Méthodologie de compilation d'algorithmes de traitement du signal pour les processeurs en virgule fixe sous contrainte de précision

Daniel Ménard

To cite this version:

Daniel Ménard. Méthodologie de compilation d'algorithmes de traitement du signal pour les processeurs en virgule fixe sous contrainte de précision. Traitement du signal et de l'image [eess.SP]. Université Rennes 1, 2002. Français. NNT : . tel-00609159

HAL Id: tel-00609159
https://theses.hal.science/tel-00609159

Submitted on 18 Jul 2011

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

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

N^o d'ordre : 2790

THÈSE

présentée

DEVANT L'UNIVERSITÉ DE RENNES 1

pour obtenir

le grade de : *DOCTEUR DE L'UNIVERSITÉ DE RENNES 1*

Mention : TRAITEMENT DU SIGNAL

par

Daniel MENARD

Équipe d'accueil : Laboratoire d'Analyse des Systèmes de Traitement de l'Information à Lannion

École Doctorale : Mathématiques, Informatique, Signal, Électronique et Télécommunications

Composante Universitaire : École Nationale Supérieure de Sciences Appliquées et de Technologie

Méthodologie de compilation d'algorithmes de traitement du signal pour les processeurs en virgule fixe sous contrainte de précision

SOUTENUE LE 12 décembre 2002 devant la commission d'examen

COMPOSITION DU JURY :

MM.	E.	MARTIN	Professeur à l'Université de Bretagne Sud		Président
	D.	DEMIGNY	Professeur à l'ENSEA		Rapporteurs
	T.	RISSET	Chargé de Recherche à l'INRIA (HDR)	ENS Lyon	
	F.	CHAROT	Chargé de Recherche à l'INRIA	IRISA	Examinateurs
	P.	LE GUERNIC	Directeur de Recherche à l'INRIA	IRISA	
	O.	SENTIEYS	Professeur à l'Université de Rennes 1	ENSSAT	

Résumé

L'implantation efficace des algorithmes de traitement numérique du signal (TNS) dans les systèmes embarqués requiert l'utilisation de l'arithmétique virgule fixe afin de satisfaire les contraintes de coût, de consommation et d'encombrement exigées par ces applications. Le codage manuel des données en virgule fixe est une tâche fastidieuse et source d'erreurs. De plus, la réduction du temps de mise sur le marché des applications exige l'utilisation d'outils de développement de haut niveau, permettant d'automatiser certaines tâches. Ainsi, le développement de méthodologies de codage automatique des données en virgule fixe est nécessaire. Dans le cadre des processeurs programmables de traitement du signal, la méthodologie doit déterminer le codage optimal, permettant de maximiser la précision et de minimiser le temps d'exécution et la taille du code. L'objectif de ce travail de recherche est de définir une nouvelle méthodologie de compilation d'algorithmes spécifiés en virgule flottante au sein d'architectures programmables en virgule fixe sous contrainte de respect des critères de qualité associés à l'application. Ce travail de recherche s'articule autour de trois points principaux.

Le premier aspect de notre travail a consisté à définir la structure de la méthodologie. L'analyse de l'influence de l'architecture sur la précision des calculs montre la nécessité de tenir compte de l'architecture cible pour obtenir une implantation optimisée d'un point de vue du temps d'exécution et de la précision. De plus, l'étude de l'interaction entre les phases de compilation et de codage des données permet de définir le couplage nécessaire entre les phases de conversion en virgule fixe et le processus de génération de code.

Le second aspect de ce travail de recherche concerne l'évaluation de la précision au sein d'un système en virgule fixe à travers la détermination du Rapport Signal à Bruit de Quantification (RSBQ). Une méthodologie permettant de déterminer automatiquement l'expression analytique du RSBQ en fonction du format des données en virgule fixe est proposée. Dans un premier temps, un nouveau modèle de bruit est présenté. Ensuite, les concepts théoriques pour déterminer la puissance du bruit de quantification en sortie des systèmes linéaires et des systèmes non-linéaires et non-récursifs sont détaillés. Finalement, la méthodologie mise en œuvre pour obtenir automatiquement l'expression du RSBQ dans le cadre des systèmes linéaires est exposée.

