a d'autres recours que la synthèse manuelle de la partie mémoire de l'architecture qui s'avère très coûteuse en temps.
Bien qu'une approche complètement manuelle se fondant sur l'expertise du concepteur aide à diminuer la surface de la puce et améliore certaines performances, le temps passé par un concepteur pour optimiser ces SoCs est trop important, et l'approche manuelle est à proscrire [Dut98]. Ainsi, la plupart des systèmes spécifiques doivent se fonder sur l'utilisation d'outils de CAO qui génèrent des circuits électroniques à partir d'une description de haut niveau d'une application.

Ainsi cette thèse rentre dans le cadre de la conception automatique de systèmes multiprocesseurs monopuce.

## *I.2. Le besoin de mémoire dans les systèmes multiprocesseurs monopuce*

La conception des systèmes digitaux modernes est influencée par les tendances technologiques et celles du marché. Ainsi, de nos jours on est capable de fabriquer des systèmes de plus en plus complexes sur une même puce avec la mémoire et la logique.

Cette évolution technologique est accompagnée par l'apparition d'applications dans les domaines tels que le multimédia (audio et vidéo) et le traitement d'images qui manipulent des données très volumineuses et/ou fortement dépendantes. Elles exigent par conséquent l'intégration de mémoires de différents types et tailles et en particulier des mémoires globales partagées dans le cas de données fortement dépendantes.

Pour illustrer ce besoin de mémoire partagée, supposons un système de traitement vidéo composé d'une dizaine de processeurs. L'objectif est de reconstituer une image, chaque pixel étant calculé en connaissant l'état de ses voisins. Il existe alors deux grands types d'architecture mémoire pour concevoir un tel système.

- une mémoire globale partagée, accessible par tous les processeurs, dans laquelle l'image est intégralement structurée,
- une mémoire distribuée, c'est-à-dire que l'image est découpée et répartie dans les mémoires de chaque processeur. La mémoire associée à un processeur peut être accessible aux autres processeurs (mémoire distribuée partagée), où non (dans ce cas, les processeurs émettent une demande pour recevoir un segment de l'image).

Le choix n'est pas du tout évident et il est contraint par le type de mémoire utilisé, le type de processeur (DMA par exemple), le réseau de connexion des processeurs, la bande passante des bus, mais aussi par la nature de l'application (nombre d'accès aux données, périodicité des accès, …, etc).
Pour répondre à cette question, le chapitre IV présente un algorithme efficace pour décider laquelle de ces architectures est la mieux adaptée à l'application.

## *I.3. Motivations*

Dans les systèmes monopuce, la mémoire est un composant de plus en plus dominant. Des prévisions récentes (ITRS 2000) prédisent que la mémoire sera de plus en plus utilisée dans de tels systèmes et occupera près de 95% des puces à l'horizon 2014 (Figure I. 1).

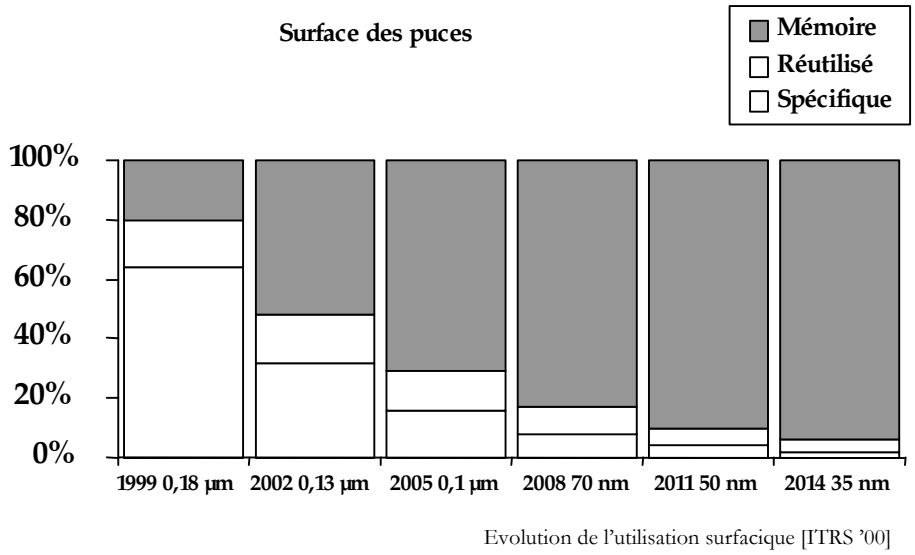


**Figure I. 1.** Evolution de l'utilisation surfacique des puces

Dans ces systèmes, l'architecture mémoire est plus ou moins choisie librement, mais elle dépend des contraintes de l'application. Les différents choix mènent à des solutions dont les coûts sont très différents, ce qui signifie qu'il est important de faire le bon choix.

La Figure I. 2 illustre cette multitude de choix dont dispose un concepteur concernant l'architecture mémoire. Pour cette raison, l'allocation des blocs mémoire devient une des étapes les plus importantes dans les flots de conception de systèmes monopuce. Le but de l'allocation mémoire est de profiter de cette liberté pour optimiser les coûts et les performances du système. En effet, des modèles mathématiques optimisés doivent être développés pour pouvoir trouver l'architecture mémoire la mieux adaptée à l'application.

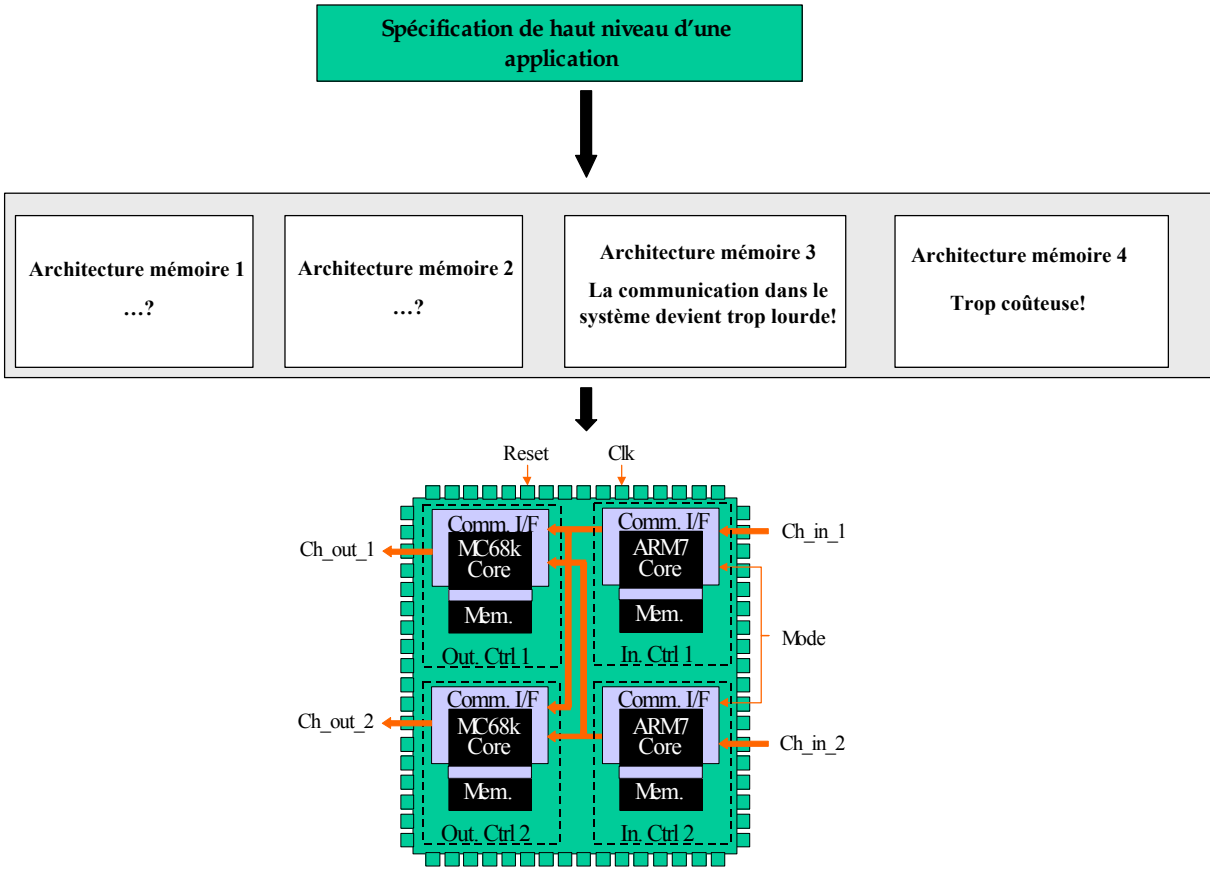


**Figure I. 2.** Allocation mémoire dans les systèmes multiprocesseurs monopuce.

## *I.4. Objectifs*

L'objectif de ce travail est de définir et d'intégrer un flot automatisable de conception d'une architecture mémoire partagée, dans un flot de conception d'architectures multiprocesseurs monopuce. Ce flot doit permettre une allocation optimisée de la mémoire, et de l'intégrer automatiquement dans le système. Pour pouvoir atteindre un tel objectif, plusieurs questions se posent :

- Comment reconnaître les éléments à mémoriser dans la spécification d'un système ?
- Quelles sont les architectures mémoire possibles pour une application donnée ?
- Quelle est l'architecture mémoire la mieux adaptée à une application donnée ?
- Quel est le volume de la ou des mémoires ?
- Comment modifier automatiquement l'architecture et le code de l'application pour tenir compte de l'architecture mémoire choisie ?
- Comment affecter les données de l'application dans ces différentes mémoires de façon optimale ?

## *I.5. Plan de la thèse*

Le chapitre II présente une taxonomie des éléments de mémorisation. Puis les systèmes multiprocesseurs classiques, les systèmes monopuce et les multiprocesseurs monopuce spécifiques à une application donnée sont introduits. Dans la seconde partie du chapitre, les différentes architectures mémoire sont exposées et critiquées en abordant les problèmes de cohérence mémoire. Le flot du groupe SLS pour la conception de systèmes multiprocesseurs monopuce est ensuite présenté.

Dans le chapitre III, le problème d'allocation mémoire est défini, puis l'essentiel des travaux dans ce domaine est résumé. La deuxième partie du chapitre est consacrée à la présentation des différentes étapes d'un flot cohérent et systématique, pour la conception de l'architecture mémoire pour les systèmes multiprocesseurs monopuce. Les différents modèles pour représenter les mémoires à chaque niveau d'abstraction utilisé dans le flot de conception sont détaillés dans la dernière partie de ce chapitre.

Les étapes de notre flot d'allocation mémoire sont détaillées dans le chapitre IV. Après la description de l'étape d'analyse du code de l'application pour extraire les paramètres nécessaires à l'allocation mémoire, un modèle optimal d'allocation basé sur la programmation linéaire en nombres entiers est présenté. Puis l'étape du raffinement du système et des transformations automatiques du code de l'application, pour l'adapter à l'architecture mémoire choisie, est détaillée.

Une application est étudiée dans le chapitre V pour illustrer les étapes de conception présentées dans cette thèse. Il s'agit d'une version du VDSL.

Le chapitre VI conclut ce manuscrit et donne quelques perspectives du travail présenté.

# Chapitre II

# LES MEMOIRES DANS LES SYSTEMES MONOPUCE

Le début de ce chapitre décrit une classification des types de mémoires présents dans les systèmes électroniques. Puis les différents systèmes : les multiprocesseurs, les systèmes monopuce, et plus particulièrement les systèmes multiprocesseurs monopuce sont introduits.

La seconde partie de ce chapitre est consacrée à la présentation des principales architectures mémoires utilisées dans les systèmes multiprocesseurs (mémoire partagée, mémoire distribuée, mémoire distribuée partagée), et on présente leurs implémentations. Puis le problème de la cohérence mémoire ainsi que les principales méthodes utilisées pour le résoudre sont discutés.

Dans la dernière partie de ce chapitre, le flot de conception du groupe SLS pour la conception de systèmes multiprocesseurs monopuce sans mémoire est présenté.

## *II.1. Introduction*

Dans les systèmes électroniques, on trouve diverses informations à mémoriser. On distingue les programmes qui doivent être exécutés sur les processeurs et les données de l'application.
Selon la taille de l'information, plusieurs types de mémoires peuvent être utilisés. Un registre peut servir à mémoriser des petites capacités (quelques dizaines de bits). Pour des capacités plus importantes, on peut associer plusieurs registres pouvant jouer le rôle d'un seul registre de grande taille (un banc de registres). Dans le cas d'informations de grande taille, les registres deviennent trop coûteux, on préfère alors des circuits associant plusieurs cellules élémentaires, ce sont les circuits mémoires.

Suivant le mode d'accès à l'information mémorisée dans la cellule de base, on distingue deux catégories de mémoires :

- Les mémoires à structure linéaire, ce sont des mémoires à accès séquentiel (du type registre à décalage). L'accès à l'information n'est pas direct, on décale le contenu des cases mémoire jusqu'à trouver le bon emplacement.
- Les mémoires à structure matricielle, l'accès est aléatoire où chaque point mémoire est directement adressable. On peut distinguer deux types de mémoires directement adressable : les mémoires mortes (ROM « Read-Only Memory ») qui sont des mémoires à lecture seulement, et les mémoires vives auxquelles on peut accéder en lecture ou/et en écriture.

### *II.1.1. Classification structurelle des mémoires*

#### II.1.1.1. Mémoires à structure linéaire

Ce sont des mémoires à adressage implicite, l'accès aux données dépend de l'ordre dans lequel elles ont été enregistrées. Ce type de mémoire est réalisé par un registre à décalage. L'utilisateur peut accéder soit au premier étage, soit au dernier. Dans cette catégorie il existe la pile et la file.

- Les piles : Ce sont des éléments de mémorisation largement utilisés dans les systèmes numériques et, plus particulièrement, dans les architectures à base de processeurs. Elles permettent, de façon relativement souple, d'échanger des données entre processeurs et périphériques intelligents ou entre les processeurs mêmes. L'ordre de lecture des données sur une pile est dit LIFO (Last In, First Out), ce qui signifie que le dernier mot écrit dans la mémoire sera le premier lu.
- Les files : Contrairement aux piles, dans les files les données mémorisées sont consommées dans l'ordre de leur mémorisation (première donnée mémorisée, première donnée consommée). Cet ordre de lecture des données est dit FIFO (First In First Out).

#### II.1.1.2. Mémoires à structure matricielle

Une mémoire matricielle est accessible d'une manière aléatoire, elle est constituée de plusieurs cellules élémentaires qui forment une matrice (lignes et colonnes). Une cellule correspond à l'intersection d'une ligne et une colonne.

On accède à un mot en positionnant l'adresse d'une ligne et l'adresse d'une colonne sur le bus d'adresse. Ce dernier est généralement multiplexé, en divisant ce bus en deux parties : adresse de ligne et adresse de colonne. Les adresses sont envoyées en deux fois sur les mêmes lignes, validées respectivement par les signaux adresse de ligne (« RAS » Row Adress Strobe) et adresse de colonne (« CAS » Column Adress Strobe).

La capacité de la mémoire est représentée par le nombre total de bits qui peuvent être mémorisés (nombre de points mémoire). L'organisation correspond à l'arrangement de ces points mémoires. Ainsi deux mémoires de 2048 mots de 1 bit et une autre de 256 mots de huit bits (octet) ont la même capacité. Dans le premier cas, on lit ou on écrit un seul bit à la fois tandis que dans le second on traite simultanément 8 bits. Généralement on exprime la capacité des mémoires en multiples de 1024, représentés par K pour kilo. Un exemple de mémoire matricielle à $2^{10}$ lignes et $2^{10}$ colonnes est donné dans la Figure II. 1.

.

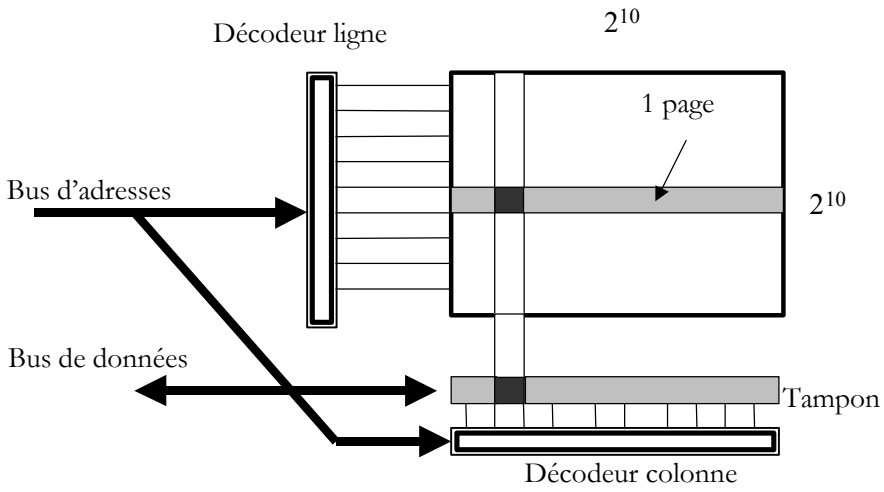


**Figure II. 1.** Mémoire matricielle à $2^{10}$ lignes et $2^{10}$ colonnes

## *II.1.2. Classification fonctionnelle des mémoires*

Selon le type d'opération que l'on peut effectuer sur les mémoires (lecture, écriture) on peut distinguer deux types de mémoires. Les mémoires mortes n'autorisent que des accès en lecture, et les mémoires vives permettent les accès en lecture et en écriture.

### II.1.2.1. Les mémoires mortes

La principale caractéristique des mémoires mortes est qu'elles ont la particularité de garder l'information après une coupure d'alimentation. On peut distinguer plusieurs types :

- Les ROM (Read Only Memory), n'acceptent que des accès en lecture. Une fois écrit, le contenu ne peut plus être changé.
- Les PROM (Programmable Read Only Memory) sont des mémoires ROM qui offrent à l'utilisateur la possibilité de les programmer. Quand le contenu de la mémoire ne convient plus, on programme un autre composant.

- Les EPROM (Erasable PROM) sont des PROM effaçables qui permettent de modifier le contenu des boîtiers sans les jeter. Le contenu peut être effacé par exposition à une source d'ultraviolets, à travers une fenêtre quartz placée sur le boîtier. Par contre l'écriture se fait électriquement avec une tension d'alimentation plus élevée.
- Les EEPROM (Electric EPROM), offrent la possibilité d'effacer le contenu d'un boîtier mémoire tout en restant sur la carte qui le contient, sans être obligé d'extraire les boîtiers pour les exposer à une source d'ultraviolets comme dans le cas des EPROM. Ainsi, les EEPROM peuvent être écrites et effacées électriquement.

#### II.1.2.2. Les mémoires vives

Contrairement aux mémoires mortes, les mémoires vives sont des mémoires volatiles dans lesquelles l'information mémorisée s'efface en absence d'alimentation. Une mémoire vive permet d'enregistrer et de restituer à la commande des informations. Dans les mémoires vives adressables, ces informations sont organisées en mots de taille fixe, désignés par une adresse. Les mémoires adressables sont appelées des RAM (Random Acces Memory : mémoire à accès aléatoire) car on peut avoir accès à tous les mots indépendamment de ceux auxquels on a accédé précédemment.
Suivant la structure des points mémoires, on distingue deux types de RAM : les RAM statiques (SRAM) et les RAM dynamiques (DRAM).

- Les SRAM sont qualifiées de statiques car elles permettent de garder l'information enregistrée pendant une durée illimitée tant que le circuit est sous tension. Elles peuvent offrir des temps d'accès très courts (quelques ns). Les SRAM sont utilisées généralement pour réaliser des mémoires spécifiques qui servent pour la communication entre processeurs, tampon d'entrée/sortie, des mémoires centrales de petits systèmes ou encore des mémoires «caches» pour améliorer le temps d'accès d'un microprocesseur à sa mémoire principale. L'inconvénient des SRAM est la densité d'intégration qui est moindre qu'avec des mémoires dynamiques, ceci est dû à la complexité des points mémoire qui occupent beaucoup de place (au moins quatre transistors par point mémoire).
- Les DRAM : sont constituées de cellules élémentaires instables dans lesquelles l'information est stockée sous forme de charge électrique dans la capacité de structure d'un transistor. La charge électrique stockée ne conserve pas éternellement son état de charge, elle peut disparaître à cause des résistances de fuie de la capacité. Il faut donc assurer périodiquement le rafraîchissement de l'information enregistrée. Ce rafraîchissement consiste à venir lire la cellule à intervalles fixes, et récrire l'information avant que la charge stockée ne se dégrade totalement. Pour cette raison, ces mémoires sont qualifiées de dynamiques. Ce type de mémoire est plus dense que les mémoires statiques (un transistor par bit), mais il est plus délicat à employer à cause des circuits de rafraîchissement. Il en existe de nombreuses variétés telles que les SDRAM, DPRAM, etc.

### *II.1.3. Hiérarchie mémoire*

La conception mémoire a suscité une attention considérable des concepteurs d'ordinateurs à cause de l'importance des mémoires dans les calculs effectués par les processeurs. En effet, l'exécution de chaque instruction simple comporte un ou plusieurs accès à la mémoire.

Avec l'explosion de la taille mémoire et le dépassement des vitesses d'accès mémoire par les vitesses des processeurs, la mémoire est devenue rapidement un goulot d'étranglement. Le temps d'accès à une mémoire est directement lié à sa taille : plus la taille est grande, plus le temps d'accès est lent. Ce qui a conduit souvent les concepteurs à envisager des ordinateurs à plusieurs niveaux de mémoires (hiérarchie mémoire).

Le niveau supérieur correspond à des mémoires de très grande capacité avec un temps d'accès très important, inversement au niveau inférieur qui correspond généralement aux registres.

La hiérarchie mémoire se base sur le principe de la localité. En effet l'accès fréquent à des instructions et à des données placées dans des mémoires du haut niveau de la hiérarchie, implique une pénalité de temps d'accès qu'on peut éviter en plaçant ces données et ces instructions dans des mémoires de niveau plus bas qui sont plus rapides.

Dans la majorité des architectures multiprocesseurs, les niveaux de la hiérarchie mémoire sont : les registres, un ou quelques niveaux de mémoires caches, mémoire principale et les mémoires secondaires (disques). Un exemple typique de cette hiérarchie est donné dans la Figure II. 2.

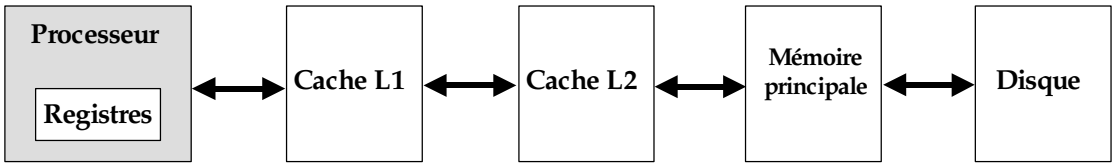


**Figure II. 2.** Exemple d'une hiérarchie mémoire

Les éléments qui constituent généralement une hiérarchie mémoire sont brièvement présentés dans ce qui suit.

***Registre*** : les registres offrent aux processeurs des accès très rapides. Généralement ils sont intégrés à l'intérieur des CPU grâce à leur petite taille (quelques kilo octets). Ils offrent un temps d'accès avoisinant 1 ns.

***Mémoire cache*** : les mémoires caches sont dans le niveau qui suit le niveau des registres. Elles sont utilisées pour mémoriser les données et les instructions qui sont récemment accédées. Les caches sont généralement intégrées avec les CPU dans la même puce, ce type de mémoire se caractérise par la rapidité d'accès qui est de l'ordre de 5 ns. Les niveaux de la hiérarchie mémoire correspondant aux caches ont généralement une capacité de quelques dizaines de kilo octets.

***Mémoire principale*** : l'accès à la mémoire principale est relativement lent par rapport aux niveaux précédents, il est de l'ordre de 50 ns. Par contre la taille de ce type de mémoire est importante, pour les mémoires des processeurs modernes, elle est de l'ordre de 512 Mega octets.

***Mémoire secondaire*** : c'est le plus haut niveau de la hiérarchie. L'espace mémoire virtuel de l'application peut être entièrement sauvegardé dans la mémoire secondaire (généralement dans le disque dure). L'accès à la mémoire secondaire est très coûteux (avoisinant les 10 ms), la taille est très importante (plusieurs Giga octets).

Les différents types de mémoires présentés dans cette section sont de plus en plus présents dans les systèmes électroniques modernes. Tout le long de cette thèse, on s'intéresse particulièrement aux systèmes multiprocesseurs sur une seule puce. Ces systèmes sont introduits dans la section qui suit.

## *II.2. Les systèmes monopuce*

L'importante évolution dans la technologie de fabrication de systèmes électroniques lors des dernières années a permis l'apparition d'une nouvelle génération de systèmes contenant au moins un processeur. Ils sont différents des ordinateurs, leurs ascendants, par le fait qu'ils sont intégrés sur une seule puce électronique. Ces systèmes sont appelés les systèmes monopuce *(SoC: System on a Chip).* Ils apportent des changements importants dans les méthodologies de conception des systèmes classiques [Jer01b].

Dans les systèmes électroniques classiques [Hen99], [Cul98] (sur une carte électronique, constitués de plusieurs puces), une des difficultés était liée à la dimension de ces cartes. Ceci limitait notamment la vitesse de communication, même entre des composants rapides. Ceci est particulièrement critique pour la communication entre le processeur et la mémoire, car quelle que soit la vitesse du processeur, il doit lire ses instructions en mémoire. Pour pallier à ces problèmes, il est nécessaire, dans les systèmes classiques (multi puce) d'utiliser des mémoires caches. L'inconvénient des caches est que ce sont des systèmes très complexes à étudier de part leur comportement et par conséquent de gros facteurs d'indéterminisme. Avec les systèmes monopuce, la communication reste toujours un goulet d'étranglement, mais avec un facteur bien moindre car le fait d'avoir sur la même puce l'ensemble du système raccourcit les chemins.

### *II.2.1. Les systèmes embarqués spécifiques*

Les systèmes électroniques sont de plus en plus présents dans la vie courante. Les ordinateurs et micro-ordinateurs sont des systèmes électroniques bien connus. Mais l'électronique se trouve maintenant embarquée dans de très nombreux objets usuels : les téléphones, les agendas électroniques, les voitures. Ce sont ces systèmes électroniques enfouis dans les objets usuels qui sont appelés systèmes embarqués.

### *II.2.2. Problématique de la conception des multiprocesseurs monopuce spécifiques*

Dans le cas des systèmes multiprocesseurs embarqués, on trouve certaines contraintes particulières qui les distinguent des autres systèmes multiprocesseurs classiques. En effet, ce sont des systèmes intégrés sur une seule puce. La plupart d'entre eux sont spécifiques à une application, c'est-à-dire qu'ils sont conçus « sur mesure » pour l'application.

En raison de ces particularités, on peut distinguer essentiellement les contraintes suivantes dans ces systèmes :

- un encombrement réduit,
- une consommation très faible (alimentation par piles ou batteries),
- le temps : ils sont souvent employés dans des applications avec des contraintes de temps et de performance,
- sécurité : ils sont souvent employés dans des domaines tels que l'aéronautique ou l'automobile

Ces systèmes multiprocesseurs monopuce, peuvent (et souvent exigent) l'intégration de certaines architectures mémoire plus ou moins sophistiquées. La section suivante de ce chapitre présente une classification de ces architectures mémoire.

## *II.3. Les architectures mémoires pour les systèmes multiprocesseurs*

Dans cette section, on présente une classification classique des multiprocesseurs, à partir de laquelle découle la classification des architectures mémoire.

### *II.3.1. Classification des architectures parallèles*

L'évolution des ordinateurs depuis leur apparition n'a jamais cessé de s'accroître. En effet, l'idée d'utiliser plusieurs processeurs à la fois pour améliorer la performance et pour améliorer la fiabilité date des tout premiers ordinateurs électroniques. Il y a environ 30 ans, Flynn [Fly66] a proposé un modèle simple pour classer tous les ordinateurs qui est encore utilisé aujourd'hui. Il examine le parallélisme dans les flots d'instructions et de données induit par l'exécution des instructions de la partie critique de la machine. Il range tous les ordinateurs dans une des quatre catégories suivantes [Hen99] :

- Un seul flot d'instructions, un seul flot de données (SISD) - C'est le monoprocesseur.
- Un seul flot d'instructions, plusieurs flots de données (SIMD) - La même instruction est exécutée par plusieurs processeurs utilisant différents flots de données.
- Plusieurs flots d'instructions, un seul flot de données (MISD) - Aucune machine commerciale de ce type n'a été construite à ce jour, mais pourrait l'être dans le futur.
- Plusieurs flots d'instructions, plusieurs flots de données (MIMD) - Chaque processeur lit ses propres instructions et opère sur ses propres données. Les processeurs sont souvent des micro processeurs standards.

Les premiers multiprocesseurs étaient principalement des SIMD, et ce modèle a fait l'objet d'une attention renouvelée dans les années 80. Ces dernières années, cependant, le MIMD a surgi comme l'architecture évidente à choisir pour des multiprocesseurs d'usages généraux. Deux facteurs sont principalement responsables de l'émergence des machines MIMD :

- Une machine MIMD fournit de la flexibilité. Avec les supports matériels et logiciels adéquats, elle peut fonctionner comme une machine mono utilisateur destinée à la haute performance pour une application, comme une machine avec multiprogrammation exécutant plusieurs tâches simultanément ou selon une certaine combinaison de ces fonctions.

- Une machine MIMD peut être construite en s'appuyant sur les avantages coût - performance des microprocesseurs standards. En fait, presque tous les multiprocesseurs construits aujourd'hui utilisent les mêmes microprocesseurs que ceux des stations de travail et des petits serveurs monoprocesseurs.

Les systèmes MIMD actuels sont classables en deux groupes, selon le nombre de processeurs qui lui-même impose une structure mémoire et une stratégie d'interconnexion. Nous désignons les systèmes selon leur organisation mémoire, car le nombre de processeurs (petit ou grand) a toutes les chances de changer avec le temps. Ainsi, on distingue les systèmes à mémoire partagée et ceux à mémoire distribuée.

### *II.3.2. Les systèmes à mémoire partagée (Shared-memory systems)*

Les machines du premier groupe que nous appelons les architectures à mémoire partagée centralisée (Figure II. 3), ont au plus quelques douzaines de processeurs au milieu des années 90. Les multiprocesseurs avec un faible nombre de processeurs peuvent se partager une mémoire centralisée unique et un bus pour interconnecter les processeurs et la mémoire. Avec de gros caches, le bus et la mémoire unique peuvent satisfaire les besoins mémoire d'un petit nombre de processeurs. Puisqu'il y a une seule mémoire principale avec un temps d'accès uniforme pour chaque processeur, ces machines sont parfois appelées *UMA (Uniform Memory Access*) pour *Accès Mémoire Uniforme*. Ces systèmes offrent un modèle de programmation général et «commode » permettant un partage simple des données, à travers un mécanisme uniforme de lecture et d'écriture des structures partagées dans la mémoire globale.

La facilité et la portabilité de la programmation sur de tels systèmes réduisent considérablement le coût de développement des applications parallèles. Par contre, ces systèmes souffrent d'un handicap qui est la grande latence dans l'accès à la mémoire, ce qui rend la flexibilité (l'extensibilité de l'architecture pour d'autres applications) assez limitée. Ce type d'architectures à mémoire partagée centralisée reste de loin l'organisation la plus populaire actuellement dans les multi-ordinateurs distribués sur réseau.

