UNIVERSITÉ DE MONTRÉAL

# ANALYSE DE PERFORMANCE MULTI-NIVEAU ET PARTIONNEMENT D'APPLICATION RADIO SUR UNE PLATEFORME MULTIPROCESSEUR

DÉPARTEMENT DE GÉNIE ÉLECTRIQUE

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION DU

DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES

(GÉNIE ÉLECTRIQUE)

DÉCEMBRE 2009

© JOSÉ-PHILIPPE TREMBLAY, 2009.

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

ANALYSE DE PERFORMANCE MULTI-NIVEAU ET PARTIONNEMENT D'APPLICATION RADIO SUR UNE PLATEFORME MULTIPROCESSEUR

Présenté par : TREMBLAY José-Philippe

En vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées

A été dûment accepté par le jury d'examen constitué de :

M. DAVID Jean-Pierre, Ph.D., président

M. SAVARIA Yvon, Ph.D., membre et directeur de recherche

M. THIBEAULT Claude, Ph.D., membre et codirecteur de recherche

M. LANGLOIS Pierre, Ph.D., membre

# REMERCIEMENTS

Je te tiens d'abord à remercier M. Savaria et M. Thibeault, respectivement directeur et co-directeur de maitrise, pour leur disponibilité, leur support et divers conseils menant à terme avec succès mes travaux de recherche. Je tiens également à souligner le soutien financier d'Octasic, principal partenaire du projet OPERA. Je remercie également mes coéquipiers du GRM et du Lacime pour leur support et encadrement lors de la réalisation du projet. Je souligne plus particulièrement l'appui de Mme Mbaye pour ses précieux conseils. Je termine par remercier ma famille et mes amis qui m'ont soutenu tout au long de mes études.

# RÉSUMÉ

Depuis déjà un bon nombre d'années, une demande importante est apparue dans le domaine des télécommunications en ce qui a trait aux systèmes portables. En plus des fonctions téléphoniques déjà existantes, plusieurs applications connexes viennent maintenant se greffer à ces appareils. Afin de pouvoir surmonter les défis imposés par ces nouvelles classes d'applications, de nouvelles méthodologies et topologies de circuit ont vu le jour. Ce mémoire s'inscrit dans le projet global OPERA qui a pour but d'explorer différentes classes d'applications par rapport à une implémentation matérielle existante de manière à guider les phases de configuration, de vérification et possiblement de modification du design actuel.

De manière plus spécifique à ce projet, un des buts était de faire ressortir rapidement et automatiquement si une application sélectionnée dans le domaine des télécommunications pouvait être exécutée selon les limites imposées par les différentes normes sur la puce Vocallo fournie par les concepteurs d'Octasic, le partenaire industriel qui supporte ce projet. Pour arriver à cet objectif, la méthodologie dite de conception basée sur les plateformes (Platform based design) a été sélectionnée. Cette technique repose sur la modélisation des différentes couches du système global, soit la couche physique représentant la puce Vocallo et la couche applicative servant à abstraire les algorithmes choisis. Une fois la modélisation des couches effectuées, il suffit de les relier en propageant les contraintes du niveau applicatif vers la couche matérielle de manière à obtenir une estimation de performance selon les caractéristiques abstraites dans chacun des modèles.

La puce Vocallo comporte plusieurs caractéristiques peu communes. En effet, le Vocallo est composé d'une matrice de 15 cœurs reliés à une mémoire externe par un seul et unique bus de communication. Chacun des ces cœurs est lui-même constitué de 16 unités opératives asynchrones (ALUs) fonctionnant en parallèle.

De manière à suivre la méthodologie choisie, il était nécessaire de sélectionner une représentation de l'application ainsi que d'effectuer une modélisation de la plateforme matérielle. Un graphe de flots de contrôle et de données (Control and Data Flow Graph) a été sélectionné afin de représenter les applications décrites en C ou C++. Afin de mieux visualiser le processus de développement d'une application, un processus de hiérarchisation et de caractérisation des nœuds a été instauré par rapport au graphique CDFG de base. Par la suite, à l'aide de notre infrastructure de travail basée sur SUIF2, plusieurs modélisations d'un cœur de la plateforme Vocallo ont été développées. La principale difficulté de ce procédé repose sur l'abstraction de l'asynchronisme présent dans les unités de calcul à l'intérieur d'un modèle synchrone.

La principale contribution de ce mémoire est la conception d'un outil automatique permettant d'obtenir des estimations de performance de l'exécution d'un algorithme exprimé en C ou C++ sur un des cœurs de la plateforme Vocallo. Les performances sont obtenues en termes de quantité requise de mémoire et de temps d'exécution. De nombreuses autres caractéristiques sont également annotées dans le graphique afin de pouvoir aisément comprendre les goulots d'étranglement dans les cas problématiques. Plusieurs exemples présentés dans le mémoire viennent montrer comment utiliser les estimations produites comme bases pour l'ordonnancement d'applications sur la matrice de processeurs. Notre outil rassemble finalement toutes les estimations effectuées sous une représentation graphique exprimée sous forme XML afin de servir pour la deuxième phase dynamique du projet OPERA.

# ABSTRACT

Over the last few years, an important increase in demand has appeared in the telecommunication world in terms of portable system. New classes of applications, like video capability, are now supported by these devices. In order to satisfy the challenges of these new types of software, new circuit topology and emerging methodology became a hot topic of research. This thesis is a small part of the OPERA project that tries to explore several classes of applications in regards to a specific hardware implementation in order to guide efficiently the phases of configuration and verification and possibly could lead to modifications of the design itself.

A main goal of this project was to quickly and automatically know if a selected application within the world of telecommunication could be executed on the Vocallo chipset, developed by Octasic, the industrial partner sponsoring this project, while satisfying the requirements imposed by the different standards. In order to achieve this goal, the Platform Based Design approach was chosen. This technique rests on the modeling of the different layers of the overall system. The physical layer serves to represent the Vocallo chip, while the application layer is used to capture the algorithm's requirements. Upon the completion of the modeling of these layers, the constraints of the application are propagated towards the physical layer to obtain an estimation of performances based on the selected models.

The Vocallo chip is characterised by several uncommon particularities. It is composed of a matrix of 15 cores linked to an external memory by a single communication bus. It is also important to specify that each of the cores is composed of 16 asynchronous ALUs working in parallel. By following the selected methodology, an intermediate representation of the application is selected. The control and data flow graph representation was chosen to represent applications described in C or C++. In order to better visualise possible tradeoffs when mapping applications, a hierarchical structure embedding detailed node models was created. Several versions of the Vocallo model

were then developed within the SUIF2 infrastructure. A major issue in this process was the abstraction of the asynchronous behaviour of the processing units.

The main contribution of this thesis is the implementation of an automated tool producing performance estimates of how Vocallo would behave when executing on one of the chip's core some key algorithms expressed in C or C++. These performances are reported in terms of memory requirements and overall execution time. Several other characteristics are also annotated on the graph to easily identify bottlenecks in the application. Several examples presented in the thesis show how to use these estimates as a basis for mapping target algorithms on the Vocallo core processors. Lastly, our tool also summarizes all of the estimates under a graphical form expressed with a XML format.

# TABLE DES MATIÈRES

Remerciements ..... III
Résumé ..... IV
Abstract ..... VI
Table des matières ..... VIII
Listes des figures ..... XII
Liste des Tableaux ..... XIV
Sigles et abréviations ..... XV
Liste des annexes ..... XVI

**CHAPITRE 1 INTRODUCTION ..... 1**

1.1 Conception basée platefeforme (Platform Based Design) ..... 3
1.2 Contexte du projet ..... 6
1.3 Architecture Vocallo ..... 6
1.4 Applications visées ..... 7
1.5 Contributions du mémoire ..... 8
1.6 Plan du mémoire ..... 9

**CHAPITRE 2 MISE EN CONTEXTE ..... 11**

2.1 Architecture MPSoC ..... 12
2.1.1 Historique des MPSoC ..... 12
2.1.2 Défis des MPSoC ..... 16
2.2 Graphe de flots de contrôle et de données ..... 18
2.2.1 Principes de base ..... 18
2.2.2 Variantes du CDFG ..... 21
2.3 Partitionnement et ordonnancement ..... 24

2.4 Applications ..... 27

**CHAPITRE 3 L'ARCHITECTURE DU VOCALLO ..... 30**

3.1 Spécification générale du Vocallo ..... 31

3.2 Architecture Inter-Noyaux ..... 31

3.2.1 Matrice de processeurs ..... 32

3.2.2 Communication inter-noyau ..... 33

3.2.3 Circuits périphériques ..... 34

3.3 Spécification d'un noyau ..... 34

3.3.1 Mémoire Locale ..... 35

3.3.2 Registres ..... 36

3.3.3 Architecture des ALU ..... 37

3.4 Conclusion ..... 38

**CHAPITRE 4 LES GRAPHES DE FLOTS DE CONTRÔLE ET DE DONNÉES ..... 39**

4.1 Cadre de travail ..... 40

4.1.1 SUIF2 ..... 40

4.1.2 MACHSUIF ..... 41

4.1.3 Flot de conception du CDFG ..... 41

4.2 Processus de création du CDFG ..... 44

4.2.1 Algorithme de conversion du CFG vers le CDFG ..... 45

4.2.2 Extraction des dépendances de données ..... 47

4.3 Hiérarchisation ..... 50

4.3.1 Algorithme de hiérarchisation du CDFG ..... 51

4.3.2 Structures de contrôle ..... 53

4.3.2.1 Structure séquentielle ..... 53

4.3.2.2 Boucle ..... 54

4.3.2.3 Branchement ..... 56

4.4 Caractérisations des nœuds ..... 57

4.4.1 Nœuds de base ..... 57
4.4.2 Nœuds hiérarchiques ..... 58
4.5 Visualisation ..... 59
**CHAPITRE 5 ESTIMATION DES BESOINS EN MÉMOIRE ..... 61**
5.1 Estimation des besoins en mémoire ..... 62
5.1.1 Caractérisation de la mémoire ..... 62
5.1.2 Extraction des besoins en mémoire ..... 64
5.1.3 Validation ..... 65
5.2 Partitionnement ..... 68
5.2.1 Principes de partionnement ..... 68
5.2.2 Exemple de partionnement ..... 71
5.3 Conclusion ..... 73
**CHAPITRE 6 MODÉLISATION ET ORDONNANCEMENT ..... 75**
6.1 Modélisation ..... 75
6.1.1 Premier modèle ..... 75
6.1.2 Deuxième modèle ..... 77
6.1.3 Validation des modèles ..... 80
6.2 Ordonnancement d'application ..... 83
6.2.1 Taux d'utilisation ..... 83
6.2.2 Limite dues aux unités de contrôle ..... 85
6.2.3 Exemple ..... 87
6.3 Variabilité de la vitesse ..... 89
6.3.1 Cause architecturale ..... 89
6.3.2 Effet du parallélisme ..... 90
6.4 Conclusion ..... 93
**CHAPITRE 7 CONCLUSION ..... 95**
7.1 Synthèse des travaux ..... 95

7.2 Limitations des travaux ..... 97

7.3 Possibilités de travaux futurs ..... 98

**RÉFÉRENCES……..99**

**ANNEXES……..104**

# LISTES DES FIGURES

Figure 1-1. Évolution des modèles d'entrée de design utilisés dans le domaine de la conception de circuits intégrés ..... 2
Figure 1-2. Interaction entre les différentes couches d'abstraction ..... 5
Figure 2-1. MPSoC Daytona de Lucent ..... 13
Figure 2-2. Réseau de processeurs C-5 ..... 14
Figure 2-3. Le Viper Nexperia de Philips ..... 15
Figure 2-4. Le OMAP 5912 de Texas Instrument ..... 16
Figure 2-5. Exemple de CFG et de DFG ..... 19
Figure 2-6. Exemple de code et de son CDFG équivalent ..... 20
Figure 2-7. Transformations de branchement : A) Branchement complet B) Branchement étendu ..... 22
Figure 2-8. Transformations de boucle : A) Boucle complète B) Boucle étendue ..... 23
Figure 2-9. Exemples de partionnement : A) Architecture MPSoC à 8 processeurs B-C-D) Stratégies de partionnement ..... 25
Figure 3-1. Schéma bloc de l'architecture complète du Vocallo [25] ..... 32
Figure 3-2. Schéma bloc de l'architecture d'un noyau du Vocallo ..... 35
Figure 4-1. Flot de conception d'un CDFG ..... 42
Figure 4-2. Algorithme de conversion du CFG vers le CDFG ..... 45
Figure 4-3. Exemples de dépendances de données ..... 50
Figure 4-4. Algorithme de hiérarchisation du CDFG ..... 52
Figure 4-5. Exemples de représentations de boucles ..... 55
Figure 4-6. Exemples de représentations de branchements ..... 56
Figure 5-1. Première étape de partitionnement ..... 69
Figure 5-2. Deuxième étape de partitionnement ..... 70
Figure 6-1. Algorithme d'ordonnancement du premier modèle ..... 76
Figure 6-2. Deuxième algorithme d'ordonnancement ..... 79

Figure 6-3. Exemple du deuxième modèle d'ordonnancement ..... 80
Figure 6-4. Effet des dépendances de données ..... 85
Figure 6-5. Temps d'exécution du décodage Turbo en fonction du nombre d'Alu utilisés ..... 86
Figure 6-6. Variation de l'accélération relative en fonction du niveau de parallélisme pour un cœur lent..... 91
Figure 6-7. Variation de l'accélération relative en fonction du niveau de parallélisme pour un cœur rapide..... 92

# LISTE DES TABLEAUX

Tableau 1-1. Ressources matérielles typiques et exemple de caractéristiques ..... 4
Tableau 3-1. Type et quantité des registres d'un noyau ..... 36
Tableau 3-2. Étape d'exécution d'une instruction ..... 37
Tableau 4-1. Caractéristiques des nœuds de base ..... 57
Tableau 4-2. Caractéristiques des nœuds hiérarchiques ..... 58
Tableau 5-1. Catégories de mémoire ..... 63
Tableau 5-2. Estimation de la quantité de mémoire requise pour l'addition vectorielle .. 66
Tableau 5-3. Estimation mémoire pour le Maximum vectoriel ..... 66
Tableau 5-4. Estimation mémoire pour le FIR Complexe ..... 66
Tableau 5-5. Estimation mémoire pour le Filtre LMS ..... 67
Tableau 5-6. Partitionnement final avec une mémoire locale de 10 Ko ..... 72
Tableau 5-7. Partitionnement final avec une mémoire locale de 5 Ko ..... 72
Tableau 5-8. Partitionnement final avec une mémoire locale de 3 Ko ..... 73
Tableau 6-1. Durée moyenne des instructions selon leur classe ..... 78
Tableau 6-2. Validation pour l'addition vectorielle ..... 81
Tableau 6-3. Validation pour le maximum vectoriel ..... 81
Tableau 6-4. Validation pour le filtre complexe ..... 81
Tableau 6-5. Validation pour le filtre LMS ..... 82
Tableau 6-6. Validation pour le codage ..... 82
Tableau 6-7. Temps d'exécution en fonction du nombre de paquets traités ..... 88

# SIGLES ET ABRÉVIATIONS

| | |
|---|---|
| ALAP | : As Late As Possible |
| ALU | : Arithmetic and Logical Unit |
| ASAP | : As Soon As Possible |
| CDFG | : Control and Data Flow Graph |
| CDMA | : Code Division Multiple Access |
| CFG | : Control Flow Graph |
| DDR | : Double Data Rate |
| DFG | : Data Flow Graph |
| DMA | : Direct Memory Access |
| DSP | : Digital Signal Processor |
| GDL | : Graphical Description Language |
| IP | : Intellectual Property |
| MIMD | : Multiple Instruction Multiple Data |
| MPSoC | : Multi-Processor System on Chip |
| NoC | : Network on Chip |
| OFDM | : Orthogonal Frequency Division Multiplexing |
| OPERA | : Octasic Polytechnique ÉTS Radio Application |
| PBD | : Platform Based Design |
| RISC | : Reduced Instruction Set Computer |
| RTW | : Real-Time Workshop |
| SoC | : System on Chip |
| SUIF | : Stanford University Intermediate Format |
| VCG | : Visualization of Compiler Graphs |
| WCDMA | : Wideband Code Division Multiple Access |
| XML | : eXtensible Markup Language |

# LISTE DES ANNEXES

**Annexe A**. Code de l'application synthétique servant de démonstration pour l'outil de partitionnement

**Annexe B**. Représentation XML du partitionnement final avec une mémoire locale de 10K

**Annexe C**. Représentation XML du partitionnement final avec une mémoire locale de 5K

**Annexe D**. Représentation XML du partitionnement final avec une mémoire locale de 3K

**Annexe E**. Algorithme synthétique servant l'effet du parallélisme par rapport à la vitesse d'un cœur

**Annexe F**. Représentation graphique de l'addition vectorielle

# CHAPITRE 1

## INTRODUCTION

La complexité toujours croissante des designs électroniques combinée à un nombre important de technologies devant être maitrisées afin d'obtenir un produit de qualité répondant aux demandes du marché, poussent les sociétés à se concentrer sur leurs compétences primaires. Nous pouvons assister en effet à une restructuration de l'industrie d'un mode vertical au sein d'une même entreprise vers un modèle plus horizontal combinant le savoir-faire propre de plusieurs sociétés travaillant collectivement dans l'accomplissement d'un même design. Ce dernier facteur, ainsi que la pression exercée par un temps de mise en marché, toujours plus difficile à satisfaire, associé aux coûts importants liés à la réalisation de projets de grande envergure, entraîne le besoin de développer des méthodes de conception plus structurées axées sur le principe de réutilisabilité et d'implémentation correcte sans besoin d'itérations subséquentes. Cette quête de flexibilité dans le domaine des circuits embarqués force l'industrie à migrer vers des solutions programmables pour un nombre grandissant de classes d'applications, rendant urgent le besoin de nouvelles méthodologies palliant à toutes ces nouvelles exigences. Afin de toujours pouvoir compter sur un flot de design économiquement satisfaisant, celui-ci doit entre autre permettre de restreindre l'exploration de l'espace de design afin d'obtenir des résultats supérieurs dans les contraintes de temps fixées. Cette tendance est déjà observable dans les domaines de la programmation conventionnelle et de la conception de circuits intégrés. En informatique, de nouveaux langages utilisant un niveau d'abstraction plus élevé sont venus remplacer les langages traditionnels tels l'assembleur, tandis que l'utilisation de modules encapsulant de la propriété intellectuelle, couramment appelés blocs IP (Intellectual Property), est présentement en train de graduellement déplacer les cellules standardisées comme le module de base servant à exprimer le design d'un circuit intégré. Notons que les modules IP sont des structures plus complexes que les cellules normalisées et qu'en fait, un module IP peut

être composé d'un assemblage de cellules normalisées. La figure 1.1 [28] présente cette réalité en montrant l'évolution des modèles utilisés pour exprimer les designs. La méthodologie servant de base pour ce projet peut donc être vue comme la progression naturelle vers un niveau plus élevé d'abstraction dans la conception de systèmes embarqués complexes. La méthodologie adoptée est communément appelée design basé sur des plateformes ou plus communément appelé « platform-based design » (PBD). Cette méthodologie émergente peut aussi être considérée comme une excroissance de l'évolution des SoC dans laquelle les facteurs économiques engendrés par la fabrication et le design de circuits intégrés ont été longuement étudiés.

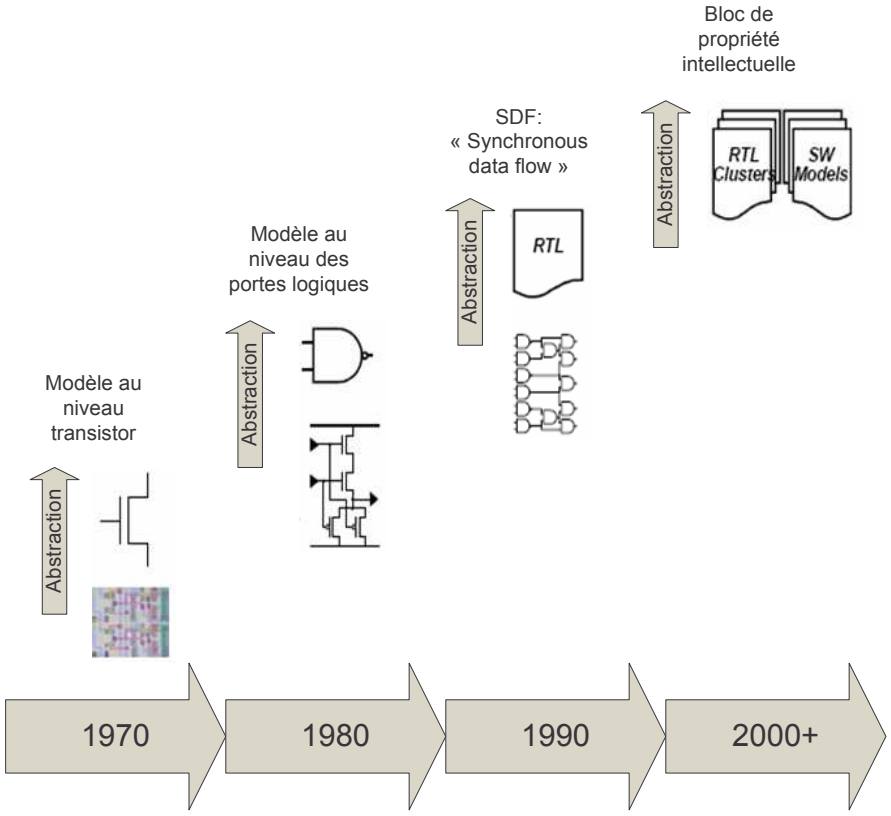


**Figure 1-1. Évolution des modèles d'entrée de design utilisés dans le domaine de la conception de circuits intégrés**

## 1.1 Conception basée platefeforme (Platform Based Design)

L'objectif global du PBD est de proposer une méthodologie permettant d'évaluer plus rapidement et de manière efficace différentes combinaisons d'éléments de calcul (processeur à usage général, DSP, …), de mémoire et d'interconnexions (bus, NOC, …) en sacrifiant le moins possible une performance potentielle atteignable. Celle-ci s'applique à tous les niveaux d'abstraction du design procurant ainsi une structure de travail où des pratiques connues de design qui peuvent être tirées de recherches académiques peuvent être évaluées afin de permettre d'en justifier l'utilisation. Globalement, le PBD peut être décrit comme une approche du type rendez-vous au milieu (*meet-in the middle*) où plusieurs raffinements descendant successifs d'une même spécification rejoignent les abstractions d'implémentations possibles.

Pour ce faire, il est important de bien identifier les couches sur lesquelles reposeront les procédés de raffinement et d'abstraction. Ces couches prennent également le nom de plateforme. On peut aussi définir une plateforme comme un niveau d'abstraction facilitant un certain nombre de raffinements de manière à être elle-même raccordée à un autre niveau d'abstraction (plateforme) dans le flot de conception. Ces couches cachent les détails de plusieurs raffinements possibles des couches inférieures. Celles-ci servent entre autres à isoler les détails des niveaux subséquents tout en laissant transparaître assez d'informations à propos de ces niveaux inférieurs pour explorer l'espace de design afin d'obtenir une prédiction la plus précise possible d'une implémentation finale. Une plateforme peut prendre la forme d'une librairie d'éléments souvent caractérisés par un modèle de fonctionnement précis qui offre une estimation plus ou moins précise dépendant du modèle choisi. Les choix des caractéristiques abstraites des plateformes à chacun des niveaux sont primordiaux et ils sont toujours régis par un compromis entre la complexité et le degré de précision de la modélisation. Les éléments abstraits sont souvent sélectionnés selon un niveau d'importance pour le concepteur. Ces différentes ressources peuvent même être implémentées à l'aide de différents types de technologies

et fournir des performances très variables en fonction des choix effectués. Le tableau 1-1 décrit les ressources matérielles typiques ainsi que les caractéristiques à modéliser.

**Tableau 1-1. Ressources matérielles typiques et exemple de caractéristiques**

| Type de ressource | Exemple | Caractéristiques | |
|---|---|---|---|
| Unité de traitement | Processeurs | Fréquence d'Horloge | MIPS |
| | - Usage général | Taille de la mémoire | Puissance |
| | - Spécifique à une application | Jeu d'instructions | Pipeline |
| | - DSP | | |
| | Dédiés | Latence | Débit |
| | - Co-processeurs | Adressage | Mode "Burst" |
| | - Hard-Soft IP | Taille | |
| Interconnexions | Personnalisé | Taille | Largeur |
| | Bus | Latence | Puissance |
| | "Network-on-Chip" | Vitesse d'accès | |
| Mémoires | DRAM | Taille | Latence |
| | SRAM | Fréquence | Puissance |

Dans le cas de plateformes représentant une architecture, il est à noter que la flexibilité se doit d'être une propriété primordiale. Ceci aura sûrement comme effet, dans les prochaines années, de satisfaire l'effort de calcul requis à l'aide d'éléments programmables, rendant ainsi le design électronique de plus en plus dépendant de sa partie logicielle et de la qualité de ce logiciel. Lorsque la composition des plateformes supérieure et inférieure est bien définie, il suffit de relier les deux pour former une pile de plateformes, communément appelée « *platfom stack* ». Pour ce faire, puisque la PBD est de type rendez-vous au milieu, il faut propager les contraintes obtenues de la plateforme supérieure vers le bas et propager les performances estimées à l'aide de la plateforme inférieure vers le haut. Ce processus est illustré à la figure 1-2. Une pile de plateformes est donc composée d'au moins deux plateformes, ou de plusieurs séries de sous-plateformes, ainsi que de tous les outils et méthodes utilisés pour relier deux niveaux adjacents.

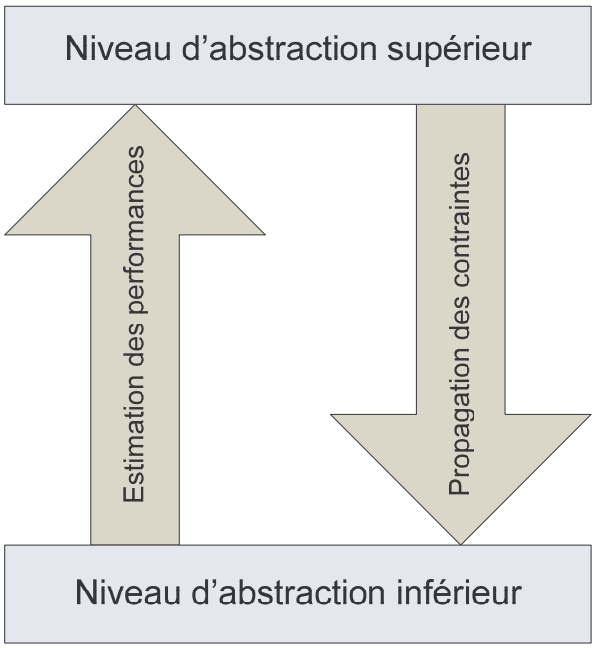


**Figure 1-2. Interaction entre les différentes couches d'abstraction**

L'utilisation de cette méthodologie émergente procure de nombreux bénéfices face aux problématiques actuelles du milieu en mettant en évidence le besoin accru de réutilisabilité et de régularité, pour en arriver à des circuits électroniques économiquement réalisables. En effet, les couches supérieures permettent le développement d'applications sans avoir à tenir compte des caractéristiques précises de leur implémentation matérielle. Les couches inférieures, quant à elles, servent à spécifier certains éléments comme partie d'une plateforme, de manière à pouvoir obtenir une estimation précise et rapide de plusieurs choix architecturaux aussi variés que complexes. Cette méthode semble en effet avoir été adoptée par plusieurs compagnies, dont Philips avec la Nexperia développée pour le multimédia et TI avec OMAP pour la téléphonie cellulaire [9]. Ces plateformes sont toutes deux basées sur des composants programmables. Il est à parier que de nombreuses autres leur emboîteront le pas dans les années à suivre.

## 1.2 Contexte du projet

Ce projet de maîtrise s'inscrit dans le cadre du projet OPÉRA qui signifie « Octasic Polytechnique ÉTS Radio Application ». L'objectif global du projet OPÉRA est d'explorer différentes classes d'applications par rapport à une implémentation matérielle existante, de manière à guider les phases de configuration, de vérification et possiblement de modification du design actuel. De manière plus spécifique, le projet de recherche OPÉRA vise à déterminer si la classe d'algorithmes choisie peut en effet être implémentée de manière efficace et correcte sur l'architecture étudiée. Il vise aussi à permettre d'identifier les goulots d'étranglement empêchant son bon fonctionnement dans le cas d'une réponse négative. La mise en lumière de ces goulots pourra entre autres servir de base dans le cas d'éventuels changements apportés à la plateforme étudiée. L'approche retenue pour effectuer cette tâche est celle de la conception basée sur les plateformes. Pour ce faire, deux types de modèles seront employés pour caractériser et relier les parties applicatives et matérielles de manière à définir un ensemble de plateformes selon les principes énoncés précédemment. Le premier, de nature statique, prendra la forme d'une bibliothèque contenant les caractéristiques retenues des deux niveaux, tandis que le deuxième sera dynamique, à l'aide de la modélisation effectuée dans le langage SystemC. Donc, dans le cadre de ce projet, la plateforme supérieure sera constituée d'une abstraction d'applications sélectionnées dans le domaine des télécommunications, alors que la couche inférieure représentera le processeur Vocallo, développé et fourni par la compagnie Octasic.

## 1.3 Architecture Vocallo

On peut décrire rapidement l'architecture Vocallo comme une matrice de processeurs homogènes de type DSP. Cette architecture s'inscrit dans la tendance des MPSoC. Malgré sa capacité à traiter des applications complexes sur une seule et même puce, à l'aide de ses multiples processeurs indépendants, l'objectif principal de sa conception était d'atteindre une consommation de puissance très faible. Cette dernière particularité,

qui constitue présentement un secteur de recherche complet en lui-même, est obtenue entre autres grâce à l'utilisation de plusieurs unités opératives asynchrones, dans chacun des cœurs de celle-ci. Plus de détails sur la complexité apparente et le fonctionnement de la puce seront donnés au chapitre 3. Comme la plateforme matérielle est fournie, les particularités suivantes devront être prises en compte, venant ainsi spécialiser la définition du PBD présentée ci-haut. En effet, la composition de la bibliothèque de caractérisation du processeur se fera principalement de manière à pouvoir identifier les goulots d'étranglement. Dans ce cas-ci, la plateforme inférieure servira spécialement à imposer les limites physiques de l'implémentation actuelle, plutôt qu'à restreindre l'exploration de l'espace de design pour respecter les exigences imposées par l'application retenue. Étant donnée la complexité du processeur au cœur de l'architecture Vocallo, un effort particulier devra aussi être accordé à une modélisation précise de ses caractéristiques, afin de pouvoir tirer les estimations les plus justes possibles. Vers la fin du projet, la mise à notre disposition d'une suite d'outils développés par Octasic en parallèle avec le déroulement du projet OPÉRA nous a offert la possibilité de pouvoir valider la justesse de nos approximations par rapport à celles prédites par les concepteurs de la puce. Dans une deuxième phase, cette validation pourra également être accomplie face à une exécution réelle de l'application sélectionnée sur les premiers prototypes physiques.