Le troisième aspect de ce travail de recherche correspond à la mise en œuvre de la méthodologie de codage des données en virgule fixe. Dans un premier temps, la dynamique des données est déterminée à l'aide d'une approche analytique combinant deux techniques différentes. Ces informations sur la dynamique permettent de déterminer la position de la virgule de chaque donnée en tenant compte de la présence éventuelle de bits de garde au sein de l'architecture. Pour obtenir un format des données en virgule fixe complet, la largeur de chaque donnée est déterminée en prenant en compte l'ensemble des types des données manipulées au sein du DSP. La méthode sélectionne la séquence d'instructions permettant de fournir une précision suffisante en sortie de l'algorithme et de minimiser le temps d'exécution du code. La dernière phase du processus de codage correspond à l'optimisation du format des données en vue d'obtenir une implantation plus efficace. Les différentes opérations de recadrage sont déplacées afin de minimiser le temps d'exécution global tant que la précision en sortie de l'algorithme est supérieure à la contrainte. Deux types de méthode ont été mis en œuvre en fonction des capacités de parallélisme au niveau instruction de l'architecture ciblée.

Cette méthodologie a été testée sur différents algorithmes de traitement numérique du signal présents au sein des systèmes de radio-communications de troisième génération. Les résultats obtenus montrent l'intérêt de notre méthodologie pour réduire le temps de développement des systèmes en virgule fixe.

Mots-clés : traitement du signal, adéquation algorithme architecture, arithmétique virgule fixe, précision finie, bruit de quantification, processeurs de traitement du signal, processeurs spécialisés, DSP, ASIP, compilation, génération de code, télécommunications, WCDMA.

Abstract

The efficient implementation of digital signal processing algorithms into embedded systems requires the use of fixed-point arithmetic in order to satisfy the cost and power consumption constraints. The manual coding of fixed-point data is a tedious and error prone task. Moreover, the reduction of the time-to-market of applications requires high-level development tools that allow the automation of some tasks. Thus, the development of automatic fixed-point conversion methodologies is needed. For Digital Signal Processor (DSP), the methodology must determine the optimal fixed-point specification which allows to maximize the accuracy and to minimize the size and the execution time of the code. The goal of this thesis is to define and develop a new methodology for the implementation of floating-point algorithms into fixed-point programmable processors with the constraint to fulfill the application quality criteria. This work is based on three main parts.

First, the structure of the methodology has been defined. The analysis of the architecture influence on the computation accuracy underlines the necessity to take account of the target architecture in order to obtain an optimized implementation in terms of execution time and accuracy. Moreover, the study of the interaction between the compilation and the fixed-point conversion process allows to define the coupling needed between these two processes.

The second part of this work deals with the evaluation of the fixed-point system accuracy through the determination of the Signal to Quantification Noise Ratio (SQNR). A new methodology which allows to determine automatically the analytical expression of the SQNR according to the fixed-point data format has been proposed. First, a new quantification noise model is presented. Then, the theoretical concepts for the determination of the output quantification noise power are detailed in the case of linear systems and non-linear and non-recursive systems. Finally, the methodology developed for determining automatically the SQNR expression of linear systems is explained.

The last part of this work corresponds to the development of the fixed-point conversion methodology. First, the data dynamic range is evaluated with an analytical approach based on two different techniques. The dynamic information are used for the determination of the data binary point position by taking account of the number of guard bits available in the processor. For obtaining a complete fixed-point format, the data word-length is determined in order to exploit the diversity of data types manipulated by the processor. The methodology selects the set of instructions which allows to obtain a sufficient accuracy and to minimize the code execution time. The last stage of the methodology corresponds to the optimization of the fixed-point data format in order to obtain a more efficient implementation. The different scaling operations are moved for minimizing the global execution time as long as the accuracy constraint is fulfilled. Two types of method have been proposed according to the instruction level parallelism capabilities of the target processor.

This methodology has been tested on different digital signal processing algorithms used in the third generation radiocommunication systems. The results show the relevance of our methodology for reducing the development time of fixed-point systems.