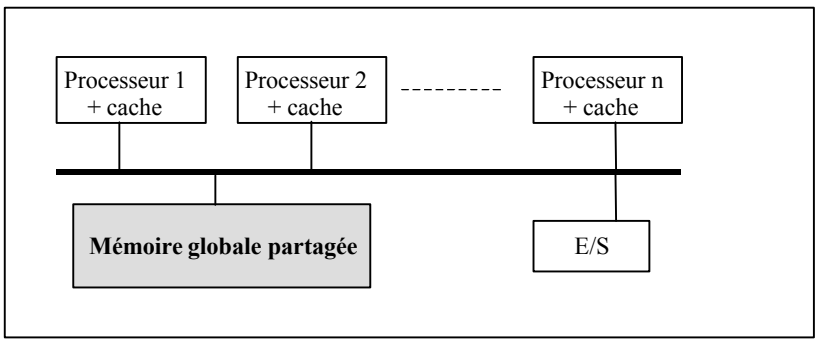


**Figure II. 3.** Architecture à mémoire partagée centralisée

### *II.3.3. Les systèmes à mémoire distribuée (Distributed memory system)*

Ces systèmes sont souvent appelés « les multi-ordinateurs ». Ils sont constitués de plusieurs nœuds indépendants. Chaque nœud consiste en un ou plusieurs processeurs et de la mémoire centrale. Les nœuds sont connectés entre eux en utilisant des technologies d'interconnexion extensibles *(scalable)* (Figure II. 4). Ces systèmes sont dits aussi machines à architecture de type NUMA *(Non Uniform Memory Acœss),* car en pratique, dans un réseau de stations de travail, l'accès à la mémoire locale de la station est nettement plus rapide que l'accès à la mémoire d'une station distante via le réseau.

La nature flexible de tels systèmes, les rend d'une très grande capacité de calcul. Mais, la communication entre des processus résidant dans des nœuds différents nécessite l'utilisation de modèles de communication par passage de messages impliquant un usage explicite de primitives du type Send/Recieve. En optant pour ce type de systèmes, le concepteur doit particulièrement faire attention à la distribution des données ainsi qu'à la gestion des communications (le transfert des processus pose un important problème à cause des différents espaces d'adressage, c'est-à-dire deux variables distinctes peuvent avoir la même adresse logique et deux adresses physiques différentes !). Ainsi, les problèmes logiciels, contrairement aux problèmes matériels sont relativement complexes dans les systèmes à mémoire distribuée.

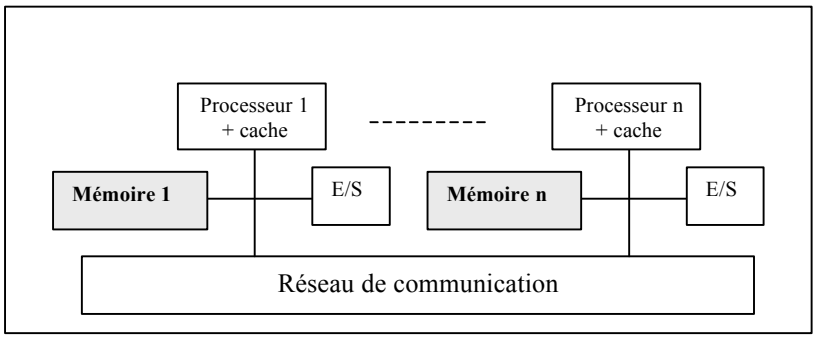


**Figure II. 4.** Architecture à mémoire distribuée

### *II.3.4. Comparaison entre les architectures à mémoire partagée et celles à mémoire distribuée*

Le Tableau II. 1 récapitule les différences essentielles entre les architectures à mémoire partagée et celles à mémoire distribuée. Nous constatons que la communication entre les processeurs est plus simple dans les systèmes à mémoire distribuée. En effet, ce type de systèmes est utilisé pour un nombre élevé de processeurs contrairement aux systèmes à mémoire partagée.

| **Architecture à mémoire partagée** | **Architecture à mémoire distribuée** |
|---|---|
| Temps d'accès à la mémoire uniforme pour tous les processeurs (UMA) | Temps d'accès dépendant de la position du mot de donnée en mémoire |
| Petit nombre de processeurs | Grand nombre de processeurs |
| Communication des données entre processeurs assez complexe | Communication facile entre processeurs |
| Les processeurs disposent généralement de plusieurs niveaux de cache (ou gros cache) | Processeurs avec des caches ordinaires |
| Architectures d'une flexibilité limitée | Architectures flexibles |
| Processeurs interconnectés par bus | Processeurs interconnectés par réseau d'interconnexion |
| Grande mémoire physiquement centralisée | Petites mémoires physiquement distribuées |

**Tableau II. 1:** Comparaison des architectures à mémoire partagée et à mémoire distribuée.

### *II.3.5. Les systèmes à mémoire distribuée partagée (DSM)*

Un concept relativement nouveau, qui est la mémoire distribuée partagée (Distributed Shared Memory), combine les avantages des deux approches. Un système DSM implante (logiquement) un système à mémoire partagée sur une mémoire physiquement distribuée. Ces systèmes préservent la facilité de programmation et la portabilité des applications sur des systèmes à mémoire distribuée, sans imposer pour autant la gestion des communications par le concepteur. Les systèmes DSM, permettent une modification relativement simple et une exécution efficace des applications déjà existantes sur des systèmes à mémoire partagée, tout en héritant de la flexibilité des systèmes à mémoire distribuée.

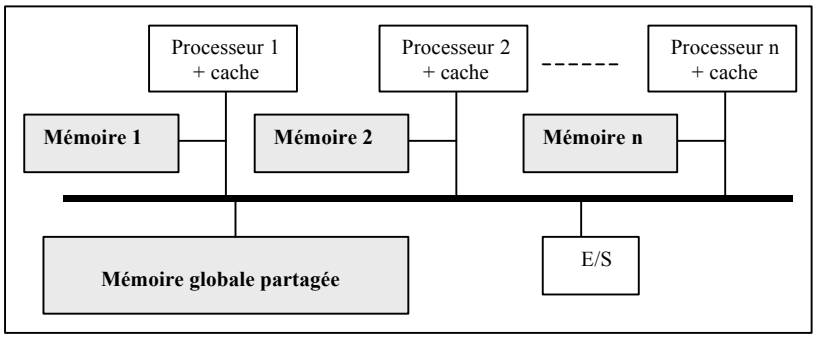


**Figure II. 5**. Architecture à mémoire distribuée partagée

Un système multiprocesseur avec mémoire distribuée partagée est généralement constitué d'un ensemble de nœuds (clusters), connectés par un réseau d'interconnexion (Figure II. 5). Un nœud peut être soit un simple processeur ou une hiérarchie qui cache une autre architecture multiprocesseur, souvent organisés autour d'un bus partagé. Les caches privés aux processeurs sont d'une grande importance afin de réduire la latence. Chaque nœud possède un module de mémoire local (physiquement), faisant partie du système DSM global, ainsi qu'une interface le connectant au système.

Les différentes méthodes utilisées pour implémenter de tels systèmes sont introduites dans la suite de cette section.

#### II.3.5.1. Implémentation des mécanismes DSM

On distingue trois implémentations possibles des mécanismes DSM [Pro96] : logicielle, matérielle et l'implémentation hybride qui est une combinaison des deux. Le choix de l'implémentation dépend généralement du compromis prix/performance. Bien qu'elles soient remarquablement supérieures en performance, les implémentations matérielles sont d'une grande complexité qu'on ne peut se permettre que pour des machines « large-scale » et haute-performance. D'autres systèmes, comme les réseaux de PC, ne peuvent tolérer un coût supplémentaire dû à une implémentation matérielle des DSM. Ainsi l'implémentation logicielle est de loin la plus utilisée pour ce type de machines. Finalement, pour des systèmes tels que les nœuds (clusters) de stations de travail, l'implémentation hybride semble être la plus appropriée.

#### II.3.5.2. Implémentation matérielle

Les implémentations matérielles des mécanismes DSM assurent la duplication automatique des données partagées dans les mémoires locales et les caches, d'une manière transparente aux couches logicielles. Ce type d'implémentations représente une extension des principes des schémas de cohérence de caches pour les architectures scalables avec mémoire partagée. Cette approche réduit considérablement les besoins de communication, cependant la conception s'avère généralement très compliquée. C'est pour cette raison qu'elle n'est utilisée que pour les machines où la performance est plus importante que le coût. On peut citer comme exemples d'implémentation matérielle des mécanismes DSM des architectures telles que : Memnet, Dash, SCI, KSR1, DDM, Merlin et RMS [Pro96].

#### II.3.5.3. Implémentation logicielle

Généralement les supports logiciels pour les DSM sont beaucoup plus flexibles que les supports matériels. Par contre, ils ne peuvent rivaliser avec ces derniers du point de vue performance. Ceci pousse généralement les concepteurs à introduire des accélérateurs matériels. Les architectures, avec une implémentation logicielle du système DSM, les plus connues dans la littérature sont : IVY, Mermaid, Murin, TreadMarks, Midway, Blizzard, Mirage, Clouds, Orca et Linda. [Pro96].

## II.3.6. Modèles de cohérence mémoire

La cohérence de la mémoire est sans doute l'un des problèmes majeurs que rencontrent les concepteurs de systèmes multiprocesseurs avec mémoire partagée. En effet, dans une architecture multiprocesseurs, ces derniers communiquent via les données partagées. De ce fait, une question importante se pose : dans quel ordre un processeur doit-il observer les écritures de données d'un autre processeur ? Puisque la seule manière d'observer les écritures des autres processeurs est la lecture, alors quelles propriétés doivent être respectées entre les lectures et les écritures dans les emplacements mémoire par les différents processeurs ?
On distingue dans la littérature [Cec01] plusieurs modèles de consistance mémoire.

### II.3.6.1. Cohérence séquentielle

Le système IVY [Li89] a introduit la duplication à la demande des pages partagées. Le système fournit une mémoire séquentiellement cohérente en utilisant un protocole à écrivain unique et lecteurs multiples. Les pages de mémoire sont protégées de telle façon qu'un accès en écriture provoque un défaut de page et l'invalidation de toutes les copies. Maintenir une cohérence forte comme la cohérence séquentielle est très coûteux. Des modèles de cohérence relâchée ont été mis au point pour minimiser les échanges de messages permettant de maintenir la mémoire distribuée cohérente. Ces modèles imposent au programmeur d'écrire des programmes où l'accès concurrent aux données est ordonné par l'utilisation des primitives de synchronisation fournies par le système.

### II.3.6.2. Cohérence par invalidation

La solution la plus simple pour maintenir la cohérence lors d'une écriture sur une page dupliquée, consiste à invalider toutes ses copies. Chaque nœud ayant eu sa copie détruite provoque, un défaut de page lors d'un nouvel accès à celle-ci. Le traitement du défaut de page déclenche l'accès à distance à la copie à jour de la page (copie maîtresse) ou la recopie de la page à partir de la copie maîtresse. Ainsi, on lit toujours le résultat de la dernière écriture effectuée sur les données.
Le nœud qui a provoqué le défaut en écriture a plusieurs choix :

- faire une mise à jour à distance sur la copie maîtresse,
- rapatrier la copie maîtresse sur le nœud local pour y effectuer la mise à jour,
- utiliser la copie locale (si le nœud en dispose d'une) pour y faire la mise à jour, supprimer la copie maîtresse et déclarer la copie locale comme nouvelle copie maîtresse. Cette technique est plus performante puisqu'il n'est pas nécessaire de recopier la page.

### II.3.6.3. Cohérence par diffusion

Cette technique consiste à diffuser la modification à tous les nœuds disposant d'une copie. Cela évite de coûteuses invalidations mais cela augmente également la quantité d'informations qui transite sur le réseau. Une cohérence par diffusion n'est pas rentable si le nombre de duplicatas est important et si le nombre d'écritures est conséquent. Toutefois, dans ce dernier cas, la duplication n'est certainement pas la meilleure stratégie pour améliorer les performances.

La mise en œuvre d'une cohérence par diffusion sur une grappe de machines nécessite un réseau d'interconnexion supportant la diffusion. En effet, les réseaux ne supportant que des

connexions point-à-point ne peuvent pas implanter efficacement une cohérence par diffusion lorsque le nombre de nœuds augmente. Le nombre de messages à envoyer pour simuler la diffusion devient alors vite prohibitif en temps et en bande passante. En résumé, la cohérence par diffusion est meilleure que la cohérence par invalidation dès lors que l'on dispose d'un réseau supportant la diffusion et que les applications n'effectuent que des écritures sporadiques sur des pages partagées dupliquées [Cou99].

#### II.3.6.4. Lazy Release Consistency

Release Consistency (RC) est un modèle de consistance mémoire relâchée qui autorise un processeur à différer le report des modifications de la mémoire partagée qu'il a effectuées jusqu'à ce qu'il ait atteint un point de synchronisation. Les modifications seront visibles par les autres processeurs uniquement après ce point de synchronisation. Ce modèle de cohérence est aussi appelé Eager Release Consistency [Car90].

Les accès aux données sont divisés en deux types : les accès ordinaires et les accès synchronisés. Dans ces derniers, les acquisitions sont distinguées des relâchements. Schématiquement, les acquisitions et relâchements des accès synchronisés peuvent être assimilés aux opérations de synchronisation sur les verrous. De même, l'arrivée à une barrière peut être considérée comme un relâchement et le départ d'une barrière comme une acquisition. RC requiert que les modifications faites sur la mémoire partagée, après une acquisition par un processeur p, deviennent visibles à un autre processeur *q,* seulement lors du prochain relâchement par p. RC se distingue du modèle courant de mémoire consistante séquentiellement (SC) en limitant le nombre de mises à jour.

La version « paresseuse » de RC, appelée Lazy Release Consistency (LRC), propose de différer la propagation des modifications jusqu'au moment de la prochaine acquisition. A cet instant, le processeur qui effectue l'acquisition met sa mémoire en cohérence pour respecter la définition de RC. [Kel94] décrit l'implémentation de LRC dans TreadMarks.

Ce type de cohérence est une version plus aboutie de la cohérence par diffusion. Elle conserve les duplicatas et permet une minimisation des mises à jour et donc des communications. Ces modèles de cohérence relâchée nécessitent une synchronisation explicite de l'accès aux variables partagées dans l'application parallèle. En revanche, ils apportent un gain important en terme d'utilisation de bande passante et d'échange de messages sur le réseau d'interconnexion.

#### II.3.6.5. Le faux partage

Jusqu'à présent, les techniques de gestion de la mémoire que nous avons présentées ont un grain qui se limite à la page mémoire. Or, il arrive fréquemment que l'utilisateur possède des données partagées en lecture, d'autres partagées en écriture, et que toutes les deux soient stockées dans une même page mémoire.

[Bol89] définit le faux partage à l'aide de la notion de partage en écriture. Un objet est dit partagé en écriture si au moins un processeur y accède en écriture et plus d'un y accède en lecture ou en écriture. Une page mémoire est dite partagée en écriture si elle remplit les mêmes conditions. Le

faux partage est alors défini par la présence d'un objet non partagé en écriture dans une page partagée en écriture.

#### II.3.6.6. Les travaux dans le domaine des mémoires distribuées partagées

De nombreux travaux ont été menés sur les mémoires partagées distribuées depuis une douzaine d'années [Cec01]. Si le modèle de programmation est attractif, les implantations successives de systèmes à DSM logicielle ont dû mettre au point des techniques permettant de maintenir la mémoire cohérente tout en offrant un niveau de performances acceptable.
Pour réduire les communications de coût très élevé, la première approche consiste à favoriser la localité des accès aux données. La technique de duplication, introduite par le système IVY [Li89], et celle de migration s'avèrent toujours efficaces même sur des machines CC-NUMA récentes [Bug97], [Ver96] où le ratio entre le coût d'un accès à la mémoire locale et celui à une mémoire distante est compris entre 2 et 3. Diverses expériences [Boly89, Cox90, Lar9l, Ran00, Ver96] ont prouvé que la migration et la duplication de pages mémoires permettent de diminuer les temps d'accès à la mémoire et de réduire la charge sur les brins d'interconnexion. Les résultats sont similaires sur les architectures avec ou sans cohérence de caches.

La DSM logicielle qui a connu le plus grand succès est certainement TreadMarks [Kel94], développée à l'université de Rice. TreadMarks implante une abstraction de mémoire partagée cohérente sur des grappes de stations de travail sans aucun support matériel pour les accès aux mémoires distantes. TreadMarks est une librairie de fonctions qui fournit les services de gestion d'une mémoire partagée en mode utilisateur. L'avantage d'une telle approche est sa portabilité sur l'ensemble des machines supportant le système d'exploitation Unix. Si la librairie consomme une partie de l'espace d'adressage du processus utilisateur, elle élimine les changements de contextes qui pourraient avoir lieu avec un démon en mode utilisateur.

Si les techniques classiques d'optimisation de la localité d'accès aux données donne de bons résultats sur un certain nombre d'applications, le maintien d'un modèle de cohérence de type *sequential consistency* est extrêmement coûteux en nombre de messages et donc en communications. La cohérence relâchée de type *eager release consistency,* implantée dans Munin [Car9l], a pu être améliorée dans TreadMarks en utilisant *lazy release concistency* qui a permis de réduire le nombre de messages nécessaires pour maintenir la mémoire distribuée cohérente. Dans le projet Cashmere-2L [Ste97], les communications utilisent un canal mémoire *(Memory Channel)* basé sur un mode en écriture seule. Dans ce cas, la cohérence est assurée de façon matérielle dans chaque nœud SMP et, entre les nœuds, c'est un protocole de cohérence au relâchement (RC) basé sur une copie maîtresse qui assure la cohérence. Les implantations récentes de mémoires partagées distribuées comme [Ran00] sur VIA exploitent les modèles de cohérence relâchée et les techniques de migration/duplication de pages de mémoires pour obtenir de bonnes performances. Ces modèles s'appliquent également avec succès dans des DSM construites dans des environnements Windows comme MILLIPEDE [Itz97].

Les DSM utilisant une cohérence forte utilisent des protocoles à écrivain unique. Ces protocoles sont simples à implanter, mais ils ne sont pas adaptés dans le cas de faux partages. Le faux partage est un problème complexe où le rapport efficacité/surcoût est difficile à doser. [Bol89]

propose des techniques plutôt statiques où le programmeur doit s'arranger pour ajouter des octets de bourrage dans ses structures pour les aligner sur des pages mémoires et éviter ainsi le faux partage. Cette méthode ne résout pas le faux partage induit par les compilateurs pour les structures internes. [Bol89] pense que des solutions au niveau du compilateur du langage devraient réduire de façon significative les problèmes de faux partage et limiter l'intervention de l'utilisateur dans ses applications. TreadMarks a introduit un protocole à écrivain multiple qui propose une réduction du coût lié au faux partage. Initialement, les pages sont protégées en écriture. A la première écriture, une faute de protection est engendrée, une copie de la page (*twin ou* jumelle) est créée et la protection contre l'écriture est enlevée. La jumelle et la copie courante peuvent alors être comparées à tout instant pour créer un < delta > collectant les modifications effectuées sur la page. Dans TreadMarks, les deltas ne sont créés que lorsqu'un processeur demande explicitement les modifications d'une page, ou lorsqu'une notification d'écriture d'un autre processeur arrive pour cette page. Cette technique est appelée technique du *twinldiff.* Le système NOA [Men98], comme SVMlib [Sch99], implante un mécanisme de type *twinldiff* sur des grappes SCI.

Le système PLATINUM [Cox90] a introduit le principe du gel/dégel périodique des pages qui migrent. Cette technique permet de réduire de façon drastique les problèmes de performance dus au phénomène de ping-pong décrit dans [Cox90]. Ce ping-pong peut être également le résultat d'un faux partage. [Lar9l] a implanté une politique de placement des pages fortement paramétrée et les résultats montrent que les seuls paramètres réellement influents sont le temps de gel d'une page et la répartition entre duplication et migration. En effet, [Laro9l] se base sur l'historique des accès en lecture et en écriture à une page pour décider s'il faut déplacer ou dupliquer. La conclusion de cette expérience est qu'une politique fortement paramétrée n'est pas nécessaire puisque seuls deux paramètres sont véritablement significatifs. Des politiques simples donnent souvent de très bons rapports surcoût de la gestion de la mémoire partagée/amélioration des performances [Bol89].

## *II.4. Le flot de conception de systèmes monopuce*

Lorsqu'un système est utilisé pour une tâche bien précise, il est souvent plus efficace et économe s'il est spécifique à cette fonctionnalité que s'il est général. Les systèmes embarqués sont très souvent utilisés dans ces conditions, et il est donc intéressant qu'ils soient conçus spécifiquement pour les fonctions qu'ils doivent remplir. Notamment, les contraintes citées dans la section II.2.2 ne peuvent souvent être respectées que si le système est conçu dès le départ pour pouvoir les respecter. Il est donc de par sa conception même spécifique.

Le problème qui se pose alors est que pour chaque nouvelle application, il faut concevoir un système spécifique différent de ceux déjà existants. Le flot de conception, présenté dans la suite de cette section, s'intéresse à la conception de ce type de systèmes embarqués, et tout particulièrement aux systèmes multiprocesseurs hétérogènes (contenant des éléments de natures différentes comme des processeurs, ASICs, mémoires, IPs) sur une puce.

Cette thèse a été effectuée dans le cadre d'un projet plus global dont le but est de définir un flot de conception et de concevoir les outils permettant d'aider à la conception des systèmes

multiprocesseurs monopuce. La motivation de ce projet est que les systèmes embarqués sont devenus trop complexes pour être développés avec des méthodes traditionnelles.

L'aide à la conception est obtenue en élevant le niveau d'abstraction des composants manipulés par les concepteurs, et en automatisant les passages d'un niveau d'abstraction à un niveau d'abstraction inférieur. Ce projet est centré sur les communications car elles apparaissent comme étant le domaine le moins traité par les méthodologies et les outils actuels. La Figure II. 10 représente ce flot.

### *II.4.1. Contexte d'utilisation du flot*

Le flot a pour but d'aider à la conception des systèmes embarqués spécifiques, et notamment les systèmes sur une puce. Les diverses équipes de conception peuvent utiliser conjointement cet outil. Mais il est tout particulièrement destiné à l'équipe d'architecture. L'aide consiste d'une part en l'apport d'une représentation multiniveaux et multilangage de l'architecture globale. Cette représentation sert de référence pour la conception de toutes les parties du système et aussi pour sa simulation. D'autre part, elle consiste en l'apport d'outils d'automatisation de diverses opérations telles que la génération des interfaces matérielles et logicielles.

### *II.4.2. Architectures cibles*

Ce flot supporte les architectures hétérogènes multiprocesseurs, multimaîtres et multitâches. Dans ce flot, chaque processeur dispose d'un système d'exploitation et d'un jeu de tâches qui lui sont propres. Chaque composant matériel est encapsulé dans une interface qui adapte ses communications locales aux communications globales de l'architecture.

### *II.4.3. Les restrictions du flot*

- La conception d'un système embarqué complet commence souvent par une spécification informelle et générale. Il existe des langages tels que UML qui permettent de représenter de manière formelle ce type de spécification, avec un niveau d'abstraction très élevé. Le flot proposé n'est pas encore capable d'intégrer ce type de spécifications : il débute plus bas dans l'abstraction, à un niveau où l'architecture globale est déjà définie.
- La spécification de départ du système est distribuée, et ne présente pas de problèmes de synchronisation.
- La conception de la partie mémoire est entièrement basée sur le savoir-faire du concepteur (manuellement).

### *II.4.4. Les modèles utilisés dans le flot*

#### II.4.4.1. La forme intermédiaire utilisée

Notre flot de conception utilise une forme intermédiaire pouvant décrire la spécification quel que soient les niveaux d'abstraction. Cette forme intermédiaire utilise le langage Colif [Ces00], [Ces01a], [Ces01b] développé au sein du groupe. Ce langage permet de décrire la structure d'un

système, à plusieurs niveaux d'abstraction, en mettant l'accent sur les communications. Les descriptions comportementales sont supposées issues de la description initiale de l'application.

### II.4.4.2. Les objets de la description

Dans tous les niveaux d'abstraction, la spécification du système est constituée de modules communicants. Ces modules peuvent représenter des parties matérielles, des parties logicielles ou des IPs. Ils communiquent par l'intermédiaire de ports au travers de canaux.
Chaque objet (module, port ou canal) est décomposé en une interface et un contenu. Le contenu de chaque objet peut être hiérarchique (c'est-à-dire contenir d'autres objets) ou faire référence à un comportement.

### II.4.4.3. Les niveaux d'abstraction utilisés dans le flot

Un système est représenté sur trois niveaux d'abstraction différents [Sva01] le niveau système (fonctionnel), le niveau architecture et le niveau micro-architecture. Ces trois niveaux sont présentés ci-dessous :

**- Niveau système**

A ce niveau (Figure II. 6), l'application est décrite avec des modules fonctionnels contenant des fonctions communiquantes par passages de messages via des canaux abstraits en utilisant des primitives de type Send/Receive. SDL [Bel91] est un langage typique pour décrire les systèmes à ce niveau d'abstraction.

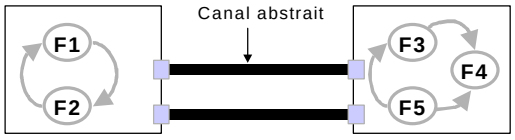


**Figure II. 6**.Niveau système

**- Niveau architecture**

A ce niveau (Figure II. 7), les modules correspondent aux blocs de l'architecture et ils communiquent via des canaux logiques par les primitives de type Read/Write. Au niveau architecture, en distingue tous les blocs de l'architecture. Des langages tels que SystemC [Sys] et VHDL [Air98] permettent de décrire des systèmes au niveau architecture.

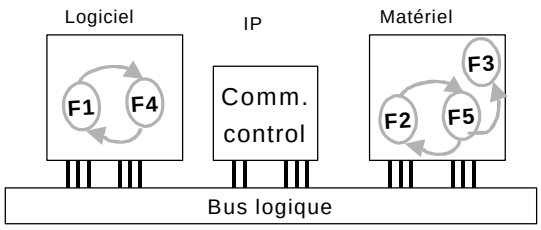


**Figure II. 7**. Niveau architecture

**- Niveau micro-architecture**

Au niveau micro-architecture (Figure II. 8), les modules sont des blocs physiques (DSP, CPU, IP,...) communiquants via des fils/canaux physiques par des primitives de type Set/Reset. VHDL et Verilog sont des langages qui permettent de décrire les systèmes à ce niveau d'abstraction.

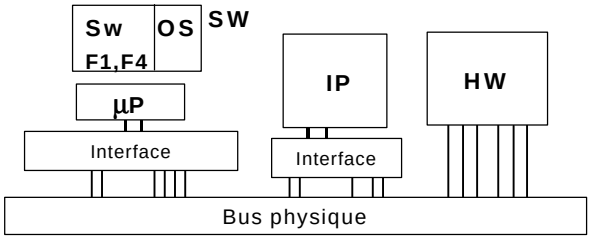


**Figure II. 8**. Niveau micro-architecture

Le Tableau II. 2 ci-dessous récapitule les différences entre les trois niveaux d'abstraction. Ainsi, on y trouve la définition d'un module, les types de données utilisés, les types de communications et quelques exemples de langages permettant de décrire les systèmes à chacun des niveaux.

| | **Niveau système** | **Niveau architecture** | **Niveau micro-architecture** |
|---|---|---|---|
| **Modules** | Fonctions communicantes | Blocs de l'architecture finale décrits au niveau logique | Blocs physiques |
| **Types de données** | Abstrait | Fixes (ex. entier) | Bit, vecteur de bit |
| **Communication** | Passage de messages (FIFO infinies) | Transmission de données/attente d'un événement | Valeurs de bits |
| **Exemples de langages** | SDL | VHDL, SystemC | VHDL, SystemC, VERILOG |

**Tableau II. 2.** Les niveaux d'abstraction

**Remarque :** il existe un niveau d'abstraction plus haut que le niveau système, appelé dans la littérature niveau service [Sva01]. Ce niveau n'est pour l'instant pas abordé. Nous commençons par le niveau système, pour arriver jusqu'au niveau micro-architecture. Ce dernier est souvent appelé niveau transfert de registres ou *RTL (Register Transfer Level).*

## *II.4.5. Architecture générale du flot*

La Figure II. 9 donne une vision simplifiée du flot de conception du groupe SLS. Ce flot débute au niveau fonctionnel, après que le partionnement logiciel/matériel ait été décidé. Il se termine au niveau micro-architecture, où une classique étape de compilation et de synthèse logique

permet d'obtenir la réalisation finale du système. C'est un flot descendant qui permet cependant de simuler à tous les niveaux d'abstraction et de revenir en arrière à chaque étape. En entrée, le flot prend une description de l'application au niveau fonctionnel. Cette description est traduite dans la forme intermédiaire multiniveaux Colif pour être raffinée au cours de deux étapes.

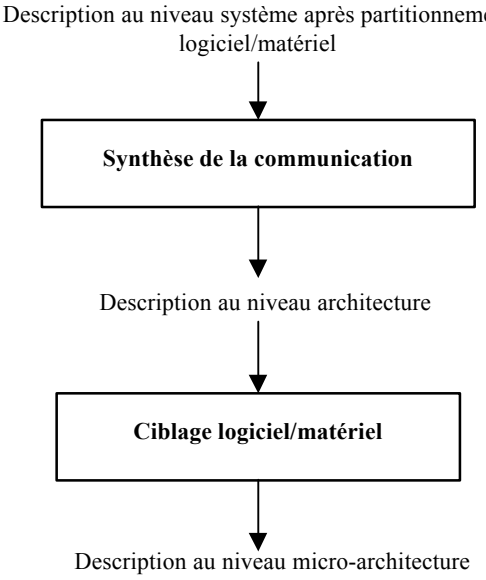


**Figure II. 9**. Représentation simplifiée du flot de conception de systèmes monopuce.

- La première étape : partant d'une description de l'application au niveau système, cette étape permet de passer à une description au niveau architecture : elle effectue la synthèse de la communication (allocation et affectation des unités de communication).
- La seconde étape fait passer la description au niveau micro-architecture : elle effectue la génération des interfaces logicielles et matérielles qui permettent l'assemblage des divers composants ainsi que leurs communications.

### *II.4.6 Architecture détaillée du flot*

Ce flot global est présenté sur la Figure II. 10. Il part d'une spécification architecturale et comportementale du système à concevoir. La description est alors raffinée de niveaux d'abstraction en niveaux d'abstractions. En sortie, nous obtenons le code logiciel (comportement des taches et système d'exploitation) et matériel (architecture du système) réalisant l'application.

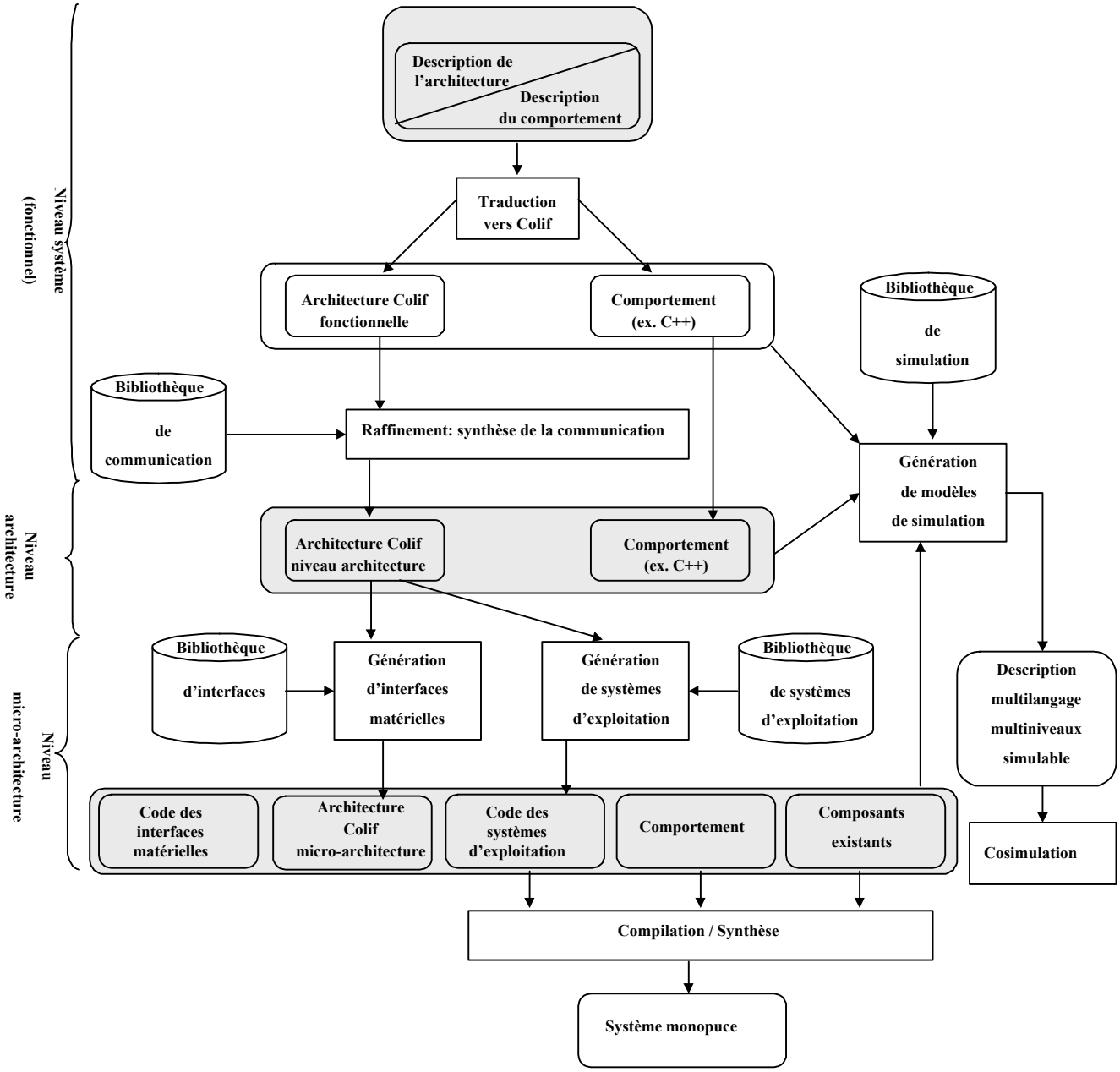


**Figure II. 10.** Flot détaillé de conception de systèmes monopuce.

### II.4.6.1. L'entrée du flot au niveau système

En entrée du flot, nous prenons une description dans le langage SystemC. Ce langage permet de décrire la structure et le comportement d'un système logiciel et matériel. Il est basé sur le langage C++ étendu avec des bibliothèques permettant la modélisation et la simulation de systèmes logiciels/matériels globalement synchrones ou asynchrones, avec un modèle à événements proches de celui du VHDL.