## 1.4 Applications visées

Ce projet de recherche vise principalement des applications de radio logicielle. Contrairement aux radios classiques, le principe de radio logicielle repose sur l'implémentation des différentes étapes de traitement de signal (filtrage, décimation, démodulation, décodage, …) de manière purement logicielle ou à l'aide de matériel physique flexible. Cette méthode émergente en télécommunications permet d'atteindre des performances accrues tout en conférant une grande adaptabilité comparativement aux traditionnels composants matériels spécifiques (filtres, oscillateurs, …). Cette flexibilité

s'obtient beaucoup plus facilement en ne changeant que des paramètres, au niveau logiciel dans le code, correspondants aux nouveaux objectifs de communication visés. Ces changements conduisent à reprogrammer une ou plusieurs unités de traitement sélectionnées (DSP, ASIC, FPGA, PC). Ce projet tend par contre à viser principalement trois grandes classes d'applications soient : les communications sans fils 3G, le WiFi et le WiMax. Toutes ces applications s'appuient elles-mêmes sur des technologies de transmission, telles les CDMA, WCDMA et OFDM. On peut rapidement caractériser les algorithmes de ces technologies comme étant très intenses en calcul sur une grande quantité de données. Ces dernières particularités semblent a priori faire de la plateforme Vocallo un bon choix permettant une grande flexibilité, parce qu'elle offre une importante puissance de calcul, particulièrement grâce à sa matrice de processeurs. Il est également à noter que la plupart des algorithmes de ce domaine sont codés en C ou en Matlab. De ce fait, l'analyse globale devra prendre en compte les différentes particularités de ces deux langages.

## 1.5 Contributions du mémoire

De manière plus spécifique, ce mémoire s'inscrit dans la première phase du projet global, soit l'analyse statique. Le but de cette recherche est de caractériser de manière automatique les applications sélectionnées en langage C afin d'en faire ressortir les différentes contraintes imposées et de pouvoir, à l'aide d'une modélisation de la plateforme Vocallo, prédire les performances de ces algorithmes dans le but éventuel de guider les décisions quant au partionnement et au « mapping » des algorithmes afin de respecter les limites imposées par les différents standards de télécommunications. Pour ce faire, la technique du PBD est utilisée. L'extraction des contraintes des applications joue le rôle de niveau supérieur, tandis que la modélisation de la matrice de processeurs tiendra celui du niveau d'abstraction inférieur. La propagation des contraintes d'un niveau vers l'autre nous permettra de faire ressortir les forces et les faiblesses de la

plateforme matérielle face aux algorithmes choisis. De manière plus concrète, la contribution peut être divisée en trois étapes distinctes :

- Le développement d'un outil automatique d'extraction des contraintes d'un algorithme en C. L'expression des ces contraintes se fera sous la forme d'un graphe de type CDFG.
- L'analyse de la plateforme Vocallo et la conception d'un modèle en C qui corresponde le plus précisément possible à l'implémentation physique actuelle.
- Le regroupement des deux niveaux d'abstraction et en faisant ressortir, en analysant les résultats obtenus à l'aide de métriques de performances, les bonnes pratiques de programmation et d'ordonnancement, menant à la meilleure utilisation possible du processeur au cœur de l'architecture Vocallo vis-à-vis les classes d'applications mentionnées ci-haut.

## 1.6 Plan du mémoire

Ce mémoire commence par une introduction des différents concepts liés à l'utilisation de la méthode « Platform Based-Design ». Une brève description des nombreux aspects de ce projet sera aussi incluse dans ce premier chapitre. Le chapitre 2 fera office de revue de littérature et aura pour but de présenter et justifier l'utilisation de techniques ayant déjà prouvé leur efficacité dans un passé rapproché. Les principales caractéristiques d'un graphe de type CDFG, ainsi que différentes techniques d'ordonnancement seront en effet discutées dans cette section, qui se terminera par une description sommaire des algorithmes utilisés dans l'obtention des résultats présentés dans ce mémoire. Puisqu'une bonne compréhension de l'architecture Vocallo est nécessaire afin de comprendre la modélisation résultante, le chapitre 3 présentera en détails une description de tous les composants que l'on retrouve sur la plateforme matérielle ainsi que leurs interactions. Le chapitre 4 se concentrera sur la production automatique de notre version d'un CDFG typique, en expliquant toutes les nouvelles modifications, par rapport aux normes

introduites dans la littérature. Le chapitre 5 présentera les premiers résultats quant au partionnement des applications, tandis que le chapitre 6 exposera le modèle du Vocallo et les différentes stratégies d'ordonnancement à l'aide des derniers résultats obtenus. La conclusion présentera un sommaire des méthodes présentées ainsi que leurs limites pour finir avec quelques indications sur les travaux futurs, notamment la phase dynamique du projet OPÉRA.

# CHAPITRE 2

## MISE EN CONTEXTE

Un des fondements de base du projet OPERA repose sur l'utilisation du PBD, afin d'obtenir une nouvelle méthodologie permettant de déterminer les performances d'une application sur le processeur Vocallo. Le présent mémoire couvre plus particulièrement l'analyse statique d'algorithmes codés en C. Donc, de par l'emploi du PBD, il est requis de choisir de manière précise les modèles et méthodes qui serviront à représenter et à relier les deux niveaux impliqués, soient l'extraction des contraintes imposées par les applications et la modélisation de la plateforme matérielle. Suite à l'abstraction complète des deux niveaux précédemment identifiés, une phase d'ordonnancement au niveau des opérations élémentaires devra être effectuée, afin de pouvoir estimer les performances. La grande majorité des étapes nécessaires à l'accomplissement de cette phase du projet ont déjà fait séparément l'objet de recherches au cours des années passées. Le fruit de ces expérimentations nous propose ainsi une multitude de techniques dont nous pouvons nous inspirer et que nous pouvons modifier selon nos besoins plus spécifiques. Le présent chapitre, servant de mise en contexte des différents éléments retrouvés dans le cadre de ce projet, commencera donc par présenter un bref historique de différentes plateformes matérielles représentant des défis similaires à ceux proposés par la puce Vocallo. Comme le choix de la représentation intermédiaire d'une application d'un niveau suffisamment bas a finalement abouti sur le graphique de type CDFG, plusieurs variantes et améliorations possibles par rapport à CDFG classique seront par la suite exposées. La section subséquente apportera quelques explications de base et exemples de méthodologies de recherche dans le domaine du partionnement d'algorithmes et d'ordonnancement d'applications dans de multiples contextes. Le chapitre 2 sera complété en dernier lieu par une description sommaire des différents algorithmes utilisés dans l'obtention des résultats présentés dans le cadre de ce mémoire.

## 2.1 Architecture MPSoC

Il est facilement possible d'affirmer que les systèmes sur puce à processeurs multiples (MPSoC; Multiprocessors System-On-Chip) constituent une classe à part entière de l'évolution des systèmes VLSI. Cette classe est justifiée par des besoins communs précis comme le respect de contraintes de temps réel, la faible consommation de puissance et la nécessité de traiter des applications toujours plus complexes qui s'inscrivent souvent dans un contexte multi-usagers. Ceci est en effet très facile à observer dans les principaux domaines d'utilisation des MPSoC, soient le réseautage, les communications, le traitement de signal et le multimédia. Depuis l'avènement du premier MPSoC, de nombreuses architectures aux particularités souvent bien différentes ont été développées afin de répondre à ces classes d'applications spécifiques. La puce Vocallo représente en 2008 une des évolutions possibles de ce type d'architecture dans le domaine du traitement de signal. La section 2.1 servira entre autres à dresser un bref historique de quelques architectures représentant les générations précédentes de MPSoC ainsi qu'à identifier les principaux défis communs à leur réalisation.

### 2.1.1 Historique des MPSoC

Parmi les premières architectures de MPSoC, on peut citer l'architecture Daytona développée par Lucent [1]. La topologie du Daytona est présentée à la figure 2-1. Le Daytona a été conçu pour être intégré dans une station de base sans fils. Chacun des cœurs effectue de manière identique le traitement de signaux provenant d'un certain nombre de canaux. Le Daytona est une architecture symétrique composé de 4 cœurs connectés par un bus rapide. Bien que technologiquement désuet, il est très facile d'en faire ressortir les grandes similarités avec la puce Vocallo, qui sera extensivement présentée dans le chapitre suivant. Suite à la sortie du Daytona, plusieurs autres MPSoC sont apparus sur le marché. Ils répondent à une plus grande variété d'applications et montrent des styles architecturaux plus divers et complexes.

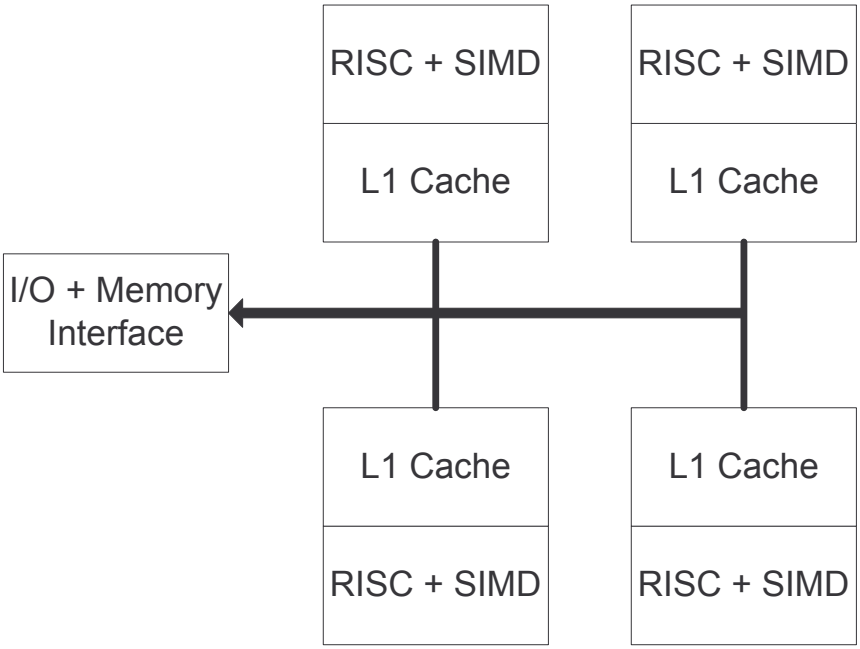


**Figure 2-1. MPSoC Daytona de Lucent**

Le *C-5 Network processor* [7], représenté à la figure 2-2, a été quant à lui réalisé pour le traitement de paquets dans différents types de réseaux. Les paquets sont traités par les 4 processeurs de canal qui sont eux-mêmes constitués de 4 unités opératives chacun. L'implémentation de 3 bus différents permet entre autres le traitement de plusieurs types de trafics. Le C-5 doit par contre utiliser quelques processeurs extérieurs additionnels afin d'effectuer un travail plus spécialisé alors que le processeur exécutif (voir Fig. 2-2) possède un jeu d'instructions réduit de type RISC.

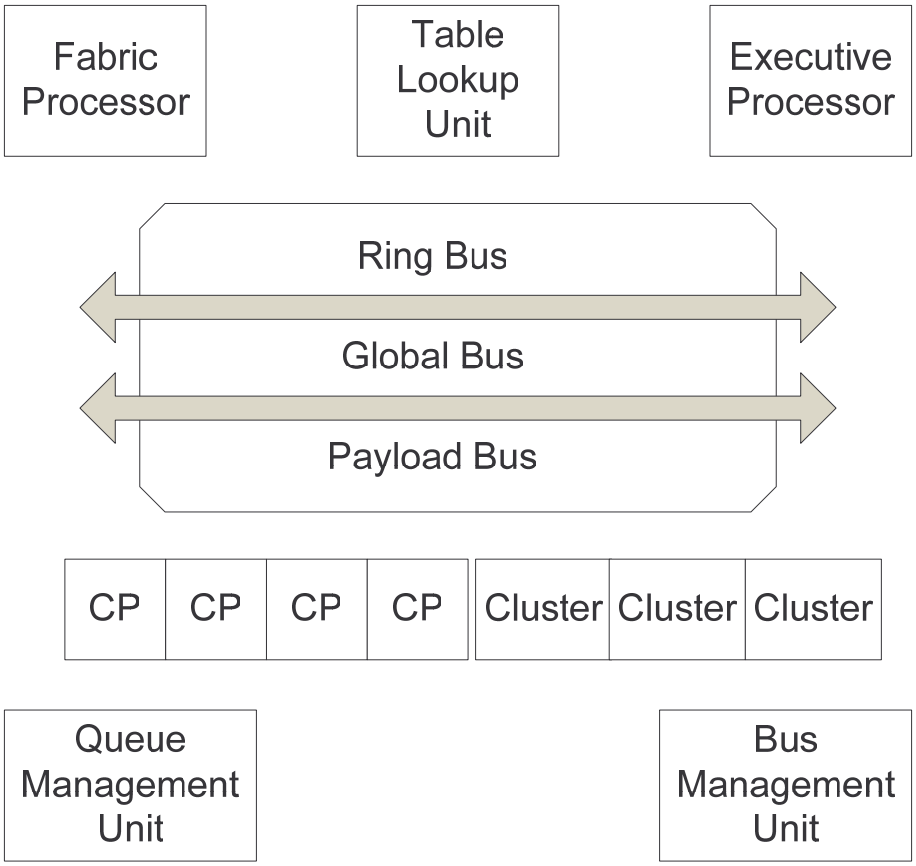


**Figure 2-2. Réseau de processeurs C-5**

Tel que mentionné précédemment, les MPSoC sont également en utilisation dans le domaine du multimédia. Le Philips Viper Nexperia [9], tel qu'illustré à la figure 2-3, se situe parmi les premiers exemples notables de processeur MPSoC en multimédia. Le Viper est principalement composé d'un MIPS servant de maître en étant responsable de prendre en charge le système d'exploitation et d'un processeur Trimedia VLIW répondant comme esclave aux commandes envoyées par le MIPS. Le système est régi par 3 bus, un pour chacun des processeurs tandis que le dernier est associé à l'interface d'accès à la mémoire externe. La partie de traitement multiprocesseur est par contre assurée par la présence de nombreux accélérateurs matériels attachés aux différents bus. Ces accélérateurs sont entre autres responsables d'effectuer les opérations plus complexes tels la conversion de couleurs, la mise à l'échelle, etc.

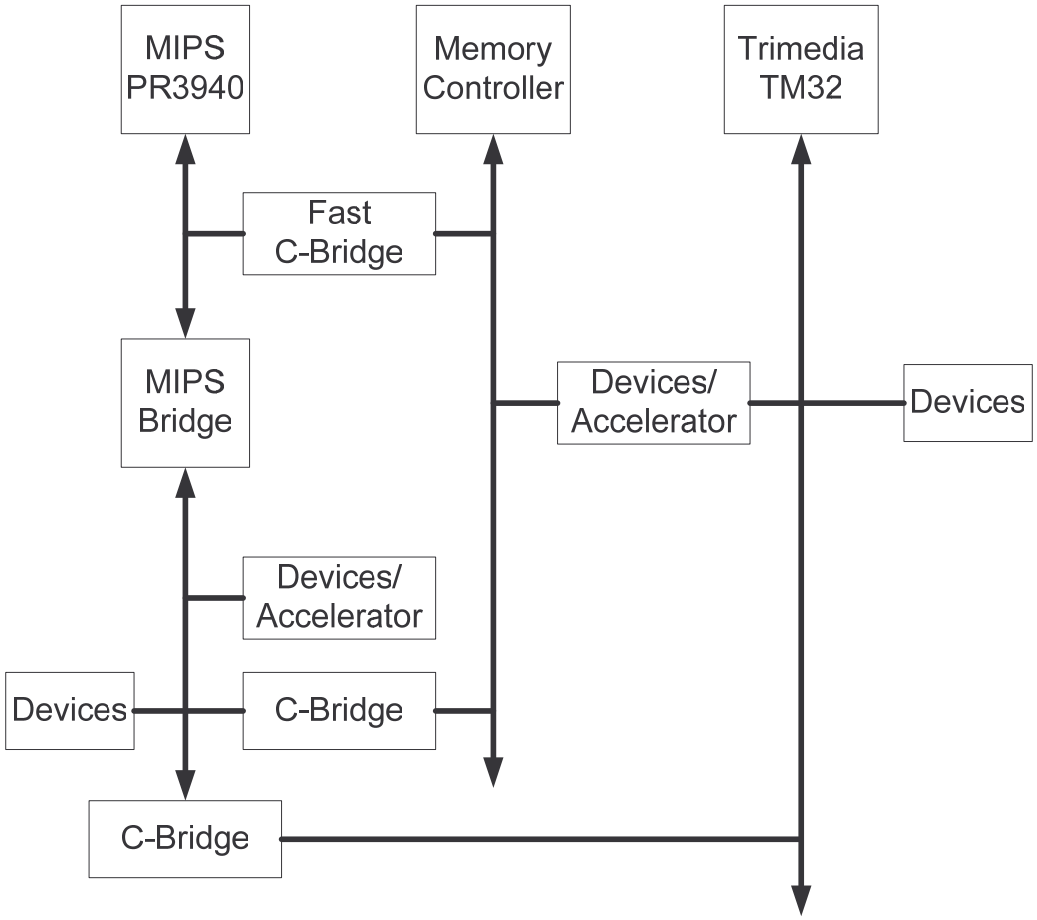


**Figure 2-3. Le Viper Nexperia de Philips**

Les processeurs de téléphones cellulaires constituent une quatrième classe importante d'applications maintenant desservies principalement par les MPSoC. En effet, depuis déjà plusieurs années, ces appareils mobiles ont souvent la possibilité d'effectuer des opérations de communications ainsi que du multimédia. Plusieurs déclinaisons disponibles de l'architecture OMAP de Texas Instrument [31] permettent le support de toutes ces nouvelles fonctionnalités. La figure 2-4 présente la version 5912 du OMAP. Cette version est composée de deux processeurs. Le ARM9 sert de maître au TMS320C55x, DSP traditionnel, qui est quant à lui responsable des opérations de traitement de signal.

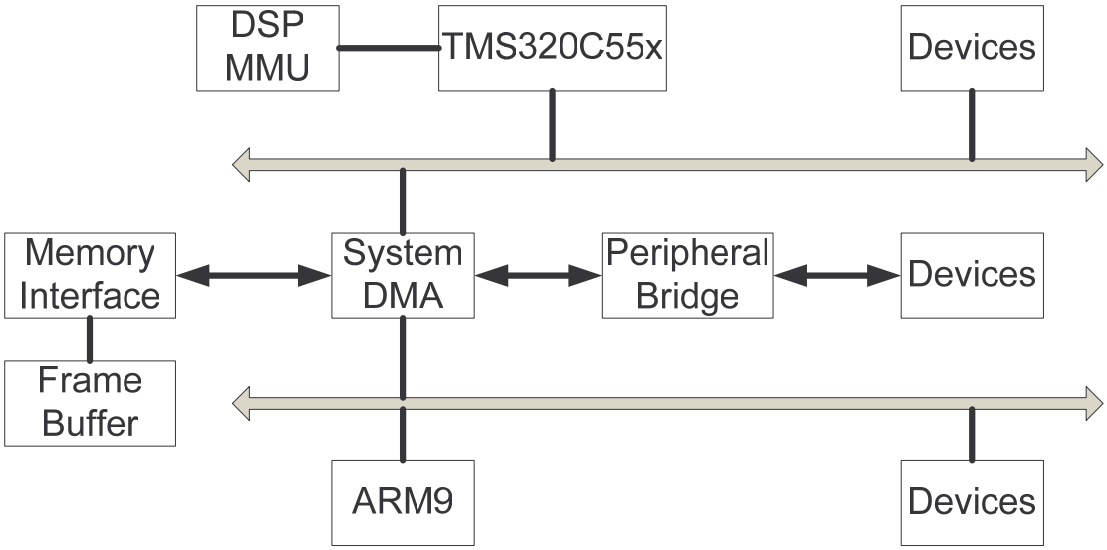


**Figure 2-4. Le OMAP 5912 de Texas Instrument**

### 2.1.2 Défis des MPSoC

Bien que tous les MPSoC présentés ci-dessus présentent des caractéristiques distinctes, ils se rejoignent quand même dans les défis reliés à leur implémentation et leur utilisation. Comme le design de MPSoC est de par sa nature basé sur les processeurs, donc sur une partie logicielle, une des difficultés majeures est de répartir adéquatement l'application sur la configuration optimale de l'implémentation matérielle. Depuis les débuts de l'informatique, les concepteurs de logiciels ont en effet appris à écrire du code séquentiel avec des machines de plus en plus rapides pour effectuer son exécution. On peut ainsi voir apparaître un contraste flagrant entre ce mode de pensée et le niveau de parallélisme offert par les nouveaux circuits de type MPSoC. On peut à présent distinguer deux classes majeures de système [22]. Le traitement symétrique (*Symetric Multiprocessing* ou SMP) est caractérisé par un regroupement de processeurs ou de cœurs, partageant souvent le même accès à une mémoire globale et traitant des tâches similaires dans un contexte global commun. Le traitement asymétrique (A*symetric*

*multiprocessing* ou AMP) est défini par une structure de processeurs habituellement connectés de manière beaucoup moins serrée, qui peuvent souvent avoir des fonctionnalités totalement différentes et qui possèdent en général des ressources locales plus importantes comme des mémoires dédiées. Mais peu importe le modèle suivi, plusieurs tâches sont exécutées de manière concurrente. Elles doivent même souvent s'échanger des données ou avoir accès à des ressources partagées ou externes à la puce elle-même. Ceci a pour effet de pouvoir causer des combats de priorités, de l'interblocage (*deadlocking*), ou un manque de données à traiter (*data starving*) et peut aller jusqu'à engendrer la production de données incorrectes [22]. De cette instabilité imprédictible, due aux raisons précédemment énoncées, découle la grande difficulté, même malgré des années de recherche, de décomposer adéquatement une application codée de façon séquentielle en une série de tâches parallèles pouvant coopérer de manière ordonnée et prédictible. Dans le modèle symétrique, les tâches peuvent être assignées aux différents processeurs, puisqu'ils partagent généralement le même jeu d'instruction, selon les niveaux de trafics et la consommation de puissance désirée. Les communications inter-tâches sont souvent plus aisées à réaliser à partir d'une structure de mémoire commune mais un souci particulier doit être accordé afin de s'assurer de respecter toutes les dépendances présentes dans l'application et de bien équilibrer les charges que représentent ces tâches, afin d'obtenir un taux d'utilisation suffisant. Du coté asymétrique, la fonctionnalité et la granularité variable des unités de calcul requiert des mécanismes de communication plus complexes, mais qui doivent chercher à rester dans l'ensemble le plus efficace possible. Donc, peu importe le type d'architecture sélectionnée, un des défis majeurs reste le découpage de l'application. Ce processus demeure encore à ce jour une tâche ardue et hautement manuelle, habituellement basée sur l'expérience du concepteur.

Le débogage devient également problématique lors de l'utilisation des MPSoC. En effet, cette opération nécessite une connaissance approfondie du fonctionnement individuel de chaque partie tout en gardant une vision globale cohérente. Les outils permettant

l'instrumentation et l'analyse au niveau de l'interaction multiprocesseur restent encore à développer.

## 2.2 Graphe de flots de contrôle et de données

En suivant le PBD, une représentation à bas niveau devait être choisie afin de refléter et de pouvoir propager les caractéristiques des applications étudiées. Il existe plusieurs représentations possibles, mais comme tous les types de dépendance présents dans une application doivent absolument être pris en compte pour modéliser le plus justement possible les particularités d'algorithmes codés en C, notre choix s'est porté sur le graphe de flots de contrôle et de données (CDFG). Ce type de graphe est couramment utilisé comme représentation intermédiaire dans de nombreux compilateurs sur lesquels sont effectuées la plupart des optimisations et autres choix de design de manière à augmenter les performances globales de l'algorithme. Plus de détails en rapport à notre outil qui permet d'extraire des CDFG seront offerts dans le chapitre 4. Pour l'instant, la présente section se concentre sur une définition plus formelle de la composition de ce type de graphe, ainsi que quelques variantes notables tirées de recherches antérieures.

### 2.2.1 Principes de base

Un graphe de flots de contrôle et de données est traditionnellement composé de deux parties, soit le graphe de flots de contrôle (CFG) représentant les dépendances de contrôle et un ou plusieurs graphes de flots de données (DFG). Dans un CFG, un nœud, également appelé bloc de base, correspond à une séquence d'instructions consécutives dans laquelle le flot de contrôle est activé au début de celle-ci et inhibé à la fin de celle-ci, sans possibilité d'arrêt ou de branchement autre qu'à la fin de la séquence. Chacun de ces nœuds peut par la suite être représenté par un DFG où seront représentées les dépendances de données présentes entre chacune des opérations. Ce principe est présenté à la figure 2-5 [16]. Le graphe de gauche correspond aux dépendances de contrôle présentes entre les différents blocs de base. Celui de droite montre le DFG correspondant

au bloc de base BB2. La réunion de ces deux graphes constitue la base de la création d'un CDFG.

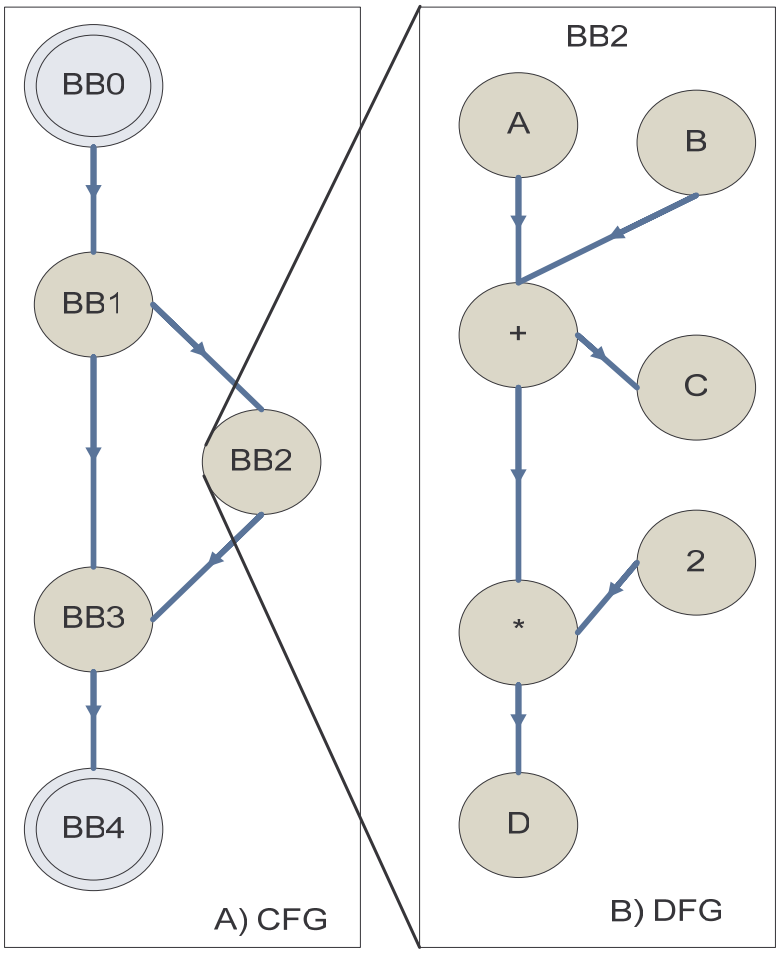


**Figure 2-5. Exemple de CFG et de DFG**

Donc un CDFG est un graphe orienté et acyclique, où les nœuds peuvent être de deux types : nœud opératif, représentant une opération arithmétique ou logique, ou nœud de contrôle servant à rendre compte des structures de contrôle telles les conditions IF/CASE ou les boucles. Tous les liens entre les nœuds sont orientés et impliquent une dépendance de données quand ils rejoignent deux nœuds opératifs et un changement de structure de contrôle lorsqu'ils servent à relier deux nœuds de contrôle. Ces liens peuvent également être traversés de manière conditionnelle en fonction de certaines conditions, comme les

différents cas d'une structure IF ou des bornes d'itérations de certaines boucles. Un exemple complet d'un CDFG représentant un court bout de code est présenté à la figure 2-6. On peut noter que les nœuds 1, 2, 5 et 6 sont des nœuds opératifs, tandis que les nœuds restants correspondent à des nœuds de contrôle. Le lien entre les nœuds 5 et 6 illustre le principe de lien conditionnel, car il n'est emprunté que lorsque la condition A > 0 s'avère être exacte. Ceci résume la base nécessaire à la bonne compréhension de ce type de graphe. Par contre, de nombreux autres types de graphes dérivés du CDFG ont été développés au cours des années, afin de répondre de manière plus adéquate à certains problèmes spécifiques. La section suivante présentera quelques-uns de ces dérivés.