Keywords : signal processing, architecture algorithm adequacy, fixed-point arithmetic, finite precision, quantification noise, digital signal processor, specialized processor, DSP, ASIP, compilation, code generation, telecommunication, WCDMA

Remerciements

Cette thèse a été effectuée au sein du groupe signal-architecture du Laboratoire d'Analyse des Systèmes de Traitement de l'Information (LASTI) de Lannion. Aussi, je tiens à remercier monsieur K. Chehdi de m'avoir accueilli dans son laboratoire.

Je remercie vivement monsieur O. Sentieys, professeur à l'ENSSAT, pour avoir accepté d'être mon directeur de thèse. Je souhaite lui exprimer ma profonde et sincère reconnaissance pour son aide et ses conseils dans la préparation de cette thèse et pour m'avoir fait profiter de ses connaissances dans le domaine des méthodologies de conception de systèmes.

Je tiens à remercier particulièrement monsieur E. Martin, professeur à l'Université de Bretagne Sud pour m'avoir fait l'honneur d'être le président du jury de cette thèse.

J'adresse mes sincères remerciements à messieurs D. Demigny, professeur à l'ENSEA et T. Risset chargé de recherche à l'INRIA pour avoir accepté de juger mon travail en tant que rapporteurs.

Je remercie également messieurs P. Le Guernic, directeur de recherche à l'INRIA et F. Charot, chargé de recherche à l'INRIA pour leur participation à se jury de thèse. Que monsieur F. Charot soit assuré de ma gratitude pour sa collaboration et pour m'avoir fait profiter de ses connaissances dans le domaine de la compilation.

Je souhaite remercier vivement messieurs T. Saïdi, ingénieur de recherche au LASTI, P. Quemerais ingénieur de recherche à l'ENSSAT, R. David, doctorant au LASTI et D. Chillet, maître de conférences à l'ENSSAT, et les différents stagiaires, ayant participé à ce projet, pour leur collaboration dans le cadre de ce travail de recherche.

Je remercie tous les membres du LASTI, le personnel technique et le personnel administratif de l'ENSSAT pour avoir contribué, de près ou de loin, à ce travail.

Enfin, je n'oublie pas de remercier chaleureusement tous les membres de ma famille et amis qui m'ont soutenu, encouragé et aidé tout au long de la préparation de cette thèse.

A Marie,

Table des matières

Introduction **1**
Cycle de développement des applications de traitement du signal 1
Problématique de l'étude 4
Plan de l'étude 5

1 État de l'art **7**
1.1 Codage des données 7
1.1.1 Les différents types de codage 7
1.1.2 Définition des règles de l'arithmétique virgule fixe 10
1.1.3 Processus de codage 11
1.1.4 Modélisation du processus de quantification 13
1.1.5 Comparaison des codages en virgule fixe et en virgule flottante 16
1.2 État de l'art des méthodologies 19
1.2.1 Introduction 19
1.2.2 Détermination de la dynamique des données 19
1.2.3 Evaluation de la précision 23
1.2.4 Implantation des algorithmes dans les DSP 26
1.3 Conclusions 32

2 Définition de la méthodologie **33**
2.1 Architecture des processeurs de traitement du signal 33
2.1.1 Introduction 33
2.1.2 Architecture de l'unité de traitement 37
2.1.3 Architectures des unités de mémoire et de contrôle 50
2.1.4 Analyse de l'influence de l'architecture sur la précision des calculs 55
2.1.5 Conclusions 61
2.2 Compilation pour les processeurs de traitement du signal 62
2.2.1 Introduction 62
2.2.2 Les techniques de compilation classiques 63
2.2.3 Les spécificités de la compilation pour les DSP 66
2.2.4 Analyse de l'interaction entre la compilation et le codage des données . 70
2.2.5 Conclusions 73
2.3 Présentation de la méthodologie 74
2.3.1 Introduction 74
2.3.2 Détermination et optimisation du codage des données 75
2.3.3 Évaluation de la qualité de l'implantation 76
2.3.4 Conclusions 78