Cette description peut être effectuée au niveau fonctionnel ou aux niveaux inférieurs. Il est aussi possible de combiner les niveaux. Elle donne les informations sur la structure, et

éventuellement le comportement. Si certains composants sont fournis sans comportement, ils sont considérés comme des boîtes noires.

Cependant il était préférable que le flot soit indépendant du langage d'entrée. C'est pourquoi la première étape du flot consiste à convertir la description SystemC en une description Colif qui sépare le comportement de la communication dans le système (Figure II. 11).

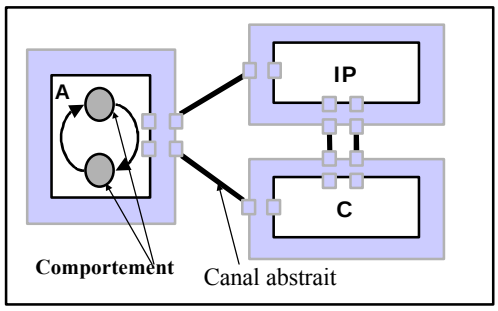


**Figure II. 11**. Exemple d'une spécification d'entrée du système

### II.4.6.3. La sortie du flot

Le flot fournit en sortie un système monopuce dédié à l'application de départ. Ce système peut être composé de plusieurs processeurs et de certains IPs. La description complète de ce système est donnée au niveau micro-architecture. L'architecture est synthétisée, le code de l'application et le système d'exploitation sont compilés.

### II.4.6.4. Les étapes du flot

***- Traduction vers Colif***

Cette étape extrait les informations structurelles de la description d'entrée (SystemC), pour les mettre sous le format Colif qui sera traité tout au long du flot. Les informations comportementales et les informations «boîte noire» sont conservées. Cette étape a été automatisée grâce aux outils développés dans l'équipe par Wander Cesârio [Ces00].

***- Synthèse de la communication***

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

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

### *- Génération d'interfaces matérielles*

La génération d'interfaces matérielles permet d'interconnecter les divers éléments de calcul : en effet ces éléments ne sont pas tous compatibles entre eux. Ces interfaces permettent aussi de réaliser des protocoles de communication non supportés nativement par les éléments.

Cette étape, décrite dans [Lyo01], fait partie des travaux de thèse de Damien Lyonnard. Elle est basée sur un assemblage de blocs à partir d'une bibliothèque. Le modèle d'interface généré au sein du flot est constituée de trois parties (Figure II. 12) : une partie dépendante de l'élément de calcul processeur, qui fait le pont entre son bus et le bus interne du processeur. La deuxième partie est le bus de l'interface, et la dernière est l'ensemble des contrôleurs de communication qui réalisent les protocoles pour l'échange avec le réseau de communication. Ce découpage, permet de générer aisément des interfaces pour tout type de processeur et tout type de protocole sans que la bibliothèque soit trop grande.

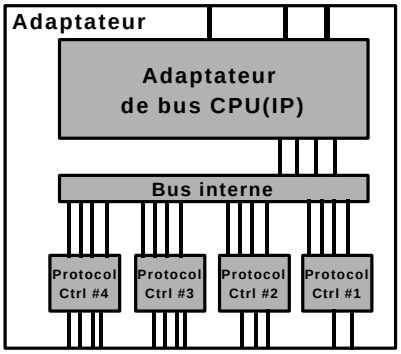


**Figure II. 12.** Exemple d'une interface matérielle (adaptateur).

### *- Génération de systèmes d'exploitation*

Les parties logicielles ne peuvent pas être exécutées directement sur les processeurs : l'exécution concurrente de plusieurs tâches sur un même processeur et les communications entre le logiciel et le matériel doivent être gérées par une couche logicielle appelée système d'exploitation. La génération de systèmes d'exploitation est réalisée par l'assemblage de composant logiciels organisés sous forme de bibliothèque. Cette dernière est organisée principalement en trois parties (Figure II. 13) : la première partie contient des éléments fournissant les services API, ce sont des éléments écrits en C et fournissant des fonctions système. La seconde partie fournit des éléments décrivant le système de gestion des interruptions du système d'exploitation. La troisième partie fournit les pilotes d'entrée/sortie (services drivers).

Cette génération de systèmes d'exploitation spécifiques a fait l'objet des travaux de thèse de Lovic Gauthier [Gau00], [Gau01a], [Gau01b], [Gau01c], [Gau01d].

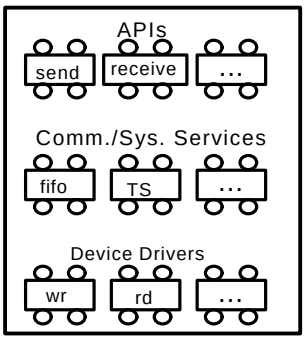


**Figure II. 13**. Bibliothèque de génération des interfaces logicielles

### *- La simulation*

On distingue principalement deux étapes pour la simulation du système : la génération des enveloppes de simulation et la cosimulation. Ces deux étapes sont décrites ci-dessous.

*1. La génération d'enveloppes de simulation*

Le flot de conception permet d'effectuer des simulations du système à tous les niveaux d'abstraction, mais aussi en mélangeant les niveaux d'abstraction. La technique de base consiste à encapsuler les divers composants à simuler dans des enveloppes qui adaptent le niveau de leurs communications à celui de la simulation globale. La Figure II. 14 illustre l'utilisation de ces enveloppes : à l'intérieur de chacune se trouve un composant qui est simulé à un niveau d'abstraction qui peut être inférieur (module A), égal ou supérieur (module B) au niveau d'abstraction de la simulation. Ces enveloppes permettent aussi d'adapter les divers simulateurs entre eux, comme par exemple un simulateur VHDL (module B) et un simulateur SystemC. Cette étape fait partie des travaux de thèse de Gabriela Nicolescu [Nic01a], [Nic01b].

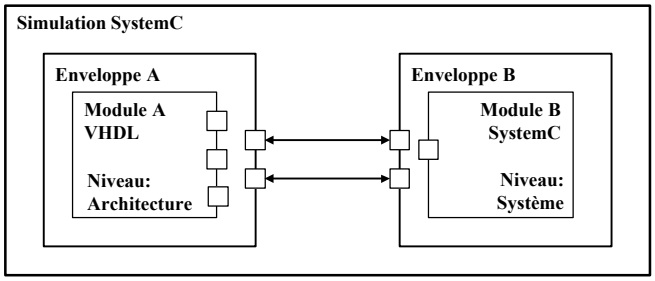


**Figure II. 14**. Enveloppes de simulation

*2. La cosimulation*

Grâce aux enveloppes, il est possible d'effectuer des validations par cosimulation pour toutes les étapes du flot (ou même pour des systèmes composés de modules décrit à différents niveaux d'abstraction). Plus le niveau d'abstraction est élevé, plus la simulation est rapide mais plus le niveau

d'abstraction est bas et plus la simulation est précise. Cette cosimulation est multiniveaux, mais elle peut être aussi multilangage. L'existence de telles cosimulations vient du fait qu'un langage est adapté pour la simulation de certaines parties d'un système, mais inadapté pour d'autres. Par exemple les simulateurs VHDL sont bons pour les circuits mais pas pour la mécanique qui est mieux modélisée par Matlab.

Pour effectuer une cosimulation, chaque simulateur est exécuté dans un processus différent. Un processus de contrôle gère l'ensemble des communications et synchronisations entre les simulateurs par le biais des IPC Unix. Le modèle temporel utilisé est le modèle synchrone : chaque simulateur exécute un pas de calcul, puis toutes les données sont échangées grâce au processus de routage. Une fois que les communications sont achevées, un autre pas de calcul peut être effectué.

Le processus de contrôle est décrit en SystemC. Cela permet de l'utiliser pour décrire des parties matérielles ou logicielles au niveau architecture, sans avoir à utiliser un simulateur externe. Il est généré en même temps que les enveloppes à partir de la bibliothèque de simulation.

***- Utilisation des résultats de simulation***

Les simulations pourront servir de base pour mettre en place une méthodologie d'évaluation qui permettra dans un premier temps de guider les choix du concepteur dans l'étape de synthèse, puis d'automatiser cette synthèse avec recherche d'optimum. Pour ce faire, il faudra définir des modèles de performances, et des heuristiques permettant de les composer en même temps que l'on compose les divers modules des systèmes à générer.

## *II.4.7. Analyse du flot*

Le flot de conception de systèmes multiprocesseurs monopuce du groupe SLS, présenté dans cette section se base sur la séparation entre la communication et le comportement dès les plus hauts niveaux d'abstraction. Ceci permet de traiter efficacement le problème crucial de la communication. Il se distingue des autres flots de conception par l'assemblage systématique et automatique des enveloppes logicielles et matérielles. La génération automatique d'un système d'exploitation spécifique à l'application augmente le niveau d'optimisation du système final et réduit considérablement le temps et l'effort de sa conception.

Ce flot n'intègre malheureusement pas la conception de l'architecture mémoire et ne permet l'utilisation d'aucune architecture mémoire sophistiquée (mémoire distribuée, mémoire partagée. .etc). En effet, il se limite à fixer les tailles des mémoires locales aux processeurs au moment de la génération des enveloppes logicielles et matérielles, et ce en se basant totalement sur l'expérience du concepteur. En fait la mémoire est implicite car on suppose que chaque processeur possède des ressources propres (RAM, ROM) pour les besoins du programme qui s'exécute.

## II.5. Extension du flot de conception SLS pour supporter les architectures mémoire

La principale contribution de cette thèse est l'extension du flot de conception du groupe SLS pour supporter différentes architectures mémoire. Dans la suite de cette section sont donnés les principaux choix et hypothèses de base de ce travail.

### II.5.1. Problèmes de cohérence mémoire et défauts de caches dans les systèmes monopuce dédiés du groupe SLS

Le problème de la cohérence mémoire est un problème crucial auquel il faut sans doute accorder une grande importance lors de la conception de systèmes multiprocesseurs utilisant des mémoires distribuées partagées.

Pour simplifier notre démarche, nous avons pris certaines hypothèses. Elles sont présentées ci-dessous ainsi que leur justification.

- La spécification de haut niveau contient explicitement la synchronisation, et gère la cohérence. La description d'une application sous forme d'un ensemble de tâches (en SDL par exemple), requiert toutes les informations en vue de la simulation. C'est donc au concepteur de les spécifier.
- L'allocation dynamique de mémoire n'est pas supportée, ce qui permet de prévoir et d'éviter tout problème de cohérence par le concepteur dès les hauts niveaux d'abstraction.
- Les caches sont utilisés pour améliorer les performances d'une architecture généraliste à une application. Le fait de tailler l'architecture pour une application donnée devrait nous permettre de s'affranchir des caches. Mais ils peuvent aussi être avantageusement remplacés par des mémoires « scratchpad », ayant les mêmes caractéristiques temporelles mais une gestion plus simple et déterministe.

### II.5.2. Architecture mémoire

Dans ce travail, nous nous intéressons uniquement aux mémoires de données. Ainsi nous distinguons principalement trois types de mémoires :

- mémoire locale privée : c'est généralement une mémoire de petite taille privée à un processeur (non accessible directement par les autres processeurs),
- mémoire locale distribuée : c'est une mémoire située sur un nœud associé à un processeur, mais directement accessible par les autres processeurs du système.
- mémoire globale partagée : c'est une mémoire de grande taille accessible par plusieurs processeurs d'une façon uniforme.

Ainsi, les systèmes multiprocesseurs monopuce que nous ciblons dans ce travail peuvent contenir une configuration de ces trois types de mémoires.

### *II.5.3. Flot de conception étendu*

Le flot de conception étendu, supportant différentes architectures mémoire contient plusieurs étapes liées à l'architecture mémoire allant de l'allocation mémoire à la synthèse mémoire. La description des étapes de ce flot fait l'objet du chapitre III.

## *II.6. Conclusion*

Comme on a pu le constater tout au long de ce chapitre, la mémoire est la partie critique dans les systèmes monopuce spécifiques de nos jours. Le traitement de tous les problèmes relatifs aux mémoires dans de tels systèmes nécessitera sûrement encore beaucoup d'effort et des années de travail. Cependant, ce qui paraît être l'un des problèmes les plus pressant est l'intégration d'un flot de conception d'une architecture mémoire sophistiquée (section II.3) dans le flot de conception global des systèmes monopuce spécifiques à partir d'un haut niveau d'abstraction (niveau système). En effet, cela permettrait de simuler le système dans son intégralité dès les niveaux d'abstraction les plus hauts (niveau architecture). Ceci fait l'objet du prochain chapitre de cette thèse.

# Chapitre III

# LE FLOT DE CONCEPTION DE SYSTÉMES MULTIPROCESSEURS MONOPUCE AVEC MEMOIRES

L'objectif de ce chapitre est de présenter les différentes étapes d'un flot cohérent et systématique, pour la conception de l'architecture mémoire pour les systèmes multiprocesseurs monopuce.

Comme notre flot est centré autour d'un modèle d'allocation mémoire, ce chapitre présente dans un premier temps en détail le problème d'allocation mémoire puis un tour d'horizon des principaux travaux dans ce domaine. Ensuite, les différentes étapes permettant d'obtenir une micro-architecture du système à partir d'une spécification de niveau système sont présentées. Les différents modèles pour représenter les mémoires à chacun des trois principaux niveaux d'abstraction utilisés dans le flot de conception sont ensuite détaillés. Nous nous intéressons dans ces modèles particulièrement à la mémoire globale partagée. Puis l'interaction du flot de conception de l'architecture mémoire avec le flot global de conception de systèmes monopuce du groupe SLS est présentée.

## *III.1. Introduction*

Nous proposons dans ce chapitre une méthodologie complète et cohérente pour la conception de systèmes multiprocesseurs multitâches avec mémoires. Notre approche est facilement automatisable et permet d'intégrer la mémoire dans les modèles de description de l'application envisagée à différents niveaux d'abstraction. Le passage par plusieurs niveaux d'abstraction permet de raffiner le modèle architectural abstrait jusqu'à un modèle d'architecture physique.

Ceci nous permet d'une part, de faire plusieurs optimisations mémoire de haut niveau telles que la minimisation du nombre d'accès et d'autres optimisations à un niveau d'abstraction plus bas, comme l'utilisation de modes d'accès performants et certaines transformations du code de l'application.

D'autre part, cela permet de pouvoir explorer plusieurs configurations possibles et de valider l'architecture à des niveaux d'abstraction assez hauts c'est-à-dire avant d'atteindre le niveau cycle d'horloge (micro-architecture). Ainsi, le concepteur a la possibilité, à l'aide de la simulation à un haut niveau d'abstraction, de tester différentes solutions, en modifiant les types et tailles des mémoires, distribution des données … etc, afin d'aboutir au meilleur choix architectural possible.

A un haut niveau de la conception, la mémoire n'est qu'un module abstrait (un processus SystemC ou SDL,..), ainsi la communication entre les autres modules et la mémoire est très abstraite (des primitives de haut niveau du type Send/Receive ). Par contre, au niveau physique, la mémoire peut être un module IP ou un bloc matériel connecté à des interfaces à travers lesquels les processeurs demandent l'accès à la mémoire via le bus ou le réseau de communication (des fils physiques). La communication entre les différents composants de l'architecture (processeur, mémoire, ASIC…) suit tout un protocole complexe (avec envoi de requêtes, des accusés de réception, …) ce qui rend la validation du choix architectural très coûteuse à bas niveau.

Notre flot de conception de l'architecture mémoire est basé sur un modèle d'allocation mémoire, permettant de trouver une architecture mémoire optimisée à l'application. Avant de présenter notre flot de conception en détail, on définit plus explicitement ce problème d'allocation mémoire dans la section suivante de ce chapitre, avant de faire un tour d'horizon des principaux travaux dans le domaine.

# *III.2. L'allocation / affectation mémoire et raffinement de l'architecture mémoire d'un système*

## *III.2.1. Introduction*

On entend principalement par l'allocation mémoire trois opérations :

- choisir une architecture mémoire (mémoire partagée, mémoire distribuée, caches, ...),
- fixer le nombre et les emplacements des différents blocs mémoire dans le système,
- fixer la taille de chacun des blocs mémoires.

L'affectation mémoire est l'opération qui attribue à chaque donnée de l'application un emplacement (adresse) dans l'un des blocs mémoire alloué dans le système.

Dans les systèmes embarqués, l'architecture mémoire est plus ou moins choisie librement, elle ne dépend que des contraintes de l'application. Les différents choix mènent à des solutions dont les coûts sont très différents. En effet, ITRS prévoit que dans l'horizon 2014, jusqu'à 95% de la surface des puces sera de la mémoire. Ainsi, un mauvais choix de l'architecture mémoire peut compromettre significativement les performances du système telle que la consommation en énergie. Pour cette raison, l'allocation des blocs mémoire devient une des étapes les plus importantes dans la conception des systèmes monopuce.

## *III.2.2. Etat de l'art de l'allocation et l'affectation mémoire*

### III.2.2.1. Approche classique

Il n'existe pas à ce jour d'outils industriels permettant de concevoir automatiquement une architecture optimisée. Ce problème est abordé par quelques travaux de recherche mais reste encore assez marginal. En effet, les flots de conception existants dans la littérature se concentrent tous sur l'automatisation de la conception de la partie communication du système, ainsi que sur la connexion des différents processeurs entre eux et/ou avec des parties matérielles spécifiques. Mais en aucun cas ils n'envisagent l'intégration d'une architecture mémoire à partir d'un haut niveau d'abstraction. En effet la mémoire n'est intégrée dans le système qu'au niveau micro-architecture. La taille énorme que peut avoir un système monopuce à ce niveau d'abstraction rend pratiquement impossible alors d'envisager d'intégrer manuellement une architecture mémoire sophistiquée (mémoire partagée) au système.

L'approche est alors une approche classique où l'on associe une mémoire DRAM à chacun des processeurs (mémoire locale).

### III.2.2.2. Approche basée sur les configurations de caches

Les chercheurs de Princeton proposent un algorithme de synthèse logiciel/matériel, qui essaye d'optimiser la taille des hiérarchies de caches pour les systèmes temps réel périodiques [Li97], [Li98].

L'algorithme synthétise un ensemble de tâches temps réel avec des dépendances de données en une architecture multiprocesseurs hétérogène satisfaisant les contraintes temps réel en essayant d'optimiser la taille des caches. Ils procèdent essentiellement en deux phases. La première extrait certains paramètres des programmes source, qui sont ensuite utilisés par la seconde phase qui est la synthèse. L'approche proposée dans ce travail donne des résultats intéressants, mais malheureusement elle n'est valable que pour un type d'applications bien restreint tel que les applications périodiques. Elle est concentrée essentiellement sur l'étape de synthèse (à un niveau d'abstraction assez bas) et ne couvre pas l'intégralité du flot de conception. Elle ne permet pas d'envisager des architectures à mémoire partagée.

Beaucoup d'autres travaux dans la littérature sont basés sur l'exploration des architectures mémoire basées essentiellement sur les hiérarchies de caches [Pan99], [Wuy98].

Les travaux de l'université d'Irvine en Californie sont particulièrement avancés et représentatifs de ce type d'exploration des architecture mémoires spécifiques. Ils se basent sur l'exploration de différentes configurations de caches, mais aussi sur l'utilisation des mémoires particulières à accès très rapide telles que les « scratchpad ». Ceci reste par contre dans un contexte monoprocesseur.

Ce sont des approches d'exploration plus ou moins exhaustives de différentes configurations de caches. Ces approches sont très peu conseillées pour des applications manipulant des données volumineuses, car dans ce cas on voit apparaître les fameux problèmes de défauts de caches liés essentiellement aux petites tailles des caches.

#### III.2.2.3. Approche basée sur des mémoires de grande taille

Certains travaux dans la littérature traitent le problème de conception de l'architecture mémoire pour certains types d'applications d'une façon plus au moins complète, en essayant de mixer les niveaux de caches avec des mémoires internes et externes. Les plus connus de ces travaux sont sans doute ceux de l'IMEC, et qui ont permis d'élaborer la méthode DTSE (Data Transfer and Storage Exploration).

##### *a - Les étapes de la méthode DTSE*

Les travaux effectués à l'IMEC depuis 1989 [Cat98a], [Cat98b], [Dan97] ont abouti à la méthode DTSE. Cette approche est destinée à des applications temps réel orientées flot de données très spécifiques telles que certaines parties des applications multimédia (audio et vidéo). Son but est d'optimiser essentiellement deux critères (la surface et la consommation) d'une grande importance pour ce type d'application. Partant d'un graphe représentant le transfert des données et les conflits d'accès, la méthode DTSE a pour objectif de trouver une bonne architecture mémoire (taille, nombre de mémoires, connexions,..). La méthode DTSE (Figure III. 1) est composée des quatre étapes essentielles suivantes :


**Figure III. 1.** Les étapes de la méthode DTSE.

*Transformation du flux de données :* le but de cette étape est de réduire les redondances dans le flot de données en supprimant les copies de données inutiles et en effectuant certaines transformations de boucles.

*Transformation globale de boucles et de flux de contrôle :* cette étape consiste à réduire la durée de vie globale des signaux, et à augmenter la localité et la régularité des accès aux données.

*Réutilisation de données et exploitation de la hiérarchie mémoire :* l'objectif de cette étape est d'exploiter l'organisation en hiérarchie de mémoires pour profiter de la localité lors des accès aux données, et de stocker ainsi celles auxquelles le processeur accède « souvent » dans des mémoires de petites tailles ayant une faible consommation en énergie.

*Optimisation et allocation mémoire :* l'objectif de cette étape est de déterminer une architecture mémoire optimale pour l'application. Ainsi, c'est lors de cette allocation et affectation mémoire que sont déterminées les tailles des mémoires et que les données sont affectées aux différents espaces adressables de l'architecture. C'est une étape très importante dans la méthodologie DTSE car elle a un effet direct sur la surface et la consommation en énergie.

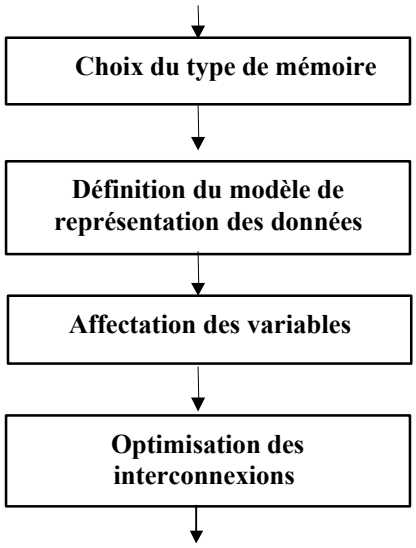


**Figure III. 2.** Allocation mémoire dans la méthode DTSE.

L'étape de l'allocation mémoire peut être décomposée à son tour en quatre sous étapes comme le montre la Figure III. 2 :

- choix du type de mémoire : cette étape consiste à :
  - 1. sélectionner le type de mémoire (DRAM, SRAM, à l'intérieur ou à l'extérieur de la puce …),
  - 2. déterminer le nombre de ports et le placement en mémoire des variables possibles. Cependant les tailles des mémoires ne sont pas encore fixées à ce niveau,
- définition du modèle de représentation de données de base : cette étape consiste à regrouper des variables ayant le même type et à adapter la taille de ces variables au type de mémoire disponible,
- affectation des variables : au cours de cette étape on affecte les variables en mémoire. Selon la taille limite des mémoires et le nombre de ports on effectue le regroupement des variables en fonction de la taille des mots. On décide où sont mémorisées les données en utilisant une méthode par séparation et évaluation exhaustive (Branch and Bound). Avant d'entamer cette étape d'affectation mémoire, on suppose dans cette méthode que le concepteur dispose d'un certain nombre de blocs mémoires avec des tailles fixées. Dans le but d'accélérer l'exploration de l'arbre de solutions possibles, certaines bornes basées sur les critères de taille et de consommation sont utilisées,
- optimisation des interconnexions : consiste à multiplexer les bus d'interconnexions dans le cas des mémoires multiport.

### *b- Avantages et inconvénients de la méthode DSTE*

La méthode DTSE semble être efficace. En effet, elle permet de réaliser une allocation mémoire et une affectation des données de l'application dans ces mémoires en réduisant le nombre d'accès mémoire ainsi que la surface mémoire utilisée. Mais malheureusement, avec cette méthode, le nombre de cycles ne fait qu'augmenter, car après les transformations appliquées sur le code, les opérations arithmétiques deviennent de plus en plus complexes [Car98a]. La limitation principale de cette méthode est sans doute le fait qu'elle cible une architecture monoprocesseur avec certaines

parties matérielles spécifiques partant d'une spécification séquentielle de l'application (ex. en C) et d'un compilateur paralléliseur. Cette méthode ne permet pas l'utilisation des mémoires distribuées partagées.

## *III.3. Flot global de conception de systèmes monopuce avec mémoires*

La description de haut niveau de l'application qui constitue l'entrée de ce flot (Figure III. 3) est une spécification exécutable de l'application sans mémoire explicite. Le résultat produit est une architecture décrite au niveau micro-architecture, ainsi que le code exécutable (application + OS) pour chacune des parties programmables de l'architecture.

Le passage d'un niveau d'abstraction à un autre s'accompagne d'un ensemble de transformations sur l'architecture, mais aussi dans les différents programmes de l'application. Des simulations valident les transformations à chaque niveau d'abstraction.

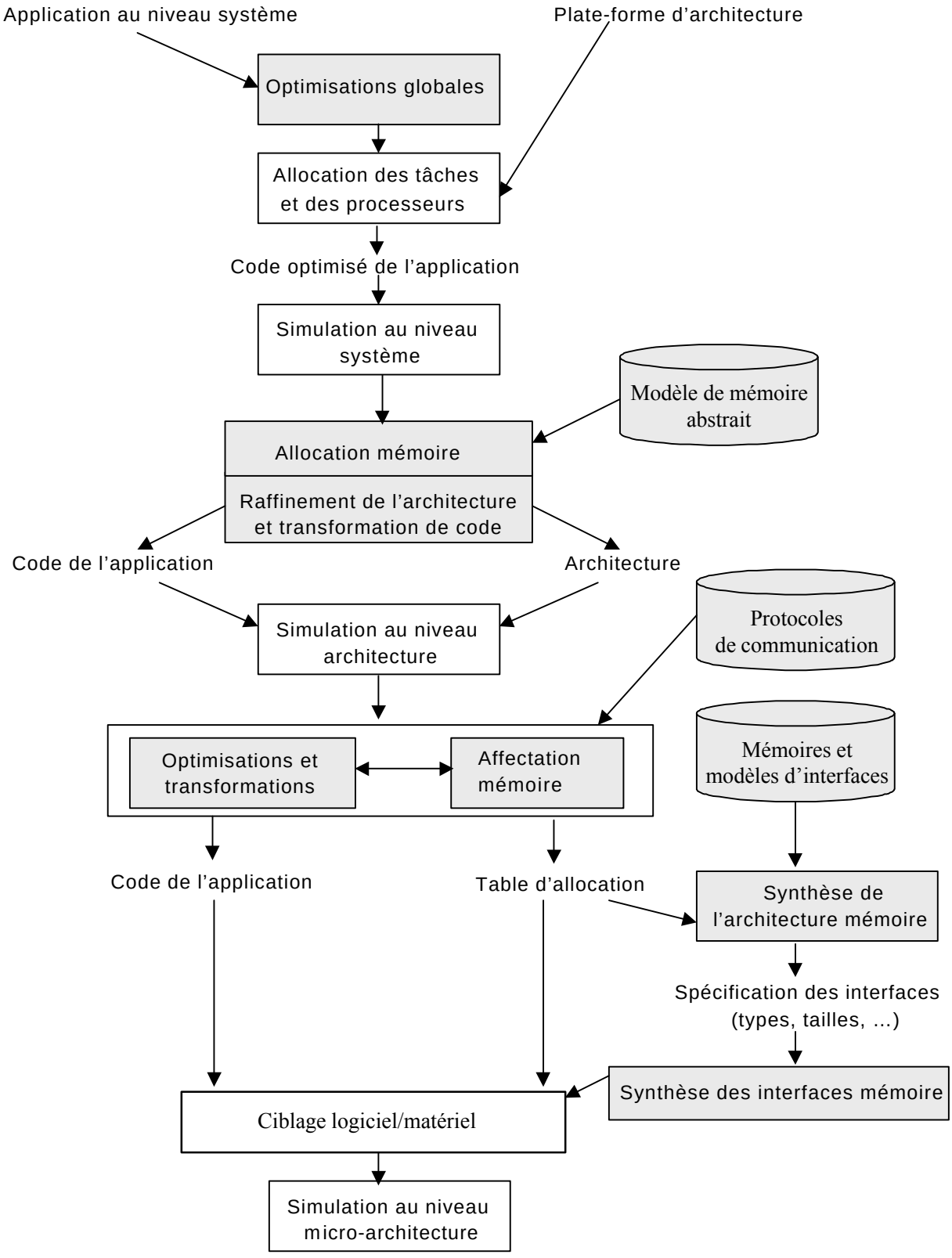


**Figure III. 3.** Flot de conception mémoire.

## *III.3.1. Entrée du flot de conception*

### III.3.1.1. Description de l'application au niveau système

Notre flot de conception accepte en entrée une spécification de l'application au niveau système (actuellement en SystemC). Elle décrit le comportement de l'application, c'est-à-dire le comportement des différentes fonctions composant le système, les échanges de données entre ces fonctions ainsi que l'utilisation des canaux de communication par le flux de données échangées entre les différents blocs.

### III.3.1.2. Plate-forme d'architecture

Dans notre flot, nous utilisons une plate-forme d'architecture générique et flexible [Bag02]. Cette architecture générique et flexible est composée d'un ensemble de modules connectés par un réseau de communication. Les modules sont les implémentations choisies par le concepteur pour réaliser les différentes fonctionnalités. Il s'agit de processeurs (ARM7 et 68k Motorola), d'IPs et d'ASICs. Le réseau de communication peut être des liaisons point-à-point ou un bus (type AMBA).
Le concepteur peut modifier certains paramètres tels que le nombre de processeurs, la taille de la mémoire et les ports E/S pour chaque processeur, l'interconnexion entre les processeurs, les protocoles de communication et les connexions externes (périphériques).
Chaque processeur dispose d'une mémoire locale et d'un système d'exploitation. Les processeurs et le réseau de communication sont connectés par des interfaces. Ces interfaces réalisent l'adaptation de protocoles.