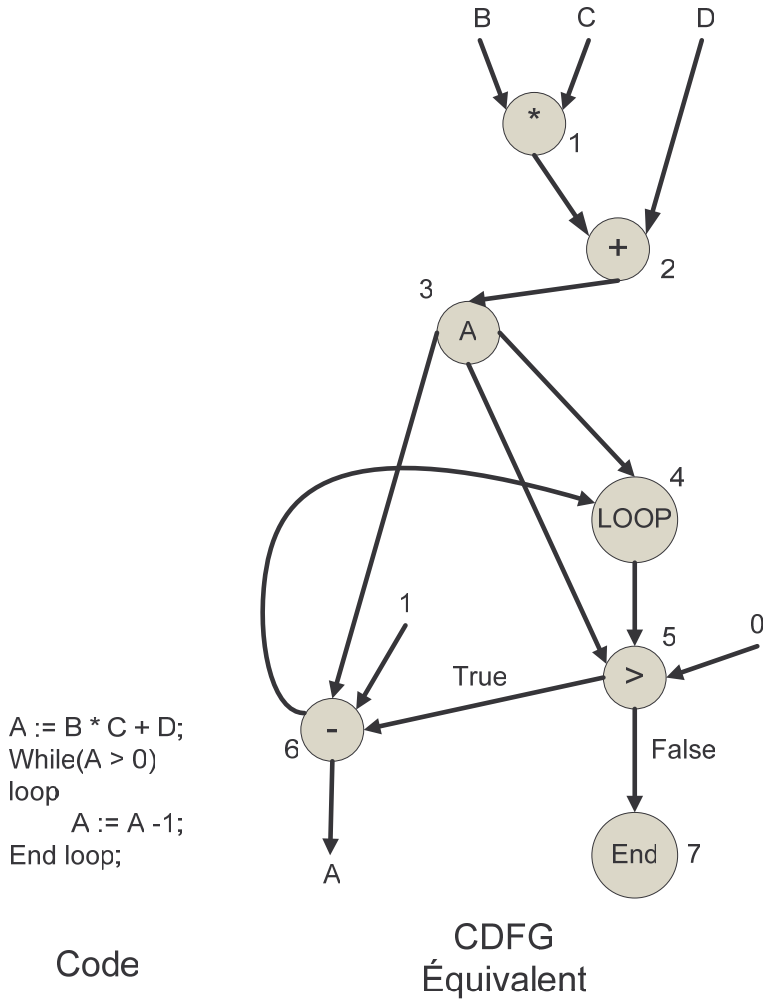


**Figure 2-6. Exemple de code et de son CDFG équivalent**

### 2.2.2 Variantes du CDFG

Depuis le début de l'utilisation des CDFG, ceux-ci ont majoritairement été employés comme représentation intermédiaire par certains compilateurs dans un but d'optimisation. Par contre, de nombreuses recherches académiques se servent maintenant des CDFG à d'autres fins en modifiant souvent l'aspect du graphe. Ce genre de graphe est particulièrement utile dans la synthèse de circuits à haut niveau. R. Namballa, dans le cadre d'une thèse intitulée *CHESS: A tool for CDFG extraction and high-level synthesis of VLSI* [24], se sert d'un CDFG afin de représenter une application codée en langage de description de matériel, soit le VHDL. L'outil développé procède selon les trois étapes suivantes : l'extraction d'un CDFG à partir de modules VHDL, l'ordonnancement et l'allocation du graphe obtenu et finalement l'association du résultat à un ensemble de composants physiques de manière à obtenir de manière automatique une architecture aux performances optimales. Ce principe est également repris par un bon nombre de groupes de recherche dont Smit et al. [29] qui ont la particularité de se baser sur des algorithmes en C. Un autre domaine d'application où le CDFG est bien visible est la synthèse mixte entre le matériel et le logiciel (*Hardware/Software Co-Design*). En effet, on peut voir dans [6] et [19] deux méthodologies différentes afin de partitionner une application afin de l'implanter en matériel ou en logiciel selon des critères bien précis pour chacun des auteurs. Dans le premier cas, la performance en termes de temps d'exécution est primordiale, tandis que la réduction des coûts de communication entre les parties matérielles et logicielles dicte les choix de design pour le second. Knudsen et al. [19] vont même jusqu'à développer leur propre variante du CDFG traditionnel afin de simplifier leur méthode. Leur graphique final découle de plusieurs transformations effectuées avant le partitionnement dans le but d'obtenir une structure permettant une meilleure estimation des communications entre certains nœuds implémentés sur des processeurs différents. Le but des transformations, qui sont surtout opérées au niveau des structures de contrôle, est d'atteindre une granularité assez fine pour une estimation correcte des communications, tout en restant suffisamment grossière afin de ne pas faire augmenter inutilement le temps requis pour le partitionnement et l'analyse du graphe. Les

figures 2-7 [19] et 2-8 [19] montrent deux raffinements possibles pour les structures de branchements et de boucles. La liberté est par la suite donnée au concepteur de pouvoir intégrer le contenu de chacun des nouveaux nœuds ainsi créés ou de les laisser tels quels. Ces nouveaux nœuds correspondent à des étapes plus spécifiques des structures de contrôle présentées. En prenant l'exemple du branchement étendu à la figure 2-7, les nœuds Hi et Ho correspondent respectivement à l'entrée et la sortie du nœud hiérarchique, BS (Branch Start) contient un traitement commun préalable aux deux branches B1 et B2 de la structure conditionnelle tandis que BE (Branch End) implique une série d'opérations devant être effectuée après l'exécution de l'une ou l'autre des deux branches. Tous ces nouveaux nœuds servent à représenter de manière plus explicite la structure de branchement complète contenue dans le nœud FB (Full Branch). Le même principe est utilisé pour la représentation de la structure de boucle dans la figure 2-8.

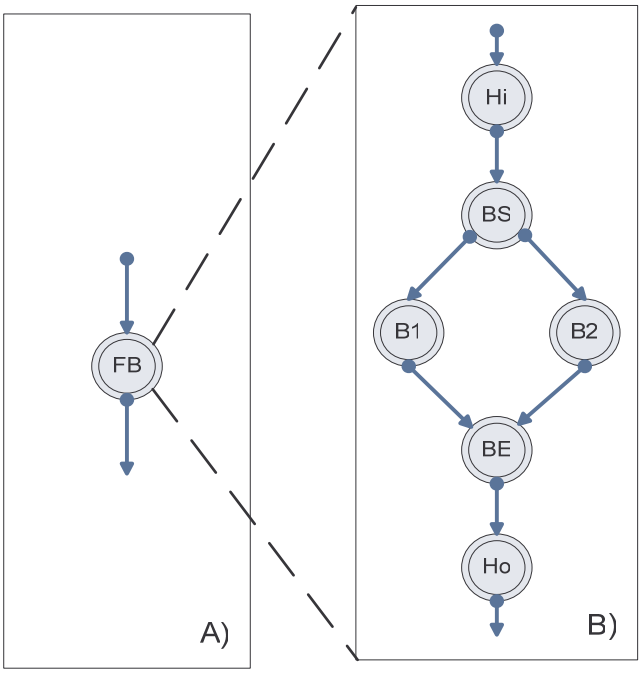


**Figure 2-7. Transformations de branchement : A) Branchement complet B) Branchement étendu**

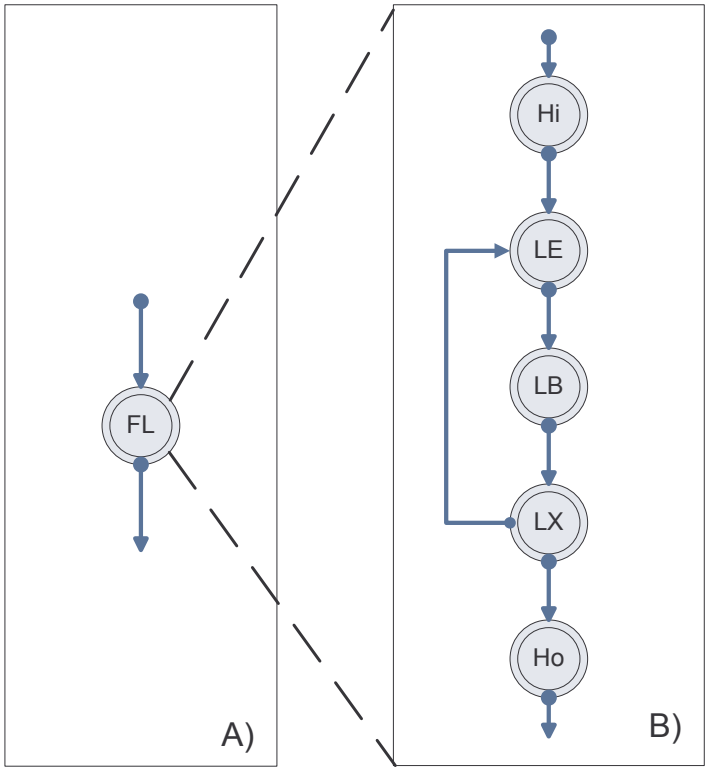


**Figure 2-8. Transformations de boucle : A) Boucle complète B) Boucle étendue**

Toujours dans le cadre du partitionnement logiciel/matériel, Wu et al. [33] introduisent le concept de hiérarchisation du CDFG en permettant à un nœud de contenir un sous graphe du même type et ce de manière récursive. Cette hiérarchisation peut être appliquée à toutes les structures de contrôle présentes dans une application conventionnelle. Leur article présente également la définition d'un nouveau type de nœud, soit le nœud de transport. Celui-ci est responsable de représenter les ressources de communications du système à implémenter. Le développement et l'inclusion de ce type de nœud dans le CDFG leur procure une flexibilité accrue dans la caractérisation de différents modes de transmission de données.

On peut brièvement conclure que le CDFG est en utilisation dans plusieurs domaines connexes au monde des compilateurs et que ceux-ci poussent souvent les concepteurs à redéfinir les bases mêmes du graphe classique afin d'arriver à un résultat final plus adapté à leurs champs d'intérêt.

## 2.3 Partitionnement et ordonnancement

Un autre aspect contenu dans ce mémoire et ayant déjà été couvert extensivement par la littérature académique concerne le partionnement d'application et son ordonnancement. La présente section portera sur l'explication sommaire de quelques principes et exemples, établis dans la littérature, portant sur l'application de ces deux disciplines dans le domaine des MPSoC. Comme mentionné précédemment dans la section sur les particularités des MPSoC, la difficulté dans le partionnement d'une application découle du parallélisme offert par ces circuits émergents face à des algorithmes codés de manière séquentielle. Pour solutionner ce problème, Xue et al. [34] proposent une méthode d'ordonnancement dynamique basée sur un partionnement effectué au préalable et qui prend en compte les différentes ressources, surtout en termes de consommation de mémoire et d'effort de calcul, requises par l'application. La première phase de leur technique consiste à prévoir les caractéristiques des différentes applications à traiter. La deuxième étape est effectuée par un outil de partitionnement dynamique assignant un nombre de processeurs, ainsi qu'une quantité de mémoire aux différentes tâches concurrentes. La figure 2-9 présente quelques stratégies de partionnement sur un MPSoC contenant 8 processeurs distincts. Le premier cas correspond à toutes les unités de traitement disponible (P1 à P8) ainsi que la quantité de mémoire commune disponible. L'assignation d'une tâche à un regroupement de processeurs ainsi qu'à une quantité de mémoire est représentée par les différents sous-groupes délimités dans la figure 2-9 par différents codes de couleur dans les trois cas subséquents.

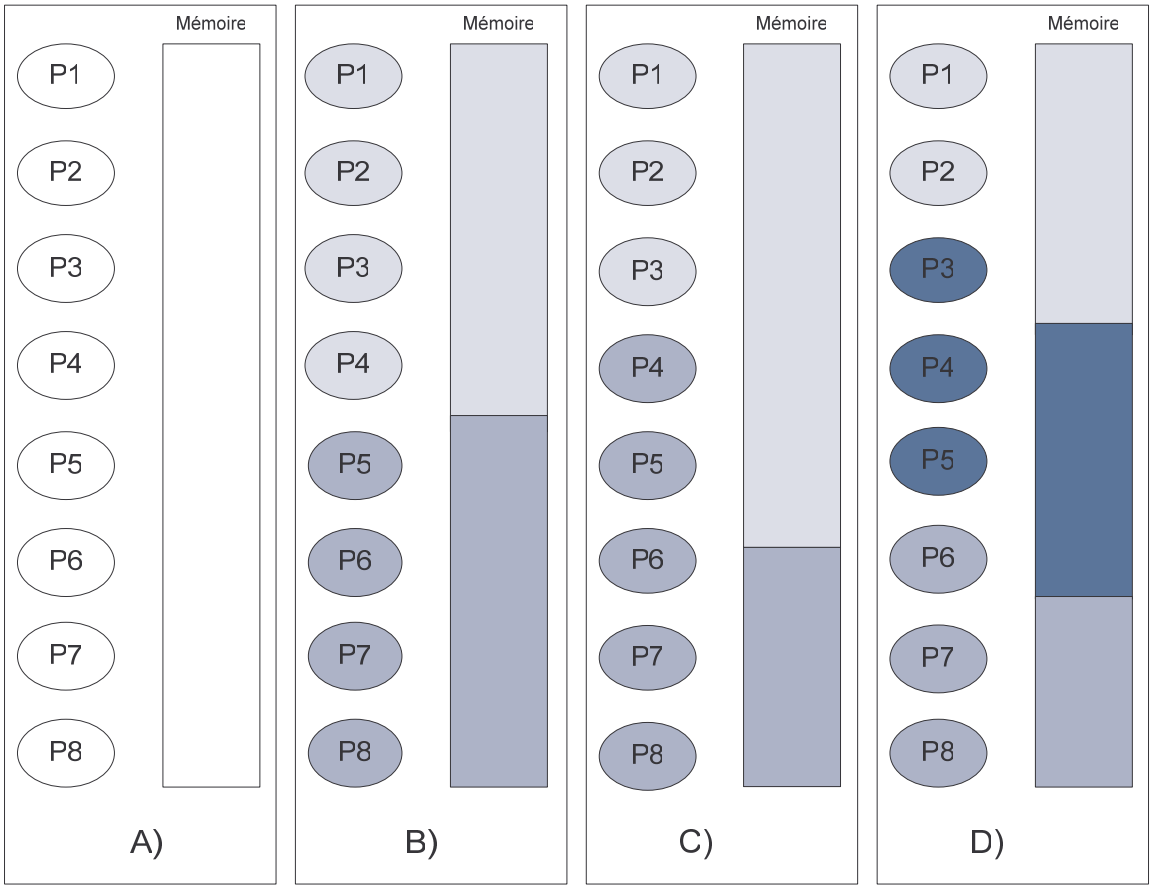


**Figure 2-9. Exemples de partionnement : A) Architecture MPSoC à 8 processeurs B-C-D) Stratégies de partionnement**

La méthode présentée permet entre autres de pouvoir évaluer adéquatement différentes stratégies de partitionnement. Les résultats obtenus tendent à démontrer encore une fois l'importance d'une bonne stratégie de partitionnement en termes de processeurs et de mémoire. Il existe également de nombreuses études se penchant sur l'optimisation de la mémoire dans un contexte multiprocesseurs [18, 21, 23]. Par contre, les techniques développées dans chacun de ces articles ne prennent pas en compte les besoins relatifs à l'effort de calcul, contrairement à notre approche et à celle, très similaire, proposée par Xue et al. [34]. Par exemple, Li et al. [21] essaient de démontrer qu'en augmentant la localité des données requises par une application dans un processeur donné, les

performances globales d'un MPSoC, de type matrice de processeurs, s'en trouvent améliorées. Leur méthode se base sur les concepts suivants. En augmentant la réutilisation des variables et données locales à un processeur, il est possible de diminuer la fréquence et la quantité de données devant être transférées à l'aide de communications entre les processeurs. Également, afin d'amortir les coûts de synchronisation entre les processeurs, il devrait y avoir une quantité suffisante de calculs à effectuer entre deux étapes de synchronisation. Finalement, une bonne répartition de l'effort de calcul sur chaque processeur tend généralement à augmenter le taux d'utilisation des processeurs de manière à passer le moins de temps possible en mode inactif. Toutes ces considérations ont été prises en compte lors de la transformation de leurs algorithmes vers une forme plus adaptée aux MPSoC. La transformation principale de leur méthode est le déroulement de boucles afin d'obtenir une application encore plus parallèle, tout en essayant de regrouper l'utilisation des données similaires sur un même processeur.

On retrouve également dans la littérature certaines méthodologies très similaires à celle du PBD utilisées dans le même but que le projet décrit dans ce mémoire, soit l'estimation de performances d'algorithmes sur une implémentation matérielle sélectionnée. En effet, ces méthodes sont principalement caractérisées par une représentation de l'application sous une forme donnée et une caractérisation de la partie matérielle. En effet, Russel et al. [26], décrivent une méthode d'analyse statique pour explorer l'implémentation de différentes architectures. Chacune des composantes de l'architecture est d'abord définie par rapport aux délais associés à chaque étape de traitement (calcul, transfert de données, …). Par la suite, les différentes parties de l'application sont réparties parmi les composantes matérielles, afin d'en tirer les performances globales. Cette technique, contrairement à celle proposé dans ce mémoire, ne tient compte que des dépendances de contrôle et n'est pas effectuée de manière totalement automatique. Hwang et al. [17] ont quant à eux avancé une méthode dynamique d'estimation des performances pour des MPSoC hétérogènes. Comme dans le présent cas, l'application est représentée sous la forme d'un graphe de type CDFG. Par contre, seul le délai engendré par l'exécution des

blocs de base est comptabilisé pour être transmis à la modélisation de la plateforme matérielle formalisée à l'aide de SystemC. Bien que leur analyse soit dynamique, les résultats obtenus sont du même ordre de grandeur que ceux produits et présentés dans le cadre de ce mémoire, mais sans tenir compte des besoins en mémoire de l'algorithme et de la capacité de la plateforme. Finalement, dans le but de pouvoir sélectionner le processeur approprié ou d'estimer les changements à effectuer avec l'utilisation de processeur configurable, Gupta et al. [12] propose une méthode basée également sur l'utilisation de SUIF pour extraire les caractéristiques de l'application. Une série de paramètres similaires à ceux extraits par l'outil produit dans le cadre du projet OPERA viennent caractériser la représentation de l'application. Ces paramètres ne viennent cependant pas montrer le détail des nœuds représentant les instructions individuelles. En se basant sur ces derniers paramètres, un ordonnanceur calcule le nombre de cycles requis pour l'exécution de l'algorithme. Bien que la technique employée par les précédents auteurs soit proche de celle mise de l'avant dans ce mémoire, on peut voir que l'erreur relative des estimations est beaucoup plus importante que celle présentée dans la suite du mémoire. Il est à noter que la visualisation de la représentation intermédiaire de l'application étudiée et que le calcul de la consommation de mémoire, ainsi que le partitionnement automatique selon la quantité de mémoire disponible, sont des fonctionnalités qui ne sont présentes que dans la méthode proposée par ce mémoire.

Toutes ces techniques en viennent à démontrer la grande complexité de la programmation des MPSoC, puisque l'on peut tenter de résoudre ce problème en considérant ou non un certain nombre de facteurs différents (effort de calcul, quantité de mémoire requise, localité des données, parallélisme, …), et en obtenant des améliorations très variables entre chacune des méthodes proposées.

## 2.4 Applications

Cette dernière section du chapitre de mise en contexte passe brièvement en revue les différentes applications qui ont été utilisées comme banc d'essai dans notre projet. Seul

un aperçu des algorithmes sera présenté dans ce mémoire. En effet le mémoire de Fredéric Plourde [25], un autre étudiant du projet Opéra, détaille explicitement le fonctionnement de nombreuses applications dans le domaine des télécommunications, dont toutes celles utilisées dans la réalisation de ce mémoire. Les bancs d'essai employés dans le développement et le test de l'outil final, ainsi que pour la production des résultats présentés au chapitre 6 ont été produits par différents membres du projet OPERA dans le cadre de leurs travaux de recherches respectifs ou de demandes particulières provenant d'Octasic. Ces algorithmes sont les suivants :

- Maximum Vectoriel
- Addition Vectorielle
- FIR Complexe
- Filtre égaliseur adaptatif LMS complexe
- Décodeur de Viterbi
- Décodeur Turbo

On peut classer les algorithmes étudiés en trois groupes sur la base de leur fonctionnalité et de leur niveau de complexité. Le premier est constitué d'opérations vectorielles, soit le maximum et l'addition. La sélection de ces opérations est due à leur grande utilisation dans le secteur des télécommunications. Elles sont utilisées pour implémenter de nombreuses fonctions plus complexes. Elles ont une relativement faible complexité algorithmique. En effet, ce faible niveau de complexité permet une première vérification rapide du fonctionnement de l'outil global, car le nombre restreint d'opérations tend à produire des CDFG aux tailles réduites, donc plus faciles à analyser de manière compréhensible par un individu. La vérification des estimations des ressources mémoires est aussi plus aisée sur du code aux demandes plus minimes en mémoires. Le deuxième groupe formé des deux types de filtres permet d'atteindre des niveaux de complexité intermédiaire, tout en considérant une fonctionnalité complète. Les filtres demeurent encore une des applications les plus couramment implémentés sur des DSP, ce qui augmente la pertinence de leur utilisation pour fins de validation. Brièvement, ces deux

configurations de filtres peuvent être employées pour produire un vaste éventail de filtres plus particuliers comme filtre passe-bas, égaliseur ou bien de mise en forme [25]. Les deux dernières applications sont deux différents types de décodeur en usage dans les classes d'applications ciblées par le projet Opéra. Ces différentes formes de codage servent à lutter contre les perturbations apportées par le support d'une transmission en remplaçant le message par un autre moins vulnérable à ces mêmes perturbations. L'emploi de celles-ci fournit comme avantage de procurer un haut niveau de complexité tout en fournissant des estimations d'algorithmes qui seront éventuellement implémentés sur la plateforme matérielle. Il est également à noter que plusieurs applications synthétiques ont été développées lors de la phase de développement de l'outil. Celles-ci avaient pour but de tester plus adéquatement certaines hypothèses ainsi que de pouvoir vérifier les limites de l'outil. Ces applications seront introduites dans le reste de ce mémoire lorsque les résultats relatifs à leur emploi seront présentés.

# CHAPITRE 3

## L'ARCHITECTURE DU VOCALLO

Le processeur Vocallo, qui est l'objet d'étude central du projet Opéra dans le cadre duquel ce projet de maîtrise a été réalisé, s'inscrit dans les courants observables dans le domaine des DSP au cours des dernières années, autant dans les sphères académique qu'industrielle. La constante réduction de la taille des transistors permet une augmentation de la complexité, pouvant se traduire en gain de performance, lorsque celle-ci est utilisée de manière efficace. Plusieurs techniques de pointe sont en usage présentement dans le domaine des DSP et ont entre autres deux objectifs principaux, soient la réduction de la puissance consommée et l'accélération de traitement des données. Grâce à l'augmentation du nombre de transistors par circuit, il est maintenant possible de concevoir des MPSoC qui sont composés de plusieurs unités de traitement permettant d'augmenter le parallélisme inhérent d'une plateforme donnée. Ce parallélisme permet d'augmenter le nombre d'opérations possibles par cycle d'horloge donc d'effectuer un nombre de calculs effectifs plus grand pour une même période de temps. Quant à la réduction de la consommation de puissance, une méthode en émergence est l'utilisation de circuits asynchrones. Cette dernière est particulièrement utile lorsqu'appliquée aux unités de calculs, responsables pour la grande majorité de la consommation de puissance. En effet, ce type de circuit, étant donné son absence de signal d'horloge, peut ne consommer virtuellement aucune énergie en mode d'attente, quand les circuits sont réalisés avec une technologie CMOS statique de longueur de grille suffisamment élevée. Les circuits de cette classe sont généralement capables de revenir en mode actif de manière quasi instantanée. Lorsque cette dernière technique est combinée avec l'utilisation de design de type MPSoC, comme dans le cas de la présente architecture, la possibilité de pouvoir désactiver un ou plusieurs cœurs selon le niveau d'activité rend cette dernière option très avantageuse. Cette variabilité du niveau d'activité est explicable en invoquant que l'effort de calcul des algorithmes de

télécommunications est fonction des conditions du canal de transmission, ainsi que le nombre d'usagers à traiter dans un contexte de temps réel. Ce chapitre présente les caractéristiques principales du processeur Vocallo.

## 3.1 Spécification générale du Vocallo

La puce Vocallo est un circuit multi-cœurs développé par Octasic pour le traitement vidéo, de la voix et de données sur IP. Comme une partie importante de ce projet consiste à évaluer les performances d'applications choisies dans le domaine des télécommunications, ce chapitre sert à dresser un portrait complet de l'architecture du Vocallo, afin de permettre une bonne compréhension de la modélisation du processeur dans les chapitres subséquents. De par la structure du processeur, nous utiliserons une approche descendante pour sa description en commençant par les caractéristiques globales de la plateforme, pour finir avec les détails fins de l'implémentation de chacun des cœurs.

## 3.2 Architecture Inter-Noyaux

Au plus haut niveau d'abstraction, on peut définir le Vocallo comme un tableau homogène de processeurs reliés entre eux par un bus de communication. La figure 3-1 montre l'architecture à haut-niveau du Vocallo, ainsi que des autres composants externes nécessaires à son bon fonctionnement. Une courte section sera consacrée à chacun des éléments identifiés dans la figure 3-1, afin de bien saisir les différentes interactions et rôle de chacun des modules.

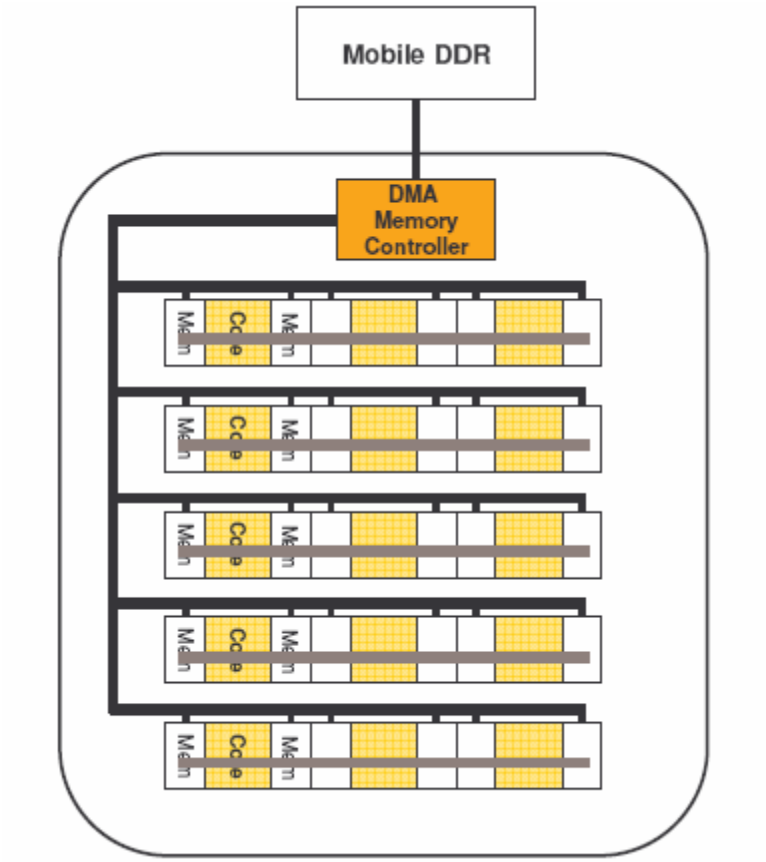


**Figure 3-1. Schéma bloc de l'architecture complète du Vocallo [25]**

### 3.2.1 Matrice de processeurs

La puce Vocallo est composée d'un ensemble de 15 noyaux identiques. Ces noyaux sont organisés selon un tableau composé de 5 rangées de 3 noyaux. Chaque noyau, contenant ses propres unités de calcul, effectue de manière autonome le traitement de données qui lui a été assigné. La description complète d'un noyau se retrouvera dans une des sections subséquentes.

Grâce à un ajout de circuiterie et au positionnement des noyaux, il est possible de désactiver les unités de traitement d'un noyau central et de répartir sa mémoire locale de manière égale entre ses voisins immédiats. Seuls les noyaux sur la même rangée peuvent être en droit d'utiliser cette mémoire supplémentaire. Cette dernière fonctionnalité peut être utile dans le cas où un partionnement logiciel implique un besoin en mémoire

supérieur à celui disponible de manière normale dans un noyau. Selon Octasic, aucune dégradation de performance ne résulte de ce type de transfert pour le noyau bénéficiant du surplus de mémoire.

### 3.2.2 Communication inter-noyau

La figure 3-1 montre que chaque noyau est connecté au seul bus responsable de tous les transferts de données entre les noyaux et l'extérieur de la puce, ainsi que des communications entre deux ou plusieurs noyaux. Les données sont transmises sur un bus d'une largeur de 32 bits à une fréquence de 1.5 GHz, qui est lui-même supporté par quelques signaux de contrôle. Ces signaux de contrôle permettent à chaque noyau de signaler, au contrôleur DMA (Direct Memory Access), le besoin de recevoir ou de transmettre des données. Ces transferts d'informations sont amorcés sur une base premier arrivé premier servi. Le contrôleur DMA est en charge de recevoir toutes les requêtes de transmission et de contrôler le bus selon les demandes à traiter. Un système de priorités est également disponible pour aider dans l'ordonnancement des communications, mais aucune documentation décrivant son utilisation précise n'est encore disponible à ce jour. Tout transfert interne doit passer par le contrôleur de mémoire qui est aussi responsable des communications vers la mémoire externe et des entrées et sorties de la puce. Il est à noter qu'un seul transfert d'information peut avoir lieu à un moment donné, puisque nous sommes en présence d'un seul bus, mais le Vocallo permet par contre de pouvoir envoyer les mêmes données à tous les noyaux ou à n'importe quel regroupement de ceux-ci de manière simultanée. Cette dernière forme de communication est dite de type universelle (broadcast). Il faut également mentionner que lorsqu'un noyau est impliqué dans une communication, peu importe qu'il soit en transmission ou en réception, ce noyau cesse complètement ses activités. Donc une communication universelle à tous les noyaux aura pour effet d'arrêter complètement l'effort de calcul de toute la puce durant toute la durée du transfert.

### 3.2.3 Circuits périphériques

Toutes les interfaces d'entrée et de sortie ainsi que la banque principale de mémoire sont implémentées à partir d'un FPGA externe à la matrice de processeurs. Les interfaces sont connectées directement à la mémoire externe par un bus pouvant faire varier sa largeur de 32 à 64 bits dépendamment du nombre de banques de mémoire présentes dans la mémoire externe. Cette dernière est constituée de Mobile-DDRAM, seul type supporté par le Vocallo. La vitesse de transfert de ce bus est de 166 MHz DDR (Double Data Rate). Comme dans le cas d'une communication entre noyaux, c'est le contrôleur DMA qui est en charge de tout transfert de données provenant ou en direction de la mémoire externe et des interfaces d'entrée et de sortie. Il est important de noter que les données provenant des interfaces d'entrée ne peuvent être acheminées directement vers un noyau, mais qu'elles doivent passer préalablement par la mémoire externe. Celle-ci joue un rôle de tampon entre les interfaces et la matrice de processeurs. Ceci veut également dire que suite au traitement effectué dans un noyau, l'information devant être retournée vers l'extérieur doit également passer par la mémoire externe. La communication limitée et obligatoire entre les interfaces et la mémoire externe implique une transmission supplémentaire causant des délais additionnels pour toute entrée et sortie de blocs de données.

## 3.3 Spécification d'un noyau

Cette section a pour but de détailler les composants internes d'un noyau de manière à permettre une meilleure compréhension de son fonctionnement. Chacun des noyaux travaille de manière autonome équivalant à un processeur DSP conventionnel. Un noyau est composé principalement d'une mémoire locale, d'une banque de registres ainsi que de 16 unités de calcul distinctes. Le jeu d'instructions supporté par cette configuration de DSP est d'inspiration RISC, tandis que l'agencement parallèle des unités opératives rend son fonctionnement similaire à un modèle plus traditionnel de DSP, soit le MIMD. Gupta et al. [11] décrivent une architecture analogue de manière à exploiter le parallélisme

extrait d'algorithmes séquentiels. Par contre, la nature asynchrone des ALU entraîne deux distinctions majeures par rapport à une architecture MIMD conventionnelle. Premièrement, le temps de complétion d'une instruction est proportionnel à sa complexité ainsi qu'aux données traitées. Le système de contrôle des unités opératives basé sur des jetons rend la cadence de lancement hautement variable. Plus de détails sur le fonctionnement de chacun des éléments d'un noyau seront discutés dans les sous-sections suivantes. L'architecture d'un noyau est également présentée à la figure 3-2.