3 Évaluation de la précision **79**
3.1 Introduction 79
3.2 Modélisation du bruit 81
3.2.1 Modélisation du bruit généré 81
3.2.2 Modèle du bruit propagé 90
3.2.3 Conclusions 91
3.3 Approche théorique pour l'évaluation de la précision 92
3.3.1 Systèmes linéaires 92
3.3.2 Systèmes non-linéaires 96
3.4 Méthodologie d'évaluation automatique de la précision 99
3.4.1 Introduction 99
3.4.2 Représentation de l'application au niveau bruit 100
3.4.3 Détermination des fonctions de transfert 102
3.4.4 Détermination de l'expression du RSBQ 110
3.4.5 Conclusions 111
3.5 Expérimentations 112
3.5.1 Estimation des paramètres statistiques du bruit généré 112
3.5.2 Estimation de la puissance du bruit dans le cadre des systèmes linéaires 113
3.5.3 Estimation de la puissance du bruit dans le cadre des systèmes non-linéaires 117
3.5.4 Temps d'exécution de l'outil 118
3.6 Détermination de la contrainte de précision minimale 120
3.6.1 Introduction 120
3.6.2 Modélisation du bruit de sortie 120
3.7 Conclusions 122

4 Codage des données **123**
4.1 Description de l'infrastructure de compilation 123
4.1.1 Description de la partie frontale SUIF 124
4.1.2 Description de la partie finale CALIFE 124
4.1.3 Description de la représentation intermédiaire GFDC 125
4.2 Détermination de la dynamique des données 128
4.2.1 Définition d'une méthode d'estimation de la dynamique 128
4.2.2 Implantation de la méthode d'estimation de la dynamique 131
4.3 Détermination de la position de la virgule 135
4.3.1 Détermination de la position de la virgule au sein d'un GFDC 135
4.3.2 Détermination de la position de la virgule au sein d'un GFD 135
4.3.3 Expérimentations 137
4.4 Détermination du type des données 139
4.4.1 Détermination de la largeur des données 139
4.4.2 Optimisation de la largeur des données transférées en mémoire 146
4.4.3 Expérimentations 147
4.4.4 Conclusions 150
4.5 Optimisation du codage des données 152
4.5.1 Présentation du problème 152
4.5.2 Architecture sans parallélisme d'instruction 154
4.5.3 Architecture avec parallélisme d'instruction 158
4.5.4 Expérimentations 162
4.5.5 Conclusions 166
4.6 Conclusions 167

5 Applications 169
5.1 Introduction . . . 169
5.2 Structure d'un émetteur et d'un récepteur WCDMA . . . 171
5.2.1 Émetteur WCDMA . . . 171
5.2.2 Récepteur WCDMA . . . 172
5.3 Implantation d'un récepteur WCDMA . . . 177
5.3.1 Définition de la contrainte de précision . . . 177
5.3.2 Filtre de réception . . . 178
5.3.3 Décodage des données au sein d'un terminal mobile . . . 181
5.3.4 Décodage des données au sein d'une station de base . . . 183
5.4 Conclusions . . . 186

Conclusions et perspectives 187
Conclusions . . . 187
Perspectives . . . 188
Évaluation de la précision . . . 188
Codage des données en virgule fixe . . . 190

A Exemple de résultats obtenus avec l'outil de détermination du RSBQ 193
A.1 Filtre IIR d'ordre 4 cascadé . . . 193
A.1.1 Spécifications . . . 193
A.1.2 Transformation $T_{21} - T_{22}$: graphe G_{eq} . . . 193
A.1.3 Transformation $T_{23} - T_{24}$: graph A_H . . . 194

B Exemple de génération d'un code C virgule fixe 197
B.1 Code virgule flottante . . . 197
B.2 Code virgule fixe . . . 197

C Estimation du canal et synchronisation 199
C.1 Estimation de l'amplitude complexe du canal . . . 199
C.2 Synchronisation des codes . . . 200

D Code C des applications 203
D.1 *Rake receiver* terminal mobile . . . 203
D.2 *Rake receiver* station de base . . . 205

Publications et Rapports 219
Revue nationale . . . 219
Conférences internationales . . . 219
Conférences nationales . . . 219
Rapports de recherche interne . . . 220