Il est possible d'intégrer dans cette plate-forme des mémoires globales partagées et des mémoires distribuées.

## *III.3.2. Etapes du flot*

### III.3.2.1. Optimisations globales

Cette étape d'optimisation consiste à transformer le code initial, dans le but de diminuer le nombre d'accès en lecture et en écriture à une même case dans un tableau. Il s'agit aussi de supprimer des tableaux intermédiaires ou de réduire la distance d'accès.
Ces transformations sont faites à un haut niveau d'abstraction et sont indépendantes de l'architecture physique.

On modifie alors le code qui manipule les tableaux, ce qui revient à travailler sur les nids de boucles. On effectue alors des transformations dont l'effet est de changer l'ordre du parcours d'un ensemble de tableaux.

De nombreux travaux sont publiés dans ce domaine, on peut en citer : [Wol92], [Kul95], [Fra99a], [Fra99b] et [Fra01].

### III.3.2.2. Simulation au niveau système

Après avoir amélioré la qualité du code initial de l'application en effectuant les transformations de code adéquates, nous vérifions le comportement du système et le validons par simulation.
En plus de la validation du comportement, cette simulation au niveau système permet d'obtenir des informations sur le partage des données entre les différents blocs ainsi que l'utilisation des canaux de communication. Ces informations sont utilisées dans la suite du flot et plus particulièrement lors de l'allocation mémoire.

En SystemC, chaque bloc d'un système est décrit avec deux fichiers : un fichier décrivant l'interface du bloc (ses entrées/sorties) et un autre décrivant son comportement.

La Figure III. 4 montre le code représentant les deux fichiers interfaces de deux blocs communicants CP1 et CP2 du système représenté dans la Figure III. 10. Ainsi, on distingue pour chaque bloc l'ensemble de ses ports d'entrée/sortie à travers lesquels il échange des données avec les autres blocs, ainsi que les variables locales des deux blocs qui correspondent dans l'exemple aux tableaux d'entiers tab1 et tab2.

.

```
// Interface file of CP1
struct cp1 : sc_sync {
 // The inputs
 const sc_signal<bool>& finl31;
 const sc_signal<bool>& finl41;
 const sc_signal<bool>& ordre1;
 // The outputs
 sc_signal<bool>& sig12;
 sc_signal<int>& data13;
 sc_signal<bool>& sig14;
 sc_signal<int>& data12;
 sc_signal<bool>& signal1;
  // Variables
 int tab1[1024];
…
…
}
```

```
// Interface file of CP2
struct cp2 : sc_sync {
 // The inputs
 const sc_signal<bool>& finl32;
 const sc_signal<bool>& finl42;
 const sc_signal<bool>& ordre2;
 // The outputs
 sc_signal<bool>& sig21;
 sc_signal<int>& data23;
 sc_signal<bool>& sig24;
 sc_signal<int>& data21;
 sc_signal<bool>& signal2;
  // Variables
 int tab2[1024];
…
…
}
```

**Figure III. 4.** Description SystemC au niveau système des interfaces de deux blocs communicants.

Toute description en SystemC d'un système donné possède un fichier de coordination qui sert à instancier les signaux, les processus et qui permet d'avoir un fichier de trace de la simulation.

Dans la Figure III. 5 on notera par exemple l'instanciation du signal entier « data » qui relie les deux processus CP1 et CP2 en connectant le port de sortie de CP1 « data12 » avec le port d'entrée de CP2 « data21 », de même pour le signal booléen « signal » sur les ports « sig12 » et « sig21 ». Ces deux signaux sont explicitement inclus dans le fichier rapport de simulation trace.vcd que l'on peut visualiser pour s'assurer du bon comportement du système, mais aussi pour des informations sur tous les signaux (données) utilisés dans l'application. En effet, on peut par exemple voir le nombre

de lectures/écritures d'une variable donnée par un processeur, le taux d'utilisation des canaux de communication reliant les différents blocs.

De telles informations sur les données peuvent être d'une grande utilité, et nous les utiliserons dans notre modèle d'allocation mémoire (chapitre IV).

```
#include"systemc.h"
#include"p1.h"
#include"p2.h"
#include"p3.h"
#include"p4.h"
#include"controleur.h"

int sc_main( int argc, char* argv[])
  {
    //instantiate the signals
      …
      …
      sc_signal<bool> signal;
      sc_signal<int> data;
…
…
// Instantiate the clock
      sc_clock clk("clk", 20, 0.5, 0.0, true);

      //trace file creation
    sc_trace_file * watch=sc_create_vcd_trace_file("trace");
    sc_trace(watch,clk.signal(),"CLOCK");
    sc_trace(watch,signal,"signal_cp1_cp2");
    sc_trace(watch,data,"data_cp1_cp2");
…
…
// Instantiate proceses
      cp1 p1("process1", clk.pos(), signal31, signal41, ord, signal,
                         don1, signal14, donnee, signal1);
      cp2 p2("process2", clk.pos(), signal32, signal42, ord, signal,
                         don2, signal24, donnee, signal2);
…
…
  // Run the simulation infinitely
  sc_start(-1);
  return 0;
}
```

**Figure III. 5.** Fichier de description de l'architecture.

La Figure III. 6 montre un exemple de visualisation du rapport de simulation d'un système décrit en SystemC, en utilisant Synopsys Waveform Viewer. On peut voir à gauche de la figure la liste des signaux que l'on veut observer. A droite de la figure on retrouve toutes les variations d'état de ces signaux sur une période de simulation donnée. On peut avoir la valeur d'une variable à tout moment de la simulation, ainsi que les taux d'accès à toutes les données par chacun des processeurs.

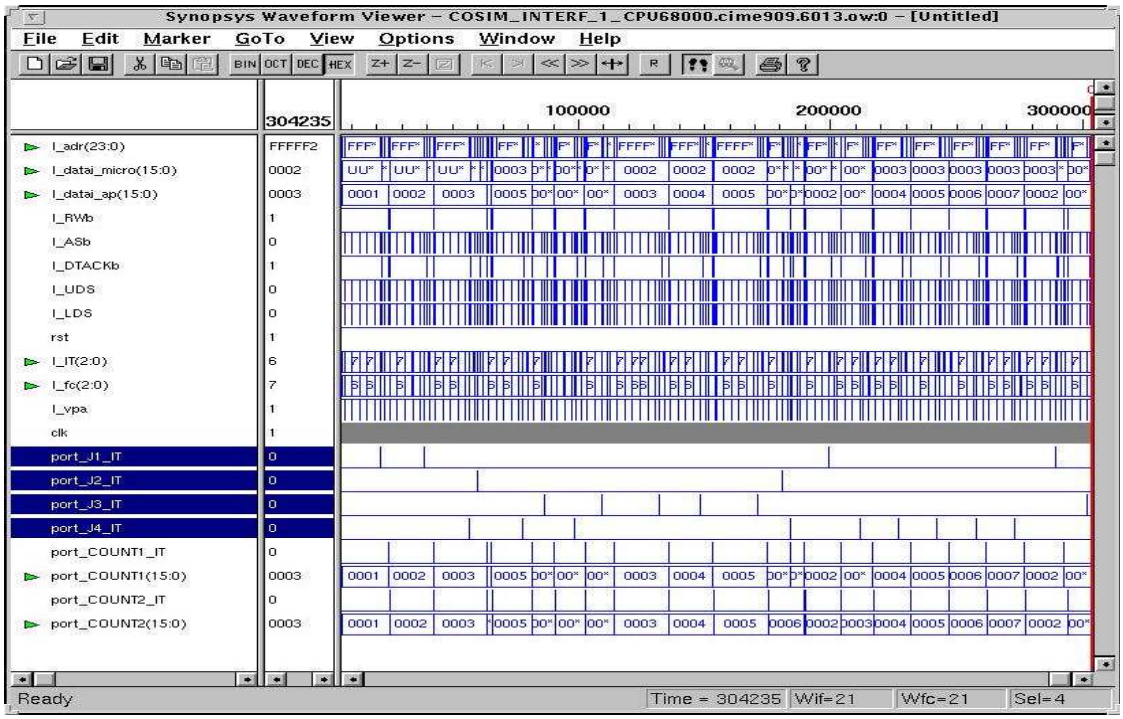


**Figure III. 6.** Visualisation des résultats d'une simulation SystemC.

### III.3.2.3. Allocation mémoire

Dans la description au niveau système d'une application, on trouve généralement des données de diffèrents types. On distingue des données locales aux fonctions (processus), des données globales à plusieurs processus et des signaux de communication entre les processus.

La question qui se pose au concepteur et de savoir quel type de mémoire est le mieux adapté pour stocker chaque donnée de l'application. Dans le cas des variables locales à un module, la donnée sera généralement stockée dans une mémoire locale privée. Par contre, dans le cas d'une variable partagée, échangée par deux processeurs (variable de communication) ou d'une variable globale, le choix d'un type de mémoire nécessite le développement d'algorithmes judicieux (chapitre IV).

Le Tableau III. 1 résume les différents types de données rencontrés dans une description de niveau système ainsi que le choix trivial ou les choix possibles des types de mémoires associées généralement à chacun de ces types.

| Type de données | Mémoire |
|---|---|
| Variable locale à une fonction/ processus | Locale |
| Variable globale à un module | Locale |
| Signal entre processus du même module | Locale |
| Signal entre processus de modules différents | Mémoire de communication (globale/locale distribuée) |
| Variable globale | Locale distribuée/globale |

**Tableau III. 1.** Types de données et types des mémoires associés.

L'étape de l'allocation mémoire est sans doute l'une des plus importante dans un flot de conception de systèmes sur une puce. En effet, cette étape permet d'allouer tous les blocs mémoire pour l'application (mémoires locales privées, locales partagées, et globale partagée). Cette allocation est déterminante pour les performances du système final (surface, consommation en énergie, performances temporelles,…).

Ainsi, l'allocation mémoire a pour objectifs :

- d'allouer des blocs mémoires d'une façon optimale pour l'application,
- de déterminer le bloc mémoire le mieux adapté au stockage de chaque donnée.

Un modèle basé sur la programmation linéaire en nombres entiers [Mef01b] a été développé pour réaliser cette allocation mémoire de façon optimale en minimisant le temps global d'accès aux variables partagées du système (variables globales et variables de communication).

### III.3.2.4. Raffinement de l'architecture et transformation de code

L'étape d'allocation implique la modification du code du modèle de l'architecture (des dizaines de fichiers) et la modification du code applicatif (comportement des tâches) s'exécutant sur les processeurs, la génération de code (éventuellement du bloc mémoire globale partagée) ainsi que l'exécution de plusieurs algorithmes d'optimisation.

Malgré cette complexité, l'étape du raffinement et transformation de code (adaptation du code de l'application à l'architecture mémoire allouée) est assez systématique, ce qui rend son automatisation très bénéfique car elle peut réduire considérablement le temps de conception et les erreurs [Mef01a], [Mef02].

Le but de cette étape est d'abord d'intégrer cette architecture mémoire au système, puis d'adapter le code initial de l'application à cette architecture mémoire.

Au cours de cette étape on doit atteindre les deux objectifs suivant :

- génération automatique des blocs mémoires,

- transformation et adaptation du code de l'application à l'architecture mémoire

Dans le flot de conception, si nous intégrons la mémoire partagée, certaines transformations doivent être effectuées sur le code de l'application. Par exemple, supposons qu'une variable A soit affectée dans la mémoire partagée, l'instruction B = A dans le code initial de l'application sera transformée en:

B = readmem(Mem_name, @A, size_of(A)).

Où :

- Mem_name est la mémoire à laquelle est affectée la variable A,
- @A est l'adresse de la variable A dans la mémoire Mem_name.

Nous supposons que les problèmes de consistance mémoire sont pris en charge par le concepteur de l'application par la synchronisation au niveau système.

**Remarque.** Nous avons défini plusieurs types de transformations de code (code du modèle de l'architecture + code du comportement des taches) liées à l'insertion de l'architecture mémoire dans le système. Elles sont présentées avec détails dans le chapitre IV.

### III.3.2.5. Simulation au niveau architecture

Après l'allocation mémoire, l'architecture finale du système est complètement définie. La simulation à ce niveau permet donc de vérifier la fonctionnalité du système avec ses blocs finaux (blocs à un niveau logique). C'est une étape très importante dans notre flot de conception car elle concerne le système décrit à un degré plus fin que le niveau système, et c'est une simulation beaucoup plus rapide que celle du niveau micro-architecture. Elle permet une première validation de l'architecture mémoire choisie.

### III.3.2.6. Affectation mémoire

L'affectation mémoire consiste à distribuer l'ensemble des données de l'application sur les mémoires allouées au cours de l'allocation mémoire. Ainsi, à l'issue de cette étape d'affectation chaque donnée de l'application doit posséder une adresse physique dans une mémoire.

Actuellement cette étape est effectuée sans algorithme d'optimisation. Nous présentons des perspectives pour son optimisation dans le chapitre VI.

Une table d'allocation mémoire est générée avec cette étape d'affectation.

### III.3.2.7. Génération de la table d'allocation finale

Quand les données de l'application sont affectées aux différents espaces adressables, un fichier intermédiaire (table d'allocation) est généré. Il contient toutes les informations sur les emplacements des variables. Ainsi, pour chaque donnée, on connaît la mémoire dans laquelle elle réside et son adresse.

Cette table d'allocation permet de pouvoir prendre en charge les variables partagées de l'application lors de la génération du système d'exploitation spécifique.

Un exemple d'allocation mémoire est donné ci-dessous.

allocation alloc1 = { ( ref data1, ref adr1 = (refmem1, 5000)),(…..

### III.3.2.8. Synthèse de l'architecture mémoire

Au cours de cette étape, le choix final des caractéristiques physiques des différentes mémoires est fixé. En effet, tenant compte de l'affectation mémoire (adresses physiques des données de l'application dans les mémoires), et de la disponibilité dans la bibliothèque des mémoires, les types des mémoires (SRAM, DRAM, SDRAM..) sont fixés ainsi que le nombre de ports et bancs de chaque mémoire.

Pour le moment, nous ne disposons pas d'outils réalisant automatiquement cette étape. Des travaux permettant son automatisation sont en cours et font partie du travail de thèse de Ferid Gharsalli [Gha02].

L'architecture mémoire est alors connue avec précision, ce qui autorise encore des optimisations. Le plus classique à ce niveau consiste à profiter des modes de transfert rapides offerts par les mémoires (accès partagé, mode burst), ce qui suppose que les données sont judicieusement placées dans la mémoire. Si les différents modules qui accèdent à la mémoire supportent ces modes d'accès rapides, il peut être nécessaire de modifier l'affectation mémoire de certaines données (adresse et banc mémoire).

La prise en compte de ces modes d'accès par les compilateurs spécifiques est décrite dans [Gru00] et [Rix00].

### III.3.2.9. Synthèse des adaptateurs de mémoires

Afin de connecter la mémoire partagée au réseau de communication, on insère entre la mémoire et le réseau une interface qui adapte le protocole d'accès à la mémoire à celui du réseau. Cette interface mémoire est dite aussi adaptateur mémoire. Il dépend uniquement du réseau de communication et de la mémoire.

Un exemple d'implémentation d'un adaptateur mémoire, mettant en œuvre une architecture composée d'une mémoire SRAM de 8 MO partagée entre deux processeurs motorola-68000 est proposé dans la Figure III. 7. Chaque processeur possède son adaptateur qui lui permet de communiquer avec les autres éléments de l'architecture via un réseau de communication point-à-point. Dans notre cas, on utilise le protocole poignée de main « hand-shake en anglais ». Une séparation verticale de la structure de l'adaptateur mémoire, permet de dissocier les dépendances inhérentes au protocole de communication utilisé par le réseau de communication (partie droite), des caractéristiques imposées par les protocoles d'accès à la mémoire.

L'adaptateur mémoire est composé de trois parties. Une partie liée au réseau de communication (partie droite) qui est constituée de deux contrôleurs et d'un arbitre. Une deuxième partie (partie gauche) qui est constituée d'un module adaptateur. La troisième partie (partie centrale) est constituée de trois bus internes (bus d'adresse, bus de donnée et de contrôle).

L'arbitre assure le partage des bus internes entre les différents processeurs qui demandent l'accès à la mémoire. Dans le cas de deux demandes d'accès simultanées à la mémoire par deux processeurs, l'arbitre donne le bus à un seul processeur et laisse l'autre en attente. On utilise une politique de priorité dans le cas d'accès simultané. Le contrôleur associé au processeur autorisé est

alors sélectionné par l'arbitre. Ce contrôleur met l'adresse sur le bus d'adresse interne et la donnée sur le bus de donnée interne dans le cas d'un accès en écriture.

Le module adaptateur est constitué de trois blocs. Un décodeur d'adresses qui transforme l'adresse virtuelle envoyée par le processeur en une adresse physique. Dans le cas de plusieurs bancs mémoires et selon l'adresse reçue, un bloc de sélection de mémoire permet de reconnaître la taille des données à transférer et sélectionne le banc mémoire. Le bloc logique de commande permet enfin d'activer les signaux de la mémoire sélectionnée et de mettre l'adresse physique sur le bus d'adresse de la mémoire et la donnée sur le bus de donnée de la mémoire dans le cas d'une écriture.

Dans le cas d'une lecture le module d'adaptation mémoire reste en attente pendant un nombre de cycles qui correspond à la latence mémoire. Ensuite, il lit la donnée sur le bus de données de la mémoire et la transmet sur le bus de donnée interne en générant un signal d'acquittement. Ce signal réveille le contrôleur qui a demandé l'accès. Ce même contrôleur réveille l'adaptateur du processeur source de la demande d'accès et lui véhicule la donnée. L'adaptateur du processeur désactive ces signaux, et un nouveau cycle d'accès peut alors commencer.

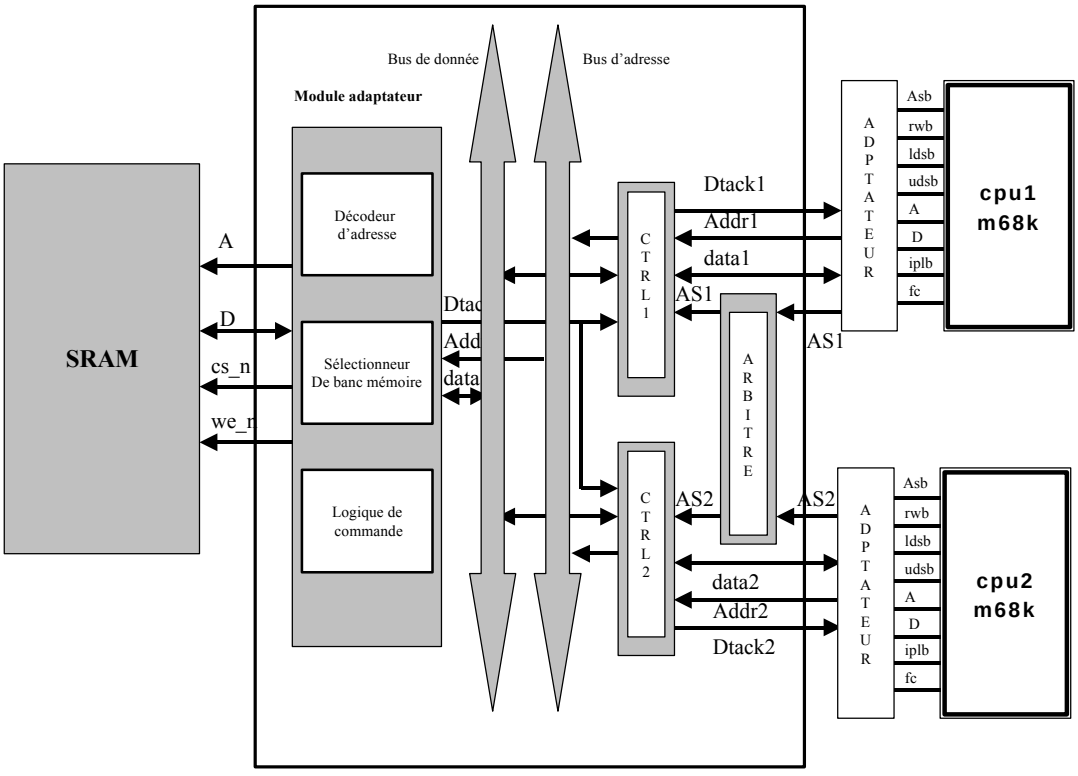


**Figure III. 7**. Adaptateur mémoire.

La Figure III. 8 montre un exemple d'un bout de code SystemC décrivant l'adaptateur d'une mémoire globale partagée par deux processeurs CP1 et CP2. Comme on le constate, dans cet exemple le processeur CP1 est prioritaire, donc en cas d'accès simultanés à la mémoire par les deux processeurs, la requête du processeur CP1 sera traitée en premier.

```
//mem_adaptor
#include ''systemc.h''
.
.
.
cpu1_end=false;
{
while(!req1.event() || !req2.event() ||
!clk.event() ) wait();
if(req2 && !req1){
  adresse=A2.read() ;
  data=D2.read() ;
.
.
ack2.write(true);

if(req1 && req2){
  adresse=A1.read() ;
  data=D1.read() ;
  .
  .
 cpu1_end=true;
 ack1.write(true);
 }
if(req2 && cpu1_end){
  adresse=A2.read() ;
  data=D2.read() ;
.
.
ack2.write(true);
}
.
.
```

**Figure III. 8.** Exemple de code SystemC décrivant l'adaptateur mémoire.

### III.3.2.10. Simulation au niveau micro-architecture

La simulation d'un système monopuce au niveau micro-architecture, même en étant très coûteuse en temps, restent une étape impérative lors de la conception de tels systèmes. En effet, cette simulation, venant après le ciblage logiciel/matériel permet de valider le système complet avec ses interfaces matérielles et système d'exploitation.

La Figure III. 9 montre un exemple de simulation au niveau micro-architecture d'un système contenant deux processeurs MC68000 et une SDRAM comme mémoire globale partagée par les deux processeurs. Ainsi le système est représenté par un ISS (Instruction Set Simulator) pour chaque processeur, un simulateur VHDL pour le code adaptant les protocoles d'accès des processeurs à ceux de la SDRAM, et un simulateur SystemC pour simuler la mémoire et son adaptateur (décrit en SystemC au niveau micro-architecture).

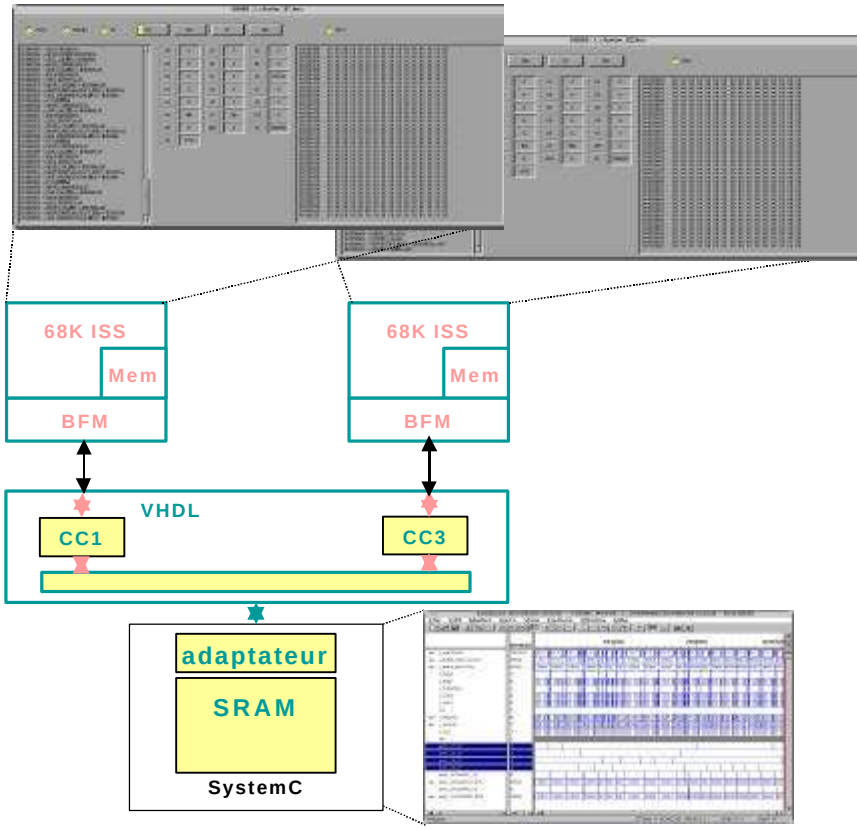


**Figure III. 9.** Exemple de simulation au niveau micro-architecture.

### *III.3.3. Sortie du flot*

Un système multiprocesseur monopuce est obtenu en fin de conception, c'est-à-dire une architecture spécifique (ensemble de processeurs, IPs, ASICs connectés entre eux par un réseau de communication via des adaptateurs de protocoles), architecture mémoire (type, taille, connexion des blocs physiques mémoire, mais aussi le placement mémoire de variables), ainsi que la partie logicielle (code de l'application, système d'exploitation pour chacun des processeurs).

## *III.4. Modèles de représentation des mémoires à travers les niveaux d'abstraction*

La taille du code d'une application augmente d'une façon très sensible entre le niveau système et le niveau micro-architecture. Ainsi, une application peut être décrite avec 50k lignes de code SDL au niveau système, 200K lignes de VHDL, C, SystemC au niveau architecture et avec 6M lignes de C exécutable et VHDL synthétisé au niveau micro-architecture.

Cette complexité, à laquelle s'ajoute la contrainte du délai de mise sur le marché qui devient de plus en plus court, les concepteurs d'aujourd'hui tentent de valider l'intégralité du système à partir du plus haut niveau d'abstraction possible.

La validation d'un système à un haut niveau d'abstraction passe généralement par la méthode de simulation. Pour simuler un système multiprocesseur monopuce à un haut niveau d'abstraction, on doit disposer de modèles pour représenter toutes ses parties (processeurs, mémoires, ..etc). Le flot de conception présenté dans le chapitre précèdent permet de modéliser toutes les parties d'un système à trois niveaux d'abstraction, sauf les mémoires. Nous présentons dans la suite de cette section trois modèles permettant de décrire les mémoires aux niveaux : système, architecture et micro-architecture.

### *III.4.1. Niveau système*

A ce niveau d'abstraction, on ne trouve pas de mémoires explicites dans l'architecture. Par contre on trouve dans la spécification de l'application des mémoires implicites sous forme de variables locales aux fonctions, variables globales à plusieurs fonctions et des variables utilisées pour la communication entre les fonctions.

La Figure III. 10 montre un exemple d'une application au niveau système. Cette application est composée de deux blocs A et B. Le premier contient les fonctions F1 et F2 et le second contient les fonctions F3, F4 et F5.

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

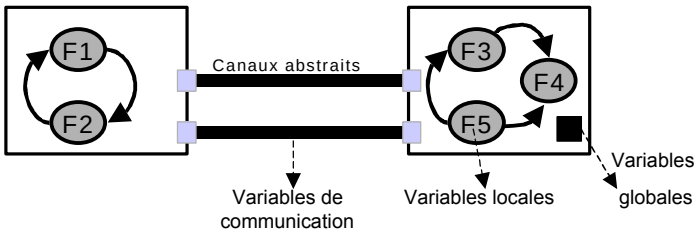


**Figure III. 10.** Niveau système.

### *III.4.2. Niveau architecture (Driver)*

A ce niveau, les modules correspondent aux blocs de l'architecture et ils communiquent via des canaux logiques. Au niveau architecture, on trouve des blocs mémoire globale explicites. Certaines variables globales ou variables de communication du modèle système sont affectées à des mémoires qui ont été allouées. Par contre les mémoires locales aux processeurs restent implicites à ce niveau d'abstraction. Les protocoles d'accès aux mémoires ne sont pas complètement définis.

La Figure III. 11 montre un exemple dans lequel on distingue un bloc mémoire représentant une mémoire globale partagée (à laquelle accèdent les deux autres blocs du système). Ce bloc mémoire est un bloc logique qui reproduit globalement le fonctionnement de la mémoire sans tenir compte des cycles d'horloge.

Dans l'exemple de la Figure III. 11, comme c'est souvent le cas à ce niveau d'abstraction, des variables globales et des variables de communication entre les deux blocs du système sont affectées au nouveau bloc mémoire. Par contre les variables locales à chaque bloc restent dans le même bloc.

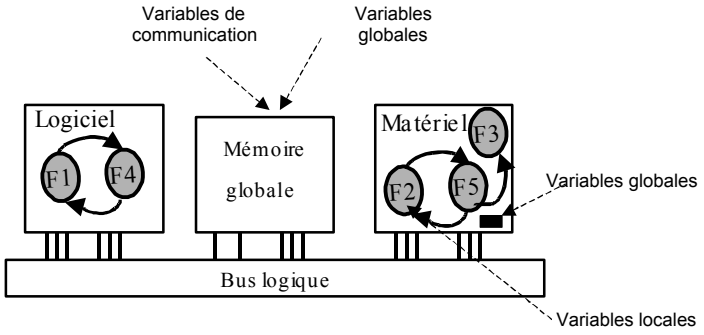


**Figure III. 11.** Niveau architecture

## *III.4.3. Niveau Micro-architecture (RTL)*

Au niveau micro-architecture, les mémoires deviennent des modules physiques tels que des SRAMs, DRAMs, SDRAMs ou autres. La mémoire globale est connectée au système via un adaptateur mémoire synthétisé. La structure de cet adaptateur mémoire est détaillée dans la suite de ce chapitre.

La Figure III. 12 donne un exemple de la micro-architecture qui peut être obtenue en raffinant le système décrit au niveau architecture de la Figure III. 11. La mémoire globale est devenue une mémoire SDRAM et les fonctions du bloc A sont exécutées par un processeur ARM7.

Le bloc B, en matériel dans la Figure III. 11 a été remplacé par un processeur MC68000.
Une mémoire locale (ROM et/ou RAM) est associée à chaque processeur. Elle est généralement connectée aux bus d'adresses et de données du processeur, ainsi qu'à quelques signaux de contrôle. Un contrôleur peut être nécessaire pour sélectionner et valider la ou les mémoires, mais aussi pour générer les signaux physiques adaptés à la mémoire utilisée.

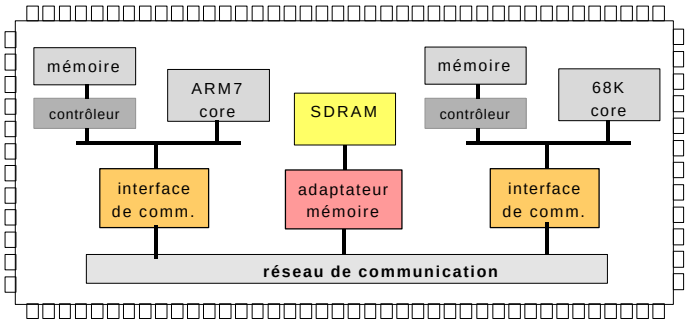


**Figure III. 12.** Niveau micro-architecture (RTL).

## *III.5. Intégration du flot de conception mémoire dans le flot global de conception du groupe SLS*

Comme cette thèse s'inscrit dans un projet global de réalisation d'un outil (le plus automatique possible) d'assistance à la conception de systèmes monopuce. Ce travail (flot de conception de l'architecture mémoire) doit être connecté à d'autres travaux réalisés ou en cours de réalisation au sein du groupe SLS.

En effet, après l'allocation mémoire et les transformations de code associées, on obtient un code de l'application ainsi qu'une table d'allocation. Ce code est repris par l'outil de génération des systèmes d'exploitation spécifiques, qui incluent par exemple des services permettants l'accès à une mémoire globale.

D'autre part, l'étape de la synthèse mémoire et des adaptateurs mémoires est étroitement liée à celle de la génération des interfaces matérielles pour les processeurs.
La simulation du système dans les différents niveaux d'abstraction fait appel aux bibliothèques de simulation et aux outils de cosimulation.