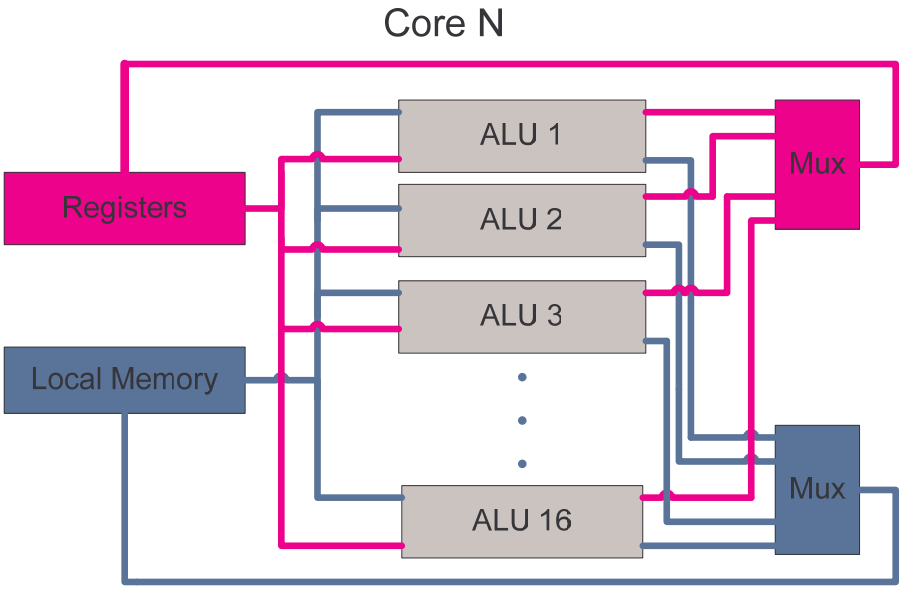


**Figure 3-2. Schéma bloc de l'architecture d'un noyau du Vocallo**

### 3.3.1 Mémoire Locale

Chacun des noyaux possède sa propre mémoire locale. Chacune de ces banques de mémoire contient 96 Koctets et peut être accédée à une fréquence de 1.5 GHz. Dans le cas où un noyau central partage sa mémoire locale avec les noyaux adjacents, ceux-ci se voient octroyer un supplément de 48 Koctets, chacun pour un total de 144 Koctets. Il est important de noter que la mémoire locale ne possède qu'un seul port d'accès et qu'elle ne peut donc pas lire et écrire des données en même temps. Ceci est la principale cause de l'arrêt local de calcul lorsqu'une communication est engagée entre le noyau et tout autre interlocuteur. L'accès simple à la mémoire locale aura aussi pour effet de limiter

l'acheminement des données vers ou à partir des unités opératives. Cette dernière conséquence sera étudiée plus en détails dans la partie sur la modélisation d'un des noyaux. Comme l'architecture d'un noyau est de type Von Neuman, la mémoire locale est responsable d'entreposer le code et toutes les données nécessaires, soient les valeurs d'entrée et de sortie ainsi que les variables temporaires utilisées par les différentes étapes du calcul.

### 3.3.2 Registres

La banque de registres décrite dans cette section est identique et propre à chacun des noyaux. Le tableau suivant présente le type et la quantité des registres appartenant à un noyau.

**Tableau 3-1. Type et quantité des registres d'un noyau**

| Type | Quantité |
|---|---|
| Usage Général | 63 |
| Null (r0) | 1 |
| Temporaire | 63 |
| Instruction Output (IOR) | 1 |

Les registres à usage général d'une largeur de 32 bits peuvent être utilisés pour stocker des données, des pointeurs d'adresse ou des conditions. Le registre « null », qui est effectivement le premier registre général, retourne toujours la valeur de 0 en cas de lecture. L'écriture dans ce registre n'entraîne aucune modification de son contenu. Ce registre est utile dans le cas où une instruction ne requiert pas de registre en entrée. Par exemple, l'instruction suivante est utilisée pour transférer le contenu du registre 24 (r24) dans le registre 4 (r4).

or r4, 24, r0

Les registres temporaires sont associés aux bascules de sortie des unités opératives. Les valeurs contenues dans ces 63 registres ne peuvent être accédées comme opérande d'entrée que dans les 16 opérations suivantes sans perdre leur validité. Le registre d'instruction en sortie est un registre temporaire particulier. Seulement quelques types

d'opérations sont susceptibles d'utiliser le registre IOR comme quatrième opérande. En effet, certaines opérations arithmétiques ou de décalage ont besoin de 3 opérandes d'entrée, tandis que la sortie est écrite systématiquement dans ce registre.

Comme le jeu d'instructions comporte plusieurs instructions du type SIMD, soit la possibilité d'effectuer la même opération sur plusieurs données de manière simultanée, il est possible de séparer les registres en 2 valeurs de 16 bits ou en 4 valeurs de 8 bits. Ceci se fait grâce à l'utilisation des instructions précédées du préfixe P dans le jeu d'instructions. L'usage de ces instructions cause par contre une baisse de la précision par rapport à la même instruction sur un registre complet de 32 bits.

### 3.3.3 Architecture des ALU

Chaque noyau est composé de 16 unités opératives (ALU) disposées en parallèle. Ces dernières ont également la particularité d'être constituées par de la logique asynchrone. Chacune des unités possède son propre pipeline de traitement des instructions afin de rendre possible l'exécution simultanée de plusieurs instructions. Ces unités possèdent un jeu d'instructions RISC augmenté de plusieurs instructions plus complexes particulièrement utiles dans les applications de traitement de signal. Toutes les unités d'un noyau se partagent certaines ressources, soient la mémoire locale, les registres et l'unité étendue de calcul. Cette dernière unité est responsable d'effectuer les opérations plus complexes, comme les instructions de type SIMD. Chaque unité est responsable d'effectuer pour son propre compte les étapes suivantes présentées dans le tableau 3-2.

**Tableau 3-2. Étape d'exécution d'une instruction**

| Étape | Opération |
|---|---|
| 1 | Chargement de l'instruction |
| 2 | Lecture des registres |
| 3 | Lecture de la mémoire locale |
| 4 | Traitement des données |
| 5 | Écriture des registres |
| 6 | Écriture en mémoire locale (Store) |
| 7 | Calcul de l'adresse de branchement |

Le contrôle de cet agencement d'unités de traitement est assuré par un système de jetons qui sont échangés par les unités opératives. Les différents jetons représentent une étape de l'exécution d'une instruction. Ces jetons sont passés à l'unité suivante lorsque l'unité courante utilise le jeton en question ou ne prévoit pas en avoir besoin. En prenant comme exemple le jeton d'écriture des registres, une unité doit attendre que l'unité précédente ait fini d'écrire ses données ou qu'elle n'ait aucune donnée à écrire pour pouvoir accéder à la banque de registres et terminer l'exécution de son instruction. Ceci a pour effet de ralentir la complétion d'une instruction lorsque l'instruction contenue dans l'unité de traitement précédente n'est pas terminée. Puisque l'utilisation de l'unité de calcul étendu requiert le jeton correspondant, il ne peut y avoir qu'une seule unité opérative pouvant exécuter une opération complexe à un instant donné.

## 3.4 Conclusion

Le présent chapitre avait pour objectif de décrire en détails la plateforme étudiée afin de mieux comprendre son effet sur la méthodologie employée. La complexité apparente jumelée à la combinaison de plusieurs courants (MPSoC, Asynchronisme, …) du domaine des DSP rend nécessaire un modèle particulier pour bien estimer les performances de cette architecture. En effet, les outils et techniques traditionnels ne permettent pas de modéliser précisément cette architecture.

# CHAPITRE 4

## LES GRAPHES DE FLOTS DE CONTRÔLE ET DE DONNÉES

La première étape du projet décrit dans ce mémoire a été le choix d'une représentation intermédiaire afin de capturer les caractéristiques majeures d'une application décrite en langage C. Étant donné la complexité d'un cœur du Vocallo au niveau de l'interaction entre ses unités opératives (ALU), la représentation choisie se devait de permettre la visualisation à niveau de granularité très fin. Comme les algorithmes étudiés sont codés en C, qui est lui-même un langage qui décrit une fonctionnalité à un niveau relativement élevé, un effort devait être fourni pour en arriver à une description plus propice à notre analyse. Pour solutionner cette problématique, nous nous sommes basés sur quelques principes qui sont présentement en utilisation dans les compilateurs modernes. En effet, un des principes de base d'un compilateur est de faire passer un algorithme d'un niveau plus haut, donc plus facile à coder et comprendre, vers un langage spécifique à une architecture donnée de processeur. De plus, la plupart des compilateurs modernes se servent d'une représentation intermédiaire de type CDFG pour y arriver comme le montre entre autre la littérature présentée dans le chapitre 2. L'utilisation de la deuxième version du compilateur expérimental SUIF, développé par l'université Stanford, nous permet à l'aide de modifications automatisées de passer d'un algorithme en C vers sa représentation sous la forme d'un CDFG. En effet, cette forme de graphe expose notamment les dépendances de données entre les instructions, en combinaison avec une modélisation d'un des DSP de la puce Vocallo. Elle nous permet d'arriver à une estimation des ressources nécessaires à l'exécution d'un algorithme en C sur un des DSP. Cette évaluation des ressources requises est calculée en termes de temps d'exécution et de la quantité de mémoire essentielle au bon fonctionnement de l'algorithme.

Le chapitre 5 se concentrera sur l'explication du processus d'extraction des besoins en mémoire, tandis que les différentes modélisations du DSP et leurs techniques respectives

d'ordonnancement menant aux approximations du temps d'exécution seront présentées et justifiées dans le chapitre 6. Le chapitre 4 sera quant à lui consacré à la description du procédé de production de ce type de graphe et il est organisé selon ses principales étapes, soient la présentation sommaire du cadre de travail, l'extraction des dépendances de données, la création de structures hiérarchiques et l'insertion de différentes métriques de performance et de caractérisation.

## 4.1 Cadre de travail

### 4.1.1 SUIF2

Afin de simplifier la création de notre CDFG, SUIF2 a été sélectionné comme environnement de travail. De manière globale, ce système est une infrastructure de compilateur conçue pour supporter la recherche et le développement de techniques de compilation basées sur une représentation de programme intitulé SUIF (Stanford University Intermediate Format). Ce système vise la réutilisation de code, en fournissant des structures de données et des méthodes utiles à la création de nouvelles routines de compilation, tout en permettant une interaction aisée entre les étapes de compilation dans un même environnement. La version 2 de SUIF diffère grandement de la première édition. Cette première variante, plutôt limitée, focalisait originalement sur l'analyse à haut-niveau de programme en C et en Fortran. SUIF2, quant à lui, rend possible l'analyse de nouveaux types de langages, comme les langages orientés objet, tel le C++ et il permet également une meilleure optimisation au niveau machine. Les concepteurs de SUIF2 l'ont élaboré de manière à pouvoir supporter facilement de nouveaux sujets de recherche en ajoutant de nouvelles bibliothèques de routine conformément à la syntaxe établie. Le fait d'avoir la capacité de développer de nouvelles routines dédiées à notre univers de recherche, pouvant s'intégrer aisément à un environnement de compilation déjà riche, combiné à la possibilité d'une grande optimisation à bas niveau, niveau requis pour la modélisation de notre DSP, ont été les facteurs décisifs qui nous ont poussés vers l'utilisation de ce compilateur expérimental.

### 4.1.2 MACHSUIF

Afin de pouvoir obtenir la représentation voulue, l'utilisation de la bibliothèque plus spécialisée MACHSUIF (contraction de *Machine* SUIF) a été nécessaire. Cette extension de la deuxième version du compilateur SUIF a été implémentée pour supporter la création de la partie dorsale du compilateur, alors que la partie frontale est toujours desservie par l'environnement SUIF2 de base. Da manière plus spécifique, Machine SUIF est une infrastructure flexible et extensible permettant la construction et la manipulation de représentation d'algorithmes exprimés au niveau machine et de pouvoir les exprimer après optimisation sous forme de code assembleur, d'objet binaire ou de retour vers un code en C. La bibliothèque vient à la base avec la possibilité de produire du code optimisé pour des machines basées sur les architectures Alpha et x86. Les concepteurs de ces bibliothèques ont laissé l'opportunité de définir de nouvelles architectures ou de modifier celles déjà présentes dans le cas d'une utilisation d'une machine similaire. Par contre, étant donné la complexité et le temps requis pour la création d'une nouvelle description complète d'une machine, nous avons utilisé les routines associées à la machine Alpha afin d'arriver à une description en langage assembleur de nos applications. Ce choix a été également justifié par la complexité du jeu d'instructions des DSP présents sur la puce Vocallo, ainsi que par l'information parfois limitée sur certains aspects des instructions qui le composent.

### 4.1.3 Flot de conception du CDFG

Le flot complet de conception de notre version du CDFG est présenté à la figure 4.1. Il est à noter que les algorithmes en C, utilisés comme principal intrant du flot, peuvent provenir majoritairement de deux sources. La première est bien entendu tout algorithme déjà implémenté directement en C. En deuxième lieu, nous avons jugé pertinent de pouvoir analyser des applications exprimées à un niveau encore supérieur au C à l'aide de Matlab. Pour ce faire, nous avons recours à l'outil RTW (Real-Time Workshop) permettant la traduction automatique de blocs Matlab vers du code en C standard. Ceci

est important, puisque cette forme de représentation est souvent utilisée dans le domaine des télécommunications et qu'elle peut souvent s'obtenir rapidement et relativement aisément par rapport à tous les autres langages pour lesquels la description d'une application se fait à un niveau inférieur.

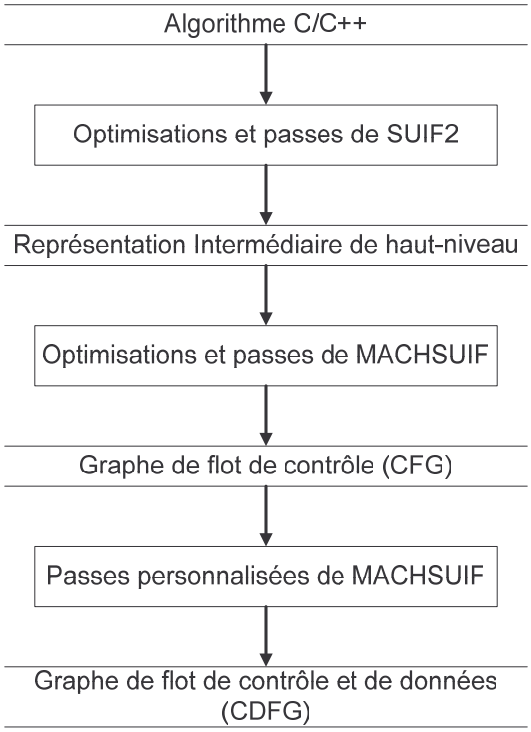


**Figure 4-1. Flot de conception d'un CDFG**

La lecture du code C en entrée et son premier abaissement vers le format intermédiaire SUIF sont effectués par SUIF2 sans aucune modification aux routines de base. Par la suite, cette dernière représentation est traitée par l'extension MACHSUIF afin de produire un CFG traditionnel. La création de ce CFG, conforme aux particularités déjà énoncées dans le chapitre 2 de ce type de graphe, s'effectue grâce à la bibliothèque CFG. De manière plus précise, cette dernière bibliothèque fournit une abstraction d'un graphe de flot de contrôle exprimée à l'aide de nœuds contenant une liste d'instructions machine. Ce format de graphe est utilisé pour remplacer la liste d'instructions linéaires représentant l'algorithme en entrée lors de sa compilation. Cette bibliothèque supporte également le

retour vers une forme linéaire ainsi que toute transformation relative à l'ajout, le retrait, le changement de connexions de différents nœuds en donnant aussi un contrôle d'une granularité plus fine pour la modification des instructions contenues dans chacun des nœuds. Dans un cadre plus traditionnel de compilation, l'utilité du CFG repose sur son abstraction de la linéarité contenue dans tout algorithme. En effet, le partionnement d'une longue liste d'instructions séquentielles en blocs de base imposés par les dépendances de contrôle permet un réordonnancement plus efficace sans avoir à séparer et rattacher des séquences linéaires de code. Par contre, dans le cadre de ce projet, notre intérêt par rapport à cette librairie s'est plutôt porté sur la possibilité d'accéder aux instructions de chacun des nœuds afin d'en extraire les dépendances de données. Donc chaque nœud du CFG est exploré de manière à en ajouter un nouveau pour chacune des opérations élémentaires et de les relier selon les dépendances trouvées afin de former le CDFG désiré. Cette dernière phase de transformation est effectuée par de nouvelles routines venant enrichir la bibliothèque CFG conventionnelle développée exclusivement pour et dans le cadre du projet OPERA. La section 4.2 servira à décrire de manière plus explicite les bases et le fonctionnement de la routine responsable de l'extraction des dépendances et de la création des nœuds nécessaires à l'obtention du CDFG. Il est à noter que la création du CDFG se fait de manière totalement automatique pour toutes les applications en C fournies en entrée. Suite à sa création, les structures du CDFG (nœuds et arcs) sont exprimées à l'aide du GDL (Graphical Description Language) pour permettre de visualiser le graphe obtenu. La production de sa forme visualisable et le choix des annotations seront explicités dans les sections subséquentes. Le CDFG correspondant est après coup envoyé à l'ordonnanceur responsable de l'estimation des performances relatives au temps d'exécution basé sur la modélisation d'un DSP qui sera présenté dans la section 6.

## 4.2 Processus de création du CDFG

Le processus de création de la représentation graphique des algorithmes étudiés est rendu plus simple grâce à l'utilisation de SUIF2. La seule opération, qui se devait d'être rajoutée aux bibliothèques de base, afin de produire un CDFG, consiste à extraire les dépendances de données présentes dans les instructions de chacun des blocs de base extraits par SUIF2. Il est aussi important de conserver l'ordonnancement de ces mêmes blocs de base dicté par les dépendances de contrôle que l'on trouve dans l'application source. La figure 4-2 présente l'algorithme global de la conversion du CFG produit par SUIF2 vers le CDFG. La prochaine sous-section viendra expliciter plus en détails les étapes avancées dans la figure 4-2. Par la suite, les différentes particularités prises en compte dans l'extraction des dépendances de données seront exposées.

```
Convert_CFG_to_CDFG(CFG)
1. cfg_size = nodes_size(cfg);
2. For (node_start; node_end; node++)
   {
3.       if (instruction_size(node) == 0)
         {
4.               adjust_control_dependencies();
         }
         else
         {
5.               add_new_start_basenode(CDFG);
6.               adjust_predecessors_control_nodes();
7.               node_size = instrs_size(node);
8.               For (instruction_start; instruction_end; instruction ++)
                 {
9.                       if (instruction == empty) {instruction ++; }
10.                      else if (instruction == control) {break; }
                         else
                         {
11.                              remove_instruction(instruction, node);
12.                              add_new_basenode(instruction);
13.                              extract_data_dependencies(instruction,node);
14.                              adjust_data_dependencies();
                         }
                 }
15.              add_new_end_basenode(CDFG);
16.              adjust_successors_control_nodes();
         }
   }
17. remove_unreachable_nodes(cdfg);
```

**Figure 4-2. Algorithme de conversion du CFG vers le CDFG**

### 4.2.1 Algorithme de conversion du CFG vers le CDFG

L'algorithme de conversion prend comme entrée le CFG de l'application source décrite à l'aide de certaines classes définies dans les bibliothèques de MACHSUIF. Brièvement, ce CFG est un ensemble de nœuds représentant chacun un bloc de base. De nombreuses méthodes variées permettent de parcourir le graphe, de sélectionner un ou des nœuds

ainsi qu'en rajouter ou en enlever. Elles viennent également permettre de relier facilement ceux-ci à l'aide de plusieurs types d'arcs ou de les séparer lors de toute transformation jugée pertinente. Chacun de ces nœuds est composé, entre autres, d'une liste séquentielle d'instructions. La classe représentant ces instructions permet l'inspection du type d'opération ainsi qu'une liste des opérandes utilisés par chacune de celle-ci. L'énumération de ces derniers concepts sera utile à une meilleure compréhension des principes énoncés dans la description plus approfondie qui suit.

L'approche implémentée telle que présentée à la figure 4-2 consiste par l'exploration systématique et la transformation de tous les blocs de bases du CFG. Chacun des nœuds est analysé et modifié dans la boucle principale de l'algorithme représenté à l'étape 2. Les CFG produits peuvent être composés de nœuds vides ne contenant aucune instruction servant uniquement à contenir certains commentaires. Même si ceux-ci sont peu nombreux, il faut en tenir compte puisqu'ils peuvent parfois refléter certaines dépendances de contrôle. Les nœuds d'entrée et de sortie du graphe source sont toujours représentés par ce type de nœuds et ils sont traités de manière spécifiques par l'opération effectuée à l'étape 4. Lors de cette même opération, les nœuds dépendants du nœud vide se voient attribuer une dépendance pour chacun des nœuds dont le nœud vide était dépendant. Suite à cette opération, il est à noter que le nœud vide se trouve à être supprimé. La première étape dans le cas des blocs de base contenant des instructions consiste à ajouter un nouveau nœud qui servira de point de départ afin de connecter tous les prédécesseurs. Ces nœuds portent la mention « START » dans le graphe final. Par la suite, la liste d'instructions contenue dans le nœud sélectionné est parcourue en s'attardant sur chacune des instructions atomiques. Les commentaires représentés par des instructions vides n'ayant entre autre aucun code d'opération ne sont pas traités et ils sont laissés tels quels dans le bloc de base d'origine. Les étapes 11 et 12 sont responsables d'enlever les instructions utiles et de les insérer dans un nouveau nœud. Par la suite, les dépendances de données de l'instruction sont extraites par rapport aux instructions qui ont déjà été traitées dans la même liste d'instructions ou en d'autres mots dans le même

bloc de base. Le processus d'extraction des dépendances de données, effectué à l'étape 13, est présenté en détails dans la section suivante. Les dépendances ainsi trouvées sont exprimées en reliant les deux instructions dépendantes à l'aide d'un arc lors de l'opération 14. Lorsqu'aucune dépendance n'est trouvée, un arc est ajouté entre l'instruction courante et le nœud de début portant la note « START ». Suite au traitement de chacune des instructions, un nœud de fin est ajouté portant la mention « END » dans le graphe final. Ces nœuds de début et de fin servent à délimiter la présence d'un bloc de base, maintenant représenté par un DFG. Ce nœud est utilisé à l'étape 16 pour relier toutes les instructions qui n'ont aucun successeur afin de respecter les dépendances de contrôle imposées par la structure de l'algorithme. Ces dernières étapes sont répétées successivement sur tous les blocs de bases du CFG originel. L'ultime étape consiste à retirer tous les nœuds qui ne sont maintenant plus connectés. Ces nœuds correspondent aux blocs de base du CFG qui sont déconnectés du graphe et qui contiennent maintenant une liste d'instructions vides. La section suivante fournira les concepts de base utilisés dans le processus d'extraction des dépendances de données.

### 4.2.2 Extraction des dépendances de données

L'extraction des dépendances de données est un processus typiquement effectué sur des applications codées à haut niveau par des compilateurs optimisant. En effet, la littérature scientifique sur le sujet montre entre autres son importance dans plusieurs types d'optimisation, comme dans l'utilisation de la technique de déroulement de boucle. Par contre, peu de recherches ont approfondi l'analyse des dépendances de données pour du code en langage assembleur [5]. À l'aide des structures disponibles dans MACHSUIF, chaque instruction est généralement exprimée sous la forme suivante :

opcode src1 src2 dest

L'opération « opcode » est effectuée sur les opérandes source (src1 et src2) et elle écrit le résultat à l'endroit indiqué par l'opérande de destination (dest). Le deuxième opérande

source est optionnel pour plusieurs types d'opérations comme les chargements (load) et les sauvegardes (store). Les opérandes peuvent quant à eux prendre trois formes distinctes, soient un registre, une adresse mémoire ou une constante. Lors de l'étape 13 de la figure 4.2, chacune des opérandes, source et destination, sont sélectionnés individuellement de manière à être comparés aux opérandes des instructions déjà extraites de la liste d'instructions du bloc de base. Un arc signifiant une dépendance est ajouté entre l'instruction courante et la dernière instruction ayant recours au même espace mémoire ou au même registre.

Une exception vient quand même s'imposer quant à l'ajout des arcs de dépendances. En effet, aucune dépendance n'est présente entre deux instructions où les deux opérandes qui pointent vers le même espace mémoire sont deux opérandes sources tel qu'il sera expliqué plus bas dans cette section. Il est à noter que les opérandes constants ne sont pas considérés, puisqu'ils ne peuvent engendrer de dépendance de données. Aucune détection de dépendance n'est effectuée sur ce type d'opérande et le processus ne fait que passer au suivant. La détection de l'équivalence entre deux registres est aisée puisque l'opérande décrit par MACHSUIF ne contient que le numéro du registre facilement accessible pour fins de comparaison. La plus grande difficulté du processus de l'étape 13 consiste à déterminer si deux adresses en mémoire locale réfèrent au même espace malgré les différentes manières de l'exprimer. Les adresses mémoire sont formulées selon la structure suivante :

$$\text{Adresse} = \text{base} + \text{index} * \text{échelle} + \text{déplacement}$$

L'extraction de chacun des paramètres doit être effectuée afin de calculer l'adresse finale de l'opérande, tout en mentionnant que plusieurs sous-classes d'opérandes ne requièrent qu'une combinaison de ces paramètres. Il est donc possible de voir une adresse exprimée de plusieurs manières différentes, ce qui explique l'effort supplémentaire devant être fourni dans l'extraction des dépendances de données lors de l'utilisation de la mémoire locale.

En procédant de cette manière, les dépendances trouvées se trouvent toujours à satisfaire certaines exigences. On peut affirmer qu'une instruction S2 est dépendante de S1 si S2 est exécuté après S1 et que les conditions suivantes sont respectées :

1. Les deux instructions accèdent au même espace mémoire M.
2. Au moins l'une d'entre elles écrit dans M.
3. Entre l'exécution de S1 et S2, aucune autre instruction S3 ne vient écrire dans M.

En respectant ces trois conditions, il est possible de définir trois types distincts de dépendance de données. La dépendance de données prend le nom de dépendance vraie (« true dependency » ou RAW (*read after write*)) si S1 écrit dans M où S2 ira lire par la suite. Une anti-dépendance (« anti dependency » ou WAR (*write after read*)) est présente si S1 vient lire M alors que S2 vient y écrire, tandis qu'une dépendance de sortie (« output dependency » ou WAW (*write after write*)) est créée lorsque les deux instructions viennent écrire dans M. La figure 4-3 vient présenter à l'aide de courts exemples ces trois types de dépendances de données. Dans le cas a), les instructions S2 et S3 sont reliées par des dépendances de type « RAW » par rapport à leurs instructions précédentes respectives. Le cas b) présente une anti-dépendance entre les instructions S2 et S3, tandis qu'une dépendance de sortie peut être observée entre les instructions S1 et S3. Il est à noter que les deux derniers types de dépendances peuvent être éliminés en utilisant d'autres registres (« *register renaming* ») plutôt que de favoriser la réutilisation des mêmes registres. La seule limite de cette technique est le nombre physique de registres disponibles. Les vraies dépendances ne peuvent, quant à elles, être complètement enlevées, mais leur effet peut être limité en utilisant des techniques comme le déroulement de boucles ou le réordonnancement des opérations. Toutes ces dernières méthodes n'ont pas été implémentées, puisqu'elles ne sont pas supportées sur la puce Vocallo autant au niveau matériel que dans les outils logiciels.

```
S1:   mov   $vr1,$vr2
S2:   add   $vr2,10,$vr3
S3:   mov   $vr3,$vr4
```

a) Dépendance vraie

```
S1:   mov   $vr1,$vr2
S2:   add   $vr2,10,$vr3
S3:   mov   $vr4,$vr2
```

b) Anti-dépendance

```
S1:   mov   5,$vr2
S2:   add   $vr2,10,$vr3
S3:   mov   10,$vr2
```

c) Dépendance de sortie

**Figure 4-3. Exemples de dépendances de données**

## 4.3 Hiérarchisation

Le CDFG ainsi obtenu, représentant toutes les instructions de base est généralement très difficilement visualisable dans son ensemble, à moins de faire l'analyse d'algorithmes très simples. Malgré une annotation de base, il demeure toujours ardu de pouvoir sélectionner une partie spécifique du code afin de bien cibler les goulots d'étranglement. Pour palier à ce problème, nous avons choisi la création de structures hiérarchiques permettant de contrôler dynamiquement la granularité lors de la visualisation du graphe. De nombreuses informations pertinentes à la composition de chacun des types de nœud viennent également donner une meilleure compréhension de l'algorithme pour le concepteur. Les nouvelles structures hiérarchiques jouent également un deuxième rôle. Elles servent dans la phase d'ordonnancement à regrouper et propager les différentes métriques servant aux estimations finales ainsi qu'à une meilleure caractérisation de l'algorithme lors de la visualisation.

### 4.3.1 Algorithme de hiérarchisation du CDFG

Débutons par mentionner qu'il existe deux types de nœuds distincts. L'usage du premier type, portant l'appellation de nœud de base, est réservé à la caractérisation des instructions simples. Chacun de ces nœuds ne peut contenir qu'une seule instruction, ce qui les rend très similaires aux nœuds produits lors du passage du CFG vers le CDFG. De manière algorithmique, une nouvelle classe, dérivée de celle implémentée dans la version standard de MACHSUIF, permet une meilleure instrumentation, tout en gardant disponible les différentes méthodes préalablement établies. Le deuxième type porte le nom de nœud hiérarchique, puisqu'il représente toutes les instructions contenues dans la structure de contrôle représentée. Au niveau algorithmique, une toute nouvelle classe vient représenter ce type de nœud. Ce nœud peut donc contenir une simple liste de nœuds de base ou un ensemble de nœuds hiérarchiques qui peuvent à leur tour en contenir d'autres. Une description plus détaillée des types de nœuds ainsi que leurs caractéristiques sera exposée dans les sous-sections suivantes.

```
Hierarchise_Graph(CDFG)
1. create_global_node();
2. current_node = node_start;
3. While (current_node != node_end)
   {
4.         if(node == "START")
           {
5.                 identify_control_structure_start();
6.                 current_node++;
7.                 create_new_hnode(control_structure);
           }
8.         else if( node == "END")
           {
9.                 identify_control_structure_end();
10.                if((end_branch || end_loop || end_graph) && current_hnode == "sequential")
11.                {return();}
12.                else
13.                {current_node++; return();}

           }
14.        else
           {
15.                add_base_node(current_hnode);
16.                current_node++;
           }
   }
```

**Figure 4-4. Algorithme de hiérarchisation du CDFG**

De manière globale, la liste de nœuds représentant le CDFG est parcourue de manière à extraire les différentes structures de contrôle (boucle, branchement) et à les ordonner hiérarchiquement. L'étape 1 consiste à créer un nœud hiérarchique qui contiendra la totalité des instructions et des structures de contrôle de l'algorithme étudié. Lors de la visualisation et suite à l'ordonnancement du graphe, les informations fournies par ce nœud représenteront entre autres les performances de l'application sélectionnée, telles que transmises à la phase dynamique du projet global sous une représentation XML. La deuxième étape d'initialisation consiste à pointer vers le premier nœud de la liste du graphe, qui sera parcourue en entier à l'aide de la boucle instanciée à l'étape 3. Il faut à présent définir que les étapes 3 à 14 correspondent à une description partielle, à un niveau

inférieur, de la fonction de création de nouveau nœud hiérarchique invoquée lors de la phase 6. La hiérarchisation du graphe requiert en effet l'utilisation d'une fonction récursive complexifiant le processus de hiérarchisation. Les détails des différentes implémentations de cette fonction seront explicités pour les trois structures de contrôle dans la section suivante. De manière générale, il faut d'abord identifier le début et la fin des blocs de base, puisque ceux-ci constituent essentiellement les structures à représenter. Les étapes 13 et 14 sont responsables d'ajouter un nœud de base à une structure hiérarchique créée précédemment avant de passer au nœud de base suivant. Lors de l'identification d'un nœud de début d'un bloc de base, un nouveau nœud hiérarchique est créé selon le type de structure de contrôle lors des étapes 5 à 7. Lors de la création d'un nœud hiérarchique, on se trouve à descendre d'un niveau dans la structure globale du graphe. Dans les cas d'une détection d'un nœud de fin, on remonte d'un niveau à l'aide de l'instruction *return()* terminant ainsi un appel de création de nœud hiérarchique. Ceci est effectué par les instructions 9 à 12. L'étape 10 est responsable de pouvoir remonter jusqu'à la bonne structure de contrôle afin de pouvoir la compléter de manière adéquate. La section suivante servira à préciser l'utilisation des opérations des étapes 5 et 9 ainsi que de la création de nouveau nœud hiérarchique pour chacun des trois types de nœuds.