Table des figures

1 Cycle de développement d'une application de TNS . . . 2
2 Synoptique de la méthodologie proposée . . . 5

1.1 Représentation des données en virgule fixe . . . 7
1.2 Représentation des données en virgule flottante [72] . . . 9
1.3 Caractéristiques des lois de dépassement . . . 12
1.4 Caractéristiques de la loi de quantification par arrondi . . . 12
1.5 Caractéristiques de la loi de quantification par troncature . . . 12
1.6 Modélisation du processus de quantification du signal x . . . 13
1.7 Comparaison de la variance théorique σ_{theo}^2 et simulée σ_{sim}^2 . . . 16
1.8 Evolution du niveau de dynamique des codages . . . 17
1.9 Evolution du RSBQ en fonction de la dynamique du signal d'entrée . . . 18
1.10 Synoptique de la méthodologie associée à l'outil FRIDGE . . . 22
1.11 Synoptique de la méthodologie *Autoscaler for C* . . . 27
1.12 Synoptique de la méthodologie de conversion pour le TMS320C25/50 . . . 30
1.13 Architecture du TMS320C25/C50 et stratégie de codage . . . 30

2.1 Caractéristiques des solutions architecturales . . . 35
2.2 Opérations arithmétiques double précision . . . 40
2.3 Opérations arithmétiques exploitant le parallélisme au niveau des données . . . 40
2.4 Structure du filtre FIR . . . 42
2.5 Architecture traditionnelle de type MAC *(a)* et double-MAC *(b)* . . . 43
2.6 Architecture du processeur DSP TMS320C62x [137] . . . 44
2.7 Coût des recadrages au sein des filtres FIR et FIR symétrique . . . 47
2.8 Représentation du pipeline pour les différents codages des instructions . . . 53
2.9 Structure du filtre FIR et représentation des sources de bruit . . . 56
2.10 Évolution du RSBQ en fonction de la largeur des données traitées . . . 56
2.11 Évolution du RSBQ en fonction des paramètres de l'architecture . . . 57
2.12 Évolution du RSBQ en sortie d'un filtre FIR pour différents DSP 16 bits . . . 58
2.13 Évolution du RSBQ en fonction du temps d'exécution du filtre . . . 58
2.14 Structures des filtres IIR et représentation des sources de bruit potentielles . . . 59
2.15 Évolution du RSBQ en fonction des paramètres de l'architecture . . . 59
2.16 Évolution du RSBQ en sortie d'un filtre IIR pour différents DSP 16 bits . . . 60
2.17 Représentation des papillons des FFT DIT radix 2 et 4 . . . 61
2.18 Synoptique de la partie frontale d'un compilateur . . . 63
2.19 Analyse de l'influence du phénomène de *spilling* . . . 72
2.20 Vue générale de la méthodologie . . . 75
2.21 Représentation du processus de détermination du $RSBQ_{min}$ . . . 77
2.22 Vue générale détaillée de la méthodologie . . . 78

3.1 Représentation des opérateurs arithmétiques élémentaires . . . 83
3.2 Représentation à base d'opérateurs élémentaires d'une addition en CA2 . . . 84

3.3 Représentation à base d'opérateurs élémentaires d'une multiplication en CA2 . 84
3.4 Probabilités des bits d'une donnée pour différents signaux 85
3.5 Modélisation des données 86
3.6 Modélisation d'une constante C 86
3.7 Exemple de multiplication d'un signal par une constante 87
3.8 Probabilités des bits en sortie d'un multiplieur et d'un additionneur 88
3.9 Modélisation du résultat de la multiplication d'un signal par une constante . . . 89
3.10 Représentation du système linéaire au niveau bruit 93
3.11 Fonction de distribution des bruits dans le cas de la valeur absolue 98
3.12 Synoptique de la méthodologie de détermination de l'expression du RSBQ . . . 99
3.13 Processus d'insertion des sources de bruit généré 101
3.14 Modèles de propagation du bruit des différents opérateurs 102
3.15 Exemple de démantèlement d'un circuit 103
3.16 Représentation des différentes étapes de la transformation T_2 104
3.17 Description de l'algorithme de détection de circuits 105
3.18 Exemple pour la transformation T_{21} 105
3.19 Exemple pour la transformation T_2 107
3.20 Exemple de graphe G_{eq} représentant les différentes fonctions linéaires 107
3.21 Algorithme récursif de détermination des fonctions linéaires 108
3.22 Exemple de substitutions de variables 109
3.23 Quantification par arrondi du résultat de la multiplication $x \times C$ 112
3.24 Quantification par troncature du résultat de la multiplication $x \times C$ 113
3.25 Erreur relative sur l'estimation de P_{b_y}, applications non-récursives 114
3.26 Analyse de l'influence du codage des coefficients 115
3.27 Erreur relative sur l'estimation de P_{b_y} pour deux filtres IIR 115
3.28 Erreur relative sur l'estimation de P_{b_y}, comparaison des modèles de bruit 116
3.29 Erreur relative sur l'estimation de P_{b_y}, applications non-linéaires 117
3.30 Détermination de la contrainte de précision minimale 120
3.31 Évaluation de l'hypothèse de bruit gaussien 121