Etant donné que toutes les transformations du code de l'application liées à l'architecture mémoire réalisées dans les étapes de notre flot de conception mémoire ne modifient en aucun cas le langage de description initial de l'application, l'introduction du flot de conception mémoire ne modifie pas les entrées ou les sorties des autres outils déjà utilisés dans le flot classique (sans mémoire).

Le changement majeur lié au flot mémoire est sans doute la prise en compte dans les interfaces matérielles et logicielles des mémoires partagées et des accès à des mémoires distantes d'un processeur donné (si ce type de mémoire a été décidé lors de l'allocation mémoire). En effet, dans le flot classique on se contentait d'allouer manuellement une mémoire locale pour chaque processeur.

L'adaptation des outils de génération des interfaces logicielles/matérielles pour la prise en compte des mémoires partagées distribuées est en cours de réalisation au sein du groupe SLS.

La Figure III. 13 situe le flot de conception mémoire dans le contexte global du flot de conception de systèmes multiprocesseurs monopuce.

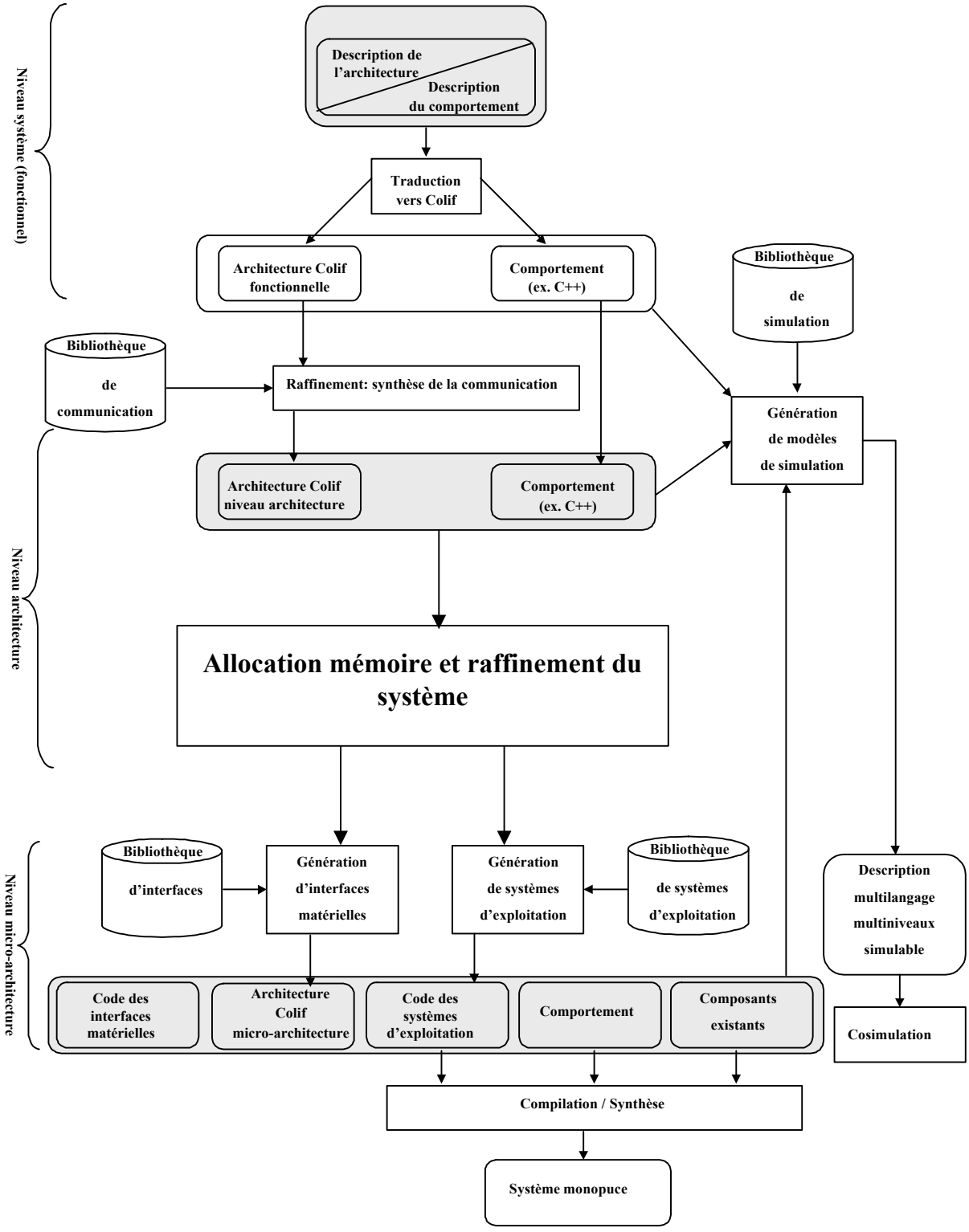


**Figure III. 13.** Flot complet de conception de systèmes multiprocesseurs monopuce.

## *III.6. Conclusion*

Nous avons décrit dans ce chapitre les étapes nécessaires à la conception d'un système multiprocesseurs monopuce. Cette conception se fait par phases en transformant conjointement l'architecture et le code applicatif.

La conception de l'architecture mémoire suit le même principe et elle a été intégrée dans le flot de conception initial. La mémoire est alors ajoutée au plus haut niveau d'abstraction et son implémentation devient de plus en plus précise à travers les différentes phases.

Concevoir l'architecture mémoire revient à prendre certaines décisions sur l'architecture (mémoire globale centralisée ou mémoire distribuée, placement mémoire). Pour aider le concepteur, dans cette phase délicate, nous proposons un algorithme d'allocation détaillé dans le chapitre suivant. Une telle décision implique de modifier le code de l'application, ce qui devient vite fastidieux. Là aussi les outils automatiques sont nécessaires et ils sont présentés au chapitre suivant.

# Chapitre IV

# ALLOCATION MEMOIRE ET RAFFINEMENT DU SYSTEME

Après avoir introduit dans le chapitre III le problème de l'allocation mémoire dans le cadre des systèmes multiprocesseurs monopuce, les différentes étapes de notre flot d'allocation mémoire et raffinement automatique du système sont détaillées dans ce chapitre. On commence par l'étape d'extraction des paramètres nécessaires à l'allocation mémoire, puis, un modèle d'allocation basé sur la programmation linéaire en nombres entiers est présenté. Ensuite l'étape du raffinement du système et des transformations automatiques du code de l'application est détaillée. Finalement, un aperçu sur l'automatisation du flot d'allocation mémoire est donné.

## IV.1. Introduction

L'allocation mémoire est une étape importante dans un flot de conception de systèmes sur une puce. En effet, c'est à l'issue de cette étape que l'architecture mémoire du système est fixée. Ainsi, tous les blocs mémoires (mémoires locales privées, locales partagées et globale partagée) deviennent explicites après l'allocation mémoire. Ceci est déterminant pour les performances du système final (surface et consommation en énergie).

Cette étape d'allocation implique la modification du code de l'application comprenant le comportement des taches et le modèle d'architecture (des dizaines de fichiers), la génération du code de l'application pour la simulation (modèle de simulation) au niveau architecture (éventuellement du bloc mémoire globale partagée) ainsi que l'exécution de plusieurs algorithmes d'optimisation.

Malgré cette complexité, cette étape est assez systématique, ce qui rend son automatisation très bénéfique car elle peut réduire considérablement le temps de conception.

Ainsi, le flot que nous présentons dans ce qui suit a pour objectifs :

- d'allouer des blocs mémoires de façon optimale pour l'application,
- de déterminer quel est le bloc mémoire le mieux adapté au stockage de chaque donnée (table d'allocation abstraite),
- de générer automatiquement le code du modèle d'architecture ainsi que celui des comportements des taches de l'application avec l'architecture mémoire au niveau architecture.

## IV.2. Flot d'allocation mémoire et de raffinement du système

Notre flot d'allocation mémoire et de raffinement du système prend en entrée une spécification de l'application au niveau système (après allocation des processeurs) et un modèle d'architecture mémoire générique (distribuée partagée).

Après une étape d'extraction d'informations (paramètres) du code de l'application initial et du rapport de simulation de niveau système, on génère un modèle linéaire en nombres entiers permettant de trouver l'architecture mémoire optimale à l'application. Ce modèle permet aussi de générer une table d'allocation mémoire abstraite spécifiant la mémoire à laquelle est affectée chaque donnée partagée de l'application.

Une dernière étape consiste à générer si nécessaire un bloc mémoire partagée et à l'insérer dans l'architecture du système. Ceci consiste à modifier le modèle structurel de l'architecture. Il faut ensuite transformer le code de l'application en adaptant les primitives d'accès aux données résidantes dans la mémoire partagée.

Ce flot est schématisé sur la Figure IV. 1.

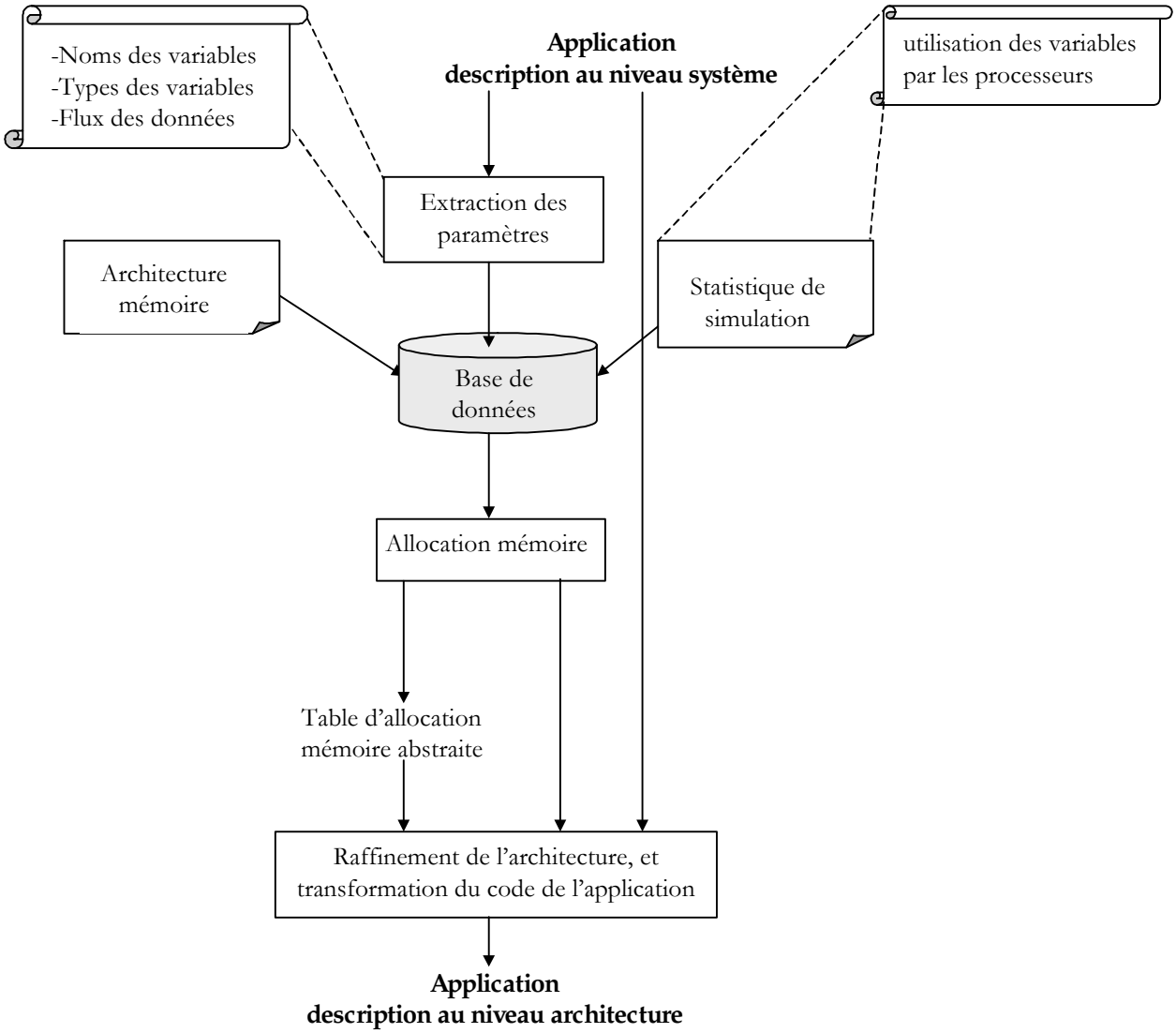


**Figure IV. 1.** Flot de conception de l'architecture mémoire à partir du niveau système.

Le flot est composé principalement de trois étapes : extraction des paramètres, allocation mémoire et raffinement du système/transformation de code. Ces étapes sont détaillées dans les sections suivantes de ce chapitre.

## *IV.3. Extraction des paramètres*

Dans le but de construire un modèle mathématique permettant de trouver une bonne architecture mémoire spécifique à une application, on analyse le code de l'application au niveau système afin d'extraire certaines informations concernant les données partagées de l'application.

Une variable est toujours caractérisée par son nom et son type (taille). En plus de ces caractéristiques, d'autres informations sont importantes, pour décider de mettre une donnée dans une mémoire partagée ou non. Ces informations sont l'utilisation des canaux de communication,

véhiculant les données, entre les modules de l'architecture. En effet, si l'utilisation d'une variable est dominée par un des processeurs, alors la meilleure solution est peut être de la faire résider dans la mémoire locale de ce processeur.

Un exemple des informations extraites de la spécification initiale de l'application, en analysant le code, est donné dans la Figure IV. 2. La variable data1 (resp. data2) et un entier de 16 bits (resp. 32 bits). Elle est utilisée en lecture par les processeurs P1 et P2 (resp.P1 et P3), et en écriture par P3 (resp. P1 et P2).

```
/*************************************************************/
/************** Extraction de paramètres **********************/
/*************************************************************/
```

| Donnée | Taille | Lecture | Ecriture |
|---|---|---|---|
| data1 | int_16 | P1, P2 | P3 |
| data2 | int_32 | P1, P3 | P1, P2 |
| … | | | |
| … | | | |

**Figure IV. 2.** Exemple des informations extraites du code de l'application.

Ainsi, pour chaque donnée partagée dans l'application, on extrait de la spécification : son nom, sa taille, la liste des processeurs qui accédent en lecture et la liste des processeurs lui accédant en écriture.

## *IV.4. Utilisation des résultats de la simulation de niveau système*

L'analyse du code de l'application donne des informations sur l'utilisation des canaux de communication, mais ces informations restent incomplètes. En effet, elles nous permettent de savoir quel processeur accède à toute variable partagée dans l'application, et avec quel type d'accès, sans toute fois avoir les taux d'accès à ces variables (c'est-à-dire à quelle fréquence un processeur accède à une donnée partagée en lecture ou en écriture).

Les taux d'utilisation des variables par les différents processeurs sont des informations dynamiques et sont difficiles à estimer rien qu'en utilisant la spécification de l'application. Ainsi, ils sont obtenus dans notre flot de conception grâce à la simulation de l'application au niveau système (chapitre III).

Toutes ces données sont stockées dans une base de données qui servira à construire le modèle d'allocation.

## IV.5. Allocation mémoire

L'algorithme utilisé pour résoudre le problème de l'allocation mémoire consiste en un programme linéaire en nombres entiers [Mef01b]. Ce programme linéaire est généré automatiquement en utilisant les informations stockées dans la base de données à l'étape précédente. Il est décrit dans la suite de ce chapitre.

### IV.5.1. Notations

Dans tout ce qui suit, nous utiliserons les notations suivantes :

- p = nombre de processeurs,
- v = nombre de variables,
- m = nombre de mémoires.

### IV.5.2. Architecture mémoire ciblée

Nous ciblons dans notre modèle d'allocation, des architectures mémoires distribuées partagées. Ainsi, l'architecture mémoire la plus générale que l'on peut obtenir avec notre modèle consiste en une mémoire globale partagée, une mémoire locale distribuée et une mémoire locale privée associées à chaque processeur.

Dans tout ce qui suit, on notera :

- $M_i$ la mémoire locale associée au processeur $P_i$,
- $M_{p+1}$ la mémoire globale partagée.

Les mémoires locales privées n'apparaissent pas de façon explicite dans notre modèle, car elles sont réservées aux données locales. Notre modèle ne traite que les données partagées de l'application. Par défaut, les données locales sont placées dans les mémoires locales aux processeurs, ceci est pris en charge par les compilateurs des processeurs.

### IV.5.3. Flexibilité du modèle d'allocation mémoire

La flexibilité d'un flot est définie par l'ensemble des paramètres que l'ont peut modifier plus ou moins facilement. Ainsi, notre flot prend certains paramètres, tels que les types de processeurs et les modèles d'accès aux mémoires, fournis sous forme de bibliothèques. On peut mettre à jours ces bibliothèques. Pour ajouter un nouveau type de processeur, il suffit d'estimer ou de mesurer ses temps d'accès aux diffèrent types de mémoires. Le contenu des bibliothèques est décrit ci-dessous.

- Bibliothèque de temps d'accès : cette bibliothèque contient les temps d'accès aux diffèrent types de mémoires (locale privée, locale distribuée, globale partagée) pour chaque type de processeur. Ces temps d'accès peuvent être d'un niveau fonctionnel ou du niveau RTL tenant compte des spécificités de nos systèmes d'exploitation (en tenant compte des interfaces) offrant ainsi au concepteur un très large domaine d'exploration de l'architecture mémoire.

- Bibliothèque mémoires : elle définit les restrictions du concepteur sur les types de mémoires qu'il souhaite ou non inclure dans l'architecture. En effet, par exemple en interdisant l'utilisation des mémoires locales distribuées, le concepteur limite l'exploration aux architectures à mémoire partagée globale. Cette bibliothèque contient aussi le coût moyen d'intégration de chaque type de mémoire dans l'architecture, ainsi que le coût moyen en fonction de la taille de chaque type de mémoire.

### *IV.5.4. Les variables de décision*

Nous utilisons dans notre modèle d'allocation mémoire les trois types de variables de décision suivants :

$X_{kj}$ est de type binaire = { 1 si et seulement si la variable j est affectée à la mémoire k. ; 0 sinon

TM(k) est de type entier = Taille de la mémoire k

$Y_k$ est de type binaire = { 1 si et seulement si la mémoire k fait partie de l'architecture. ; 0 sinon

### *IV.5.5. La fonction objectif*

Notre objectif dans cette allocation mémoire consiste à minimiser le temps d'accès global des processeurs aux données partagées, ainsi que le coût total de l'architecture mémoire.

- Le temps d'accès global des processeurs aux données partagées de l'application comprend :

  - le temps global d'accès en lecture par les processeurs aux données partagées, que l'on peut exprimer sous la forme :

$$\sum_{i=1}^{p}\Big(\sum_{j=1}^{v} Nb\_Acces\_Lect(i,j)\sum_{k=1}^{m} T\_Acces\_Lect(i,k) * X_{kj}\Big)$$

  où :

  Nb_Acces_Lect est un tableau de données à deux dimensions indiquant le nombre d'accès en lecture de chaque processeur i à chaque variable partagée j.

  T_Acces_Lect est un tableau de données donnant le temps d'accès en lecture de chaque processeur j à chaque mémoire k.

- le temps global d'accès en écriture par les processeurs aux données partagées, que l'ont peut exprimer sous la forme :

$$\sum_{i=1}^{p}\Big(\sum_{j=1}^{v} Nb\_Acces\_Ecr(i,j)\sum_{k=1}^{m} T\_Acces\_Ecr(i,k) * X_{kj}\Big)$$

où :

Nb_Acces_Ecr est un tableau de données à deux dimensions indiquant le nombre d'accès en écriture de chaque processeur i à chaque variable partagée j,

T_Acces_Ecr est un tableau de données donnant le temps d'accès en écriture de chaque processeur j à chaque mémoire k.

- Le coût total de l'architecture mémoire est donné par la somme des deux termes suivants :

- le coût du à la taille de chaque mémoire. Ce coût est proportionnel à la taille des données affectées à chaque mémoire k. Il peut etre representé comme suit :

$$\sum_{k=1}^{m} CBM_k * Taille\_Mem_k$$

où $CBM_k$ est le coût unitaire de la mémoire k

Le coût dû à l'intégration de chaque mémoire k dans l'architecture k. Ce coût est différent du coût précédent du fait qu'il ne dépend pas de la taille de la mémoire mais uniquement de son type. En effet il sera le même par exemple pour une mémoire globale partagée de 8 Mo et une autre de 32Mo.

En effet, il correspond à la fois à l'effort de l'intégration de la mémoire en tant qu'entité et au coût financier engendré par cette intégration. Il est difficilement chiffrable en valeur absolue, mais une estimation par le concepteur (même grossière) dans notre modèle d'allocation (avec des degrés d'approximation comparables pour toutes les mémoires) est suffisante.

Il est exprimé pour l'ensemble des mémoires de l'architecture par :

$$\sum_{k=1}^{m} CUM_k * Y_k$$

Ainsi la fonction objectif du modèle linéaire peut être exprimée comme suit

$$\begin{aligned} \text{Min } F = & \sum_{i=1}^{p}\Big(\sum_{j=1}^{v} nb\_read(i,j)\sum_{k=1}^{m} T\_read(i,k) * X_{kj}\Big) \\ & + \sum_{i=1}^{p}\Big(\sum_{j=1}^{v} nb\_write(i,j)\sum_{k=1}^{m} T\_write(i,k) * X_{kj}\Big) \\ & + \sum_{k=1}^{m}\Big(CBM_k * TM_k\Big) + \sum_{k=1}^{m}\Big(CUM_k * Y_k\Big) \end{aligned}$$

### *IV.5.6. Les contraintes*

Les contraintes auxquelles notre modèle d'allocation mémoire est soumis sont globalement triviales. Elles peuvent être résumées comme suit :

***- La taille d'une mémoire doit être supérieure où égale à la somme des tailles des données qu'elle contiendra :***

$$\sum_{j=1}^{v} \left( \text{Taille\_Var}(j) * X_{jk} \right) \leq TM_k \qquad k = 1,..,m$$

où

Taille_var(j) désigne la taille de la donnée j

***- Chaque donnée doit être affectée à une et une seule mémoire :***

$$\sum_{k \in S_j} X_{kj} = 1 \qquad j = 1,..,v$$

où

$S_j$ désigne l'ensemble des indices des mémoires locales associées aux processeurs utilisant la données k, plus l'indice de la mémoire globale partagée.

***- Contraintes de cohérence du modèle***

Ce sont des contraintes importantes liées à la dépendance entre certaines variables du modèle. En effet, il existe une forte dépendance entre les variables $TM_k$ et $Y_k$, car pour toute mémoire k :

a- si la mémoire k n'est pas incluse dans l'architecture mémoire du système ($Y_k = 0$) alors aucune variable ne doit être affectée à cette mémoire ($TM_k = 0$).

et

b- si il y a au moins une variable affectée à la mémoire k ($TM_k > 0$), alors la mémoire k doit être incluse dans le système ($Y_k = 1$).

Une façon triviale de décrire mathématiquement ces deux contraintes est :

$$TM_k (1 - Y_k) \leq 0 \quad k = 1,..,m$$

Ainsi, dans cette contrainte :

Si $Y_k = 0$ alors on a $TM_k \leq 0$, et comme $TM_k$ est une variable entière positive ou nulle alors on a forcement $TM_k = 0$ ce qui satisfait donc la contrainte (a).

et

Si $TM_k > 0$, on a obligatoirement $Y_k = 1$, car si $Y_k = 0$ la contrainte devient $TM_k \leq 0$ ce qui contredit l'hypothèse $TM_k > 0$. Donc ceci satisfait la contrainte (b).

Cependant, cette façon de formuler les deux contraintes (a) et (b) présente un inconvénient majeur qui réside dans le fait qu'elle n'est pas linéaire. Ce qui rendrait le modèle final non linéaire et par conséquent encore plus « difficile » à résoudre.

Pour remédier à ce problème, et pour conserver la linéarité de notre modèle, nous allons modéliser les contraintes (a) et (b) comme suit :

la formule linéaire ci-dessous (où A est un très grand nombre entier positif supérieur à tous les $TM_k$ quelque soit k) satisfait la contrainte (a) car si $Y_k = 0$, alors la formule devient $TM_k \leq 0$, et comme $TM_k$ et une variables entière positive alors ça implique que $TM_k = 0$.

$$TM_k \leq A * Y_k \qquad k = 1,..,m \qquad A >>> 0$$

la contrainte (b) est satisfaite aussi, car comme $TM_k$ est une variable entière positive ou nulle alors si $TM_k > 0$, alors si $Y_k = 0$ la formule devient $TM_k \leq 0$, ce qui contredit l'hypothèse initiale. D'où on en déduit que $Y_k = 1$, ce qui satisfait la contrainte (b).

- Les variables $X_{ij}$ doivent être binaires pour tout i, j.

$$X_{jk} \in \{0,1\} \qquad j = 1,..,v \qquad \text{et} \qquad k = 1,..,m$$

- Les variables $Y_k$ sont binaires quelque soit k, et les variables $TM_k$ sont entières pour tout k.

$$Y_k \in \{0,1\} \qquad \text{et} \qquad TM_k \in N \qquad k = 1,..,m$$

Ainsi le programme linéaire en nombres entiers final est le suivant :

**Objectif :**

$$\text{Min } F = \sum_{i=1}^{p}\Big(\sum_{j=1}^{v} nb\_read(i,j)\sum_{k=1}^{m} T\_read(i,k) * X_{kj}\Big)$$
$$+ \sum_{i=1}^{p}\Big(\sum_{j=1}^{v} nb\_write(i,j)\sum_{k=1}^{m} T\_write(i,k) * X_{kj}\Big)$$
$$+ \sum_{k=1}^{m}\big(CBM_k * TM_k\big) + \sum_{k=1}^{m}\big(CUM_k * Y_k\big)$$

**Sous les contraintes:**

$$\sum_{j=1}^{v}\big(Taille\_Var(j) * X_{jk}\big) \le TM_k \qquad k = 1,..,m$$
$$TM_k \le A * Y_k \qquad k = 1,..,m \qquad A >>> 0$$
$$\sum_{k \in S_j} X_{kj} = 1 \qquad j = 1,..,v$$
$$X_{jk} \in \{0,1\} \qquad j = 1,..,v \qquad \text{et} \qquad k = 1,..,m$$
$$Y_k \in \{0,1\} \qquad \text{et} \qquad TM_k \in N \qquad k = 1,..,m$$

**Figure IV. 3.** Programme linéaire en nombres entiers d'allocation mémoire.

## *IV.5.7. Analyse du modèle d'allocation mémoire*

### IV.5.7.1. Complexité du modèle

Pour une instance de ce modèle linéaire en nombres entiers, on obtient :

- au moins $[(P + 1).V]^a + [P + 1]^b + [P + 1]^c = (P + 1)(V + 2)$ variables.

Avec:

a = nombre de variables $X_{jk}$,

b = nombre de variables $Y_k$,

c = nombre de variables $TM_k$.

- NB_contraintes = (P + 1) + (P + 1) + V contraintes

Notons que dans le calcul du nombre de variables, nous avons supposé que chacune des "v" données partagées est utilisée par tous les processeurs (les "p" processeurs). Ceci est donc le pire cas théorique, car dans les applications réelles il est très rare de tomber sur un tel cas pareil.

### IV.5.7.2. Avantages

La modélisation et résolution du problème d'allocation mémoire en utilisant le programme linéaire ci-dessus présentent des avantages majeurs tels que :

- c'est un modèle exact qui, contrairement aux heuristiques, nous donne la solution optimale si elle existe,
- c'est un modèle très générique car il permet de tenir compte de tous les types d'architectures mémoire. En effet, on peut envisager des mémoires : locales privées, locales distribuées, et globale partagée. Ce qui permet une exploration assez exhaustive des architectures mémoire possibles,
- il permet de résoudre deux problèmes importants en même temps : l'allocation des blocs mémoire et l'affectation des variables dans ces blocs,
- on peut trouver un nombre important d'outils très bien élaborés permettant la résolution d'un tel modèle.

### IV.5.7.3. Inconvénients

Etant donné que les variables de ce modèle linéaire sont de type binaire, sa résolution peut être très lente en fonction du nombre de variables. Ainsi il est souhaitable de n'utiliser dans ce modèle que les variables « importantes » de l'application, c'est à dire les plus volumineuses où celles aux quelles les processeurs accèdent le plus souvent.

### IV.5.7.4. Solution

Dans le cas d'applications contenant un nombre « important » de processeurs et de variables partagées, il est déconseillé d'utiliser ce modèle d'allocation basé sur la programmation linéaire en nombres entiers, car son temps de résolution peut exploser d'une façon exponentielle.

Dans de tels cas, il est préférable d'utiliser des modèles heuristiques basés sur la théorie des tests statistiques par exemple. Ces derniers modèles sont présentés dans le chapitre VI de cette thèse comme perspectives.

**Remarque :** Dans ce modèle, les tailles des différentes mémoires sont fixées par le programme linéaire par rapport aux tailles des données qui leur sont affectées.

La taille des mémoires obtenue avec le modèle d'allocation (somme des tailles des variables affectées à cette mémoire est une taille minimale (borne inférieure). Le concepteur doit donc choisir une mémoire existante de taille supérieure à la taille obtenue.

## IV.6. Raffinement du système

### IV.6.1. Génération de code

Si au cours de l'allocation, il est décidé d'inclure une mémoire globale partagée dans l'application, il faut ensuite générer le modèle (structurel et comportemental) de cette mémoire au niveau architecture et l'insérer dans la spécification de l'application.

Ce bloc est constitué lui-même des trois parties suivantes :

- le bloc mémoire partagée : il est générique et indépendant de l'application avec un port de lecture, un port d'écriture et un port d'adresse. Il s'agit simplement d'une matrice de taille théoriquement infinie,

```
#include "systemc.h"
 #include "ram.h"

 void ram::entry()
 {
 while (true) {
…
  if(rw.read() = 0)
  {
  a = adrin.read();
  memoire[a] = datain.read();
  }
...
…
 if(rw.read() = 1)
  {
  a = adrout.read();
  dataout.write(memoire[a]);
  wait();
  }
  }
```

**Figure IV. 4.** Exemple de code décrivant le fonctionnement d'une mémoire au niveau architecture.

A un haut niveau d'abstraction, le modèle comportemental de la mémoire est intemporel. En effet il ne tient pas compte des cycles nécessaires aux opérations de lecture ou d'écriture, (ni rafraîchissement dans le cas d'une DRAM). Un tel modèle est donné Figure IV. 4.

Ainsi, la mémoire globale partagée est simplement un tableau (mémoire[] dans la Figure IV. 4) de grande taille pouvant contenir des entiers. Elle reçoit les données sur le port d'écriture "datain", et envoi des données sur le port de lecture "dataout".

- un contrôleur d'écriture : la résolution du programme linéaire nous donne les variables présentes dans la mémoire partagée, ainsi que le nombre de processeurs (noté k) accédant à ces données en écriture. Ce contrôleur possède alors « k » ports logiques d'entrée. Les opérations d'écriture dans la mémoire partagée sont alors gérées par une file d'attente, et le contrôleur se charge de toutes les écritures dans la mémoire globale partagée,
- un contrôleur de lecture : comme celui d'écriture, ce contrôleur possède un nombre de ports logiques égal au nombre de processeurs accédant en lecture aux variables affectées dans la mémoire globale partagée.

Ces deux contrôleurs constituent l'adaptateur de la mémoire globale partagée à un niveau d'abstraction plus bas.

Toutes les variables globales ou de communication de type non binaire affectées à la mémoire globale partagée, sont caractérisées par une étiquette (un nom) et une adresse abstraite (indice dans le tableau mémoire). Cette caractérisation des données de l'application est donnée dans une table d'allocation mémoire. Un exemple de cette table d'allocation abstraite est donné ci-dessous.

```
/*****************************************************/
/************** Abstract Allocation Table**************/
/*****************************************************/

data1       int_16        LM1
data2       int_32        LM2
data3       int_32        GSM    0
data4       int_32        GSM    1
…
…
/*****************************************************/
```

**Figure IV. 5.** Exemple d'une table d'allocation abstraite.

La Figure IV. 5, montre un exemple de résultat de l'allocation mémoire abstraite. Cette table indique que la variable data1 (resp. data2) de taille 16 bits (resp. 32 bits) est affectée à la mémoire locale LM1 (res. ML2). Par contre data3 et data4 sont affectées à la mémoire globale partagée (GSM), respectivement dans les indices 0 et 1 (adresses abstraites des données dans la mémoire globale partagée).