### 4.3.2 Structures de contrôle

Afin de pouvoir représenter toutes les configurations possibles d'un algorithme, trois structures de contrôle de base ont dû être définies. La première correspond à une simple exécution d'instructions séquentielles. La deuxième représente le concept de boucle, tandis que la dernière sert à exprimer un agencement de branchements conditionnels. Les prochaines sous-sections serviront à clarifier certaines étapes du processus de hiérarchisation du graphe en se servant d'exemples précis particuliers à chacun.

#### 4.3.2.1 Structure séquentielle

La structure séquentielle ne fait que représenter une liste d'instructions. Ce type de nœud hiérarchique ne peut que contenir une liste de nœuds de base, sauf dans un cas rencontré

dans le traitement des branchements, tandis que les deux autres types ne peuvent contenir que d'autres nœuds hiérarchiques. Lors des étapes d'identification des nœuds de début et de fin, ce type de nœud hiérarchique n'est reconnu que dans le cas où aucun autre type n'a été détecté. Lors de la création de ce type de nœud à l'étape 7, celui-ci est rattaché au nœud hiérarchique du niveau supérieur. Par la suite, les opérations 15 et 16 sont responsables d'ajouter les instructions élémentaires du bloc de base traité au nœud séquentiel. Il est à noter que les opérations 15 et 16 n'entrent en fonction que dans le cas de nœud séquentiel. Lors de la détection du nœud de fin, une fois tous les nœuds annexés, l'instruction 13 vient terminer la composition du nœud séquentiel et effectue le passage vers le prochain nœud hiérarchique.

#### 4.3.2.2 Boucle

La structure hiérarchique de boucle sert, comme son nom l'indique, à représenter des boucles. Une boucle est détectée à l'étape 5 si et seulement si un des prédécesseurs du nœud de début est situé plus loin dans la liste complète du graphique. L'exécution de l'opération 7 dans le cas d'une boucle se déroule comme suit. Contrairement au nœud séquentiel, deux nœuds hiérarchiques sont requis afin de bien représenter cette structure de contrôle. Le premier nœud, de type boucle, est crée et attaché à la structure hiérarchique du niveau supérieur. Par la suite, un nœud de type séquentiel est créé afin de recueillir les instructions élémentaires et est inséré dans le nœud de boucle instancié précédemment. Donc, la détection d'un nœud de boucle nécessite au minimum la création de deux nœuds. Le nœud de boucle n'est par contre pas limité à être composé d'un seul nœud et il peut contenir plusieurs autres nœuds hiérarchiques distincts. Suite à la création du nœud représentant une boucle, il est nécessaire d'obtenir les bornes d'itération minimale et maximale contenues dans un fichier texte externe à l'environnement SUIF. Ce fichier est produit automatiquement à l'aide d'un analyseur syntaxique en repérant des annotations insérées à cet effet dans le code de base. Le fichier ainsi obtenu contient les bornes d'itérations de chacune des boucles du programme de base en C. Suite à la

sauvegarde des bornes dans des champs prévus dans ce but, les instructions primaires sont annexées dans le nœud séquentiel. Pour détecter la fin d'une boucle, un des successeurs du nœud de fin doit être situé avant lui dans la liste complète du graphe. Les instructions 10 à 13 viennent quant à elles compléter tous les nœuds hiérarchiques dont la fin coïnciderait avec celle de la boucle, comme le nœud séquentiel dans le cas le plus simple. Deux exemples de représentation de boucles dans différents cas possibles sont présentés à la figure 4-5. Le premier exemple présente le cas le plus simple : soit une boucle composée d'un seul nœud séquentiel. Un cas plus complexe est montré par la suite en examinant un des nombreux cas de figure possible.

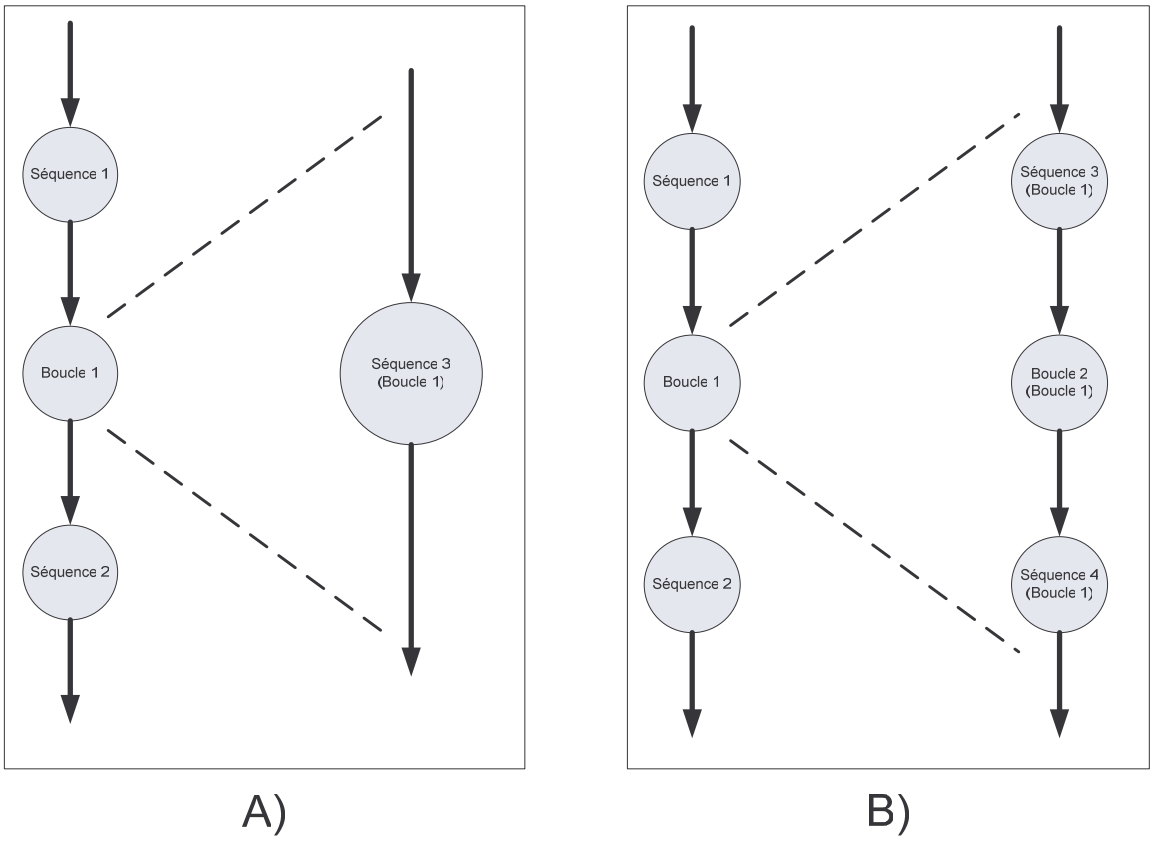


**Figure 4-5. Exemples de représentations de boucles**

#### 4.3.2.3 Branchement

Les principes reliés au nœud servant à représenter des branchements sont très similaires à ceux décrits dans le cas des boucles. Lors de l'analyse du nœud de début à l'étape 5, un branchement est détecté s'il ne possède qu'un seul prédécesseur qui possède plusieurs successeurs, qui sont tous situés plus loin dans la liste complète du graphe. Lors de sa création, le nœud hiérarchique de branchement se voit annexé un nœud séquentiel par branche. Un nœud séquentiel représentant une partie d'un branchement constitue le seul nœud séquentiel pouvant contenir d'autres nœuds hiérarchiques. Les étapes 10 à 13 sont responsables de compléter la structure de branchement et du retour au niveau supérieur, au même titre que pour les nœuds correspondants aux boucles. Deux exemples de représentation de branchements dans différents scénarios sont présentés à la figure 4-6. Le premier exemple A) montre le cas où une seule branche, dont l'exécution s'avère conditionnelle, vient composer le nœud hiérarchique. Le temps de traitement de ce nœud peut donc être nul. Le deuxième cas B) nécessite par contre l'exécution d'au moins une des branches.

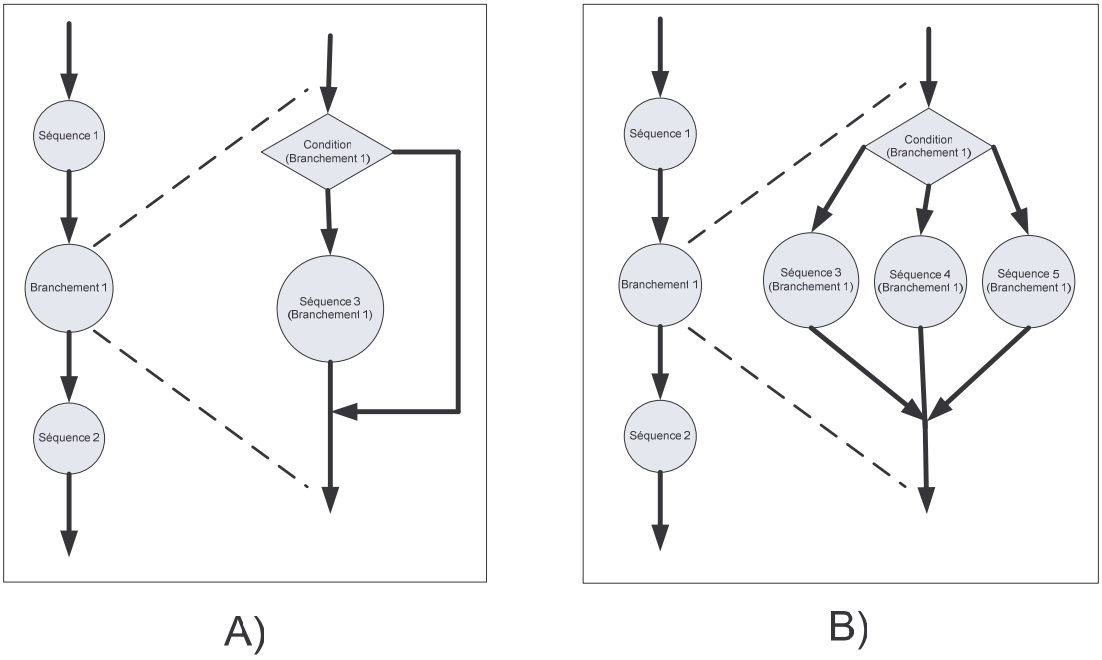


**Figure 4-6. Exemples de représentations de branchements**

## 4.4 Caractérisations des nœuds

Afin d'obtenir un graphique final comportant un niveau de détail suffisant à une bonne compréhension de l'algorithme étudié, plusieurs caractéristiques sont extraites des différent nœuds. Ces caractéristiques sont par la suite annotées dans le graphe de manière à être examinées lors de la visualisation. Certaines caractéristiques peuvent également servir durant l'étape d'ordonnancement ou sont elles-mêmes le résultat de cette étape. Plus de détails seront fournis à ce sujet dans le chapitre 6. La présente sous-section a pour but de présenter les différentes caractéristiques attachées à chaque type de nœud du graphe.

### 4.4.1 Nœuds de base

Les caractéristiques des nœuds de base pouvant être visionnées grâce à l'outil de visualisation sont présentées dans le tableau suivant.

**Tableau 4-1. Caractéristiques des nœuds de base**

| Caractéristiques | Description |
|---|---|
| Mem_In | Nombre de bits à lire en mémoire locale |
| Mem_Out | Nombre de bits à écrire en mémoire locale |
| Reg_In | Nombre de bits à lire dans les registres |
| Reg_Out | Nombre de bits à écrire dans les registres |
| ALAP | Valeur associée à l'ordonnancement ALAP |
| ASAP | Valeur associée à l'ordonnancement ASAP |
| ALU | Numéro de l'ALU responsable du traitement de l'instruction |

Puisque les nœuds de base constituent la représentation des instructions élémentaires, les premières caractéristiques affichées sont relatives aux transferts de données nécessaires au traitement de chacune des opérations. Une distinction est faite en fonction de la provenance ou de la destination du transfert ainsi que de son type (lecture ou écriture). Ces informations sont extraites avant toute phase d'ordonnancement, puisqu'elles sont utilisées par les différentes modélisations d'un cœur qui seront présentées dans la section

sur l'ordonnancement. Les caractéristiques liées à l'ordonnancement sont déterminées lors de cette dernière phase selon le procédé qui sera explicité lors du chapitre 6. L'affichage de ces informations est surtout utilisé afin de valider la phase d'ordonnancement elle-même.

### 4.4.2 Nœuds hiérarchiques

Le tableau 4-2 présente les caractéristiques principales des nœuds hiérarchiques. On peut y distinguer deux types de caractérisation. En effet, certaines informations sont obtenues suite à la création des structures hiérarchiques ou de la phase d'ordonnancement. Afin de caractériser adéquatement la variabilité du temps de calcul, ces valeurs doivent être exprimées à l'aide de bornes supérieure et inférieure. La modélisation de cette variabilité nous donne une profondeur accrue dans l'analyse des résultats d'ordonnancement et du choix de stratégies d'ordonnancement à implémenter.

**Tableau 4-2. Caractéristiques des nœuds hiérarchiques**

| Caractéristiques | Description |
|---|---|
| Nb_Ops | Nombre d'instructions dans le code |
| Max_Nb_Ops | Nombre maximal d'instructions exécutées |
| Min_Nb_Ops | Nombre minimal d'instructions exécutées |
| $\gamma$ | Mesure du Parallélisme |
| Max_time | Temps maximal d'exécution |
| Min_time | Temps minimal d'exécution |

En commençant par la plus simple, *Nb_Ops* représente le nombre d'instructions requises dans le code pour effectuer le traitement contenu dans le nœud hiérarchique. Cette valeur est surtout utilisée pour déterminer la quantité de mémoire requise pour l'entreposage du code à l'intérieur de la mémoire locale. Pour les caractéristiques bornées, soit le nombre d'instructions exécutées et le temps d'exécution, la méthode de calcul dépend du type de nœud et de sa composition. Les limites de ces deux caractéristiques sont calculées de la même manière. Dans les nœuds représentant une boucle, la borne supérieure correspond à

l'exécution du nombre maximal d'itérations tandis que le nombre minimal d'itérations sert au calcul de la borne inférieure. Pour les cas de branchement, la borne supérieure est obtenue en ne considérant que le branchement ayant le pire temps d'exécution suite à son ordonnancement. Le calcul de la borne inférieure est dépendant quant à lui de la branche prenant le moins de temps à exécuter. Il est important à noter que dans le cas de plusieurs structures de branchement, cette branche correspond au cas où aucune condition n'est atteinte et que le chemin emprunté équivaut à un contournement complet de la structure en question. Ceci a pour effet de conférer une valeur nulle au temps d'exécution minimal et au nombre d'opération exécuté. Dans le cas du parallélisme, la formule suivante est utilisée :

$$\gamma = \frac{V}{V_{LP}} \tag{1}$$

Dans cette équation, V représente le nombre d'opérations contenues dans un bloc de base tandis que $V_{LP}$ correspond au nombre d'instructions que l'on retrouve dans le chemin critique. Ce dernier est déterminé lors de la phase préliminaire d'ordonnancement ou les valeurs du ASAP et du ALAP sont obtenues. Lors de l'analyse du parallélisme, on s'attend à obtenir une valeur de $\gamma$ proche de 1 pour une partie hautement séquentielle. La possibilité de pouvoir exécuter plusieurs opérations en parallèle augmente lors de toute augmentation de $\gamma$. Comme il sera rapporté plus tard dans le chapitre 6, cette métrique donne souvent une première indication du nombre d'unités de calcul d'un seul cœur devant être alloué à une portion de code afin de pouvoir en faire le traitement dans le laps de temps le plus court possible.

## 4.5 Visualisation

Une courte section sur la visualisation des graphes produits vient conclure le chapitre 4. Les graphes obtenus sont décrits à l'aide du GDL (« *graphical description language* »). Ceux-ci peuvent être visualisés à l'aide de l'outil VCG (« *Visualization of Compiler*

*Graphs* ») [27]. Un exemple complet de la représentation graphique de l'addition vectorielle se trouve en annexe F. Étant donné l'utilisation de certains résultats dans la phase dynamique du projet, une représentation secondaire est également disponible sous la forme XML. Toutes les informations nécessaires à la création de ce fichier XML sont aussi incorporées sur le graphe complet en commentaires visibles dans le bloc d'entrée du système.

# CHAPITRE 5

## ESTIMATION DES BESOINS EN MÉMOIRE

Suivant la méthodologie de conception basée sur les plateformes, présentée dans le premier chapitre comme étant la ligne directrice de ce projet, une étape importante consiste à abstraire les caractéristiques liées à l'application afin de former la couche supérieure requise pour la propagation des contraintes vers la couche physique ou vers les couches inférieures. Pour ce faire, plusieurs attributs peuvent en effet être extraits de manière à obtenir un modèle complet et précis de l'algorithme étudié. L'estimation de la mémoire requise par une application est d'une importance capitale, surtout dans le cas présent, puisque la classe d'applications ciblées est du domaine des télécommunications et que l'implémentation se fait sur une architecture multi-DSP. Dans le domaine du traitement de signal, toute étape de calcul est effectuée sur de grandes quantités de données et elle résulte habituellement en autant de données en sortie. Donc nous pouvons affirmer qu'une attention particulière doit être accordée à la gestion de la mémoire, que ce soit dans le transfert des données ou dans leur simple sauvegarde en mémoire. D'un point de vue matériel, de nos jours, la mémoire consomme souvent la majorité de la superficie des circuits intégrés digitaux.

De manière plus spécifique par rapport au DSP en utilisation dans ce projet, la section 3.3 portant sur la mémoire disponible nous démontre que ce DSP ne peut contenir qu'une quantité très limitée de données. Nous pouvons donc facilement prédire que la mémoire sera souvent un goulot d'étranglement à l'utilisation du Vocallo. Il faut également considérer que les transferts de données doivent être considérés de manière rigoureuse, puisqu'un seul bus assure tous les transferts de données entre la mémoire principale et les 15 cœurs de la puce Vocallo. Toutes ces raisons nous ont donc fortement encouragés à élaborer et inclure une procédure d'estimation de la mémoire consommée dans la couche représentant l'application sélectionnée.

La recherche dans le domaine de l'estimation et l'optimisation de la consommation de mémoire est déjà bien en cours depuis de nombreuses années. Par contre, ce mémoire ne contiendra pas d'énumération de techniques existantes et intéressantes dans le cadre de ce projet, puisque ce domaine précis a été couvert plus en détails dans le cadre du mémoire d'un autre étudiant participant également au projet OPERA, soit celui de Frederic Plourde [25]. Ce chapitre se concentrera plutôt sur l'estimation automatique des ressources en mémoire ainsi que du partionnement possible de l'application sur la base de cette première estimation. Une brève conclusion servira finalement à présenter les enseignements finaux et améliorations possibles de notre outil quant à l'estimation de la mémoire et du partionnement.

## 5.1 Estimation des besoins en mémoire

De manière plus précise, cette section propose une définition plus poussée des caractéristiques à abstraire ainsi que la méthode employée pour extraire de manière automatique ces dernières, telles qu'implémentées dans la version finale de notre outil.

### 5.1.1 Caractérisation de la mémoire

Afin de caractériser la consommation de mémoire de manière précise, celle-ci a été divisée en plusieurs catégories. Chacune de celles-ci représente un aspect important de la consommation de mémoire par rapport à l'usage dans un système embarqué et encore plus particulièrement pour le cas du Vocallo. Le tableau 5.1 présente ces différentes subdivisions.

**Tableau 5-1. Catégories de mémoire**

| Catégories de mémoire |
|---|
| Entrée (In) |
| Sortie (Out) |
| Temporaire (Temp) |
| Code |

L'utilisation de l'ensemble de ces types de mémoire nous fournit tous les paramètres nécessaires à une bonne compréhension des besoins en mémoires de l'algorithme sélectionné. Ces catégories sont également la base de la représentation choisie dans la phase dynamique du projet global. Les informations sur la consommation de mémoire transmises par l'outil développé dans le cadre de ce mémoire sont donc sous la forme de ces quatre divisions. Toutes les quantités de mémoire présentées dans ce chapitre sont exprimées en nombre d'octets.

La mémoire en entrée correspond au nombre d'octets requis pour stocker les différents paramètres de l'application étudiée ainsi que les données nécessaires au traitement. Il est par contre important de noter, tel qu'il sera explicité dans la sous-section suivante, que la taille de toutes les données entrées à l'aide de pointeurs doit être ajoutée manuellement, puisque SUIF2 ne peut prendre en compte les structures de données externe au code sélectionné. La catégorie de sortie correspond à la quantité de mémoire requise pour sauvegarder les données qui doivent être transférées de la mémoire locale d'un cœur vers la mémoire globale. Le transfert de celles-ci est dû au besoin de réutilisation de ces données par un ou plusieurs autres segments de l'application. Ces deux dernières catégories sont extrêmement critiques dans le calcul du nombre d'octets devant être transmises sur le seul bus de communication de la puce avant et après l'exécution de l'algorithme. La mémoire temporaire correspond principalement à l'entreposage des variables temporaires apparaissant au fur et à mesure de l'exécution de l'application. Le calcul de cette catégorie ne prend par contre pas en compte la durée de vie des données. La valeur obtenue constitue donc une mesure du pire des cas. En effet, aucun effort n'a été déployé pour optimiser l'espace mémoire en regroupant les variables dont l'utilisation

est mutuellement exclusive. Cette dernière technique aurait eu pour effet de réduire, dans les meilleurs cas, la consommation de ce type de mémoire. La dernière division est utilisée pour connaître les besoins pour emmagasiner le code en assembleur de l'algorithme étudié. Ceci est particulièrement utile puisque dans le cas du Vocallo, les données et le code sont stockés dans la mémoire locale d'un cœur et que le code, à l'instar des paramètres et des données d'entrée, doit être envoyé au cœur par l'intermédiaire du bus de communication. Cette valeur constitue également une borne du pire cas dû à un manque d'information du coté des développeurs de la puce. En effet, les instructions peuvent être codées sous une forme de 32 ou de 64 bits mais aucune documentation indiquant la taille de chacune des instructions n'est disponible. Une taille de 64 bits est donc associée pour chacune des opérations forçant la valeur obtenue vers la borne supérieure pour un nombre fixe d'opérations.

### 5.1.2 Extraction des besoins en mémoire

Le processus d'extraction des besoins en mémoire est effectué comme la première étape suite à l'entrée de l'application à l'intérieur de l'infrastructure SUIF2. Suite à son entrée, SUIF2 représente l'algorithme sous la forme d'un arbre SUIF. Cet arbre est utilisé comme source pour produire le CFG à partir des bibliothèques composant MACHSUIF. Cet arbre contient toutes les informations relatives aux structures de contrôle, aux paramètres, aux variables, aux entrées et aux sorties du code. À ce niveau, une bibliothèque très riche vient caractériser et relier entre elles les structures de l'algorithme avec les différents paramètres et variables. Il est important à partir de maintenant de différencier les variables et les paramètres. Les paramètres proviennent de l'extérieur comme entrée primaire et ils ne peuvent subir de modifications lors de l'exécution du code. Les variables, quant à elles, sont instanciées par le programme lui-même. De nombreuses méthodes sont également disponibles pour parcourir l'arbre afin d'en faire ressortir les caractéristiques voulues. Ces bibliothèques sont discutées en profondeur dans la documentation de base de SUIF2 [23].

Une des branches de cet arbre, portant le nom de *SymbolTable*, énumère toutes les variables utilisées par le programme. Suite à la sélection de cette branche, il suffit de parcourir tous ses éléments et d'en faire ressortir la taille à l'aide des méthodes déjà en place. La somme de toutes ces variables et paramètres correspond à la quantité de mémoire temporaire, tel qu'expliqué précédemment. Pour connaître la quantité de mémoire en entrée, il suffit d'additionner les tailles des paramètres, puisque ceux-ci doivent être transmis au cœur par le bus central. Les données devant être recueillies à la fin du traitement sont également identifiées de manière précise dans ce bloc. L'analyse de cette branche permet donc d'extraire les quantités de mémoire requises en entrée, en sortie et de manière temporaire. L'espace requis pour le code doit être calculé plus tard dans le processus global. En effet, le nombre d'instructions nécessaires au traitement de l'application ne peut être déterminé qu'après le passage du CFG vers le CDFG, puisque l'arbre SUIF se situe à un niveau supérieur. Les informations recueillies à ce niveau sont donc envoyées plus bas par l'intermédiaire des fichiers textes utilisés par le système de partitionnement. Ce système ainsi que les principes de partitionnement seront énoncés à la section 5.2. Il est également important de souligner que le calcul des besoins en mémoire se complexifie dans le cas où un partitionnement est requis. Ceci sera aussi discuté plus tard dans le cadre de la section 5.2.

### 5.1.3 Validation

Afin de vérifier la validité des résultats ainsi que leur précision, les différentes applications énoncées à la fin du chapitre 2 ont été utilisées. Leur traitement par notre outil a engendré les résultats contenus dans les tableaux suivants. Les valeurs de référence ont été obtenues à l'aide de la première version de l'émulateur qui avait pour avantage, contrairement aux versions subséquentes, de donner la consommation en termes de mémoire pour un algorithme donné. La valeur retournée par l'émulateur ne comprend pas l'espace nécessaire pour emmagasiner le code. C'est pourquoi la colonne intitulée « Total » n'est obtenue qu'en additionnant les catégories « Entrée », « Sortie » et

« Temporaire ». Les tableaux de résultats présentent les erreurs absolues et relatives entre nos bancs d'essais et les performances obtenues de l'émulateur. Afin de mieux pouvoir identifier les erreurs ainsi que leurs sources, plusieurs cas possibles ont été analysés en changeant le nombre d'échantillons traités. Dans le cas de la deuxième classe d'applications, soient les filtres, le nombre de coefficients a également été modifié.

**Tableau 5-2. Estimation de la quantité de mémoire requise pour l'addition vectorielle**

| Nb d'échantillons | Émulateur (Octets) | Estimation Mémoire (Octets) | | | | | Erreur (Octets) | |
|---|---|---|---|---|---|---|---|---|
| | | Entrée | Sortie | Temporaire | Code | Total | | |
| 16 | 104 | 64 | 32 | 22 | 304 | 118 | 14 | 13,46% |
| 32 | 200 | 128 | 64 | 22 | 304 | 214 | 14 | 7,00% |
| 64 | 392 | 256 | 128 | 22 | 304 | 406 | 14 | 3,57% |
| 128 | 776 | 512 | 256 | 22 | 304 | 790 | 14 | 1,80% |
| 256 | 1544 | 1024 | 512 | 22 | 304 | 1558 | 14 | 0,91% |

**Tableau 5-3. Estimation mémoire pour le Maximum vectoriel**

| Nb d'échantillons | Émulateur (Octets) | Estimation Mémoire (Octets) | | | | | Erreur (Octets) | |
|---|---|---|---|---|---|---|---|---|
| | | Entrée | Sortie | Temporaire | Code | Total | | |
| 16 | 44 | 32 | 1 | 24 | 648 | 57 | 13 | 29,55% |
| 32 | 76 | 64 | 2 | 24 | 648 | 90 | 14 | 18,42% |
| 64 | 140 | 128 | 4 | 24 | 648 | 156 | 16 | 11,43% |
| 128 | 268 | 256 | 8 | 24 | 648 | 288 | 20 | 7,46% |
| 256 | 524 | 512 | 16 | 24 | 648 | 552 | 28 | 5,34% |

**Tableau 5-4. Estimation mémoire pour le FIR Complexe**

| Nb de coefficients | Nb d'échantillons | Émulateur (Octets) | Estimation Mémoire (Octets) | | | | | Erreur (Octets) | |
|---|---|---|---|---|---|---|---|---|---|
| | | | Entrée | Sortie | Temporaire | Code | Total | | |
| 4 | 16 | 284 | 116 | 128 | 88 | 1872 | 332 | 48 | 16,90% |
| 4 | 32 | 476 | 180 | 256 | 88 | 1872 | 524 | 48 | 10,08% |
| 4 | 64 | 860 | 308 | 512 | 88 | 1872 | 908 | 48 | 5,58% |
| 4 | 128 | 1628 | 564 | 1024 | 88 | 1872 | 1676 | 48 | 2,95% |
| 4 | 256 | 3164 | 1076 | 2048 | 88 | 1872 | 3212 | 48 | 1,52% |
| 8 | 16 | 332 | 164 | 128 | 88 | 1872 | 380 | 48 | 14,46% |
| 16 | 16 | 428 | 260 | 128 | 88 | 1872 | 476 | 48 | 11,21% |
| 32 | 16 | 620 | 452 | 128 | 88 | 1872 | 668 | 48 | 7,74% |
| 64 | 16 | 1004 | 836 | 128 | 88 | 1872 | 1052 | 48 | 4,78% |
| 128 | 16 | 1772 | 1604 | 128 | 88 | 1872 | 1820 | 48 | 2,71% |

**Tableau 5-5. Estimation mémoire pour le Filtre LMS**

| Nb de coefficients | Nb d'échantillons | Émulateur (Octets) | Estimation Mémoire (Octets) | | | | | Erreur (Octets) | |
|---|---|---|---|---|---|---|---|---|---|
| | | | Entrée | Sortie | Temporaire | Code | Total | | |
| 4 | 16 | 416 | 212 | 128 | 184 | 2424 | 524 | 108 | 25,96% |
| 4 | 32 | 672 | 340 | 256 | 184 | 2424 | 780 | 108 | 16,07% |
| 4 | 64 | 1184 | 596 | 512 | 184 | 2424 | 1292 | 108 | 9,12% |
| 4 | 128 | 2208 | 1108 | 1024 | 184 | 2424 | 2316 | 108 | 4,89% |
| 4 | 256 | 4256 | 2132 | 2048 | 184 | 2424 | 4364 | 108 | 2,54% |
| 8 | 16 | 464 | 260 | 128 | 184 | 2424 | 572 | 108 | 23,28% |
| 16 | 16 | 560 | 356 | 128 | 184 | 2424 | 668 | 108 | 19,29% |
| 32 | 16 | 752 | 548 | 128 | 184 | 2424 | 860 | 108 | 14,36% |
| 64 | 16 | 1136 | 932 | 128 | 184 | 2424 | 1244 | 108 | 9,51% |
| 128 | 16 | 1904 | 1700 | 128 | 184 | 2424 | 2012 | 108 | 5,67% |

En se fiant à l'exemple du filtre LMS, on peut extraire le comportement typique de l'estimation en mémoire. Il est bien sur normal d'observer une augmentation des entrées et sorties suivant une augmentation du nombre d'échantillons traités. L'augmentation quand à elle du nombre de coefficient n'influence pas la quantité de données en sorties puisque les coefficients sont caractérisés comme de simples paramètres. La quantité de mémoire temporaire et l'espace requis pour stocker le code ne sont pas influencés par la quantité de données traitées. Ces dernières valeurs sont dictées par le style de codage. En examinant l'ensemble des résultats, il est possible de conclure que notre outil produit des estimations de la consommation mémoire se rapprochant de la réalité. En effet, cette erreur se situe généralement en dessous des 10% et monte jusqu'à un maximum de 30% dans certains cas de figure. Pour une estimation basée sur le pire cas, on peut juger ces résultats comme très acceptables pour la phase dynamique du projet OPERA. On peut également constater que l'erreur absolue a tendance à rester constante lorsque l'on fait varier les différents paramètres des algorithmes. Ceci implique que dans les cas d'applications traitant une grande quantité de données, qui requièrent donc une grande consommation de mémoire, l'erreur relative tend à diminuer, procurant ainsi une estimation encore plus précise. Ceci semble encore une fois être acceptable et même favorable, puisque les applications dans le domaine des télécommunications effectuent dans la majorité des cas un traitement d'un très grand nombre de données. Un autre

avantage de l'utilisation de cet outil afin de faire ressortir automatiquement les besoins en mémoire est sa grande rapidité d'analyse.