4.1 Modélisation d'une structure de contrôle 125
4.2 Modélisation des structures de contrôle 126
4.3 Modélisation de l'algorithme de FFT 127
4.4 Synoptique du processus de détermination de la dynamique des données 132
4.5 Modélisation de la position de la virgule pour un opérateur 135
4.6 Spécification de la position de la virgule en présence de bits de garde 136
4.7 Synoptique de la méthode de détermination de la largeur des données 140
4.8 Exemple de modélisation sous forme d'arbre 143
4.9 Exemple de traitement au sein du GFD 144
4.10 Synoptique de la méthode de détermination du type des données 147
4.11 Évolution du temps d'exécution minimal d'un corrélateur complexe 148
4.12 Évolution du temps d'exécution minimal d'un filtre IIR d'ordre 2 149
4.13 Évolution du temps d'exécution du processus d'optimisation 150
4.14 Représentation du processus d'alignement des bits de garde 153
4.15 Exemple de déplacement d'une opération de décalage 154
4.16 Technique de détermination du temps d'exécution des opérations de recadrage . 156
4.17 Optimisation de la position des opérations de recadrage 157
4.18 Algorithme d'optimisation du placement des opérations de recadrage 158
4.19 Exemple de calcul de la métrique η_{ij} 160
4.20 Exemple d'ordonnancement pour un filtre FIR complexe 161
4.21 Évolution de la précision en fonction du coût des recadrages 164

4.22 Synoptique détaillé de la méthodologie de codage en virgule fixe 168

5.1 Synoptique d'un chaîne de transmission numérique 169
5.2 Synoptique des émetteurs WCDMA 171
5.3 Synoptique d'un récepteur WCDMA 172
5.4 Synoptique d'un *rake receiver* 174
5.5 Synoptique d'un *finger* du *rake receiver* 174
5.6 Synoptique de la partie décodage des données (station de base) 175
5.7 Synoptique de la partie décodage des données (terminal mobile) 176
5.8 Évolution du taux d'erreur binaire en fonction du RSB 177
5.9 Synoptique de l'expérimentation réalisée pour déterminer $\Gamma_{FP} = f(RSBQ)$ . . 178
5.10 Évolution du taux d'erreur binaire Γ_b en fonction du RSBQ 178
5.11 Évolution du niveau du bruit et du signal au sein du filtre FIR 179
5.12 Évolution du RSBQ en fonction de la largeur de l'additionneur 180
5.13 Spécification virgule fixe des deux solutions testées 181
5.14 Évolution des niveaux du bruit et du signal en fonction de SF 181
5.15 Synoptique des solutions architecturales testées 182
5.16 Niveau du bruit et du signal au sein du récepteur WCDMA 183
5.17 Largeur des données au sein du *rake receiver* (terminal mobile) 184
5.18 Synoptique de la solution architecturale testée 184
5.19 Largeur des données au sein du *rake receiver* (station de base) 185
5.20 Cycle de développement d'une application de TNS 190

A.1 Synoptique du filtre IIR 4 193
A.2 Graphe G_{eq} du filtre IIR 194
A.3 Arbre A_H du filtre IIR 194

C.1 Synoptique du traitement pour l'estimation de canal 199
C.2 Synoptique de la boucle d'asservissement de retard 201

Liste des tableaux

1.1 Description des cas particuliers des représentations SVA et CA2 9
1.2 Règles de propagation de la dynamique 21

2.1 Largeur naturelle des données de quelques cœurs de DSP 39
2.2 Présentation des capacités SWP de certains processeurs DSP 41
2.3 Types des données manipulées par certains processeurs DSP 42
2.4 Surcoût du recadrage interne ($\mathcal{C}_r$) pour différentes applications 73