Dans cet exemple on constate qu'il n'y a pas d'adresses logiques (indices) pour les données affectées aux mémoires locales. Ceci est dû au fait que notre modèle d'allocation ne traite que les données affectées à des mémoires éloignées, celles affectées aux mémoires locales étant prises en charge par les compilateurs des différents processeurs.

## IV.6.2. Transformation de code

Comme au niveau système une application est constituée uniquement de processeurs communiquant par passage de messages, la mémoire n'existe pas explicitement. L'échange de données entre les blocs à ce niveau est réalisé par des primitives simples de type Send/Receive.

Après une éventuelle décision par l'algorithme d'allocation mémoire d'affecter certaines données de l'application dans une mémoire globale partagée, un problème d'accès à ces données partagées apparaît dans le code comportemental des taches de l'application. En effet, les simples primitives Send/Receive ne suffisent plus pour permettre aux processeurs d'accéder aux données partagées éloignées (affectées à la mémoire partagée). Ainsi, nous distinguons deux types de données : les variables globales et les variables de communication, nécessitant des transformations appropriées de code dans leurs primitives d'accès afin de produire une nouvelle spécification du comportement des tâches de l'application tenant compte de l'architecture mémoire partagée.

### IV.6.2.1. Variables de communication

Ce sont toutes les variables que les modules utilisent pour communiquer entre eux. Il s'agit de variables binaires ou de données non binaires échangées entre les différents modules

***- Variables binaires***

Nous ne traitons pas ce type de données dans notre modèle d'allocation mémoire. Il reste toujours affecté aux mémoires locales des processeurs. En effet, dans une spécification de niveau système, les variables binaires correspondent généralement à des signaux de synchronisation. Dans nos applications, on suppose que la synchronisation est entièrement prise en charge par le concepteur de niveau système. On garde ces signaux de synchronisation dans leur état initial dans un soucis de conserver la cohérence du système.

***- Variables non binaires***

Après l'allocation mémoire, chaque variable partagée est affectée à une mémoire donnée, et est caractérisée, dans la table d'allocation abstraite, par une adresse abstraite sur cette mémoire (indice dans le tableau mémoire).

Supposons que l'on dispose d'un système constitué de deux processeurs P1 et P2. « X » est une variable partagée par P1 et P2.

***Primitive de lecture***

Pour recevoir la variable "X" envoyée par P2, le processeur P1 doit tout simplement exécuter l'instruction *Receive(X)* après la réception d'un signal de synchronisation comme suit :

```
{
..
Wait for synch_signal_P2
Receive(X);
..
}
```

Au niveau architecture, si l'on décide d'insérer une mémoire globale partagée dans le système, et si la donnée "X" et affectée à cette mémoire globale partagée, alors la simple primitive *Receive(X)* ne suffit plus au processeur P1 pour pouvoir lire « X ». En effet, après réception d'un signal de P2

attestant la disponibilité de la donnée dans la mémoire, P1 envoie une requête, via le port le connectant au contrôleur de lecture de mémoire, demandant la donnée. Elle lui est renvoyée par le contrôleur de lecture de la mémoire au prochain cycle comme décrit ci-dessous.

***Processeur récepteur***

```
{
..
Wait for synch_signal_P2;
Ask_for_data_in_shared_memory_output_contr("X");
Wait();
Read_from_output_contr(X);
..
}
```

A la réception de la requête de P1, le contrôleur de lecture de la mémoire récupère l'indice de l'emplacement de « X », sur la mémoire, dans la table d'allocation abstraite, puis récupère la valeur de « X » dans la mémoire pour l'envoyer à P1 via le canal les connectant. Un bout du code correspondant à l'envoi de « X » à P1 est donné ci-dessous.

***Contrôleur mémoire***

```
{
...
Ind = read_data_index_from_adstract_alloc_table("X");
X = Read_cell_in_memory_matrix(ind);
Write_data_in_port(X);
Wait();
...
}
```

***Primitive d'écriture***

Pour envoyer "X" à P2, le processeur P1 au niveau système doit simplement envoyer la valeur directement à P2, suivi d'un signal de synchronisation, comme décrit ci-dessous.

```
{
...
Send(X);
Send(synch_signal);
Wait();
...
}
```

Au niveau architecture, si "X" est affectée à la mémoire globale partagée, la primitive *Send(x)* ne suffit plus pour envoyer "X" par P1. En effet, l'envoi de "X" doit passer par la mémoire partagée. Ainsi comme décrit ci-dessous, P1 envoi le nom et la valeur de "X" au contrôleur d'écriture de la mémoire suivi d'un signal de synchronisation à P2.

***Processeur écrivant la donnée***

```
{
...
write_signal_data_shared_mem_input_contr("X", X);
write(synch_signal);
wait();
...
}
```

***Contrôleur mémoire***

Après réception de la valeur de « X » la contrôleur mémoire récupère son indice en mémoire dans la table d'allocation abstraite. Puis écrit la valeur dans l'emplacement mémoire correspondant à cet indice, comme dans le code suivant :

```
{
...
ind = read_data_index_from_adstract_alloc_table("X");
write_data_in_cell(ind, X);
...
}
```

On notera que pour l'écriture dans la mémoire partagée, il ne suffit pas que le processeur donne la valeur de la variable, mais il faut aussi son étiquette (nom).

### IV.6.2.2. Les variables globales

Dans la spécification d'une application au niveau système, on trouve dans le comportement de certains processus des accès à des variables globales dans des expressions ou dans des affectations.

En effet, si "X" est une variable globale, on peut trouver dans une fonction "A" de la spécification système des expressions telles que :

```
Y = X + 2;
Or
X = Y/2;
```

La première instruction correspond à un accès en lecture de "X", et la seconde instruction à un accès en écriture de "X".

Si l'on décide de mettre « X » dans la mémoire globale partagée à l'issue de l'allocation mémoire, alors on doit remplacer tous les accès à « X » dans les deux expressions précédentes comme suit:

$Y = X + 2$ (lecture) devient

```
{
Sig_to_read_shared_mem(X); ------------------(1)
wait(); -------------------------------------(2)
var = read_shared_meme(X); ------------------(3)
Y = var + 2; --------------------------------(4)
}
```

L'instruction (1) consiste à envoyer une requête via le canal connectant "A" au contrôleur de lecture de la mémoire globale partagée, pour demander la valeur de "X.

Après réception d'une telle requête (après l'instruction de synchronisation (2)), le contrôleur récupère l'indice correspondant à "X" de la table d'allocation abstraite, puis envoie la valeur de « X » qui est récupérée dans une variable temporaire « var » (instruction (3)). Finalement l'expression est calculée dans (4).

La seconde expression (écriture) X = Y/2 devient

```
{
write_shared_mem("X", Y/2); -----------------(5)
wait(); -------------------------------------(6)
}
```

L'instruction (5) consiste à l'envoi d'une requête d'écriture au contrôleur d'écriture de la mémoire globale partagée. Cette requête est constituée d'un message contenant le nom « X » et la valeur « Y/2 » de la variable, comme dans le cas des variables de communication.

**Remarque :** les transformations de code présentées ci-dessus sont des transformations bloquantes**,** car un processeur ne fait aucun autre traitement pendant qu'il accède à la mémoire partagée.

## *IV.7. Automatisation du flot d'allocation mémoire*

Le flot d'allocation mémoire et le raffinement présenté dans ce chapitre a été implémenté en C. L'implémentation se compose essentiellement des trois modules suivants :

- Analyseur de code : ce module est un analyseur simplifié du code SystemC. En effet, il extrait toutes les informations nécessaires à la génération du programme linéaire en nombres entiers, en parcourant principalement les fichiers d'interfaces des modules (les fichiers SystemC ayant l'extension « .h ») et le fichier de coordination de l'application « main.cc ».

Les informations extraites sont stockées dans des fichiers de données.

- Générateur et solveur du programme linaire en nombres entiers : ce second module génère le programme linéaire en utilisant les fichiers de données obtenus précédemment, puis fait appel aux bibliothèques Cplex de Ilog pour résoudre le programme et obtenir ainsi l'architecture mémoire, et une affectation abstraite des données aux différentes mémoires.
- Générateur et transformateur de code : ce module se charge de la génération du blocs mémoire partagée au niveau architecture, des contrôleurs et de la transformation des primitives d'accès aux données partagées dans la spécification initiale de l'application. Il constitue le module le plus fastidieux à implémenter surtout dans le cas où les primitives de communication sont cachées dans le reste du code de l'application. Dans l'état actuel des travaux, ce module est automatisé partiellement et il nécessite donc l'intervention manuelle sur le code de l'application.

## *IV.8. Conclusion*

Dans ce chapitre, nous avons présenté un modèle optimal d'allocation mémoire pour les systèmes multiprocesseurs. Ce modèle est déterministe et est facilement automatisable.

Le programme linéaire en nombres entiers utilisé pour allouer les blocs mémoires du système, peut être étendu pour résoudre d'autres problèmes importants dans la conception des systèmes multiprocesseurs monopuce, tels que le problème classique d'allocation des unités de communication (registres, FIFO, ..etc). Ces perspectives sont discutées dans le chapitre VI de cette thèse.

# Chapitre V

# APPLICATION : LE VDSL

Ce chapitre présente l'application du VDSL pour illustrer les étapes de la méthodologie proposée dans cette thèse. Il commence par décrire le principe du VDSL et sa spécification au niveau système. Puis les étapes de raffinement menant à une description au niveau architecture sont détaillées. La présentation et la discussion des résultats obtenus concluent ce chapitre.

## *V.1. Introduction*

La technologie VDSL est une nouvelle technologie pour la communication haut débit sur ligne téléphonique. VDSL signifie «Very high data rate Digital Subscriber Line ». C'est une technologie qui permet la transmission de données à grande vitesse sur des lignes téléphoniques de cuivre torsadées, avec un intervalle des vitesses dépendant de la longueur de la ligne réelle. Elle fait partie des technologies dites xDSL, toutes dérivées de la technologie DSL utilisée dans le cadre de liaisons numériques RNIS (Réseau Numérique à Intégration de Services). Cette famille regroupe quatre technologies différentes : l'ADSL, HDSL, SDSL et VDSL.

Actuellement, L'ADSL est la technologie la plus au point et est commercialement prête. Le VDSL est une technologie voisine moins complexe qui permet des débits plus élevés. Avec l'ADSL, les débits maximums sont de 800 Kbps en émission et de 8 Mbps en réception tandis que le VDSL permet d'atteindre les vitesses maximales de 16 Mbps en émission et 52 Mbps en réception.

Le principe du VDSL est de substituer à la traditionnelle modulation du courant électrique une modulation nommée DMT pour Discrete MuliTone modulation. Le DMT est une modulation quadratique d'amplitude qui tronçonne la bande passante en tronçons de 4 kHz.

Avec cette technologie, l'abonné peut téléphoner et se connecter à Internet en même temps sur une seule prise téléphonique classique. A l'arrivée de la ligne de cuivre, les fréquences vocales sont acheminées vers le réseau téléphonique tandis que les autres données sont dirigées vers le réseau Internet. Sans recâblage ni changement de postes téléphoniques, l'utilisation d'un modem spécifique suffit pour utiliser cette technologie.

## *V.2. L'architecture VDSL*

STMicroelectronics a développé un modem spécifique au VDSL. La particularité de ce système est de calculer le débit optimal en fonction de la qualité de la ligne (longueur, état etc…). Les parties émission, réception, codage et décodage sont implémentées par trois blocs matériels et un DSP avec une DPRAM (Dual-Port Random Acces Memory). Le DSP permet aussi, avec un stockage périodique de données transmises, de mesurer le débit maximal supporté par la ligne. L'ARM7 qui communique avec la DPRAM gère la partie évaluation de l'état de la ligne. Un microcontrolleur (MCU) est chargé de contrôler, configurer et synchroniser la chaîne de transmission en fonction de l'état de la ligne.

Afin de soulager le DSP de l'architecture initiale, il a été proposé d'ajouter un processeur ARM (Figure V. 1). L'ARM7 qui communique avec la DPRAM gère la partie évaluation de l'état de la ligne. Le deuxième processeur configure et synchronise la chaîne de transmission en fonction des données émises par le premier ARM7.

La partie du système à concevoir était donc composée de deux processeurs et d'un bloc matériel implémentant la chaîne de transmission.

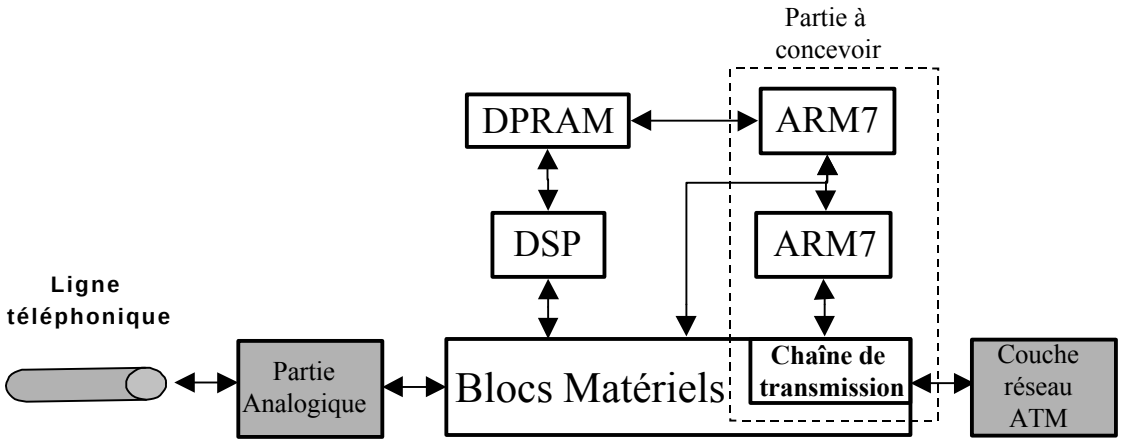


**Figure V. 1.** Architecture VDSL de ST.

## *V.3. Description du sous-ensemble de test*

Dans l'objectif d'évaluer le flot de conception SLS, nous avons voulu développer une application inspirée de l'application VDSL. Le but était de conserver l'ensemble de deux processeurs et d'un bloc communicant, d'essayer de mimer le fonctionnement d'origine tout en constituant un véritable test pour le flot.

Nous allons vous présenter la structure et le partitionnement logiciel/matériel du sous-ensemble de test ainsi que le comportement des tâches logicielles.

### *V.3.1. Structure et partitionnement.*

La Figure V. 2 représente une description du sous-ensemble réalisé. Le système est modélisé par un assemblage de modules, fils et ports.

Les deux premiers modules (M1, M2) représentés par de grands rectangles correspondent aux processeurs et le troisième (M3) à la chaîne de transmission. Chaque module processeur comporte un ensemble de tâches communicantes selon des protocoles de communication spécifiés.

Le processeur M2 envoie des paramètres pour configurer et synchroniser l'IP (M3). Il est aussi chargé d'envoyer périodiquement des données permettant au modem, avec lequel communique le système, de se synchroniser.

Les tâches sont représentées par des cercles, leur communication est modélisée par des ports et des nets respectivement représentés par des petits carrés dans lequel on indique le sens de transfert des données (I : Input, O : Output etc…) et des traits.

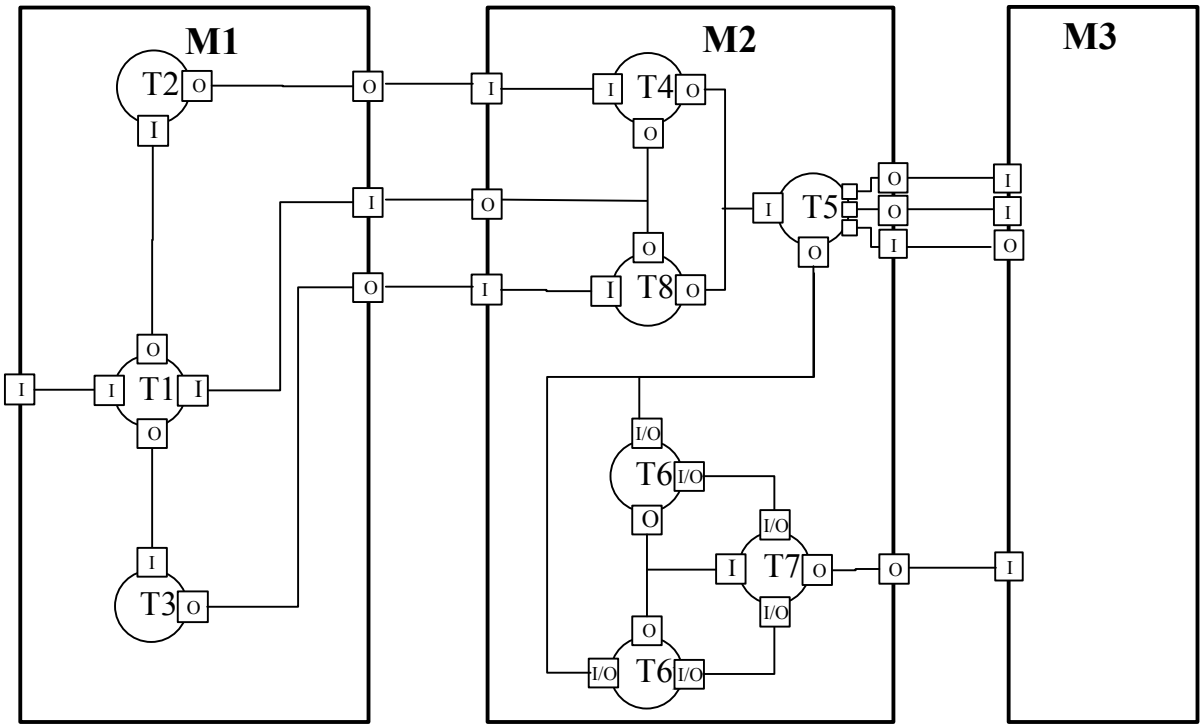


**Figure V. 2.** Description de la structure du sous-ensemble de test.

## *V.3.2. Représentation en Vadel du VDSL*

Pour pouvoir cosimuler l'ensemble du système, les deux modules M1, M2 et l'IP M3 ont été encapsulés chacun dans une enveloppe de simulation. L'enveloppe permet d'adapter l'interface de chaque module à celles des autres modules avec lesquels il communique.

On a obtenu ainsi un système composé de trois modules virtuels (un module virtuel désigne le module du système initial avec l'enveloppe dans laquelle il est encapsulé). Ces modules virtuels sont reliés par des canaux abstraits véhiculant les données et les signaux de synchronisation entre les modules (Figure V. 3).

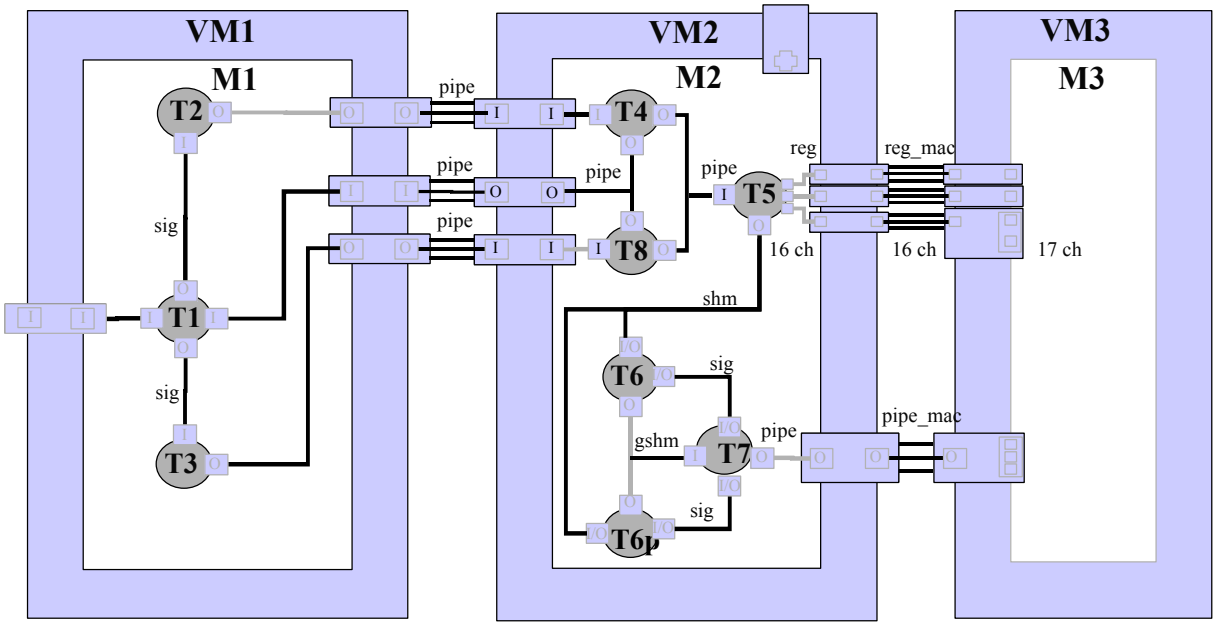


**Figure V. 3.** Description de l'application au niveau système en vue d'une cosimulation.

### *V.3.3. Description du comportement des tâches.*

Les entrées/sorties (interface) et les actions réalisées par chaque tâche appartenant aux modules M1 et M2 sont résumées dans les tableaux ci-dessous.

**Le module M1 :**

| | Entrées | Sorties | Actions |
|---|---|---|---|
| **TACHE 1** | - Un signal de l'extérieur.<br>- Un compte rendu de T4 et de T8. | - Un ordre de génération de configuration à T2 (signal).<br>- Un signal de RESET à T3. | - Envoi un signal de RESET à T3.<br>- Si réception d'un signal de l'extérieur, envoi d'un signal à T2.<br>- Si réception d'un signal de l'extérieur, envoi d'un signal à T3. |
| **TACHE 2** | - Un ordre de génération de configuration de T1. | - Un numéro de configuration à T4. | - Si réception d'un signal :<br>- Génération aléatoire d'un numéro entre 1 et 5.<br>- Envoi du numéro à T4. |
| **TACHE 3** | - Un signal de RESET. | - Le numéro de configuration d'initialisation. | - Si réception d'un signal de RESET :<br>- Envoi du numéro de configuration d'initialisation. |

**Tableau V. 1.** Comportement des tâches composants le module M1.

**Le module M2 :**

| | Entrées | Sorties | Actions |
|---|---|---|---|
| **TACHE 4** | - Un numéro de configuration de T2. | - Un numéro de configuration à T5.<br>s de la méthodologie présentée dans ce | - Si réception d'un nouveau numéro de configuration :<br>- Envoi du numéro de configuration à T5.<br>- Envoi du numéro de configuration à T1. |
| **TACHE 5** | - Un numéro de configuration de T4 ou T8. | - Des paramètres de configuration vers les registres de M3. | - Si réception d'un nouveau numéro de configuration :<br>- Demander au DATA PATH de passer à l'état V_INIT (écrire dans le registre TCS).<br>- Attendre le passage à l'état V_INIT (= attendre que TXIN = V_INIT)<br>- Attendre 10ms.<br>- Ecriture des paramètres dans le registre.<br>- Attendre le passage à l'état V_O/R_DATA (TXIN = O/R_DATA).<br>- Ecrire 'default' dans le TCS.<br>- Donner la valeur 2 dans la shm de synchro. |
| **TACHE 6** | - Une indication de lecture mémoire effectuée par T7 (signal). | - Les voc bytes de synchronisation dans la mémoire partagée. | - Si T7 n'est pas en train de lire :<br>- Si la shm de synchro vaut 1 ou 2 :<br>- Initialisation des messages de synchronisation.<br>- Décrémentation dans la shm de synchro de 1.<br>- Sinon :<br>- Ecriture des messages de synchronisation dans la mémoire partagée. |
| **TACHE 6P** | - Une indication de lecture mémoire effectuée par T4 (signal). | - Les voc bytes de sécurité dans la mémoire partagée. | - Si T7 n'est pas en train de lire :<br>- Si la shm de synchro vaut 1 ou 2 :<br>- Initialisation des messages de sécurité.<br>- Décrémentation dans la shm de 1.<br>- Sinon :<br>- Ecriture des messages de sécurité dans la mémoire partagée. |
| **TACHE 7** | - Les voc bytes de sécurités et de synchronisation par la mémoire partagée. | - Une indication de lecture dans la mémoire effectuée à T6 et T6'.<br>- Les voc bytes dans la FIFO de M3. | - Lecture de voc bytes dans la mémoire partagée.<br>- Indique à T6 et T6' que la lecture à été effectuée.<br>- Ecritures des voc bytes dans la FIFO de M3. |
| **TACHE 8** | - Le numéro de configuration d'initialisation de T3. | - Le numéro de configuration d'initialisation à T5.<br>- Un compte rendu de l'état T1. | - Si réception d'un nouveau numéro de configuration :<br>- Envoi de ce numéro à T5.<br>- Envoi du numéro à T1. |

**Tableau V. 2.** Comportement des tâches composants le module M2.

### *V.3.4. Variables partagées de l'application*

Le sous-ensemble du VDSL est composé de trois modules dont un IP (M3). Nous ne disposons pas du code décrivant le comportement de ce dernier et nous n'avons accès qu'à son interface de communication avec l'extérieur (les autres modules). C'est pour cette raison que nous ne considérons, dans l'étape d'allocation mémoire, aucune donnée lue ou écrite par le module M3.

En ce qui concerne les deux autres modules M1 et M2, ils ne comportent pas de variables globales. Ainsi on ne distingue que des variables de communication (entre les tâches d'un même module ou entre les deux modules) dans ce sous-ensemble du VDSL. Les noms de ces variables de communication, leur type et l'ensemble des tâches qui accèdent en lecture et/ou en écriture sont données dans le Tableau V. 3.

| Nom | Lecture | Ecriture | Type |
|---|---|---|---|
| D1 | T4 | T2 | Longint |
| D2 | T1 | T4, T8 | Longint |
| D3 | T8 | T3 | Longint |
| D4 | T5 | T4, T8 | Longint |
| D5 | T7 | T6, T6P | Longint |
| D6 | T6, T6P | T6, T6P, T5 | Longint |

**Tableau V. 3.** Variables de communication de l'application et leur utilisation en écriture/lecture.

Dans le Tableau V. 3, nous constatons une communication spéciale entre T5, T6 et T6P (D6) et entre T7, T6 et T6P (D5). C'est une communication par mémoire partagée logicielle (Figure V. 4), à ne pas confondre avec l'architecture mémoire partagée (matérielle). En effet, il s'agit d'un protocole, décrit avec des fonctions de niveau système qui seront interprétées par le système d'exploitation spécifique, qui sera généré à l'étape du ciblage logiciel (voir chapitre II) pour permettre aux tâches d'accéder aux données dans la mémoire locale du processeur sur lequel tourneront T4, T8, T5, T6, T6P et T7.

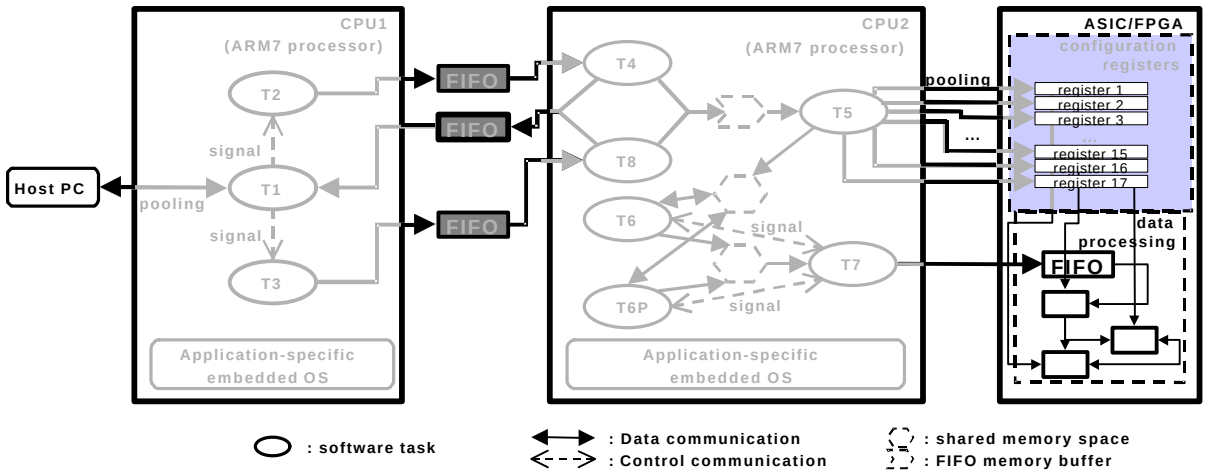


**Figure V. 4.** Communication des données dans le sous-ensemble du VDSL.

### *V.3.5. Simulation de niveau système*

La simulation de niveau système (voir chapitre III) nous permet de suivre les flots de données dans le système et de voir ainsi les différents accès aux données par les processus (tâches). Cela consiste, pour chaque donnée partagée (communication), à observer les tâches qui la partagent puis à noter le taux d'accès en lecture et/ou en écriture à la donnée (Tableau V. 4).

| | **Lecture** | | **Ecriture** | | |
|---|---|---|---|---|---|
| **D1** | 1 | | 1 | | |
| **D2** | 1 | | 0.51 | 0.49 | |
| **D3** | 1 | | 1 | | |
| **D4** | 1 | | 0.49 | 0.51 | |
| **D5** | 1 | | 0.5 | 0.5 | |
| **D6** | 0.5 | 0.5 | 0.45 | 0.27 | 0.28 |

**Tableau V. 4.** Taux d'accès en lecture/ écriture par les tâches aux données partagées de l'application.

### *V.3.6. Programme linéaire en nombres entiers*

Les éléments composant le programme linéaire en nombres entiers d'allocation mémoire sont présentés ci-dessous.

#### V.3.6.1. Variables de décision

Comme le module M3 est un IP, nous n'avons pas accès au code décrivant son comportement, et donc nous ne pouvant pas envisager une quelconque modification de ce module.

Ainsi, l'architecture mémoire la plus générale que l'on peut envisager pour cette application est composée d'une mémoire globale partagée, une mémoire locale distribuée pour chacun des modules M1 et M2 et deux mémoires locales privées.

Nous ne considererons que le cas des mémoires globales partagées et locales privées. En effet, le cas des mémoires locales distribuées ne peut etre traité actuellement pour des raisons de disponibilité d'outils de ciblage.

Donc, les variables de décision utilisées dans le programme linéaire en nombres entiers d'allocation mémoire sont :

$$Y1 = \begin{cases} 1 \text{ si la mémoire locale privée associée au module M1 fait partie de l'architecture.} \\ 0 \text{ sinon.} \end{cases}$$

$$Y2 = \begin{cases} 1 \text{ si la mémoire locale privée associée au module M2 fait partie de l'architecture.} \\ 0 \text{ sinon.} \end{cases}$$

$$Y3 = \begin{cases} 1 \text{ si la mémoire globale partagée fait partie de l'architecture.} \\ 0 \text{ sinon.} \end{cases}$$

TM1 = taille de la mémoire locale privée associée à M1 (ML1).
TM2 = taille de la mémoire locale privée associée à M2 (ML2).
TM3 = taille de la mémoire globale partagée (MGP).

X11 = 1 si D1 est affecté à ML1, 0 sinon.
X12 = 1 si D1 est affecté à ML2, 0 sinon.
X13 = 1 si D1 est affecté à MGP, 0 sinon.
X21 = 1 si D2 est affecté à ML1, 0 sinon.
X22 = 1 si D2 est affecté à ML2, 0 sinon.
X23 = 1 si D2 est affecté à MGP, 0 sinon.
X31 = 1 si D3 est affecté à ML1, 0 sinon.
X32 = 1 si D3 est affecté à ML2, 0 sinon.
X33 = 1 si D3 est affecté à MGP, 0 sinon.
X42 = 1 si D4 est affecté à ML2, 0 sinon.
X43 = 1 si D4 est affecté à MGP, 0 sinon.
X52 = 1 si D5 est affecté à ML2, 0 sinon.
X53 = 1 si D5 est affecté à MGP, 0 sinon.
X62 = 1 si D6 est affecté à ML2, 0 sinon.
X63 = 1 si D6 est affecté à MGP, 0 sinon.