## 5.2 Partitionnement

La section explique les fondements du potentiel de partitionnement de notre outil et présente un exemple simple démontrant le flot complet en termes de consommation de mémoire.

### 5.2.1 Principes de partionnement

Le but principal de l'opération de partitionnement est de pouvoir fournir à la phase dynamique du projet une représentation graphique de l'algorithme analysé sous forme d'un graphe XML où chacune des tâches élémentaires peut être exécutée sans interruption par un seul cœur du Vocallo. Ceci signifie, en termes de consommation de mémoire, qu'une tâche ne peut prendre plus de 96 Ko, ou de 144 Ko lors de la fermeture de certains cœurs, pour s'exécuter puisqu'un cœur ne peut contenir plus de mémoire. De manière générale, une seule tâche est nécessaire lorsqu'un algorithme consomme moins que la limite permise pour un cœur. L'algorithme sera par contre séparé en plusieurs partitions, qui représenteront finalement une tâche chacune, lorsque la limite est dépassée. Cette section vient expliciter le fonctionnement et les principes de base de l'outil de partionnement qui est lui aussi totalement automatisé.

On peut commencer par mentionner que le processus de partitionnement se déroule en deux étapes distinctes. La première est effectuée à partir de l'infrastructure SUIF2 lors de la première partie de l'extraction des besoins en mémoire. La deuxième phase doit se dérouler suite à la création des structures hiérarchiques et de l'évaluation de la taille totale des instructions, puisque ces derniers stades requièrent au préalable l'abaissement du niveau de langage à l'aide des bibliothèques de MACHSUIF. La figure 5-1 présente la première partie du processus de partionnement.

```
1. list_variables();
2. list_parameters();
3. For(statement_start; statement_end; statement++)
   {
4.        type = statement.get_type();
5.        create_new_partition(type);
6.        adjust_parameters(partition);
7.        adjust_variable(partition);
   }
8. write_partitions_to_file(partition_list);
```

**Figure 5-1. Première étape de partitionnement**

Les deux premières actions entreprises débutent également l'extraction des besoins en mémoire. Pour le déroulement spécifique de ces deux étapes, la section précédente couvre le sujet en détail. Une règle importante du partitionnement apparaît à l'opération 3. En effet, on considère que les structures hiérarchiques du premier niveau ne peuvent être divisées. Par exemple, l'outil de partitionnement ne viendra jamais séparer une boucle en deux afin de lui faire respecter les limites en mémoire d'un cœur. SUIF2 utilise ici aussi les mêmes trois types de structures que celles présentés lors de la génération du CDFG. Les opérations 4 à 7 sont effectuées systématiquement pour chacune des structures hiérarchiques. Les opérations 4 et 5 servent à créer une partition de base servant à emmagasiner toutes les informations extraites par rapport à l'utilisation des paramètres et variables dans la structure hiérarchique associée. En effet, les étapes 6 et 7 servent à déterminer si une partition utilise ou non chacun des paramètres et des variables. Ces informations seront utiles dans les étapes suivantes afin de calculer les nouvelles tailles des entrées et sorties si un partitionnement doit avoir lieu. Toutes ces données sont par la suite écrites dans un fichier texte, puisque la transmission directe de structures de données n'est pas possible entre SUIF et MACHSUIF. La deuxième partie de l'algorithme de partitionnement est présentée à la figure 5-2.

```
1. read_chip_info();
2. read_partitions_from_file();
3. match_partition_to_hierarchical_structure();
4. memory = calculate_memory_for_one_partition();
5. If(memory > 96K)
   {
6.         create_new_final_partition();
7.         For(partition_start; partition_end; partition++)
           {
8.                 memory = calculate_memory(partition) + calculate_memory(final_partition);
9.                 If(memory > 96K)
                   {
10.                        create_new_final_partition();
11.                        add_to_final_partition(partition);
                   }
12.                Else
                   {
13.                        add_to_final_partition(partition);
                   }
           }
     }
14. Else
     {
15.        create_new_final_partition();
16.        add_partitions_to_final_partition();
     }
```

**Figure 5-2. Deuxième étape de partitionnement**

Cette deuxième phase de l'opération de partitionnement est responsable de regrouper les différentes partitions initiales résultantes de la première étape en une série de partitions finales qui répondent à la limite en mémoire d'un cœur. Les premières instructions consistent à aller chercher les caractéristiques de la puce ainsi que la liste des partitions de base de l'algorithme. Lors de l'étape 1, la quantité de mémoire d'un cœur, la taille en bits des instructions ainsi que le nombre d'unités opératives utilisées sont déterminés à l'aide d'un fichier texte (« *info_chip.dat* »). Un changement dans ce fichier permet de modifier aisément les caractéristiques d'un cœur afin de pouvoir en observer les effets sur le partionnement et l'ordonnancement. Lors de la fermeture d'un cœur afin de faire bénéficier ses voisins d'un supplément de mémoire, il est possible d'obtenir un

partionnement sur cette nouvelle base en ne modifiant que la quantité de mémoire disponible contenue dans le fichier. À l'étape 3, les partitions initiales extraites des fichiers textes sont associées aux structures hiérarchiques de manière à pouvoir compléter la quantité de mémoire requise pour le stockage du code. L'opération 4 consiste à calculer les besoins en mémoire dans le cas où l'application n'aurait pas à être séparée. Si l'agglomération de toutes les partitions qui résultent entraîne une demande en mémoire pouvant être octroyée à un simple cœur, une partition finale est créée afin de regrouper toutes les partitions initiales aux étapes 15 et 16. Dans le cas contraire, le processus de regroupement est enclenché à l'aide des opérations 6 à 13. Une première partition finale est d'abord créée en 6. Par la suite, chacune des partitions initiales est prise séquentiellement dans l'ordre d'exécution de l'algorithme par la boucle 8 et est considérée pour faire partie de la partition finale courante. Les besoins en mémoire de cette nouvelle partition théorique sont calculés en 8 et servent à déterminer si une nouvelle partition finale doit être utilisée à l'étape 9. L'étape 8 sert également à ajuster les besoins en termes d'entrée et de sortie puisque la représentation XML doit pouvoir fournir les détails pour chacune des partitions ainsi que de son ensemble. Un exemple d'une représentation XML plus complexe sera présenté dans la section suivante, démontrant le processus de partitionnement à l'aide d'un exemple. Les instructions 10 et 11 entrent en fonction lorsque l'on dépasse la limite de mémoire imposée par un cœur, tandis qu'en 13, on ne fait qu'ajouter une partition initiale à la partition finale courante. Si une structure est trop volumineuse pour être contenue dans un cœur, le concepteur en est avisé et il peut par la suite examiner le CDFG produit, afin d'en identifier les causes.

### 5.2.2 Exemple de partionnement

Afin de démontrer le processus de partitionnement, un exemple précis sera utilisé. Dans le but d'obtenir une démonstration claire, une application synthétique a été développée et employée. Le code de cette application se retrouve en annexe [A]. De manière globale, cet algorithme sans aucune fonctionnalité réelle ne fait que manipuler de lourdes

structures de données dans une série de boucles. Afin de faire ressortir les résultats de l'outil de partitionnement, l'application a été traitée en variant la quantité de mémoire disponible sur un cœur. Les tableaux suivants montrent le résultat final de trois partitionnements possibles en fonction de la taille de la mémoire locale. Les quantités de mémoire sont exprimées en bits. Tous ces résultats ont été extraits directement de la représentation XML produite pour la phase dynamique du projet. Ces fichiers XML sont mis en annexe [B-C-D]. Les tableaux ne font que présenter tous les champs servant à caractériser une application dans ce type de représentation. Le premier tableau montre que l'on obtient une seule partition lorsque la mémoire locale est supérieure à 10 Ko. Deux partitions sont nécessaires pour une taille de 5 Ko tandis qu'une autre doit être créée lorsque l'on continue à diminuer la mémoire disponible. On peut aussi observer que la quantité totale de mémoire ne dépasse jamais la limite de mémoire imposée pour les partitions présentées dans les trois tableaux suivants.

**Tableau 5-6. Partitionnement final avec une mémoire locale de 10 Ko**

| Partition 1 | |
|---|---|
| <task name> | test1 |
| <cpu cycles> | 3274 |
| <input> | 38560 |
| <output> | 32 |
| <used> | 38688 |
| <code> | 10880 |
| <in> | n1 |
| <out> | n2 |

**Tableau 5-7. Partitionnement final avec une mémoire locale de 5 Ko**

| Partition 1 | | Partition 2 | |
|---|---|---|---|
| <task name> | test1 | <task name> | test2 |
| <cpu cycles> | 2412 | <cpu cycles> | 862 |
| <input> | 28896 | <input> | 9792 |
| <output> | 96 | <output> | 32 |
| <used> | 29024 | <used> | 9792 |
| <code> | 8000 | <code> | 2880 |
| <in> | n1 | <in> | test1 |
| <out> | test2 | <out> | n2 |

**Tableau 5-8. Partitionnement final avec une mémoire locale de 3 Ko**

| Partition 1 | | Partition 2 | | Partition 3 | |
|---|---|---|---|---|---|
| <task name> | test1 | <task name> | test2 | <task name> | test3 |
| <cpu cycles> | 946 | <cpu cycles> | 1466 | <cpu cycles> | 862 |
| <input> | 9632 | <input> | 19392 | <input> | 9792 |
| <output> | 96 | <output> | 96 | <output> | 32 |
| <used> | 9760 | <used> | 19392 | <used> | 9792 |
| <code> | 3136 | <code> | 4864 | <code> | 2880 |
| <in> | n1 | <in> | test1 | <in> | test2 |
| <out> | test2 | <out> | test3 | <out> | n2 |

Globalement, on peut remarquer qu'en réduisant la quantité de mémoire disponible sur un cœur, le nombre de partitions tend à augmenter. L'outil de partitionnement ne change en rien la phase d'ordonnancement qui fera l'objet du prochain chapitre. En effet, on peut voir que le nombre total de cycles pour exécuter cet algorithme reste le même sans égard au nombre de partitions. En effet, le partitionnement ne fait que regrouper certaines structures hiérarchiques sous différentes partitions selon la limite imposée. Il faut cependant faire remarquer que la séparation en plusieurs partitions devrait occasionner une augmentation du nombre ainsi que la taille des transferts entre la mémoire externe du Vocallo et un cœur.

## 5.3 Conclusion

Bien que l'outil de partitionnement réponde aux besoins du projet, de nombreuses modifications ou améliorations notables pourrait y être ajoutées. En effet, la présente version de cette partie de l'outil implémenté ne prend en compte que la quantité de mémoire disponible comme facteur de décision. La littérature dans ce domaine montre que de nombreux autres critères peuvent être utilisés afin de subdiviser une application. Parmi ces derniers, on peut citer l'analyse du temps de vie des variables, afin de permettre une meilleure utilisation de la mémoire en regroupant des partitions compatibles, ou le coût des transferts de données. Comme l'emphase a été mise dans le développement du Vocallo sur la réduction de la consommation de puissance, cette

dernière condition pourrait également servir. Un modèle sur la consommation de puissance devrait par contre être implémenté au préalable. Un autre point qui n'a pas été exploré dans le cadre de ce projet concerne les effets au niveau multi-cœur engendrés par un partitionnement particulier. Cette dernière suggestion serait sûrement très utile puisque tous les cœurs du Vocallo ne sont desservis que par un seul bus commun. L'outil d'estimation de mémoire produit quant à lui des résultats satisfaisants malgré une simplicité apparente dans la méthode d'extraction. Ceci reflète bien par contre le modèle de consommation de mémoire du Vocallo, puisqu'aucune étape d'optimisation n'est effectuée, selon notre expérience avec les différentes versions des émulateurs.

# CHAPITRE 6

## MODÉLISATION ET ORDONNANCEMENT

## 6.1 Modélisation

Dans le cadre de ce projet de maîtrise, deux modèles se basant sur le fonctionnement du DSP présent sur un cœur ont été développés. Ces deux modèles ont servi au cours du projet à mieux comprendre les subtilités liées à l'emploi du Vocallo. La principale difficulté de la modélisation de ce DSP découle de ses caractéristiques asynchrones. En effet, le passage vers un modèle synchrone plus traditionnel a été nécessaire en établissant une référence de temps fixe. Cette section a pour but de présenter les bases de chacun des ces modèles et de comparer les résultats obtenus par rapport aux outils de développement fournis par Octasic.

### 6.1.1 Premier modèle

La première modélisation repose sur la proposition de considérer le temps moyen requis pour exécuter une instruction comme une référence de temps stable et fixe. Durant ce temps, on suppose que chacune des unités opératives dispose du temps nécessaire à la complétion d'une instruction, peu importe laquelle. Ce temps prendra, à partir de maintenant, le nom d'étape de contrôle, mais il joue le même rôle de synchronisation qu'un signal d'horloge conventionnel. De manière précise, une étape de contrôle de ce modèle a une durée de 10656 picosecondes, soit 16 fois le temps requis pour compléter un accès en mémoire locale. Il est donc important de noter que cette itération de la modélisation ne prend pas en compte la durée réelle d'une instruction, basée sur sa complexité. Cette exécution de 16 opérations simultanées, dues aux ALU agencées de manière parallèle, ne constitue par contre qu'un maximum théorique, puisque la vraie limite provient plutôt du nombre d'accès à la mémoire locale pouvant être effectuées dans cette limite de temps. Ce maximum pourrait se produire dans le cas où 16

opérations, ne requérant aucun autre accès mémoire que l'opération de chargement de l'instruction, s'enchaîneraient séquentiellement. Ceci revient à dire que pour une étape de contrôle, on peut effectuer un nombre d'accès mémoire équivalent au nombre d'unités opératives en présence dans un cœur. Il faut à présent déterminer le nombre d'accès en mémoire requis pour chacune des instructions. Comme chaque instruction provient de la mémoire locale et qu'une opération de chargement doit être nécessairement effectuée, le nombre requis d'accès en mémoire est toujours plus grand ou égal à 1. Chaque accès supplémentaire en entrée ou en sortie viendra incrémenter cette valeur de 1. Une addition demandant deux données en mémoire et impliquant une écriture du résultat contiendrait un total de 4 accès en mémoire locale. Le processus d'ordonnancement basé sur les considérations précédentes est présenté à la figure 6-1.

```
Schedule_application(CDFG, Nb_Alu)
1. control_step = 0;
2. create_new_dsp(Nb_Alu);
3. nb_acc = 0;
4. while(!finished)
   {
5.       find_free_instructions();
6.       instr_acc = get_number_of_acc(found_instr);
7.       if((instr_acc + nb_acc) < Nb_Alu)
         {
8.                nb_acc = nb_acc + instr_acc;
9.                assign_control_step(found_instr);
         }
10.      if(nb_access == Nb_Alu || last_instr_reached)
         {
11.               control_step++;
12.               nb_acc = 0;
         }
   }
```

**Figure 6-1. Algorithme d'ordonnancement du premier modèle**

Les instructions 1,2 et 3 servent à initialiser les paramètres de départ en créant entre autres un DSP contenant un nombre d'unités opératives spécifié par l'utilisateur. La boucle principale qui débute en 4 ne se termine qu'une fois l'ordonnancement complété. La fonction find_free_instructions() est responsable de parcourir toutes les instructions et d'en faire ressortir une instruction éligible à être ordonnancée. Les instructions possédant une valeur ALAP plus faible sont considérées prioritaires. Dans le cas où plusieurs instructions seraient disponibles à l'étape 5, celles dont la valeur ALAP est la plus faible sera sélectionnée. Si cette dernière instruction ne peut être ordonnancée à cause d'un manque de capacité d'accès en mémoire, celle-ci ne sera plus considérée tant que l'étape de contrôle ne sera pas incrémentée. Suite à la sélection d'une instruction, le nombre d'accès en mémoire requis par celle-ci est extrait. La condition 7 vérifie si le nombre d'accès restants est suffisant. Lorsque ceci se trouve être vrai, le nombre d'accès courant est ajusté à l'étape 8, tandis que la phase 9 vient assigner l'étape de contrôle correspondant à l'instruction venant d'être ordonnancée. Si le nombre d'accès n'est pas suffisant, on ne fait que passer à une prochaine instruction en retournant à l'étape 5. La condition 10 est responsable de passer à la prochaine étape de contrôle. Ceci se produit lorsque le nombre maximal d'accès en mémoire a été atteint ou que la dernière instruction a été considérée et rejetée. Les étapes 11 et 12 sont responsables du passage vers la prochaine étape de contrôle.

### 6.1.2 Deuxième modèle

La base de temps utilisée lors de la deuxième modélisation correspond au temps nécessaire à la complétion d'un accès en mémoire locale. Ceci permet un processus d'ordonnancement plus proche du comportement réel du DSP asynchrone. Le choix de cette base de temps a surtout été motivé par le fait que tout accès mémoire, en lecture ou en écriture, demande le même temps pour toutes les unités opératives, soit un temps de 666 picosecondes. Sur cette base, on ne peut effectuer plus d'un accès en mémoire par étape de contrôle. Une instruction ne peut donc pas être lancée lorsqu'un autre ALU est

en train d'accéder à la mémoire locale. En se servant de cette base de temps, on peut donc caractériser le temps d'exécution des instructions selon différentes classes. Il est à noter que la variabilité due aux dépendances de données n'est pas prise en considération à cause de la nature statique de notre analyse. Le tableau 6-1 présente la durée des instructions selon leur classe. Les durées pour l'exécution des instructions ont été obtenues à l'aide de la première version de l'émulateur fourni par Octasic.

**Tableau 6-1. Durée moyenne des instructions selon leur classe**

| Classe | Nombre moyen de cycles/Instruction | Exemple |
|---|---|---|
| NOP | 2 | NOP |
| Logique | 4 | And, Or, ... |
| Arithmétique simple | 8 | Add, Sub, ... |
| Arithmétique complexe | 16 | Mul, ... |

De manière globale, l'état de chacune des unités opératives est évalué à chaque étape de contrôle. Ce deuxième modèle se trouve à suivre la majorité des étapes d'exécution d'une instruction, présentées dans le tableau 3-2. La lecture et l'écriture dans les registres ne sont pas simulées, puisque le temps nécessaire à ces transferts est jugé négligeable par rapport au temps d'un accès en mémoire locale. Le processus d'ordonnancement basé sur les considérations précédentes est présenté à la figure 6-2.

```
Schedule_Application(CDFG, Nb_Alu)
1. control_step = 0;
2. create_new_dsp(Nb_Alu);
3. while(!finished)
   {
4.         if(nb_active_unit > 0)
           {
5.                 check_for_completed_instruction();
           }
6.         if(nb_idle_unit > 0 && !writing_memory)
           {
7.                 find_next_instruction();
8.                 if(found_instruction)
                   {
9.                         assign_to_idle_alu(next_instruction);
                   }
           }
10.        control_step++;
   }
```

**Figure 6-2. Deuxième algorithme d'ordonnancement**

Les deux premières étapes servent à initialiser les multiples paramètres des classes algorithmiques représentant le DSP. La boucle principale de la phase 3 ne se termine qu'une fois la dernière instruction complétée. La condition 4 est responsable au début de chaque étape de contrôle de vérifier la complétion des instructions dans les unités opératives actives lors de l'étape 5. Ceci revient à dire que la complétion d'une instruction prend préséance par rapport à la recherche d'une nouvelle instruction à assigner. La condition 6 est par la suite responsable de décider si une nouvelle instruction peut être ordonnancée ou si l'on doit passer à la prochaine étape de contrôle. Dans le cas où aucune instruction ne s'est terminée et qu'au moins une unité est libre, on cherche une nouvelle instruction à l'étape 7. La fonction find_next_instruction() cherche une instruction dont tous les prédécesseurs ont déjà été ordonnancés. Comme dans la précédente modélisation, l'instruction ayant la plus petite valeur de ALAP sera sélectionnée dans le cas où plusieurs instructions seraient disponibles en même temps. Lorsqu'une instruction est trouvée, elle est assignée à la prochaine ALU libre. La figure suivante vient présenter une représentation graphique d'un exemple d'ordonnancement

simple. Dans ce cas précis, l'instruction 2 ne peut démarrer tant que la mémoire locale est toujours utilisée par l'unité 1. Il en est de même pour l'instruction 3 et l'unité 2.

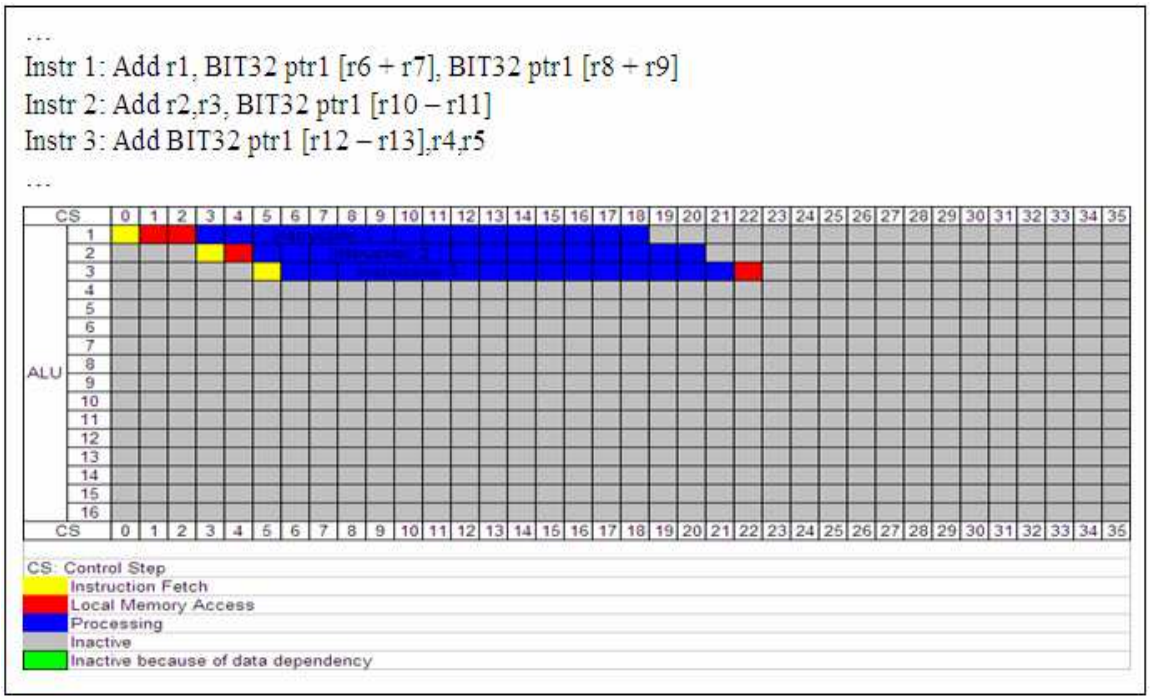


**Figure 6-3. Exemple du deuxième modèle d'ordonnancement**

### 6.1.3 Validation des modèles

Les tableaux suivants présentent les résultats des estimations produites à l'aide des méthodes décrites précédemment. Les temps d'exécution de référence ont été obtenus en utilisant la première version de l'émulateur développé par Octasic. La validation se fait par rapport à cette version, puisque la caractérisation des instructions utilisées dans le deuxième modèle a été effectuée à partir de celle-ci. Afin de pouvoir réaliser une analyse pertinente, les résultats présentés proviennent du traitement de tous les bancs d'essais en combinaison avec une variation des paramètres importants. Les temps d'exécution montrés dans les tableaux correspondent au scénario du pire cas.

**Tableau 6-2. Validation pour l'addition vectorielle**

| NB Samples | Émulateur | Estimation 1 | | | Estimation 2 | | |
|---|---|---|---|---|---|---|---|
| | Temps CPU (ps) | Temps CPU (ps) | Erreur | | Temps CPU (ps) | Erreur | |
| | | | (ps) | % | | (ps) | % |
| 16 | 1,39E+06 | 2,56E+06 | 1,17E+06 | 83,80% | 1,43E+06 | 4,18E+04 | 3,00% |
| 32 | 2,76E+06 | 5,11E+06 | 2,35E+06 | 85,02% | 2,87E+06 | 1,02E+05 | 3,69% |
| 64 | 5,46E+06 | 1,02E+07 | 4,77E+06 | 87,35% | 5,73E+06 | 2,73E+05 | 4,99% |
| 128 | 1,09E+07 | 2,05E+07 | 9,55E+06 | 87,51% | 1,15E+07 | 5,55E+05 | 5,08% |
| 256 | 2,18E+07 | 4,09E+07 | 1,92E+07 | 88,01% | 2,29E+07 | 1,17E+06 | 5,37% |

**Tableau 6-3. Validation pour le maximum vectoriel**

| NB Samples | Émulateur | Estimation 1 | | | Estimation 2 | | |
|---|---|---|---|---|---|---|---|
| | Temps CPU (ps) | Temps CPU (ps) | Erreur | | Temps CPU (ps) | Erreur | |
| | | | (ps) | % | | (ps) | % |
| 16 | 3,91E+06 | 6,22E+06 | 2,31E+06 | 59,21% | 3,62E+06 | -2,86E+05 | -7,31% |
| 32 | 7,62E+06 | 1,24E+07 | 4,83E+06 | 63,44% | 7,25E+06 | -3,69E+05 | -4,85% |
| 64 | 1,54E+07 | 2,49E+07 | 9,44E+06 | 61,13% | 1,45E+07 | -9,56E+05 | -6,19% |
| 128 | 3,12E+07 | 4,98E+07 | 1,85E+07 | 59,31% | 2,90E+07 | -2,27E+06 | -7,25% |
| 256 | 6,17E+07 | 9,96E+07 | 3,79E+07 | 61,50% | 5,80E+07 | -3,68E+06 | -5,97% |

**Tableau 6-4. Validation pour le filtre complexe**

| NB Coefficients | NB Samples | Émulateur | Estimation 1 | | | Estimation 2 | | |
|---|---|---|---|---|---|---|---|---|
| | | Temps CPU (ps) | Temps CPU (ps) | Erreur | | Temps CPU (ps) | Erreur | |
| | | | | (ps) | % | | (ps) | % |
| 4 | 16 | 1,29E+07 | 2,27E+07 | 9,73E+06 | 75,15% | 1,47E+07 | 1,80E+06 | 13,91% |
| 4 | 32 | 2,65E+07 | 4,54E+07 | 1,89E+07 | 71,17% | 2,95E+07 | 3,00E+06 | 11,32% |
| 4 | 64 | 5,31E+07 | 9,07E+07 | 3,76E+07 | 70,81% | 5,90E+07 | 5,89E+06 | 11,09% |
| 4 | 128 | 1,06E+08 | 1,81E+08 | 7,54E+07 | 71,17% | 1,18E+08 | 1,20E+07 | 11,32% |
| 4 | 256 | 2,06E+08 | 3,63E+08 | 1,57E+08 | 76,15% | 2,36E+08 | 3,00E+07 | 14,57% |
| 8 | 16 | 2,52E+07 | 4,38E+07 | 1,86E+07 | 73,90% | 2,92E+07 | 4,00E+06 | 15,88% |
| 16 | 16 | 5,30E+07 | 8,61E+07 | 3,31E+07 | 62,38% | 5,81E+07 | 5,07E+06 | 9,57% |
| 32 | 16 | 1,03E+08 | 1,71E+08 | 6,78E+07 | 65,87% | 1,16E+08 | 1,30E+07 | 12,64% |
| 64 | 16 | 2,11E+08 | 3,40E+08 | 1,29E+08 | 60,97% | 2,31E+08 | 2,04E+07 | 9,66% |
| 128 | 16 | 4,22E+08 | 6,78E+08 | 2,56E+08 | 60,80% | 4,63E+08 | 4,10E+07 | 9,72% |

**Tableau 6-5. Validation pour le filtre LMS**

| NB Coefficients | NB Samples | Émulateur | Estimation 1 | | | Estimation 2 | | |
|---|---|---|---|---|---|---|---|---|
| | | Temps CPU (ps) | Temps CPU (ps) | Erreur (ps) | Erreur % | Temps CPU (ps) | Erreur (ps) | Erreur % |
| 4 | 16 | 1,60E+07 | 2,90E+07 | 1,30E+07 | 81,15% | 1,80E+07 | 2,03E+06 | 12,69% |
| 4 | 32 | 3,22E+07 | 5,80E+07 | 2,58E+07 | 80,17% | 3,61E+07 | 3,89E+06 | 12,08% |
| 4 | 64 | 6,18E+07 | 1,16E+08 | 5,41E+07 | 87,50% | 7,21E+07 | 1,03E+07 | 16,64% |
| 4 | 128 | 1,31E+08 | 2,32E+08 | 1,01E+08 | 76,68% | 1,44E+08 | 1,30E+07 | 9,91% |
| 4 | 256 | 2,70E+08 | 4,64E+08 | 1,93E+08 | 71,45% | 2,88E+08 | 1,80E+07 | 6,65% |
| 8 | 16 | 3,13E+07 | 5,35E+07 | 2,23E+07 | 71,21% | 3,43E+07 | 3,00E+06 | 9,59% |
| 16 | 16 | 6,12E+07 | 1,03E+08 | 4,14E+07 | 67,58% | 6,67E+07 | 5,50E+06 | 8,98% |
| 32 | 16 | 1,12E+08 | 2,01E+08 | 8,91E+07 | 79,79% | 1,32E+08 | 2,00E+07 | 17,90% |
| 64 | 16 | 2,42E+08 | 3,97E+08 | 1,56E+08 | 64,41% | 2,62E+08 | 2,00E+07 | 8,28% |
| 128 | 16 | 5,00E+08 | 7,90E+08 | 2,90E+08 | 58,02% | 5,23E+08 | 2,29E+07 | 4,58% |

**Tableau 6-6. Validation pour le codage**

| Décodage | Émulateur | Estimation 1 | | | Estimation 2 | | |
|---|---|---|---|---|---|---|---|
| | Temps CPU (ps) | Temps CPU (ps) | Erreur (ps) | Erreur % | Temps CPU (ps) | Erreur (ps) | Erreur % |
| Viterbi | 6,10E+09 | 1,22E+10 | 6,06E+09 | 99,39% | 6,64E+09 | 5,42E+08 | 8,88% |
| Turbo | 3,66E+10 | 6,19E+10 | 2,53E+10 | 68,99% | 39139903584 | 2,54E+09 | 6,94% |

En examinant ces derniers résultats, on peut noter que dans la majorité des cas, l'erreur se traduit comme une surestimation. Ceci est logique puisque les tableaux présentent les estimations obtenues dans les pires cas. L'analyse du maximum vectoriel à l'aide du deuxième modèle est le seul cas où l'estimation est inférieure au temps de référence. Contrairement à l'estimation en mémoire, on peut voir que le nombre d'échantillons et de coefficients n'influence pas directement la qualité des estimations. L'observation la plus flagrante concerne la différence de précision entre les deux modèles implémentés. En effet, la première version présente des résultats avec une erreur relative se situant habituellement autour de 70% et pouvant même atteindre les 90% dans certains cas de figures. On obtient par contre des résultats satisfaisants à l'aide du deuxième modèle. Ceux-ci se situent généralement autour de 10% dans la très grande majorité des cas et ne dépasse jamais les 18%. On peut voir ici le compromis entre un modèle plus simple produisant des résultats moins précis et un deuxième modèle plus précis mais requérant plusieurs informations supplémentaires comme la caractérisation des instructions. Ceci vient entre autre refléter dans la méthodologie du PBD l'importance du choix des

modèles dans l'abstraction des différentes couches. Comme la littérature l'avait déjà indiqué, on voit apparaître un compromis entre la complexité et la précision.

L'utilisation du deuxième modèle donne des résultats ayant une erreur assez faible pour être utilisable dans la phase dynamique du projet global. On peut juger ces résultats comme acceptables puisqu'ils résultent d'une modélisation synchrone d'un DSP asynchrone aux ALUs multiples. Dans la grande majorité des cas, l'erreur se trouve à être une légère surestimation, dû entre autres à la nature statique de l'analyse. En effet, le temps requis pour l'exécution des instructions correspond au temps moyen tel qu'extrait de l'émulateur. La durée d'une instruction est par contre variable en fonction des données traitées à cause de la nature asynchrone des ALUs, ce qui induit une variabilité qui n'est pas prise en compte dans notre modèle. Les résultats présentés correspondent au pire cas, ce qui justifie également cette légère surestimation.

## 6.2 Ordonnancement d'application

En plus de fournir des estimations en termes de consommation de mémoire et de temps d'exécution des applications choisies, notre outil nous permet également de pouvoir guider le processus d'ordonnancement de ces dernières sur un cœur du Vocallo. La méthode qui sera présentée dans cette section a pour but de maximiser l'utilisation du parallélisme offert par l'agencement parallèle des unités opératives. Pour ce faire, nous mettons de l'avant le principe de fils d'exécution multiple (*« multithreading »*) afin d'augmenter le parallélisme offert par le coté applicatif. La section commencera par introduire une nouvelle métrique servant à comparer les différentes stratégies d'ordonnancement entre elles. Nous présenterons par la suite un exemple concret en nous servant du décodage Turbo.

### 6.2.1 Taux d'utilisation

De manière à pouvoir comparer plusieurs stratégies d'ordonnancement entre elles, nous en sommes arrivés à la formulation du taux d'utilisation, défini par l'équation 2. Cette

métrique nous permet de déterminer le nombre de cycles de calcul qui sont réellement effectués par rapport au nombre théorique maximal d'opérations qui auraient pu être effectuées sur les données.

$$U_{cpu} = \frac{\sum_{i=0}^{i<N_{inst}} L_i}{N_{cycles} * N_{ALU}} \tag{2}$$

Dans cette équation, $N_{inst}$ correspond au nombre d'instructions exécutées, $L_i$ représente la durée de l'instruction i exprimé en cycles, $N_{cycles}$ est le nombre de cycles requis pour l'exécution de l'algorithme tandis que $N_{ALU}$ correspond au nombre d'ALU utilisés. De manière plus spécifique, cette métrique nous permet de voir si une stratégie favorise l'effort de calcul ou si l'exécution de celle-ci est ralentie par différents goulots d'étranglement. Parmi ceux-ci, on peut citer les accès mémoires et les dépendances de données. La figure 6-3, présentée et décrite précédemment, permet de comprendre les cycles perdus à cause des accès en mémoire locale. La figure 6-4, quant à elle, vient présenter un court exemple montrant visuellement les cycles perdus à cause des dépendances de données. On peut effectivement voir le nombre de cycles perdus par l'ALU 3 dû à la présence d'une dépendance de données entre les instructions 2 et 3.

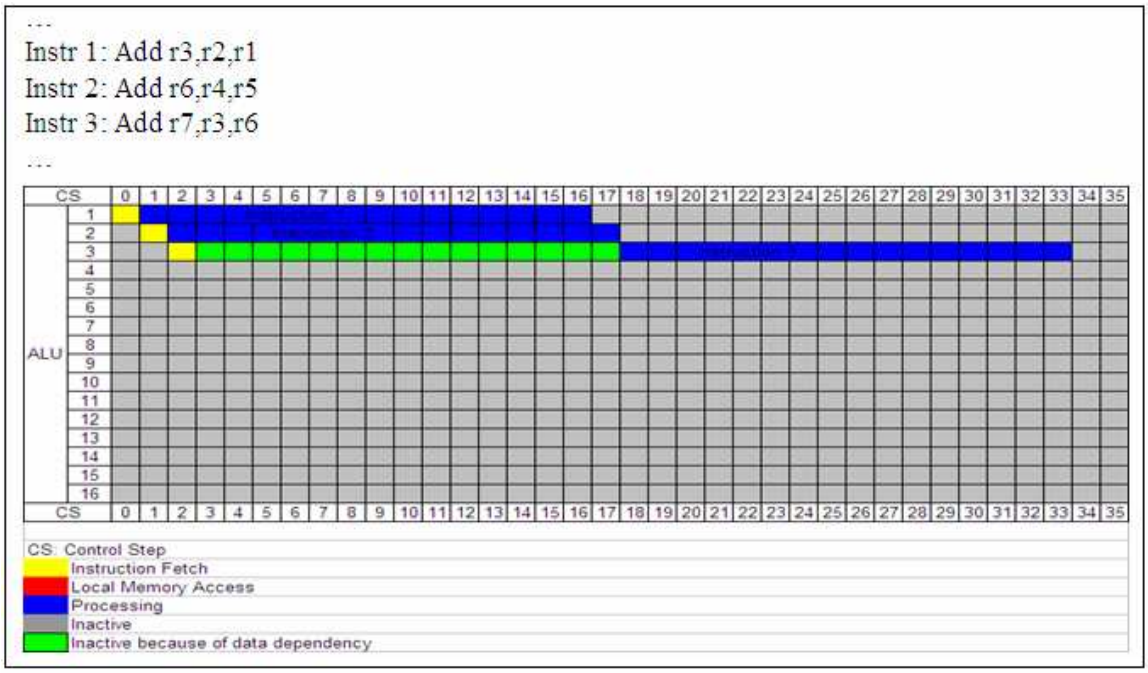


**Figure 6-4. Effet des dépendances de données**

### 6.2.2 Limite dues aux unités de contrôle

De manière à pouvoir bien saisir le parallélisme offert par un cœur du Vocallo, une application, soit le codage Turbo, a été ordonnancée en utilisant un nombre restreint d'unités opératives. Les résultats présentés ont été obtenus en exécutant le décodage Turbo traitant 660 bits par paquet avec un rapport d'encodage de 1/3. Pour une lecture en continu (*« streaming »*) de données à 128 kps, on peut assumer une charge de travail d'environ 18 Gops. Tous les résultats supposent également une quantité de mémoire et surtout de bande passante suffisante. Donc, la première stratégie élaborée consiste à traiter l'algorithme en faisant varier le nombre d'ALU présents dans un cœur. Ces résultats sont présentés à la figure 6-5.

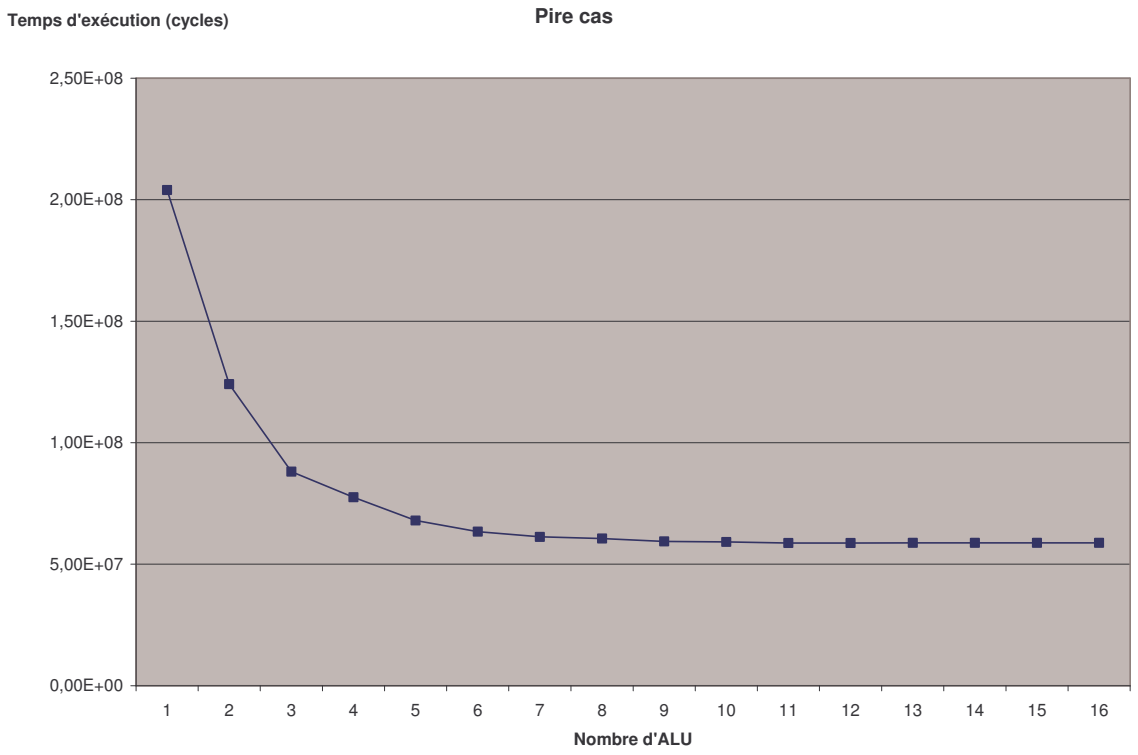


**Figure 6-5. Temps d'exécution du décodage Turbo en fonction du nombre d'Alu utilisés**

Comme on peut l'observer dans la figure précédente, le temps d'exécution tend à décroître de manière asymptotique avec l'augmentation du nombre d'unités de traitement. Ceci confirme notre intuition qu'une application possède une limite dure quant au nombre d'ALU pouvant être utilisés pour réduire le temps de traitement d'un paquet. En effet, on peut voir que dans ce cas, traiter l'application avec un nombre d'ALU supérieur à 9 n'entraîne que de très faibles diminutions du temps d'exécution. Ce phénomène est principalement dû aux dépendances de données. Le degré de parallélisme, tel que calculé à partir de l'équation 1, constitue aussi un indicatif préliminaire du nombre maximum d'ALU à allouer à un algorithme. En effet, cette limite ne peut pas prendre une valeur supérieure au plus grand degré de parallélisme détecté dans l'application.

### 6.2.3 Exemple

Pour l'ordonnancement du décodeur Turbo, nous considérons le modèle d'application suivant. En regardant le standard WIMAX, dont le décodage Turbo constitue un des éléments, on peut décomposer cette application complète en une série de segments devant être exécutés pour chacun des usagers. Un paquet de données est associé à chacun des ces usagers. La stratégie d'ordonnancement de base serait d'assigner un segment et un paquet à un cœur et de traiter de nouveaux segments de code ou de nouveaux paquets lors de la complétion du traitement du premier paquet. Par contre, de manière à augmenter l'efficacité globale du système, nous considérons le traitement sur le même cœur de plusieurs paquets en parallèle. On suppose que pour chaque segment de code à exécuter, plusieurs paquets pourraient être concaténés pour former une sous-trame. Le traitement de ces paquets peut donc être entrelacé puisqu'ils sont tous indépendants les uns des autres. Pour simuler ce genre de traitement multi-fils, plusieurs copies du CDFG sont introduites dans notre outil d'ordonnancement. Le tableau 6-7 présente les résultats du traitement de plusieurs paquets en parallèle. Ces derniers ont été produits en supposant un cœur contenant 16 unités opératives. Les temps d'exécution qui sont présentés correspondent au pire cas et l'accélération compare la stratégie courante par rapport au traitement d'un seul paquet. L'accélération est obtenue en comparant pour chaque stratégie le nombre de cycles par paquets par rapport à la stratégie où un seul paquet est traité.

**Tableau 6-7. Temps d'exécution en fonction du nombre de paquets traités**

| NB de paquets | Temps d'exécution | Taux d'utilisation | Cycles/ Paquets | Accélération |
|---|---|---|---|---|
| 1 | 58768624 | 19,69% | 58768624 | 1,00 |
| 2 | 80869040 | 28,61% | 40434520 | 1,45 |
| 3 | 107295200 | 32,35% | 35765067 | 1,64 |
| 4 | 135170960 | 34,24% | 33792740 | 1,74 |
| 5 | 163034880 | 35,48% | 32606976 | 1,80 |
| 6 | 191460336 | 36,26% | 31910056 | 1,84 |
| 7 | 219902928 | 36,83% | 31414704 | 1,87 |
| 8 | 248723024 | 37,21% | 31090378 | 1,89 |
| 9 | 277430336 | 37,53% | 30825593 | 1,91 |
| 10 | 306961856 | 37,69% | 30696186 | 1,91 |
| 11 | 336583712 | 37,81% | 30598519 | 1,92 |
| 12 | 366209248 | 37,91% | 30517437 | 1,93 |
| 13 | 395822560 | 38,00% | 30447889 | 1,93 |
| 14 | 425435776 | 38,07% | 30388270 | 1,93 |
| 15 | 455031456 | 38,14% | 30335430 | 1,94 |
| 16 | 484609568 | 38,20% | 30288098 | 1,94 |

En se basant sur le tableau précédent, on peut noter que le taux d'utilisation (Eq. 2) augmente asymptotiquement lors d'une augmentation du nombre de paquets traités en parallèle. Conséquemment, le nombre de cycles requis par paquet se trouve à diminuer avec cette même augmentation. En prenant par exemple le traitement de 5 paquets en parallèle, le débit est augmenté d'un facteur de 1,8 par rapport à la stratégie de base. On peut également penser que la consommation d'énergie nécessaire au traitement du même nombre de paquets se trouve à être globalement diminuée du même facteur. Le tableau 6-7 permet également d'effectuer un ordonnancement au niveau multi-cœur. Considérons par exemple le cas où 18 paquets doivent être décodés dans un délai maximum de 30M cycles. Une approche possible consisterait à assigner 9 paquets à 2 cœurs différents en fermant le reste des cœurs, réduisant encore ici la consommation de puissance. On peut conclure que nos résultats démontrent que le traitement simultané de plusieurs paquets dans le contexte d'un DSP asynchrone à ALU multiples peut augmenter le taux d'utilisation ainsi qu'il permet de réduire la consommation de puissance. Par contre, ce traitement multi-fils entraîne des effets secondaires qui ne sont pas visibles dans le

tableau 6-7. En effet, cette stratégie implique une augmentation de la latence et de la consommation de mémoire due à une augmentation de la taille des transferts.

## 6.3 Variabilité de la vitesse

Notre outil nous permet également d'observer un aspect particulier de la plateforme Vocallo lié à la vitesse relative entre chacun des cœurs. Cette section commence d'abord par expliquer les causes possibles de cette variation de vitesse entre les DSP, bien qu'ils soient tous implémentés de manière identique. Par la suite, l'effet de cette variabilité combinée au parallélisme au niveau applicatif sera exposé afin de faire ressortir le comportement d'un cœur et d'ainsi pouvoir élaborer une stratégie d'ordonnancement plus complète.

### 6.3.1 Cause architecturale

Il est très plausible de considérer que dans une puce comme le Vocallo, composée d'une matrice de processeurs, les caractéristiques de certains cœurs puissent différer d'un cœur à l'autre. En effet, une certaine variabilité quant à la vitesse de calcul peut exister entre les cœurs. Cette variabilité a été observée par les développeurs de la puce sans pouvoir la quantifier avec exactitude. Cette dernière peut être due entre autres aux procédés de fabrication, qui de nos jours peuvent entraîner des disparités importantes, croissantes avec la réduction de l'échelle, entre des modules similaires, dues à leur emplacement sur la puce. Il serait également possible de croire qu'une itération future de la puce puisse incorporer un système de contrôle dynamique de la tension. Ce dernier pourrait servir à ralentir un cœur en diminuant sa tension afin de réduire la consommation de puissance lors du traitement d'application dont les demandes imposées par le temps réel sont facilement respectées. La littérature fait état de plusieurs puces bénéficiant déjà de ce genre de circuiterie pour réduire la consommation de puissance [20]. Une dernière cause de l'apparition de cette variabilité repose sur la température des cœurs. En effet, la température d'un cœur peut jouer sur sa vitesse et il serait très improbable que la

température de chacun des cœurs soit identique. Cette disparité de température de fonctionnement influence certainement la vitesse des cœurs DSP.

### 6.3.2 Effet du parallélisme

Peu importe la cause de cette variabilité de vitesse d'exécution des unités de traitement, son effet peut également varier en fonction du niveau de parallélisme de l'application. Suite à une réduction ou à une augmentation de la vitesse d'un cœur par rapport à un cœur à vitesse nominale, nous avons pu observer une différence dans le comportement d'applications selon leurs niveaux de parallélisme. Afin de pouvoir simuler des cœurs avec une vitesse variable, un paramètre (*« speedup »*) a été implémenté afin de pouvoir faire varier la durée des instructions exécutées par le cœur. Il est ainsi possible de faire varier la vitesse d'un cœur afin de simuler des cœurs aux différentes tensions ou températures telles qu'exposées dans la section précédente. Des bancs d'essais spécifiques ont été développés afin d'obtenir des algorithmes dont le parallélisme peut être ajusté en modifiant le nombre de registres temporaires utilisés. Un exemple de ces algorithmes, celui possédant le niveau de parallélisme le plus élevé, est présenté à l'annexe E. Tous les codes ont la même fonctionnalité, soit l'addition de tous les entiers contenus dans un tableau. La différence entre les différentes versions de cette application provient de la méthode employée lors du traitement. En effet, pour arriver à la somme voulue, le nombre de sommes intermédiaires change de manière à modifier le niveau de parallélisme de l'application. Prenons par exemple le cas où tous les entiers sont additionnés un à un dans un même registre. Il est facile de constater que cet algorithme possède une mesure de parallélisme très faible puisque chaque opération d'addition doit attendre la complétion de la précédente sur un cœur de Vocallo. Conséquemment, une seconde mise en œuvre de l'algorithme utilisant deux registres pour produire des sommes partielles indépendantes qui sont ensuite sommées à leur tour comporte un plus grand parallélisme. Comme toutes les versions de l'application ne sont composées que d'une seule structure de contrôle, il nous est possible de déterminer leur niveau de parallélisme

respectif à l'aide de l'équation 1. Donc en utilisant notre outil sur ces applications synthétiques, il nous est possible d'observer l'effet de la variation de la vitesse d'un cœur par rapport au parallélisme offert par l'application étudiée.

Les deux figures suivantes présentent l'accélération relative de la stratégie d'ordonnancement sélectionnée en fonction du niveau de parallélisme des applications étudiées. Le niveau de parallélisme a été déterminé à l'aide de l'équation 1. Les accélérations relatives sont déterminées par rapport à la même stratégie utilisée sur un cœur fonctionnant à vitesse nominale et sont toujours rapportées en fonction du pire cas d'exécution.

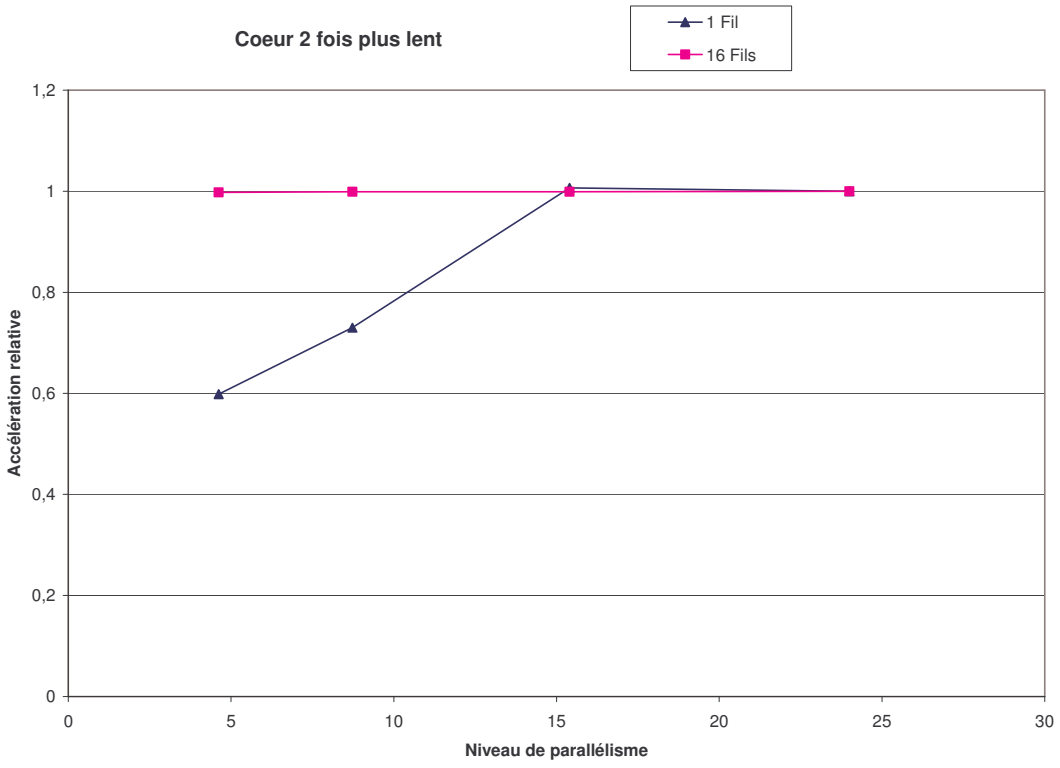


**Figure 6-6. Variation de l'accélération relative en fonction du niveau de parallélisme pour un cœur lent**

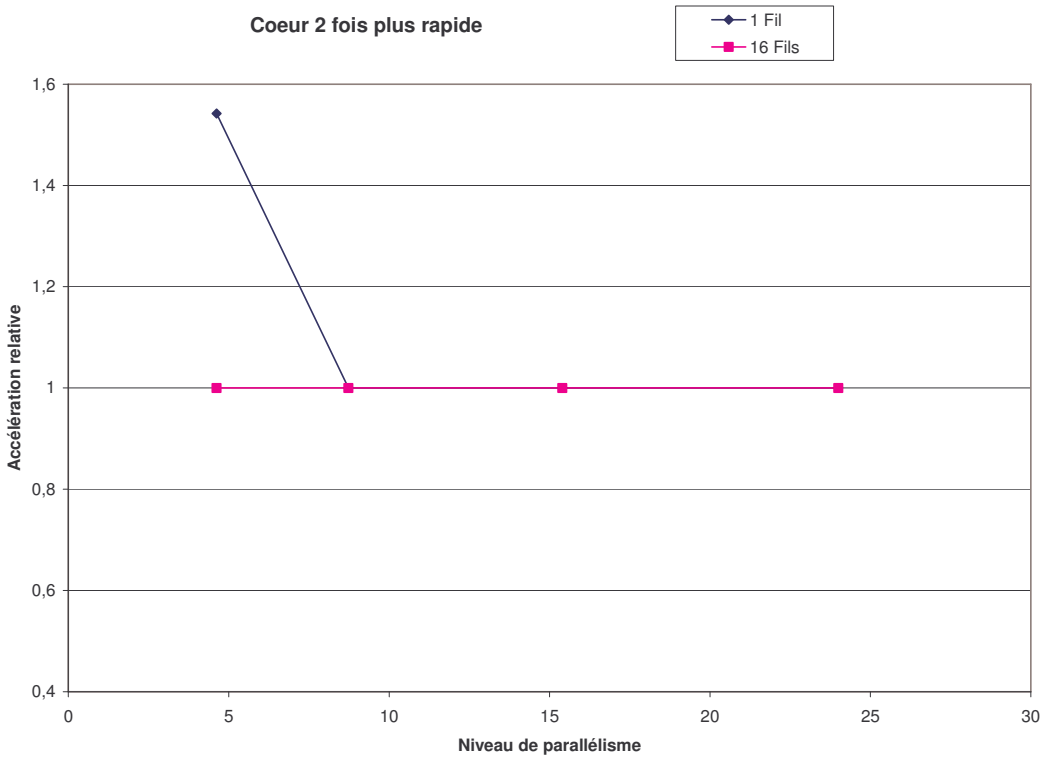


**Figure 6-7. Variation de l'accélération relative en fonction du niveau de parallélisme pour un cœur rapide**

En observant la figure 6-6, on peut conclure qu'une application possédant un niveau de parallélisme très faible est beaucoup plus affectée par le ralentissement d'un cœur qu'une application au niveau élevé de parallélisme. Dans le cas du traitement d'un seul paquet avec un algorithme peu parallèle (niveau d'environ 5), son temps d'exécution se voit diminué de 40%. En se fiant toujours à la figure 6-6, on peut voir que le temps d'exécution des algorithmes au parallélisme plus élevé est très peu affecté par le ralentissement d'un cœur. En effet, le ralentissement du cœur affecte plus particulièrement ces applications puisque le nombre d'instructions pouvant être exécutées de manière simultanée est très retreint. Une application au parallélisme élevé ne voit son accélération diminuée que faiblement puisque de nouvelles instructions peuvent être lancées sans attendre la complétion des instructions en traitement. Cette conséquence vient annuler partiellement le fait que l'on doive attendre plus longtemps pour la

complétion individuelle des instructions. La figure 6-7, quant à elle, nous apprend qu'une application possédant un fort niveau de parallélisme ne prendra pas avantage d'un gain de vitesse de traitement d'un cœur. Les effets sont par contre beaucoup plus importants pour les applications très peu parallèles, puisque l'exécution de nouvelles instructions est principalement tributaire des dépendances de données plutôt que d'un manque d'unités opératives. Suite à ces observations, on peut avancer les conclusions suivantes. Un cœur plus rapide aura avantage à exécuter des applications au faible niveau de parallélisme afin de profiter au maximum d'un gain en accélération. De manière similaire, des applications possédant un niveau de parallélisme élevé devraient être jumelées à un cœur moins rapide afin d'en minimiser les effets sur le temps d'exécution.

## 6.4 Conclusion

De manière globale, on peut affirmer que l'utilisation du deuxième modèle nous permet d'arriver à des estimations du temps d'exécution dont l'erreur est jugée acceptable. L'élaboration des modèles repose sur l'établissement d'une référence de temps fixe servant à abstraire les caractéristiques asynchrones d'un cœur du Vocallo. Les deux modèles implémentés nous prouvent que plusieurs modélisations peuvent être faites sur cette base en atteignant chacune des performances différentes. Par la suite, à l'aide de ces estimations ainsi que du calcul du taux d'utilisation, il nous est possible de comparer différentes stratégies d'ordonnancement entre elles afin de sélectionner celle qui permet d'atteindre les performances requises par le standard. Il nous a été possible de déterminer le nombre maximal d'unités opératives pouvant mener à une diminution significative du temps d'exécution. Le lien entre cette limite et le niveau de parallélisme a été exposé grâce à l'outil d'ordonnancement. Ce dernier a également servi à prendre avantage au maximum d'un cœur rapide et à minimiser l'influence d'un cœur lent en sélectionnant une application au niveau de parallélisme adéquat. La prochaine étape de cette partie du projet serait de pouvoir observer les effets des stratégies d'ordonnancement au niveau

multi-cœur sur les besoins en bande passante, en consommation de puissance et de mémoire.

# CHAPITRE 7
# CONCLUSION

Le dernier chapitre sera consacré à une courte récapitulation des travaux faisant l'objet du présent mémoire. Par la suite, les différentes limites des modèles seront exposées, ainsi que certaines améliorations possibles. La dernière section viendra clore ce mémoire en suggérant des avenues de recherches futures.

## 7.1 Synthèse des travaux

Le projet OPERA, dans lequel se sont inscrits les travaux présentés dans le cadre de ce mémoire, avait pour objectif global d'explorer différentes classes d'applications par rapport à une implémentation matérielle existante, de manière à guider les phases de configuration, de vérification et possiblement de modifications du design actuel. De manière plus spécifique à ce projet, un des buts était de faire ressortir rapidement et automatiquement si une application sélectionnée dans le domaine des télécommunications pouvait être exécutée selon les limites imposées par les différentes normes existantes sur la puce Vocallo fournie par les concepteurs d'Octasic. Pour arriver à cet objectif, la méthodologie de la conception basée sur les plateformes a été sélectionnée. Cette technique repose sur la modélisation des différentes couches du système global, soit la couche physique représentant la puce Vocallo et la couche applicative servant à abstraire les algorithmes choisis. Une fois la modélisation des couches effectuées, il suffit de les relier en propageant les contraintes du niveau applicatif vers la couche matérielle de manière à obtenir une estimation de performance selon les caractéristiques abstraites dans chacun des modèles. Différentes modélisations de ces couches permettent donc de pouvoir voir l'effet de plusieurs types de configuration afin de satisfaire aux mêmes exigences.