### V.3.6.2. Hypothèses

Au niveau architecture, on ne considère pas la notion de temps dans la spécification du système. Ainsi une tâche peut accéder instantanément à une donnée qui réside dans la mémoire locale du module contenant cette tâche. Donc si l'on considère que le temps d'accès par une tâche à une donnée locale représente l'unité, le temps de lecture, par une tâche, d'une donnée résidante dans la mémoire locale associée à un module diffèrent de celui où est définie la tâche correspond à :

- Envoi d'une requête par le processeur où réside la tâche,
- Lecture de la donnée par le processeur lointain,
- Envoi d'un signal au processeur qui a demandé la donnée,
- Envoi de la donnée.

A ce niveau, on considère que chacune des opérations ci-dessus est réalisée en une unité de temps, on obtient donc un temps d'accès global de quatre unités.

Pour ce qui est d'un accès d'une tâche à une donnée résidant dans la mémoire globale partagée : comme cette dernière est un module esclave dans le système, on considère que la lecture s'effectue en deux temps (requête par le processeur et réception de la donnée).

On suppose que l'on utilisera le même type de mémoires dans le système, c'est-à-dire de même caractéristiques (SRAM, DRAM, SDRAM…) et du même constructeur, ce qui fait que le coût unitaire des mémoires est identique. Donc, dans la fonction objectif, on pondère les variables de type TM par le coefficient un.

Finalement, pour le coût de l'intégration des mémoires dans le système, on considère à ce niveau d'abstraction qu'il est le même pour les mémoires locales ou la mémoire globale.

### V.3.6.2. Objectif

L'objectif étant de minimiser simultanément le temps global d'accès aux données partagées de l'application et le coût de l'architecture mémoire, on a spécifié la fonction objectif du modèle linéaire en considérant non pas les nombres d'accès aux données partagées et le coût de l'architecture mémoire, mais les taux d'utilisation des données partagées par les tâches et les rapports mutuels des coûts des différents types de mémoires (ce qui revient au même du point de vue optimisation).

Ainsi on obtient, pour le sous-ensemble du VDSL, la fonction objectif suivante :

Min F = 4 X11 + X12 + 2 X13 + X21 + 4 X22 + 2 X23 + X31 + X32 + 2 X33 + X42 + 2 X43 + X52 + 2 X53 + X62 + 2 X63 +

X11 + 4 X12 + 2 X13 + 4 X21 + X22 + 2 X23 + 4 X31 + X32 + 2 X33 + X42 + 2 X43 + X52 + 2 X53 + X62 + 2 X63 +

TM1 + TM2 + TM3 + Y1 + Y2 + Y3

### V.3.6.3. Contraintes

Les contraintes associées aux variables de décision définies ci-dessus sont les suivantes :

- Tailles des mémoires :

32 X11 + 32 X21 + 32 X31 <= TM1

32 X12 + 32 X22 + 32 X32 + 32 X42 + 32 X52 + 32 X62 <= TM2
32 X13 + 32 X23 + 32 X33 +32 X43 + 32 X53 + 32 X63 <= TM3

- Cohérence du modèle :
TM1 <= 10000 Y1
TM2 <= 10000 Y2
TM3 <= 10000 Y3
Où on a pris A = 10000 (voir chapitre IV).

- Unicité de l'affectation des données (une donnée est affectée à une et une seule mémoire)
X11 + X12 + X13 = 1
X21 + X22 + X23 = 1
X31 + X32 + X33 = 1
X42 + X43 = 1
X52 + X33 = 1
X62 + X63 = 1

#### V.3.6.4. Résultats

Après résolution du programme linéaire en nombres entiers on obtient :
Y1 = Y2 = Y3 = 1.
X13 = X23 = X33 = X42 = X52 = X62= 1.

Toutes les autres variables de type Xij sont égales à zéro. Ainsi, l'allocation mémoire obtenue pour les données partagées considérées dans cette application est que le VDSL requière une mémoire globale partagée. Les trois variables de communication entre les modules M1 et M2 sont affectées à cette mémoire globale partagée. Cependant les autres données partagées considérées dans le modèle resteront dans la mémoire locale de M2 (ce qui est tout a fait prévisible). Ainsi le système final obtenu est représenté dans la Figure V. 5.

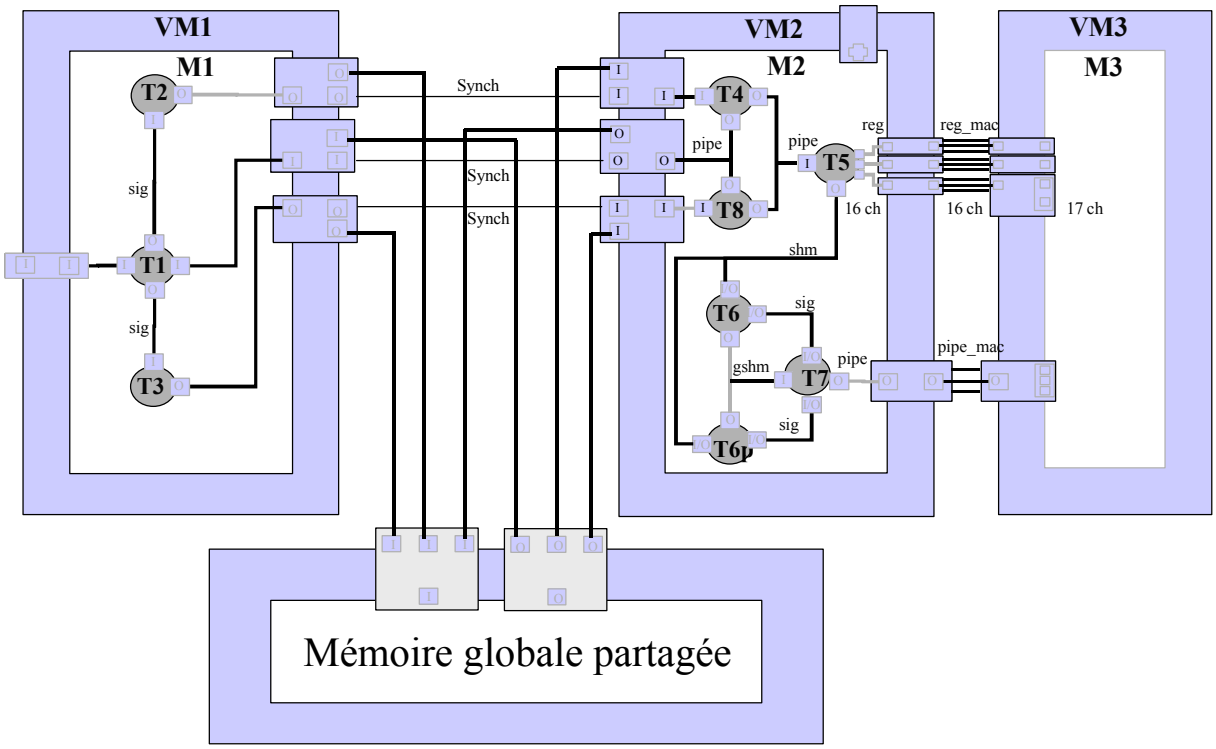


**Figure V. 5.** Description de l'application au niveau architecture.

## V.4. *Transformation de code*

Cette étape consiste à remplacer les accès en lecture/écriture aux variables partagées dans le code comportemental des modules M1 et M2 par des accès explicites à la mémoire globale partagée.

Les accès à ces données de l'application représentent environ 14,13% du code décrivant le comportement des tâches contenues dans les modules M1 et M2 au niveau architecture. Cette étape de notre flot de conception et pénible et délicate car elle nécessite l'intervention manuelle sur l'ensemble du code de l'application. Elle est actuellement en cours d'automatisation.

## V.5. *Autres Architectures mémoire pour le VDSL*

Dans cette section, nous présentons les principales architectures mémoire possibles pour l'application du VDSL, autres que celle trouvée en utilisant le modèle d'allocation mémoire dans la section précédente. Nous discutons les particularités de chaque architecture, ses avantages et inconvénients ainsi que les possibilités de son implémentation au niveau micro-architecture.

### *V.5.1. Architecture 1 : mémoires locales*

L'architecture mémoire la plus triviale que l'on puisse imaginer pour le VDSL, (ainsi que pour toute autre application), n'est constituée que de mémoires locales privées (une avec chaque processeur). Cette architecture est représentée dans la Figure V. 6.

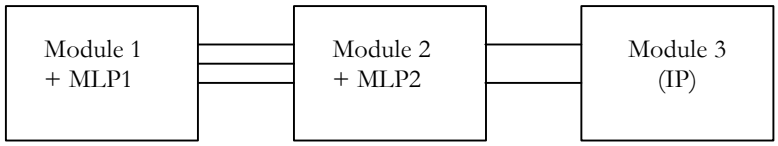


**Figure V. 6.** Le VDSL au niveau architecture avec uniquement des mémoires locales privées

Le choix d'une telle architecture mémoire pour l'application du VDSL, revient à choisir de raffiner la spécification du niveau système au niveau architecture sans ajouter aucun nouveau module à la description initiale. En effet, initialement les données de l'application résident déjà implicitement dans les mémoires locales aux processeurs.

Cette architecture mémoire n'implique aucune transformation de code particulière à part les transformations ordinaires liées au passage vers un niveau d'abstraction plus bas, comme le remplacement des primitives de type send/receive par des primitives de type write/read. Ceci permet évidemment d'obtenir un code applicatif de taille relativement réduite. En effet, dans ce cas on obtient respectivement : 968 lignes de code SystemC pour l'application et un système d'exploitation de 1484 octets, 1872 lignes en SystemC pour l'application et 2624 octets pour le système d'exploitation, respectivement pour les modules 1 et 2.

En affectant toutes les données partagées de l'application du VDSL dans les mémoires locales privées aux processeurs (X1, X2, X3 à la mémoire locale privée du module 1 « ML1 » et X4, X5, X6 à la mémoire locale privée du module 2 « ML2 »), la fonction objectif considérée dans le modèle PLNE d'allocation mémoire augmente d'un rapport de 4.23%. Ce rapport est relatif au coût de l'architecture mémoire ainsi qu'au temps d'accès global aux données partagées considérées dans le modèle. Sachant que nous avons pris en compte dans la fonction objectif des coefficients approximatifs et abstraits ne tenant pas compte des temps d'accès réels par les processeurs aux mémoires (notamment les cycles liés au système d'exploitation et aux adaptateurs mémoire), ce taux d'accroissement de la fonction objectif ne peut qu'augmenter considérablement en considérant le VDSL au niveau micro-architecture (RTL).

### *V.5.2. Architecture 2 : mémoire distribuée*

La seconde architecture mémoire que l'on peut imaginer pour l'application VDSL, est celle constituée de mémoires locales privées (comme dans l'architecture 1), et d'une mémoire locale distribuée pour chaque processeur (directement accessible par l'autre processeur).
La spécification du VDSL au niveau architecture obtenue avec cette architecture mémoire est représentée sur la Figure V. 7.

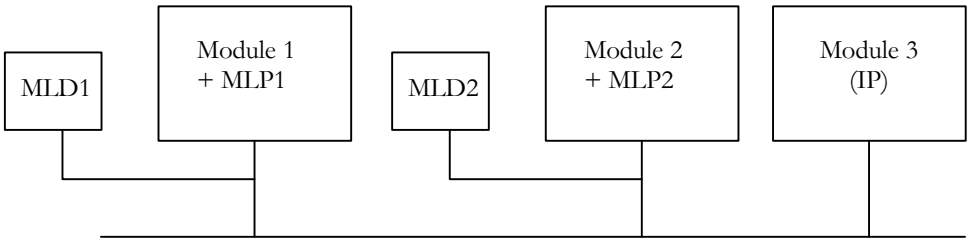


**Figure (a)**

**Figure V. 7.** Le VDSL au niveau architecture avec deux mémoires distribuées

Dans cette architecture les mémoires locales distribuées sont représentées chacune par un module mémoire fonctionnel comme dans le cas d'une mémoire globale partagée (chapitre IV).
Avec le choix de cette architecture 2, les données affectées à la mémoire locale distribuée du module i ne nécessitent pas de transformations dans codes des tâches quand elles sont accédées par ce même module i. Cependant dans le cas d'un accès par un autre module, elles nécessitent des transformations de code pouvant être très complexes dues à la distribution de la mémoire.

Dans les architectures du VDSL des figures (b) et (c) (Figure V. 8) nous n'avons intégré qu'un seul bloc mémoire distribué avec l'un des deux modules, cela simplifie considérablement les transformations des codes des taches impliquées car, à part les temps d'accès, cette mémoire peut être considérée exactement comme une mémoire globale partagée si elle est unique, et donc ne risque pas de générer des problèmes de conflits d'adressage liés à la distribution des données. Ainsi, les transformations de code devant être réalisées sont semblables à celles concernant la mémoire globale partagée présentée dans le chapitre IV. Ceci fait que le choix de l'une de ces deux architectures donne une taille de code de l'application identique à celle que l'on obtient en choisissant une architecture mémoire globale partagée. Le coût de l'architecture reste aussi très comparable vu qu'il s'agit toujours d'insérer un module mémoire partagée (globale ou locale à un module) dans l'architecture. La seule différence entre ces deux architectures et celle avec mémoire globale partagée est l'accroissement de la fonction objectif due à l'augmentation du temps d'accès global des modules aux données partagées.

Cependant, dans le cas de l'architecture (a), on distingue deux blocs mémoire distribués (un bloc avec chaque module). Cela rend l'architecture mémoire du VDSL réellement distribuée (dans le sens des multiprocesseurs classiques), donc les messages d'accès aux données dans les mémoires deviennent sensiblement plus complexes par rapport à ceux des architectures (b) et (c).

En effet, dans ce cas, deux données distinctes et ayant deux adresses physiques différentes peuvent avoir la même adresse logique ! Ceci implique au niveau architecture d'envisager une table d'allocation mémoire abstraite, contenant les emplacements des données dans les blocs mémoire abstraits, pour chacune des mémoires distribuées. Ainsi, en affectant par exemple la donnée X1 à la mémoire locale distribuée du module 1 dans l'architecture (a), un accès en lecture à cette donnée par le module 2 du type « receive (X1) » au niveau système doit être transformé, au niveau architecture, en un message d'accès semblable au cas d'une mémoire globale partagée, dans lequel on précise en

plus de quelle mémoire il s'agit. Le message sera donc du type : read (X1, **MLD1**). En précisant le nom de la mémoire, on sélectionne dans la table d'allocation mémoire, la mémoire concernée (MLD1).

En affectant par exemple les données X1, X2 et X3 à la mémoire locale distribuée du module 1 «MLD1», et les données X4, X5 et X6 à «MLD2», la fonction objectif du programme linéaire en nombres entiers d'allocation mémoire augmente d'un rapport de 8.13%.

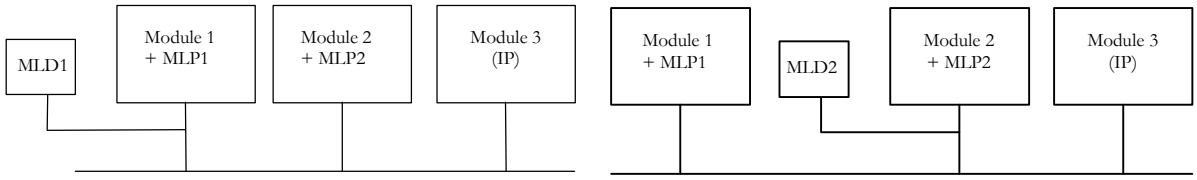


**Figure (b)** **Figure (c)**

**Figure V. 8.** Le VDSL au niveau architecture avec une mémoire locale distribuée.

## *V.5.3. Architecture 3 : mémoire distribuée partagée*

La troisième architecture mémoire que l'on peut envisager pour l'application du VDSL est l'architecture mémoire distribuée partagée, contenant en plus une mémoire globale partagée (Figure V. 9). Cette architecture est certainement la plus générale et la plus complexe parmi celles présentées dans ce chapitre.

Les architectures des figures (a), (b) et (c) contiennent respectivement en plus des mémoires locales privées aux modules : une mémoire locale distribuée et une mémoire globale partagée (figure (a)), une mémoire locale distribuée et une mémoire globale partagée (figure (b)), deux mémoires locales distribuées et une mémoire globale partagée (figure (c)). Ces architectures sont au moins aussi complexes que l'architecture avec deux mémoires locales distribuées décrite dans la section précédente et posent donc le même problème dû à la distribution des données de l'application.

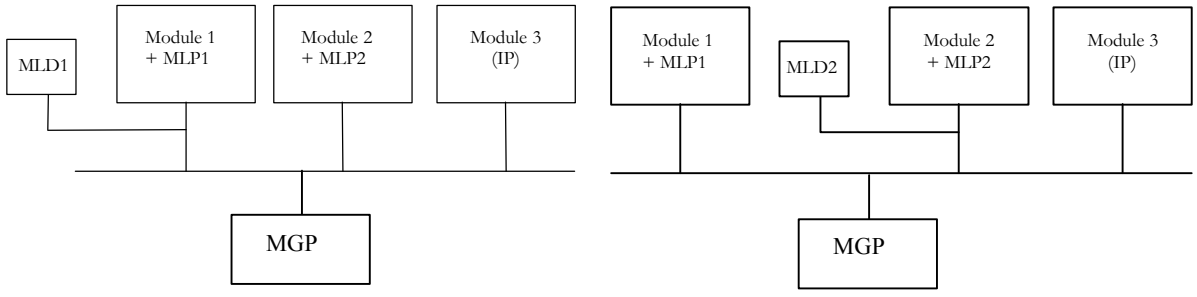


**Figure (a)** **Figure (b)**

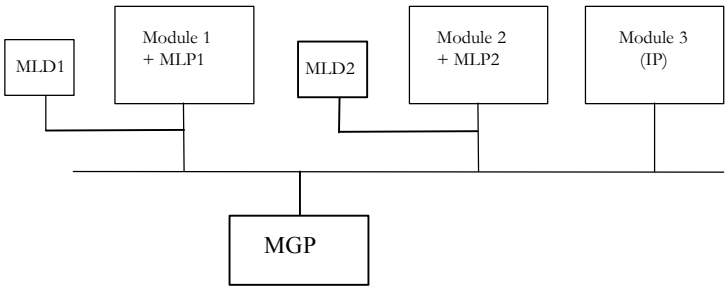


**Figure (c)**

**Figure V. 9.** Le VDSL au niveau architecture avec une architecture mémoire distribuée partagée.

Pour cette application, le nombre de données traitées ne nécessite pas l'intégration de mémoires partagées distribuées et une mémoire globale partagée à la fois. En utilisant le même programme linéaire d'allocation toutes les données de l'application seront bien sûr affectées seulement à la mémoire globale partagée sauf dans le cas où l'inéquation suivante serait valide :

$T\_read(i, m) + T\_write(i, m) \geq T\_read(i, k) + T\_write(i, k)$

Où :

m : représente l'indice de la mémoire globale partagée et k celui de l'une ou l'autre mémoire locale distribuée.

Ceci n'est pas possible dans l'application du VDSL vue l'uniformité des accès vers les données partagées par les modules. En effet les données présentes dans le module 1 sont lues trop souvent par le module 2 pour que l'inéquation ci-dessus soit vérifiée.

Mais juste à titre comparatif, on constate qu'en taille de code, les architectures (a) et (b) de la figure 4 augmentent la taille du code de l'application d'au moins 200 lignes au niveau architecture et du double dans le cas de l'architecture (c), par rapport à l'architecture avec seulement une mémoire globale partagée, dues essentiellement à l'intégration des modules mémoires.

Cette architecture mémoire ne doit être envisagée que pour une architecture contenant plus de modules que le VDSL, et si les données ne sont pas échangées d'une façon uniforme entre les modules.

## *V.6. Comparaison entre les différentes architectures mémoire du VDSL*

On peut comparer les différentes architectures mémoire possibles pour l'application du VDSL ainsi que l'architecture mémoire obtenue pour la même application avec le modèle ILP d'allocation, sur plusieurs points. En effet, sans prétendre être exhaustif, on peut distinguer principalement les cinq critères de comparaison suivants :

- La fonction objectif du modèle ILP de l'allocation mémoire : ce critère, même ne donnant pas des valeurs exactes concernant le coût de l'architecture et le temps d'accès aux données partagées (performance) à cause de l'estimation grossière à ce niveau d'abstraction des

paramètres de l'ILP, permet d'avoir une bonne vision sur l'évolution globale de ces deux critères (coût et temps) dans les différentes architectures mémoire.

- Transformations de code : le nombre et la difficulté des transformations du code de l'application et de la structure de l'architecture du système, relatives à l'intégration d'une architecture mémoire donnée est un critère important de comparaison. En effet, ces transformations peuvent être des plus triviales dans certaines architectures mémoire et très complexes dans d'autres.
- Flexibilité : ce critère nous donne une idée sur la facilité d'apporter des modifications à l'architecture du VDSL dans le cas où l'on envisagerait une modification de la fonctionnalité de l'application initiale.
- Implémentation au niveau micro-architecture : l'implémentation au niveau micro-architecture de chacune des architectures du VDSL décrites dans la section précédente implique, entre autres, la disponibilité de certains services dans le système d'exploitation spécifique, la prise en compte de messages complexes par les adaptateurs des différents modules de l'architecture,…, etc. Ces besoins, plus au moins spécifiques à chacune des architectures mémoires, peuvent être très complexes à implémenter et à intégrer dans l'outil global de conception.

Ainsi, concernant le premier critère (la fonction objectif du modèle d'allocation mémoire), l'architecture avec une mémoire globale partagée est évidemment la meilleure car elle optimise la fonction. Les architectures à mémoire distribuée partagée sont les moins bonnes pour ce critère car elles augmentent non seulement le temps global d'accès aux données partagées mais aussi augmentent considérablement le coût global de l'architecture. Concernant les transformations de code relatives au choix de l'architecture mémoire, l'architecture mémoire avec uniquement des mémoires locales privées aux modules est bien sûr la meilleure puisqu'elle n'implique quasiment aucune transformation de code, les architectures mémoire distribuée partagée, ainsi que l'architecture (a) de la Figure V. 7 sont les moins bonnes et cela est essentiellement dû à la distribution des données de l'application sur plusieurs mémoires partagées. La flexibilité de l'architecture est bonne dans le cas de l'architecture 1 et l'architecture (a) de la Figure 7, ceci étant dû à l'absence d'une mémoire globale partagée. Enfin, l'architecture avec uniquement des mémoires locales privées est bien sûr la plus facile à implémenter au niveau micro-architecture. Les architectures avec une mémoire distribuée partagée et celles avec plusieurs mémoires distribuées sont les plus difficiles à implémenter car elles nécessitent la mise à jour des outils de ciblage logiciel/matériel pour permettre la gestion de la distribution physique des données partagées de l'application.

## *V.7. Critique de l'application et de sa réalisation*

L'état actuel des outils d'aide à la conception ne permettent pas d'aller automatiquement vers une réalisation RTL.

La transformation du modèle SystemC de simulation de l'application au niveau architecture vers un modèle architecture avec mémoires est partiellement réalisé, et fonctionne correctement dans le cas d'une mémoire globale partagée. La transformation du code de l'application résultant de ce choix d'architecture mémoire est aussi opérationnel.

L'étape de synthèse mémoire (choix du type des mémoires, tailles,…, etc) n'est pas intégrée et reste entièrement manuelle. Et les outils de ciblage logiciel/matériel ne prennent pas encore en compte les différentes architectures mémoires, ce qui empêche encore de prendre les valeurs temporelles réelles dans le modèle d'allocation mémoire.

Les données considérées dans l'application sont de même type (taille), ceci est fixé dans la spécification initiale de l'application. Cependant, des changements des tailles de ces données (en tableaux par exemple) ne change pas le comportement du modèle d'allocation. Ceci est dû au fait que les tailles des variables n'apparaissent pas dans la fonction objectif.

## *V.8. Conclusion*

Un sous-ensemble de l'application du VDSL a été présenté dans ce chapitre pour illustrer la conception de l'architecture mémoire spécifique et particulièrement l'étape de l'allocation mémoire.

Le modèle linéaire d'allocation mémoire spécifique à l'application a été généré et résolu en utilisant les bibliothèques Cplex C++ de Ilog. La résolution du modèle a été particulièrement rapide (28 secondes en explorant 48 nœuds dans l'arbre des solutions) montrant ainsi son efficacité due à sa simplicité malgré le fait qu'il ne soit pas polynomial. Les transformations de code impliquées par l'intégration de la mémoire globale partagée dans le système ont été réalisées manuellement et concernent environ 14% du code comportemental des modules M1 et M2.

Nous ne pouvons pas générer le code de niveau micro-achitecture de cette application avec la mémoire globale partagée, car une telle architecture mémoire n'est pas encore supportée par les outils de ciblage logiciel/materiel du groupe SLS (ceci étant en cours de réalisation).

L'architecture mémoire obtenue a été comparée aux autres architectures mémoire possible pour l'application du VDSL sur quatre critères.

La micro-architecture du VDSL sans mémoire partagée obtenue est representée dans la Figure V. 10. La Figure V. 11 schématise la micro-architecture ciblée par notre flot de conception mémoire pour cette application.

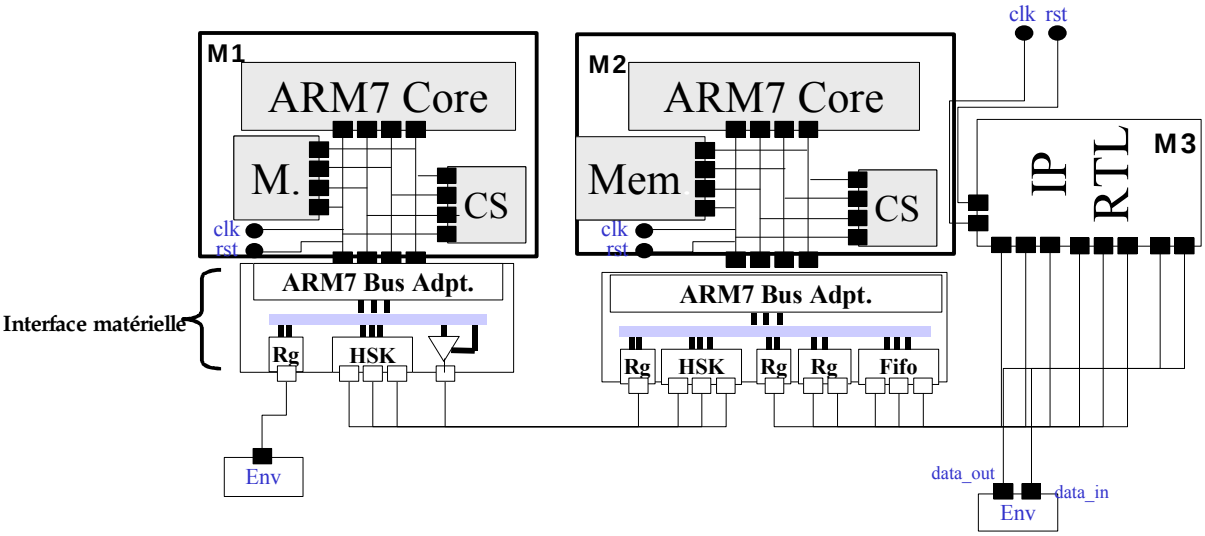


**Figure V. 10.** Architecture RTL du sous-ensemble du VDSL sans mémoire globale.

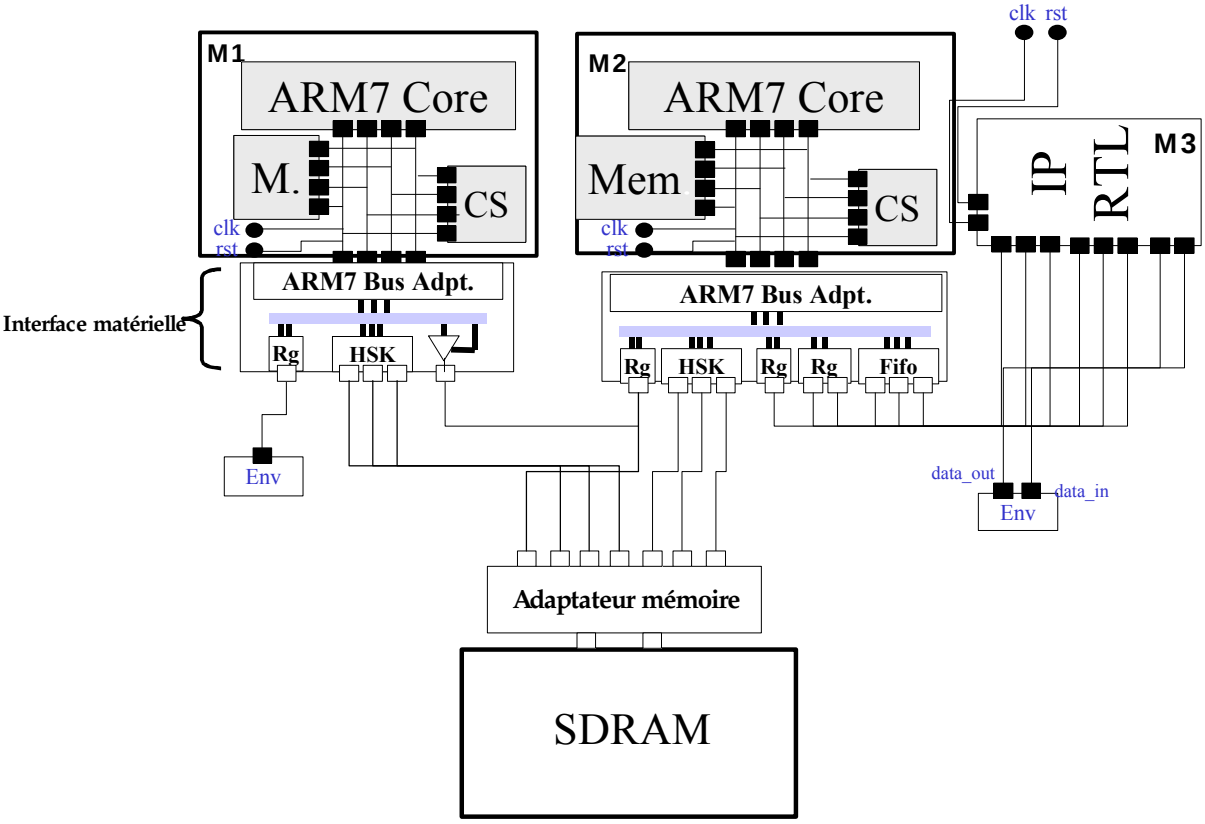


**Figure V. 11.** Architecture RTL du sous-ensemble du VDSL avec mémoire globale partagée.

# Chapitre VI

# CONCLUSION ET PERSPECTIVES

Une conclusion globale de ce mémoire est donnée dans la première partie de ce chapitre. La seconde partie présente quelques axes de réflexions qui constituent des perspectives visant à compléter le travail présenté dans cette thèse.

## *VI.1. Conclusion*

Les systèmes embarqués sont les parties électroniques qui prennent place progressivement dans les objets usuels allant des téléphones mobiles aux voitures.

Récemment la demande pour les systèmes embarqués et le nombre de fonctionnalités souhaitées ce sont fortement accrue tandis que les délais de conception requis diminuent. Des architectures multiprocesseurs hétérogènes semblent devenir la clé pour que les systèmes embarqués puissent supporter cette complexité. En parallèle, l'intégration a fait de grands progrès : il est maintenant possible d'intégrer sur une même puce plus de 100 millions de transistors. Il est dès lors possible d'intégrer complètement un système sur une seule puce.

La plupart de ces systèmes monopuce spécifiques à des applications modernes telles que le traitement d'image et les jeux, requièrent une grande capacité de mémoire. Ces dernières occuperont plus de 95% des surfaces des puces, à l'horizon 2014, selon des prévisions récentes (ITRS). Ainsi les performances de ces systèmes monopuce dépendent fortement de leur architecture mémoire.