La première étape de ce projet de maîtrise a donc été de se familiariser avec l'ensemble des caractéristiques peu communes présentes sur le Vocallo. En effet, le Vocallo est composé d'un tableau de 15 cœurs reliés à une mémoire externe par un seul et unique bus de communication. Il est également très important de noter que chacun des ces cœurs est lui-même constitué de 16 unités opératives asynchrones (ALUs) fonctionnant en parallèle. Suite à l'examen poussé de l'architecture, le choix de la représentation intermédiaire des applications étudiées s'est fixé sur un graphe de flots de contrôle et de données. Dans sa version de base, cette représentation nous permet de définir le lien entre les instructions individuelles. Par contre, les graphiques produits souffraient souvent d'un manque de point de repère par rapport au code originel. Pour palier à ce manque lors de la visualisation, un processus de hiérarchisation a été implémenté afin de pouvoir faire varier le niveau de granularité. La caractérisation de chacun des nœuds en fonction de leurs types respectifs a également été ajoutée pour faciliter l'identification des goulots d'étranglement. La caractérisation des applications choisies se termine par l'estimation de leurs performances en termes de consommation de mémoire et de temps d'exécution. Pour la mémoire, notre infrastructure de travail, SUIF2, nous a permis aisément de faire ressortir et de compiler toutes les données utilisées par un algorithme. Malgré la simplicité de cette dernière estimation, les résultats produits se trouvent à demeurer à l'intérieur d'une marge d'erreur acceptable pour la phase suivante du projet OPERA. Les estimations du temps d'exécution ont requis l'élaboration d'un modèle synchrone représentant un cœur possédant 16 unités asynchrones de calcul. Deux modèles ont été ainsi développés. La comparaison de ces deux modèles a fait ressortir le compromis entre la précision du modèle et sa complexité. La réunion des estimations mémoire et du temps d'exécution sous un format de graphe xml sert finalement à la représentation des algorithmes en C ou C++ pour la phase dynamique du projet.

## 7.2 Limitations des travaux

Tout au cours de ce projet, plusieurs limites ont été atteintes dû à plusieurs causes dont la quantité de temps fini à notre disposition, certaines limites des outils utilisés ainsi qu'un manque de certaines informations pertinentes au fonctionnement de la puce et de ses outils de développement.

La première limite importante rencontrée concerne la bibliothèque de passage de la représentation intermédiaire Suif vers un code en assembleur. En effet, l'utilisation de l'infrastructure SUIF2 nous permet rapidement d'arriver à une liste d'instructions ainsi qu'un graphe de flot de contrôle pouvant être transformé vers notre format de graphe. Par contre, les bibliothèques déjà implémentées et responsables de la production du code en assembleur sont basées sur des modèles de processeurs standards tel le ALPHA ou le X86. L'implémentation d'une nouvelle bibliothèque se basant sur le jeu d'instruction des unités opératives auraient demandé de nombreuses informations sur le compilateur utilisé pas Octasic. Malheureusement, ces informations n'étaient pas disponibles. Malgré tout, l'utilisation de la bibliothèque développée pour le processeur ALPHA a quand même permis d'arriver à des estimations du temps d'exécution très acceptables. Une autre limite imposée par l'utilisation de SUIF2 est la nature statique de son fonctionnement. En effet, lors de la caractérisation de la durée des instructions, il a été déterminé que celles-ci étaient dépendantes entre autre des données traitées. Suif2 ne nous permettait malheureusement pas de pouvoir simuler l'exécution d'un algorithme à l'aide de données réelles, ce qui nous a forcé à utiliser les valeurs obtenues dans le pire cas.

Le deuxième modèle implémenté comporte également une abstraction pouvant finalement venir influer sur la précision des estimations du temps d'exécution. L'effet des accès aux registres n'a pas été inclus dans les étapes de traitement d'une instruction. Au départ, ceux-ci n'ont pas été pris en compte puisque le temps requis pour ce type d'accès a été jugé négligeable par rapport au temps nécessaire à un accès à la mémoire locale. À l'aide de l'émulateur fourni par Octasic, nous avons par contre observé plusieurs cas de figures où de multiples accès simultanés aux registres causaient l'apparition d'un goulot

d'étranglement. La prise en considération des accès aux registres demanderait l'élaboration d'un nouveau modèle puisque le temps nécessaire pour accéder à un registre est inférieur à celui d'un accès en mémoire et que le choix de la durée d'une étape de contrôle doit représenter la plus petite mesure de temps fixe.

## 7.3 Possibilités de travaux futurs

La première possibilité de continuité ou d'amélioration de ce projet concerne l'outil de partitionnement. Dans le cadre du projet, la seule base de partitionnement est la consommation de mémoire. Tel que discuté dans le chapitre 5, de nombreux autres critères auraient pu être utilisés.

La plus grande possibilité de continuité repose sur l'analyse des impacts du choix de stratégies au niveau multi-cœur. En effet, le traitement multi-fils, tel que proposé dans le chapitre 6, induit une augmentation de la taille des transferts entre la mémoire externe et les cœurs. Cette augmentation est majoritairement causée par l'envoi et le retour de plusieurs séries de données. Dans la présente version de nos outils, la quantité de mémoire requise pour stocker le code et toutes les données représente la seule limite matérielle quant au nombre d'usagers à traiter simultanément. Par contre, la bande passante sur le bus pouvant être accordée à chaque cœur devrait aussi venir limiter le nombre d'usagers. Comme la première phase du projet OPERA ne s'attaque qu'au comportement des applications dans un cœur, la phase dynamique aura pour but de permettre la visualisation des effets au niveau de l'interaction entre les cœurs.

# RÉFÉRENCES

[1] ACKLAND, B., ANESKO, A., BRINTHAUPT, D., DAUBERT, S.J., KALAVADE, A., KNOBLOCH, J., MICCA, E., MOTURI, M., NICOL, C.J., O'NEIL, J.H., OTHMER, J. SACKINGER, E., SIGNH, K.J., SWEET, J., TERMAN, C.J. and WILLIAMS, J. (2000). A single-chip, 1.6-billion, 16-b MAC/s multiprocessor DSP. IEEE J. Solid-State Circuits, vol. 35, no. 3, pp. 412–424, Mars 2000.

[2] AIGNER, G., DIWAN, A., HEINE, D., LAM, M., MOORE, D., MURPHY, B. and SAPUNTZAKIS, C. (2000). An overview of the SUIF2 Compiler Infrastructure. Computer Systems Laboratory of Stanford University, Août 2000. Consulté le 4 Janvier 2010, tiré de http://suif.stanford.edu/suif/suif2/

[3] AIGNER, G., DIWAN, A., HEINE, D., LAM, M., MOORE, D., MURPHY, B. and SAPUNTZAKIS, C. (2000). The SUIF program representation. Computer Systems Laboratory of Stanford University. Août 2000. Consulté le 4 Janvier 2010, tiré de http://suif.stanford.edu/suif/suif2/

[4] AIGNER, G., DIWAN, A., HEINE, D., LAM, M., MOORE, D., MURPHY, B. and SAPUNTZAKIS, C. (2000). The basic SUIF programming guide. Computer Systems Laboratory of Stanford University, Août 2000. Consulté le 4 Janvier 2010, tiré de http://suif.stanford.edu/suif/suif2/

[5] AMME, W., BRAUN, P., ZEHENDNER, E. and THOMASSET, F. (1998). Data Dependence Analysis of Assembly Code. Proceedings of the 1998 international Conference on Parallel Architectures and Compilation Techniques, Oct. 1998, Page(s):340 – 347.

[6] BENNOUR, I., LANGEVIN, M. and El ABOULHAMID, M. (1996). Performance Analysis for Hardware Software Co-synthesis. Canadian Conference

on Electrical and Computer Engineering, Volume 1, Mai 1996, Page(s):162 – 165.

[7] C-Port Corp (2001). C-5 Network Processor Architecture Guide, North Andover, MA, Mai 31, 2001. Consulté le 4 Janvier 2010, tiré de http://www.freescale.com/files/netcomm/doc/ref_manual/C5NPD0-G.pdf?fsrch=1

[8] CARLONI, L., DE BERNARDINIS, F., PINELLO, C., SANGIOVANNI-VINCENTELLI, A. and SGROI, M. (2005) Platform-Based Design for Embedded Systems. University of California at Berkeley, 2005.

[9] DUTTA, S., JENSEN, R. and RIECKMANN, A. (2001). Viper: A multiprocessor SOC for advanced set-top box and digital TV systems. IEEE Des. Test. Comput., vol. 18, no. 5, pp. 21–31, Sep./Oct. 2001.

[10] GALANIS, M., MILIDONIS, A., THEODORIDIS, G., SOUDRIS, D. and GOUTIS, C. (2005). A Framework for Partitioning Computational Intensive Applications in Hybrid Reconfigurable Platforms. Proceedings of the 19th IEEE international Parallel and Distributed Processing Symposium, Avril 2005, Page(s):8.

[11] GUPTA, R. and LEE, S. (1992). Exploiting Parallelism on a Fine-Grained MIMD Architecture Based Upon Channel Queues. International Journal of Parallel Programming, Vol. 21, No 3, 1992.

[12] GUPTA, T.V.K., SHARMA, P., BALAKRISHNAN, M. and MALIK, S. (2000). Processor evaluation in an embedded systems design environment. 13th International Conference on VLSI Design, 3-7 Jan. 2000, Page(s):98 – 103.

[13] HOLLOWAY, G. and SMITH, M. (2001). An Introduction to Machine SUIF and its Portable Libraries for Analysis and Optimization. Division of Engineering and Applied Science, Harvard University, Octobre 2001. Consulté le 4 Janvier 2010, tiré de http://www.eecs.harvard.edu/hube/software/software.html.

[14] HOLLOWAY, G. and SMITH, M. (2001). The Machine-SUIF Machine Library. Division of Engineering and Applied Science, Harvard University, Octobre 2001. Consulté le 4 Janvier 2010, tiré de

http://www.eecs.harvard.edu/hube/software/software.html.

[15] HOLLOWAY, G. and SMITH, M. (2001). The Machine-SUIF Control Flow Graph Library. Division of Engineering and Applied Science, Harvard University, Octobre 2001. Consulté le 4 Janvier 2010, tiré de http://www.eecs.harvard.edu/hube/software/software.html.

[16] HOLZER, M., KNERR, B., BELANOVI'C, P. and RUPP, M. (2006). Efficient Design Methods for Embedded Communication Systems. EURASIP Journal on Embedded Systems, Vol. 2006, Article ID 64913, Pages 1–18.

[17] HWANG, Y., ABDI, S. and GAJSKI, D. (2008). Cycle-approximate Retargetable Performance Estimation at the Transaction Level. Proceedings of the 2008 Design, Automation and Test in Europe, DATE 2008.

[18] KANDEMIR, M., OZTURK, O. and KARAKOY, M. (2004) Dynamic on-chip memory management for chip multiprocessors. In Proceedings of International Conference on Compilers, Architectures, and Synthesis for Embedded Systems, Washington D.C., Septembre 2004.

[19] KNUDSEN, P. and MADSEN, J. (1999). Graph Based Communication Analysis for Hardware/Software Codesign. Proceedings of the 7th international Workshop on Hardware/Software Codesign, Mai 1999, Page(s):131 – 135.

[20] LEWIS, M. and BRACKENBURY, L. (2001). Low power asynchronous DSP for digital mobile phones. IEEE Seminar on Low Power IC Design. 2001. Page(s): 13/1 – 13/6.

[21] LI, F. and KANDEMIR, M. (2005). Locality-conscious workload assignment for array-based computations in MPSOC architectures. Design automation conference 2005**,** 13-17 Juin 2005, pp. 95- 100.

[22] MARTIN, G. (2006). Overview of the MPSoC Design Challenge. DAC 2006, 24-28 Juillet, 2006, San Francisco.

[23] MEFTALI, S., GHARSALLI, F., ROUSSEAU, F. and JERRAYA, A. (2001). An Optimal Memory Allocation for Application-Specific Multiprocessor System-

on-Chip. In Proceedings of the International Symposium on Systems Synthesis, Montreal, Canada, 2001.

[24] NAMBALLA, R. (2003). CHESS: A tool for CDFG extraction and high-level synthesis of VLSI. 8 Juillet, 2003.

[25] PLOURDE, F. (2009). Développement d'une méthodologie d'estimation de l'utilisation des ressources mémoires sur une puce DSP multi-noyaux. École de technologie supérieure, Montréal, Québec, Canada.

[26] RUSSEL, J.T. and JACOME, M.F. (2003). Architecture-level performance evaluation of component-based embedded systems. Proceedings of the 2003 Design, Automation and Test in Europe, DATE 2003, Page(s): 396-401.

[27] SANDER, G. Visualization of Compiler Graphs. Consulté le 4 Janvier 2010, tiré de http://rw4.cs.uni-sb.de/users/sander/html/gsvcg1.html

[28] SANGIOVANNI-VINCENTELLI, A. (2005). Platform-Based Design. , University of California at Berkeley, 2005.

[29] SANGIOVANNI-VINCENTELLI, A. and MARTIN, G. (2001). Platform-Based Design and Software Design Methodology for Embedded System. Design and Test of Computers, Volume 18, Issue 6, Nov.-Dec. 2001, Page(s): 23 – 33.

[30] SMIT, G., ROSIEN, M., GUO, Y. and HEYSTERS, P. (2003). Overview of the tool-flow for the Montium Processing Tile. Enschede, The Netherlands, 2003.

[31] Texas Instruments Inc (2004). OMAP5912 Multimedia Processor Device Overview and Architecture Reference Guide. Dallas, TX, Mars 2004. Consulté le 4 Janvier 2010, tiré de http://www.ti.com

[32] WOLF, W., JERRAYA, A. and MARTIN, G. (2008). Multiprocessor System-on-Chip (MPSoC) Technology. IEEE Transactions on computer-aided design of integrated circuits and systems, Vol. 27, No. 10, Octobre 2008.

[33] WU, Q., WANG, Y., BIAN, J., WU, W. and XUE, H. (2002). A Hierarchical CDFG as Intermediate Representation for Hardware/Software. IEEE 2002 International Conference on Communications, Circuits and Systems and West Sino Expositions, Volume 2, 29 June-1 July 2002 Page(s):1429 - 1432 vol.2.

[34] XUE, L., OZTURK, O., LI, F. and KANDEMIR, M. (2006). Dynamic Partitioning of Processing and Memory Resources in Embedded MPSoC Architectures. Proceedings of the 2006 Design, Automation and Test in Europe, DATE 2006, Volume 1, 6-10 Mars 2006 Page(s):6 pp.

# ANNEXES

## A) Code de l'application synthétique servant de démonstration pour l'outil de partitionnement

```
typedef struct{
  int x[100];
  int y[100];
  int z[100];
}pos;

int test(int in ,pos* in1, pos* in2, pos* in3, pos* in4)
{
  int a,b,c,i;

  a = 0;
  b = 0;
  c = 0;

  for(i = 0; i < 100; i++)
  {
     a = a + in1->y[i];
     b = b + in1->z[i];
     c = c + in1->x[i];
  }

  for(i = 0; i < 100; i++)
  {
     a = a + in2->y[i];
     b = b + in2->z[i];
     c = c + in2->x[i];
  }

  for(i = 0; i < 100; i++)
  {
     a = a + in3->y[i];
     b = b + in3->z[i];
     c = c + in3->x[i];
  }

  for(i = 0; i < 100; i++)
  {
     a = a + in4->y[i];
     b = b + in4->z[i];
     c = c + in4->x[i];
  }

  c = in + a + b + c;

  return(c);

}
```

**B) Représentation XML du partitionnement final avec une mémoire locale de 10K**

```
<task name="test">     <mem>
          <input>38560</input>
          <output>32</output>
          <used>38688</used>
          <code>10880</code>
     </mem>
     <cpu_cycles=3274/>
     <in>n1</in>
     <out>n1</out>
</task>
```

### C) Représentation XML du partitionnement final avec une mémoire locale de 5K


```
<task name="test"><mem>
     <input>38560</input>
     <output>32</output>
     <used>38688</used>
     <code>10880</code>
</mem>
<cpu cycles=3274/>
<in>n1</in>
<out>n1</out>
     <task name="test1">     <mem>
          <input>28896</input>
          <output>96</output>
          <used>29024</used>
          <code>8000</code>
     </mem>
     <cpu cycles=2412/>
     <in>n1</in>
     <out>test2</out>
     </task>
     <task name="test2">     <mem>
          <input>9792</input>
          <output>32</output>
          <used>9792</used>
          <code>2880</code>
     </mem>
     <cpu cycles=862/>
     <in>test1</in>
     <out>n1</out>
     </task>
</task>
```

**D) Représentation XML du partitionnement final avec une mémoire locale de 3K**


```
<task name="test"><mem>
     <input>38560</input>
     <output>32</output>
     <used>38688</used>
     <code>10880</code>
</mem>
<cpu_cycles=3274/>
<in>n1</in>
<out>n1</out>
     <task name="test1">     <mem>
          <input>9632</input>
          <output>96</output>
          <used>9760</used>
          <code>3136</code>
     </mem>
     <cpu_cycles=804/>
     <in>n1</in>
     <out>test2</out>
     </task>
     <task name="test2">     <mem>
          <input>19392</input>
          <output>96</output>
          <used>19392</used>
          <code>4864</code>
     </mem>
     <cpu_cycles=1608/>
     <in>test1</in>
     <out>test3</out>
     </task>
     <task name="test3">     <mem>
          <input>9792</input>
          <output>32</output>
          <used>9792</used>
          <code>2880</code>
     </mem>
     <cpu_cycles=862/>
     <in>test2</in>
     <out>n1</out>
     </task>
</task>
```

## E) Algorithme synthétique servant l'effet du parallélisme par rapport à la vitesse d'un cœur

```
int sum(int data[64])
{
        int sum;
        int sum1,sum2,sum3,sum4,sum5,sum6,sum7,sum8;
        int sum12,sum34,sum56,sum78;
        int sum14,sum58;

        sum = 0;

        sum1 = 0;
        sum2 = 0;
        sum3 = 0;
        sum4 = 0;
        sum5 = 0;
        sum6 = 0;
        sum7 = 0;
        sum8 = 0;

        sum12 = 0;
        sum34 = 0;
        sum56 = 0;
        sum78 = 0;

        sum14 = 0;
        sum58 = 0;

        sum1 = sum1 + data[0];
        sum1 = sum1 + data[1];
        sum1 = sum1 + data[2];
        sum1 = sum1 + data[3];
        sum1 = sum1 + data[4];
        sum1 = sum1 + data[5];
        sum1 = sum1 + data[6];
        sum1 = sum1 + data[7];

        sum2 = sum2 + data[8];
        sum2 = sum2 + data[9];
        sum2 = sum2 + data[10];
        sum2 = sum2 + data[11];
        sum2 = sum2 + data[12];
        sum2 = sum2 + data[13];
        sum2 = sum2 + data[14];
        sum2 = sum2 + data[15];

        sum3 = sum3 + data[16];
        sum3 = sum3 + data[17];
        sum3 = sum3 + data[18];
        sum3 = sum3 + data[19];
        sum3 = sum3 + data[20];
        sum3 = sum3 + data[21];
        sum3 = sum3 + data[22];
        sum3 = sum3 + data[23];
```

```
        sum4 = sum4 + data[24];
        sum4 = sum4 + data[25];
        sum4 = sum4 + data[26];
        sum4 = sum4 + data[27];
        sum4 = sum4 + data[28];
        sum4 = sum4 + data[29];
        sum4 = sum4 + data[30];
        sum4 = sum4 + data[31];

        sum5 = sum5 + data[32];
        sum5 = sum5 + data[33];
        sum5 = sum5 + data[34];
        sum5 = sum5 + data[35];
        sum5 = sum5 + data[36];
        sum5 = sum5 + data[37];
        sum5 = sum5 + data[38];
        sum5 = sum5 + data[39];

        sum6 = sum6 + data[40];
        sum6 = sum6 + data[41];
        sum6 = sum6 + data[42];
        sum6 = sum6 + data[43];
        sum6 = sum6 + data[44];
        sum6 = sum6 + data[45];
        sum6 = sum6 + data[46];
        sum6 = sum6 + data[47];

        sum7 = sum7 + data[48];
        sum7 = sum7 + data[49];
        sum7 = sum7 + data[50];
        sum7 = sum7 + data[51];
        sum7 = sum7 + data[52];
        sum7 = sum7 + data[53];
        sum7 = sum7 + data[54];
        sum7 = sum7 + data[55];

        sum8 = sum8 + data[56];
        sum8 = sum8 + data[57];
        sum8 = sum8 + data[58];
        sum8 = sum8 + data[59];
        sum8 = sum8 + data[60];
        sum8 = sum8 + data[61];
        sum8 = sum8 + data[62];
        sum8 = sum8 + data[63];

        sum12 = sum1 + sum2;
        sum34 = sum3 + sum4;
        sum56 = sum5 + sum6;
        sum78 = sum7 + sum8;

        sum14 = sum12 + sum34;
        sum58 = sum56 + sum78;

        sum = sum14 + sum58;

        return (sum);
}
```

## F) Représentation graphique de l'addition vectorielle

```
graph: { title: "Vector Add"

x: 30
y: 30
height: 800
width: 500
stretch: 60
shrink: 100
layoutalgorithm: minbackward
node.borderwidth: 3
node.color: white
node.textcolor: black
node.bordercolor: black
edge.color: black

node: { title:"0" label:"Entry" info1:
"Worst time: 2237.000000
Best time: 2237.000000
Worst nb of ops: 96
Best nb of ops: 96
Size of code: 38
Parallelism: 0.387755"
 }
graph: { title:"2" label:"Seq" folding: 1
info1:
"Worst time: 121.000000
Best time: 121.000000
Worst nb of ops: 7
Best nb of ops: 7
Size of code: 7
Parallelism: 3.500000"
node: { title:"100" label:"Start Node" }
node: { title:"200" label:"mov" info1:
"ASAP: 1
ALAP: 2
ALU: 35
reg_in: 32
reg_out: 32"}
node: { title:"300" label:"mov" info1:
"ASAP: 1
ALAP: 2
ALU: 57
reg_in: 32
reg_out: 32"}
node: { title:"400" label:"mov" info1:
"ASAP: 1
ALAP: 2
ALU: 73
reg_in: 32
reg_out: 32"}
```

```
node: { title:"500" label:"ldil" info1:
"ASAP: 1
ALAP: 1
ALU: 10
mem_in: 32
mem_out: 32"}
node: { title:"600" label:"mov" info1:
"ASAP: 2
ALAP: 2
ALU: 89
reg_in: 32
reg_out: 32"}
node: { title:"700" label:"End Node" }
edge: {sourcename:"0" targetname:"100" }
edge: {sourcename:"100" targetname:"200" }
edge: {sourcename:"100" targetname:"300" }
edge: {sourcename:"100" targetname:"400" }
edge: {sourcename:"100" targetname:"500" }
edge: {sourcename:"500" targetname:"600" }
edge: {sourcename:"600" targetname:"700" }
edge: {sourcename:"400" targetname:"700" }
edge: {sourcename:"300" targetname:"700" }
edge: {sourcename:"200" targetname:"700" }
}
graph: { title:"701" label:"Loop" folding: 1
info1:
"Worst time: 2091.000000
Best time: 2091.000000
Worst time: 3
Best time: 3
Worst nb of ops: 87
Best nb of ops: 87
Size of code: 29
Parallelism: 4.142857"
node: { title:"800" label:"Start Node" }
node: { title:"900" label:"mull" info1:
"ASAP: 3
ALAP: 7
ALU: 335
mem_in: 64
mem_out: 32"}
node: { title:"1000" label:"mov" info1:
"ASAP: 3
ALAP: 6
ALU: 186
mem_in: 32
mem_out: 32"}
node: { title:"1100" label:"mov" info1:
"ASAP: 4
ALAP: 7
ALU: 380
reg_in: 32
```

```
reg_out: 32"}
node: { title:"1200" label:"addl" info1:
"ASAP: 5
ALAP: 8
ALU: 583
reg_in: 64
reg_out: 32"}
node: { title:"1300" label:"ldwu" info1:
"ASAP: 3
ALAP: 3
ALU: 10
mem_in: 64
mem_out: 16"}
node: { title:"1400" label:"mov" info1:
"ASAP: 4
ALAP: 4
ALU: 69
reg_in: 16
reg_out: 16"}
node: { title:"1500" label:"mov" info1:
"ASAP: 5
ALAP: 5
ALU: 131
reg_in: 16
reg_out: 32"}
node: { title:"1600" label:"mull" info1:
"ASAP: 3
ALAP: 7
ALU: 407
mem_in: 64
mem_out: 32"}
node: { title:"1700" label:"mov" info1:
"ASAP: 3
ALAP: 6
ALU: 211
mem_in: 32
mem_out: 32"}
node: { title:"1800" label:"mov" info1:
"ASAP: 4
ALAP: 7
ALU: 445
reg_in: 32
reg_out: 32"}
node: { title:"1900" label:"addl" info1:
"ASAP: 5
ALAP: 8
ALU: 601
reg_in: 64
reg_out: 32"}
node: { title:"2000" label:"ldwu" info1:
"ASAP: 3
ALAP: 3
```

```
ALU: 35
mem_in: 64
mem_out: 16"}
node: { title:"2100" label:"mov" info1:
"ASAP: 4
ALAP: 4
ALU: 89
reg_in: 16
reg_out: 16"}
node: { title:"2200" label:"mov" info1:
"ASAP: 5
ALAP: 5
ALU: 153
reg_in: 16
reg_out: 32"}
node: { title:"2300" label:"addl" info1:
"ASAP: 6
ALAP: 6
ALU: 245
reg_in: 64
reg_out: 32"}
node: { title:"2400" label:"zapnot" info1:
"ASAP: 7
ALAP: 7
ALU: 461
mem_in: 32
reg_in: 32
reg_out: 16"}
node: { title:"2500" label:"mull" info1:
"ASAP: 3
ALAP: 7
ALU: 489
mem_in: 64
mem_out: 32"}
node: { title:"2600" label:"mov" info1:
"ASAP: 3
ALAP: 6
ALU: 266
mem_in: 32
mem_out: 32"}
node: { title:"2700" label:"mov" info1:
"ASAP: 4
ALAP: 7
ALU: 526
reg_in: 32
reg_out: 32"}
node: { title:"2800" label:"addl" info1:
"ASAP: 5
ALAP: 8
ALU: 617
reg_in: 64
reg_out: 32"}
```

```
node: { title:"2900" label:"stw" info1:
"ASAP: 8
ALAP: 8
ALU: 633
reg_in: 16
reg_out: 64"}
node: { title:"3000" label:"ldil" info1:
"ASAP: 3
ALAP: 4
ALU: 106
mem_in: 32
mem_out: 32"}
node: { title:"3100" label:"mov" info1:
"ASAP: 4
ALAP: 5
ALU: 169
reg_in: 32
reg_out: 32"}
node: { title:"3200" label:"addl" info1:
"ASAP: 5
ALAP: 6
ALU: 291
mem_in: 32
mem_out: 32
reg_in: 32"}
node: { title:"3300" label:"mov" info1:
"ASAP: 6
ALAP: 7
ALU: 542
reg_in: 32
reg_out: 32"}
node: { title:"3400" label:"ldil" info1:
"ASAP: 3
ALAP: 7
ALU: 558
mem_in: 32
mem_out: 32"}
node: { title:"3500" label:"cmpult" info1:
"ASAP: 7
ALAP: 8
ALU: 650
mem_in: 32
mem_out: 64
reg_in: 32"}
node: { title:"3600" label:"bne" info1:
"ASAP: 9
ALAP: 9
ALU: 675
reg_in: 64"}
edge: {sourcename:"700" targetname:"800" }
backedge: {sourcename:"3600" targetname:"800" }
edge: {sourcename:"800" targetname:"900" }
```

```
edge: {sourcename:"800" targetname:"1000" }
edge: {sourcename:"1000" targetname:"1100" }
edge: {sourcename:"1100" targetname:"1200" }
edge: {sourcename:"900" targetname:"1200" }
edge: {sourcename:"800" targetname:"1300" }
edge: {sourcename:"1300" targetname:"1400" }
edge: {sourcename:"1400" targetname:"1500" }
edge: {sourcename:"800" targetname:"1600" }
edge: {sourcename:"800" targetname:"1700" }
edge: {sourcename:"1700" targetname:"1800" }
edge: {sourcename:"1800" targetname:"1900" }
edge: {sourcename:"1600" targetname:"1900" }
edge: {sourcename:"800" targetname:"2000" }
edge: {sourcename:"2000" targetname:"2100" }
edge: {sourcename:"2100" targetname:"2200" }
edge: {sourcename:"1500" targetname:"2300" }
edge: {sourcename:"2200" targetname:"2300" }
edge: {sourcename:"2300" targetname:"2400" }
edge: {sourcename:"800" targetname:"2500" }
edge: {sourcename:"800" targetname:"2600" }
edge: {sourcename:"2600" targetname:"2700" }
edge: {sourcename:"2700" targetname:"2800" }
edge: {sourcename:"2500" targetname:"2800" }
edge: {sourcename:"2400" targetname:"2900" }
edge: {sourcename:"800" targetname:"3000" }
edge: {sourcename:"3000" targetname:"3100" }
edge: {sourcename:"3100" targetname:"3200" }
edge: {sourcename:"3200" targetname:"3300" }
edge: {sourcename:"800" targetname:"3400" }
edge: {sourcename:"3300" targetname:"3500" }
edge: {sourcename:"3400" targetname:"3500" }
edge: {sourcename:"3500" targetname:"3600" }
edge: {sourcename:"2900" targetname:"3600" }
edge: {sourcename:"2800" targetname:"3600" }
edge: {sourcename:"1900" targetname:"3600" }
edge: {sourcename:"1200" targetname:"3600" }
}
graph: { title:"3601" label:"Seq" folding: 1
info1:
"Worst time: 25.000000
Best time: 25.000000
Worst nb of ops: 2
Best nb of ops: 2
Size of code: 2
Parallelism: 2.000000"
node: { title:"3700" label:"Start Node" }
node: { title:"3800" label:"ret" info1:
"ASAP: 10
ALAP: 10
ALU: 9
reg_in: 64"}
edge: {sourcename:"3600" targetname:"3700" }
```

```
edge: {sourcename:"3700" targetname:"3800" }
}
node: { title:"4000" label:"Exit" }
edge: {sourcename:"700" targetname:"800" }
edge: {sourcename:"3600" targetname:"3700" }
edge: {sourcename:"0" targetname:"100" }
edge: {sourcename:"3800" targetname:"4000" }
}
```