Cependant, de nos jours les concepteurs ne disposent d'aucun outil permettant de concevoir et d'intégrer automatiquement des architectures mémoires sophistiquées (partagée centralisée, distribuée, distribuée partagée) dans les systèmes monopuce.

En effet, le concepteur d'aujourd'hui compte encore uniquement sur sa propre expérience et son effort manuel pour choisir des architectures mémoires, se résumant souvent à des mémoires locales à chaque processeur, pour les systèmes monopuce qu'il conçoit. Pour ces raisons, les concepteurs n'arrivent plus à concevoir de tels systèmes dans des délais raisonnables, car ils manquent de méthodologies et d'outils permettant la conception automatique de la partie la plus dominante dans les systèmes modernes, à savoir les mémoires.

Ce travail se situe donc dans le cadre de la conception de systèmes multiprocesseurs monopuce spécifiques à des applications modernes.

Ce mémoire propose une méthodologie complète et systématique de conception d'architectures mémoires sophistiquées et optimisées, spécifiques à une application donnée, à partir d'une spécification distribuée de haut niveau.

La méthodologie présentée consiste en un flot de conception d'architectures mémoires avec plusieurs étapes d'optimisations. Elle permet de réaliser certaines transformations de haut niveau sur le code de l'application afin de réduire le nombre d'accès aux données. Elle intègre aussi d'autres optimisations et transformations à des niveaux d'abstraction plus bas comme la transformation du code afin d'utiliser certains modes d'accès particuliers aux mémoires tels que l'accès en mode page, « bust »,...etc.

Le flot de conception proposé dispose d'un modèle d'allocation optimal. En effet, ce modèle basé sur la programmation linéaire en nombres entiers permet d'obtenir une architecture mémoire « sur mesure » à l'application en optimisant les critères du coût et du temps d'accès global aux données partagées de l'application. Aucune restriction n'est faite a priori sur le type de l'architecture mémoire pouvant être obtenue avec le modèle linéaire présenté. En effet, elle peut comporter des

mémoires locales privées aux processeurs, locales distribuées ainsi que des mémoires globales partagées.

La méthodologie comporte également une étape très importante qui consiste à intégrer l'architecture mémoire obtenue avec le modèle d'allocation au reste du système au niveau architecture, puis à transformer le code de l'application pour l'adapter à l'architecture mémoire. Le système obtenu à l'issue de cette étape comporte ainsi tous les modules finaux du système monopuce, y compris les modules mémoire. Ceci nous donne ainsi un modèle simulable du système à un niveau d'abstraction intermédiaire entre les niveaux système et micro-architecture (RTL). Ce modèle simulable permet de simuler et de valider le système dans son intégralité dans des temps très raisonnables, chose qui peut sans doute s'avérer très bénéfique du point de vue des délais de mise sur le marché des systèmes monopuce.

Les principaux avantages de la méthodologie présentée dans ce travail sont :

- Elévation du niveau d'abstraction pour la conception de l'architecture mémoire : en effet, le flot présenté prend en entrée une spécification distribuée, de niveau système, de l'application. Contrairement à la méthode classique consistant à intégrer les mémoires au reste du système au niveau micro-architecture, il propose et intègre l'architecture mémoire au système au niveau architecture.
- L'automatisation : les étapes constituant notre flot de conception sont systématiques et ne nécessitent que peu d'intervention manuelle du concepteur.
- Optimisation : en plus des différentes étapes d'optimisation proposées tout au long du flot de conception, le modèle d'allocation proposé est basé sur une technique déterministe et optimale qui est la programmation linéaire en nombres entiers.
- Liberté de choix : le flot proposé permet d'intégrer aux systèmes non seulement les mémoires locales comme c'est le cas dans les flots de conceptions actuels, mais aussi des architectures mémoires plus générales.

Ses limitations sont principalement :

- Modèle d'allocation non polynomial : en effet le modèle d'allocation proposé est basé sur la programmation linéaire en nombres entiers (mixtes) et donc ne peut pas être résolu en temps polynomial pour des grandes instances du problème d'allocation. C'est pour cette raison qu'il faut réfléchir sur d'autres méthodes pas forcement déterministes (stochastiques) pour des systèmes ayant un grand nombre de processeurs et beaucoup de données partagées. Quelques grandes lignes et réflexions sur un tel modèle sont données dans la section suivante de ce chapitre.
- Estimation de performances : dans la méthodologie présentée dans cette thèse nous n'avons pas encore inclus une estimation de performances. Ceci pourra faire l'objet de travaux futurs.

## *VI.2. Perspectives*

Une méthodologie de conception de l'architecture mémoire pour les systèmes multiprocesseurs monopuce spécifiques a été proposée au cours de cette thèse. Toutefois, dans le but d'optimiser encore plus cette méthodologie et l'étendre pour traiter d'autres problèmes de conception, certains axes de réflexion méritent d'être considérés. Sans prétendre être exhaustif, en voici une liste :

### *VI.2.1. Approche stochastique d'allocation mémoire*

Dans le cas d'applications faisant intervenir un grand nombre de processeurs qui manipulent beaucoup de variables, la modélisation du problème de l'allocation mémoire sous forme d'un programme linéaire en variables binaires est peu conseillée vu la nature très combinatoire de ce problème.

Comme alternative, nous proposons dans ce cas une approche stochastique basée sur des tests statistiques. Le principe de cette approche est le suivant :

ordonner les variables suivant le taux d'utilisation relatif des données par les processeurs :

$$C_{vj} = \frac{\text{Taux d'utilisation de la variable v par le processus j}}{\sum_{vi} \text{Taux d'utilisation de la variable v par le processus i}}$$

avec vi = ensemble des processus utilisant la variable v

a - Le critère $C_{vj}$ est proportionnel au taux d'utilisation de la variable v par le processus j. Après avoir calculé ce critère pour toutes les variables et tous les processus, pour un « v » fixe, on calcule l'écart-type par rapport à « j ». Ce dernier nous donne une idée sur la dispersion des critères calculés pour une variable donnée. Ainsi, s'il n' y a aucun $C_{vj}$ qui domine « statistiquement » pour un « v » fixe, on décide d'affecter cette variable à la mémoire globale.

On peut aussi utiliser en plus d'autres règles telles que :

b - Si une variable est volumineuse (tableau) et si elle est utilisée par plus d'un processus, alors elle sera en mémoire partagée.

c - Si le nombre d'accès à une variable (même de petite taille) en lecture et en écriture par 2 processus différents est élevé, alors elle sera en mémoire partagée.

### *VI.2.2. Affection mémoire*

L'affectation mémoire est une étape non moins importante que les autres étapes de conception d'une architecture mémoire. Elle consiste à attribuer des adresses physiques aux différentes données de l'application dans les mémoires allouées en tenant compte généralement de deux critères d'optimisation : la consommation en énergie et l'espace mémoire occupé. Afin d'illustrer ce problème d'affectation mémoire considérons l'exemple suivant :

Supposons que l'on dispose de cinq groupes de mots mémoire (des tableaux par exemple), A, B, C, D et E. La taille d'un mot de chaque groupe ainsi que le nombre de mots de chaque groupe sont donnés dans le Tableau VI. 1.

| Groupe | Longueur du mot | Nombre de mots |
|---|---|---|
| **A** | 24 | 1096 |
| **B** | 24 | 900 |
| **C** | 24 | 1100 |
| **D** | 32 | 100 |
| **E** | 24 | 256 |

**Tableau VI. 1.**Caracteristiques des groupes de données

On dispose de trois mémoires M1, M2 et M3, de tailles respectivement : 1096, 2192 et 1096 avec des mots de 32 bits.

Il est clair que l'on peut affecter les cinq groupes de données aux trois mémoires de différentes façons qui pouvant engendrer généralement des coûts très différents. Une première affectation consiste à mettre A dans M3, C, D et B dans M2 et E dans M1. Cette affectation est l'unique solution que l'on peut obtenir en considérant les groupes dans l'ordre A, C, D, B, E et les mémoires dans l'ordre M3, M2, M1. Ainsi avec cette solution les trois mémoires doivent être utilisées.

Par contre si on considère les groupes dans l'ordre A, C, E, D, B et les mémoires dans l'ordre M2, M1, M3 comme dans la seconde affectation de la Figure VI. 1, on obtient une affectation de toutes les données avec seulement les mémoires M1 et M2 et donc M3 est inutile dans le système !

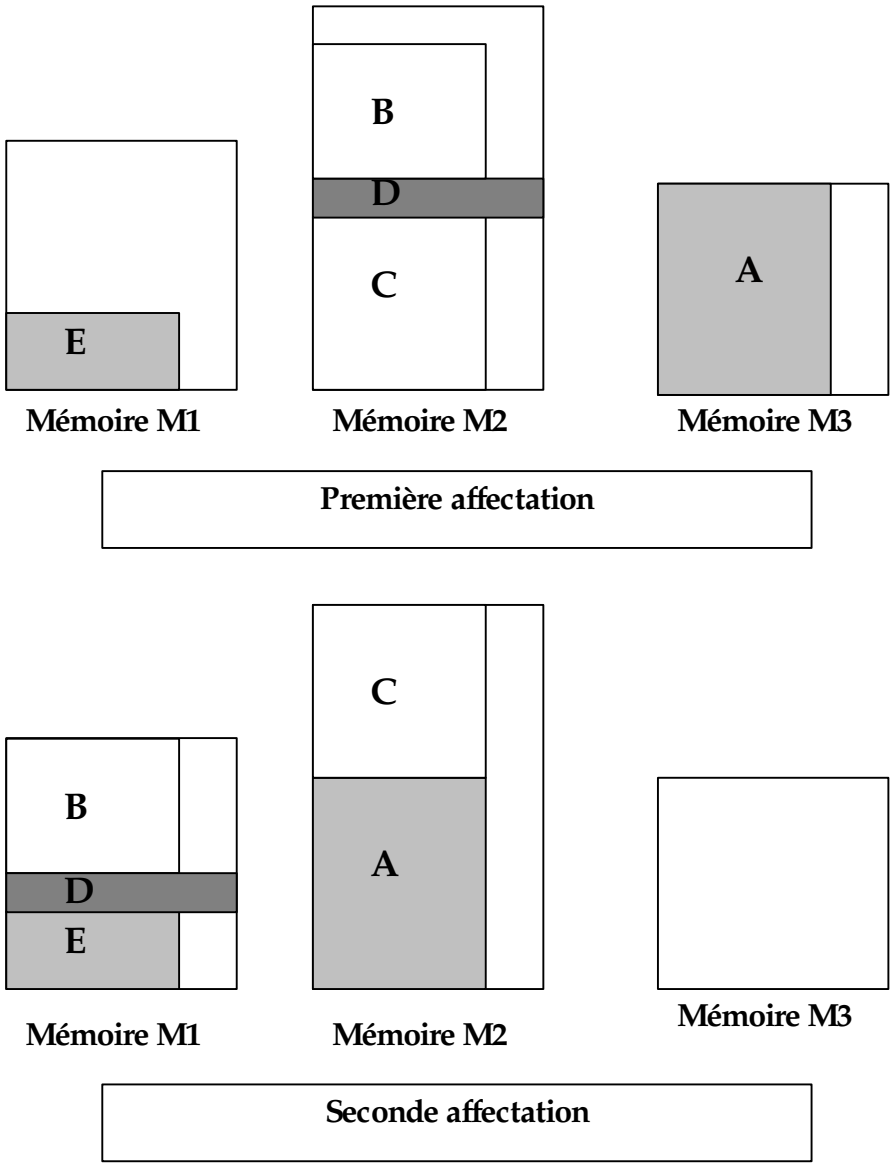


**Figure VI. 1.** Deux affectations differentes des données de l'exemple.

Pour résoudre ce problème d'affectation mémoire, les concepteurs d'aujourd'hui utilisent l'une des trois méthodes suivantes :

- Affectation manuelle : c'est le concepteur qui se charge de décider quelle donnée affecter à quelle mémoire en se basant uniquement sur son expérience.
- Algorithme de séparation et évaluation exhaustive : c'est une méthode d'exploration de toutes les solutions possibles ayant comme critères d'évaluation : la taille mémoire utilisée et une estimation de la consommation en énergie des mémoires après l'affectation de chaque nouvelle donnée (cette estimation est donnée par les constructeurs des mémoires). Ce type d'algorithme donne forcement l'affectation optimale, mais dès que les nombres de données et de mémoires deviennent important ces algorithmes ne convergent pas en un temps raisonnable. Ce sont des

algorithmes non polynomiaux qui ne peuvent pas être utilisés pour des grandes instances du problème d'affectation.

- Algorithmes gloutons : ce sont des algorithmes basés sur une technique très utilisée dans la résolution des problèmes dit « difficiles » en recherche opérationnelle. Ils convergent très rapidement vers une solution sans toute-fois garantir qu'il s'agit de la solution optimale ni même d'une « bonne » solution.

Pour résoudre efficacement ce problème d'affectation mémoire, deux axes peuvent être envisagés:

- Correspondance avec le problème classique du « Bin-Packing » (qui consiste à ranger des objets de tailles connues dans d'autres objets « containers » de tailles limitées en optimisant l'espace utilisé). Ce problème est aussi NP-Complet (difficile) mais étant un problème très classique, il existe dans la littérature de nombreux algorithmes permettant de lui trouver de « bonnes » solution dans des temps très raisonnables [Bou97], [Lab00].
- Séparation et évaluation avec utilisation de certaines heuristiques pour estimer les critères d'optimisation dans une solution finale à laquelle mènerait tout nœud dans l'arbre de solutions. Cette technique est très utilisée dans les problèmes d'ordonnancement, en la mixant avec des méthodes efficaces de parcours de l'arbre de solutions [Dup98], elle peut donner la solution optimale pour des instances « assez » importantes du problème en un temps raisonnable.

### *VI.2.3. Extension du modèle de programmation linéaire en nombre entiers pour le problème de la synthèse de la communication*

Un système multiprocesseur est décrit avec des modules communiquant via des canaux abstraits au niveau système. Le concepteur de tels systèmes dispose généralement de plusieurs unités de communication (FIFO par exemple) capables d'exécuter une même primitive de communication requise par une tâche dans l'un des modules du système.

Ainsi, l'allocation d'unités de communication prend en entrée un ensemble de processus communicants à travers des canaux abstraits et une bibliothèque d'unités de communication [Dav97]. Ces unités de communication sont une abstraction de composants physiques. Cette étape consiste à choisir dans la bibliothèque un ensemble d'unités de communication capables d'exécuter toutes les primitives de communication requises par les processus. Le choix d'une unité de communication particulière ne dépend pas seulement de la communication à exécuter (données à transférer) mais aussi des performances requises et de la réalisation des processus concernés. Cette étape fixe le protocole de chaque primitive de communication en choisissant une unité de communication avec un protocole défini pour chaque canal abstrait. Elle fixe aussi la topologie du

réseau d'interconnexion en déterminant le nombre d'unités de communication allouées et les canaux abstraits exécutés sur chacun d'entre eux.

Ce problème à une assez forte similitude avec le problème d'allocation des blocs mémoire, car il s'agit de sélectionner une unité pour réaliser une primitive (sélectionner un type de mémoire pour contenir une donnée partagée dans le problème d'allocation), chaque choix engendre un coût diffèrent et a des répercutions sur les performances du système final, comme dans le problème d'allocation. Il serait donc intéressant de réfléchir sur la possibilité d'étendre le modèle linéaire d'allocation mémoire pour modéliser le problème de la synthèse de la communication.

### *VI.2.4. Prise en comptes des caches*

Très souvent dans les systèmes multiprocesseurs on utilise plusieurs caches pour profiter de certaines caractéristiques de l'application considérée comme l'anticipation des accès à la mémoire. Tout au long de cette thèse on a considéré que les systèmes spécifiques à une seule application donc des systèmes où la mémoire et le réseau de communication sont fait sur mesure à l'application, par conséquent ils ne nécessitent pas l'introduction des caches. Par contre pour pouvoir étendre l'utilisation de la méthodologie présentée dans cette thèse pour la conception de systèmes multiprocesseurs monopuce plus au moins généraux (pas spécifiques à une seule application), il sera intéressant d'inclure les hiérarchies de caches dans le modèle d'allocation mémoire.

### *VI.2.5. Prise en compte des architectures mémoire sophistiquées dans les outils de ciblage logiciel/ matériel*

Dans l'état actuel des travaux, l'architecture mémoire globale partagée n'est pas encore prise en compte dans les outils de ciblage logiciel/matériel. En effet, l'intégration d'une telle architecture mémoire dans le système au niveau architecture implique l'utilisation de certains appels système pour accéder à la mémoire globale par les différents processeurs (voir chapitre IV). Au niveau architecture, ces appels système sont des fonctions de haut niveau. Mais à un niveau d'abstraction plus bas (micro-architecture) ils doivent être interprétés par le système d'exploitation spécifique au processeur et donc doivent être intégrés dans les bibliothèques de génération de systèmes d'exploitation.

# Liste des figures

# Liste des tableaux

# Bibliographie

[Adv99] S.V. Adve, V.S. Pai, P. Ranganathan, «Recent Advances in Memory Consistency Models for Hardware Shared Memory Systems», Special issue on distributed Shared-Memory Systems, Mars 1999.

[Air98] R.. Airiau, J. M. Bergé and al, VHDL «langage, modélisation, synthèse», Presses polytechniques et universitaires Romandes, 1998.

[Amz99] C. Amza, A. Cox, S. Dwarkadas, L. Jin and al, «Adaptive Protocols for Software Distributed Shared Memory», Special issue on distributed Shared-Memory Systems, Mars 1999.

[Bag01] A. Baghdadi, D. Lyonnard, N.E. Zergainoh, A.A. Jerraya, «An Efficient Architecture Model for Systematic Design of Application-Specifie Multiprocessor SoC», In Proceedings of Design Automation and Test in Europe, Mars 2001.

[Bag00] A. Baghdadi, N.E. Zergainoh, W. Cesàrio, T. Roudier, A.A. Jerraya, «Design Space Exploration for Hardware /Software Codesign of Multiprocessor Systems», In Proceedings of the 11th IEEE International Workshop on Rapid System Prototyping, Juin 2000.

[Bel91] F. Belina, D. Hogrefe, A. Sarma, «SDL with application from protocol specification», Prentice Hall International (UK) Ltd, 1991.

[Bol89] J. Bolosky, Robert P. Fitzgerald, Michael L. Scott, «Simple But Effective Techniques for NUMA Memory Management», Proceedings of the 12th ACM Symposium on Operating System Principles, Décembre 1989.

[Bou97] J-M. Bourjolly, V. Rebetez, «An Analysis of Lower Bound Procedures for the Bin Packing Problem», CRT Publishers, Mai 1997.

[Bug97] E. Bugnion, S. Devine, M. Rosenblum, «Disco : Running commodity operating systems on scalable multiprocessors», Proceedings of the 16th ACM Symposium on Operating System Principles, Octobre 1997.

[Car91] J.B. Carter, J.K. Bennett, W. Zwaenepoel, «Implementation and performance of Munin», Proceedings of the 13th ACM Symposium on Operating System Principles, Octobre 1991.

[Cat98a] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, A. Vandecappelle, «Custom Memory Management Methodology - Exploration of Memory Organisation

for Embedded Multimedia System Design», Kluwer Academic Publishers, Boston, 1998.

[Cat98b] F. Cathoor, S. Wuytack and al, «System level transformations for low power data transfer and storage», In paper collection low power CMOS design, IEEE Press, 1998.

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

[Cec01] E. Cecchet, «Apport des réseaux à capacité d'adressage pour les grappes à mémoire partagée distribuée logicielle – Conception et applications», These de doctorat de l'INPG, Spécialité Informatique, Grenoble, France, Juillet 2001.

[Ces00] W. Cesario, G. Nicolescu, L. Gauthier, D. Lyonnard, A.A. Jerraya, «An XML-based meta-model for the design of multiprocessor embedded systems»,VHDL International User's Forum (VIUF) Fall Workshop, Orlando, FL, USA, Octobre 2000.

[Ces01a] W. Cesario, G. Nicolescu, L. Gauthier, D. Lyonnard, A.A. Jerraya, «Colif: a design representation for application-specific multiprocessor SOCs», IEEE Design & Test of Computers, Vol. 18, No. 5, Septembre/Octobre, 2001.

[Ces01b] W. Cesario, G. Nicolescu, L. Gauthier, D. Lyonnard, A.A. Jerraya, «Colif: a multilevel design representation for application-specific multiprocessor system-on-chip design», 12th IEEE International Workshop on Rapid System Prototyping (RSP 2001), Monterey, California, USA, Juin 2001.

[Cha95] A.P. Chandrakasan, M. Potkonjak, R.. Mehra, J. Rabaey, R..W. Brodersen, «Optimizing Power Using Transformations», IEEE Transactions on CAD, Vol. 14, No. 1, Janvier 1995.

[Com90] D. Comer, J. Griffioen, «A new design for distributed systems : The remote memory model», 1990 USENIX Conference, Juin 1990.

[Cou99] G. Coulouris, J. Dollimore, T. Kindberg, «Distributed Systems, Concepts and Design, Second Edition», Addison Wesley, ISBN, 1999.

[Cox90] A.L. Cox, R.J. Fowler, «The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor. Experiences with PLATINUM», Proceedings of the 12th ACM Symposium on Computer Architecture, Décembre 1990.

[Cul98] D. Culler, J.P. Singh, A. Gupta, «Parallel computer architecture :A Hardware/Software approach», Maurgan Kauffman publishers, Aout 1998.

[Dah94] M. Dahlin, R.. Wang, T. Anderson D. Patterson, «Cooperative caching: Using remote client memory to improve file system performance», USENIX Symposium on Operating Systems Design and Implementation (OSDI), Novembre 1994.

[Dan97] K. Danckaert, F. Cathoor and al, «System-level memory optimization for hardware-software co-design», Proceedings of Workshop on hardware/software co-design, Braunschweig, Germany, Mars 1997.

[Dav97] J-M. Daveau, «Spécification système et synthèse de la communication pour le codesign logiciel/matériel», Thèse de doctorat, INPG, Spécialité : Microélectronique, TIMA, Grenoble, Decembre 1997.

[Dup98] L. Dupont, M.-C. Portmann, C. Proust, A. Vignier, «New Separation scheme for Hybrid Flowshop», Journal Européen des Systèmes Automatisés, vol 32, n° 4, 1998.

[Fee95] M J. Feeley, W E. Morgan, F H. Pighin and al, «Implementing a global memory management in a workstation cluster», Proceedings of the 15th ACM Symposium on Operating System Principles, Décembre 1995.

[Fra99a] A. Fraboulet, G. Huard, A. Mignotte, «Optimisation de la consommation et de la place mémoire par transformation de boucles», au Colloque CAO de circuits intégrés et systèmes, Aix-en-Provence, France, Mai 1999.

[Fra99b] A. Fraboulet, G. Huard, A. Mignotte, «Loop Alignement for Memory Accesses Optimization», Proceedings of International Symposium on System Synthesis, San Jose, California, USA, Novembre 1999.

[Fra01] A. Fraboulet, K. Kodary, A. Mignotte, «Loop Fusion for Memory Space Optimization», Proceedings of International Symposium on System Synthesis, Montreal, Canada, Octobre 2001.

[Gau00] L. Gauthier, A.A. Jerraya, «Software & RTOS targeting for multiprocessor architectures», MEDEA Conference on Embedded System Design, Munich, Germany, Octobre 2000.

[Gau01a] L. Gauthier, S. Yoo, A.A. Jerraya, «Automatic Generation and Targeting of Application Specific Operating Systems and Embedded Systems Software», In Proceedings of Design Automation and Test in Europe, Mars 2001.

[Gau01b] L. Gauthier, S. Yoo, A.A. Jerraya, «Automatic Generation and Targeting of Application Specific Operating Systems and Embedded Systems Software», TCAD, IEEE Transactions on Computer-Aided Design, Vol. 20 No. 11, Novembre 2001.

[Gau01c] L. Gauthier, S. Yoo, A.A. Jerraya, «Application-Specific Operating Systems Generation and Targeting for Embedded SoCs», SASIMI 2001, The 10th Workshop on Synthesis And System Integration of MIxed Technologies, Octobre 2001.

[Gau01d] L. Gauthier, «OS generation for multitask software targeting on heterogeneous multiprocessor architectures for specific embedded systems», Thèse de Doctorat INPG, Spécialité Microélectronique, TIMA, Grenoble, Décembre 2001.

[Ger01] P Gerin, Y Sungjoo, G Nicolescu, A A Jerraya, «Scalable and flexible cosimulation of SoC designs with heterogeneous multi-processor target architectures», Asia and South Pacific Design Automation Conference (ASP-DAC 2001), Yokohama, Japan, Janvier - Février 2001.

[Gha02] F. Gharsalli, S. Meftali, F. Rousseau, A.A. Jerraya, «Embedded Memory Wrapper Generation for Multiprocessor SoC», Proceedings of Design Automation Conference, New Orleans, USA, Juin 2001.

[Gru00] P. Grun, N. Dutt, A. Nicolau, «Memory aware compilation through accurate timing extraction», Proceedings. of Design Automation Conference, Los Angeles, USA, June 2000.

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

[Ibe97] M. Ibel, K. E. Schauser, C J. Scheinman, et M. Weis, «High-Performance Cluster Computing Using SCI», Hot Interconnects Symposium V, Août 1997.

[Itz97] A. Itzkovitz, A. Schuster, L. Shalev, «Supporting Multiple Programming Paradigms for Distributed Clusters on top of a Single Virtual Parallel Machine - The MILLIPEDE Concept», Proceedings of the 2nd International Workshop on High Level Programming Models and Supportive Environments (HIPS'97), Genève, Avril 1997.

[Jay96] H. Jay, «Conception et réalisation d'une mémoire partagée répartie», Thèse de doctorat de l'Institut National Polytechnique de Grenoble, Novembre 1996.

[Jer01a] A. A. Jerraya, R.. Bergamaschi, I. Bolsens and, «Are Single-Chip Multi-processors in Reach?», Roundtable, IEEE Design & Test of Computers», Vol 18, No 1, Janvier - Février 2001.

[Jer01b] A.A. Jerraya, W. Wolf, «Guest Editors , IEEE Design & Test of Computers, Special Issue on Application-Specific Multi-Processor System On Chip», Vol. 18, No 5, Septembre - Octobre 2001.

[Kel94] P. Keleher, A. Cox, S. Dwarkadas, W. Zwaenepoel, «TreadMarks : Distributed Shared Memory on Standard Workstations and Operating Systems», Winter 94 Usenix Conference, Janvier 1994.

[Kol96] D.J. Kolson, A. Nicolau, «Elimination of Redundant Memory Traffic in High-Level Synthesis», IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 15, No. 11, Novembre 1996.

[Kul95] D. Kulkarni, M. Stumm, «Linear loop transformations in optimizing compilers for parallel machines», The australian computer journal, Mai 1995.

[Lab00] M. Labbé, G. Laporte and S. Martello, «An exact algorithm for the dual bin packing problem», Operations Research Letters, Vol. 17, 2000.

[Lar91] R..P. LaRowe Jr, C. Schlatter Ellis, L. S. Kaplan, «The Robustness of NUMA Memory Management», Proceedings of the l0th ACM Symposium on Operating System Principles, Octobre 1991.

[Li89] K. Li, P. Hudak, «Memory coherence in shared virtual memory systems», ACM Transactions on Computer Systems, Novembre 1989.

[Li97] Y. Li, W. Wolf, «Allocation of Multirate Systems on Multiprocessors with Memory Hierarchy Modeling and Optimization», Proceedings of CODES/CASHE'97, Braunschweig, Germany, Mars 1997.

[Li99] Y. Li, W. Wolf, «Hardware/Software Co-Synthesis with Memory Hierarchies», IEEE transaction on computer-aided design of integrated circuit and Systems, Octobre 1999.

[Lyo01] D. Lyonnard, S. Yoo, A. Baghdadi, A. A. Jerraya, «Automatic Generation of Application-Specific Architectures for Heterogeneous Multiprocessor System-on-Chip», Proceedings of Design Automation Conference, Las Vegas, USA, Juin 2001.

[Mef01a] S. Meftali, F. Gharsalli, F. Rousseau, A.A. Jerraya, «Automatic Code-transformations, and Architecture Refinement for Application-specific Multiprocessor SoCs with Shared Memory», ln XIth IFIP VLSI-SOC01, Décembre 2001.

[Mef01b] S. Meftali, F. Gharsalli, F. Rousseau, A. A. Jerraya, «An Optimal Memory Allocation for Application-Specific Multiprocessor System-on-Chip», Proceedings of International Symposium on System Synthesis, Montreal, Canada, Octobre 2001.

[Mef02] S. Meftali, F. Gharsalli, F. Rousseau, A. A. Jerraya, «Automatic Architecture Refinement and Code-Transformations for Application-Specific Multiprocessor SoCs with Shared Memory», Chapitre de livre, Kluwer Academic Publishers, Juin 2002.

[Men98] D. Mentré, T. Priol, «NOA : A Shared Virtual Memory over a SCI cluster», SCI Europe'98, Septembre 1998.

[Nic01a] G. Nicolescu, K. Svarstad, O. Meunier, W. Cesàrio and al, «Desiderata d'un langage de spécification pour la conception des systèmes électroniques: concepts de base et niveaux d'abstraction», Revue TSI, 2001.

[Nic01b] G. Nicolescu, S. Yoo, A.A. Jerraya, «Mixed-Level Cosimulation for Fine Gradual Refinement of Communication in SoC Design», In Proceedings of Design Automation and Test in Europe, Mars 2001.

[Pan96] P. R. Panda, N. Dutt, «Reducing Address Bus Transition for Low Power Memory Mapping», Proceedings of the European Design and Test Conference, Paris, Mars 1996.

[Pan99] P. R. Panda, N. Dutt, A. Nicolau, «Memory Issues in Embedded Systems-on-chip : Optimization and exploration», Kluwer Academic Publishers, 1999.

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

[Ran00] M. Rangarajan, L. Iftode, «Software Distributed Shared Memory over Virtual Interface Architecture: Implementation and Performance», 3rd Extreme Linux Workshop, Atlanta, USA, Octobre 2000.

[Rix00] S. Rixner, W. J. Dally, U. J. Kapasi and al, «Memory Access Scheduling», Proceedings of ISCA'00, Vancouver, Canada, Juin 2000.

[Sam94] H. Samsom, F. Franssen, F. Catthoor and al, «Verification of Loop Transformations for Real Time Signal Processing Applications», Proceedings of VLSI Signal Processing, IEEE, 1994.

[Sch99] K. Scholtyssik, M. Dormanns, «Simplifying the use of SCI shared memory by using software SVM techniques», 2nd Workshop on Cluster Computing, Mars 1999.

[Sch98] M. Schulz, H. Hellwagner, «Global Virtual Memory based on SCI-DSM», SCI Europe'98, Septembre 1998.

[Ste97] R.. Stets, S. Dwarkadas, N. Hardavellas and al, «Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network», Proceedings of the 16th ACM Symposium on Operating System Principles, Octobre 1997.

[Sva01] K. Svarstad, G. Nicolescu, A. A. Jerraya, «A Model for Describing Communication between Aggregate Objects in the Specification and Design of Embedded Systems», Proceedings. of the European Design and Test Conference, Munich, Germany, Mars 2001.

[Sys] SystemC, «user's manual», http://www.systemc.org.

[Ver96] B. Verghese, S. Devine, A. Gupta, M. Rosenblum, «Operating System Supportfor Improving Data Locality on CC-NUMA Compute Servers», Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems, Octobre 1996.

[Wol92] M. Wolf, «Improving locality and parallelism in nested loops», Ph.D dissertation, Stanford University, USA, Aout 92.

[Wuy98] S. Wuytack, J. P. Diguet and al, «Formalized methodologie for data reuse exploration for low-power hierarchical memory mapping», IEEE Transactions on VLSI Systems, Vol6, 1998.

[Yoo01] S. Yoo, G. Nicolescu, D. Lyonnard and al, «A generic wrapper architecture for multi-processor SoC cosimulation and design», International Symposium on Hardware Software Codesign (CODES 2001), Copenhague, Denmark, Avril 2001.