3.1 Paramètres statistiques du bruit généré 91
3.2 Temps d'exécution de la transformation T_{22} 118
3.3 Temps d'exécution de la transformation T_{21} 119

4.1 Nombre d'exécutions des opérations de décalage en fonction de b_g 138
4.2 Optimisation du placement des opérations de recadrage 165
4.3 Optimisation du placement des opérations de recadrage (VLIW) 166

Glossaire

AS-DSP	Application Specific Digital Signal Processor
ASIC	Application Specific Integrated Circuit
ASIP	Application Specific Instruction-set Processor
ATM	Asynchronous Transfer Mode
CDMA	Code Division Multiple Access
DAG	Directed Acyclic Graph
DIT	Decimation-in-time
DPCCH	Dedicated Physical Control Channel
DPDCH	Dedicated Physical Data Channel
DSP	Digital Signal Processor
FFT	Fast Fourier Transform
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
GPRS	General Packet Radio Service
IIR	Infinite Impulse Response
ILP	Instruction Level Parallelism
LSB	Last Significant Bit
MAC	Multiplication accumulation
MPEG	Moving Pictures Experts Group
MP3	MPEG-1, Layer 3
MSB	Most Significant Bit
OVSF	Orthogonal Variable Spreading Factor
PN	Pseudo-Noise
QPSK	Quadrature Phase Shift Keying
RISC	Reduced Instruction Set Computer
SoC	Systems on Chip
SWP	Sub-Word Parallelism
UMTS	Universal Mobile Telecommunications System
VLIW	Very Long Instruction Word
VLSI	Very Large Scale Integration
WCDMA	Wideband Code Division Multiple Access

ACS	Addition, Comparaison, Sélection
CA2	Complément à deux
CAN	Convertisseurs analogique-numérique
GFC	Graphe Flot de Contrôle
GFD	Graphe Flot de Données
GFDC	Graphe Flot de Données et de Contrôle
GFS	Graphe Flot de Signal
RSBQ	Rapport Signal à Bruit de Quantification
SVA	Signe Valeur Absolue
TNS	Traitement Numérique du Signal
UAL	Unité Arithmétique et Logique

Introduction

Au cours des vingt dernières années, les applications embarquées de traitement du signal ont connu un essor important. Ces applications sont présentes dans des domaines très variés tels que les télécommunications, la navigation, le domaine du traitement des images, de la vidéo, de la parole ou de l'audio ou le domaine des applications militaires ou médicales [157]. Dans ces différents domaines, l'évolution des applications se traduit par une augmentation de la complexité des algorithmes de traitement numérique du signal (TNS) utilisés. Par exemple, pour les nouveaux systèmes de radiocommunications de troisième génération, des techniques avancées de traitement numérique du signal sont mises en œuvre afin d'augmenter les débits de transmissions. Cette augmentation des débits permet la mise en place de nouveaux services dans le domaine de l'internet et du multimédia. De plus, la diversité des standards peut nécessiter la cohabitation au sein d'un même produit de plusieurs systèmes de transmissions.

La technologie actuelle permet d'intégrer quelques dizaines de millions de transistors au sein d'une puce de silicium et à l'horizon 2010, il est prévu de pouvoir intégrer un milliard de transistors [51]. Ainsi, cette augmentation des capacités d'intégration au sein des systèmes VLSI, permet d'accompagner les besoins croissants en termes de capacité de traitement pour les nouvelles applications. Cependant, l'accroissement de la productivité des outils de conception actuels ne permet pas de suivre l'augmentation des besoins au niveau des applications [50]. De plus, la durée de vie des produits étant de plus en plus faible, il est nécessaire de réduire le temps de mise sur le marché (*time-to-market*) de ces produits. Ces contraintes impliquent la nécessité de diminuer les temps de développement des produits et la mise en œuvre de plate-formes relativement flexibles pour intégrer les futurs évolutions du produit. Ainsi, la complexité croissante des algorithmes de traitement numérique du signal implantés dans les systèmes embarqués combinée à la réduction des temps de mise sur le marché des applications, nécessite la mise en œuvre d'outils de conception et de développement de haut niveau.

Les besoins en outils de conception et de développement sont présents à tous les niveaux du cycle de développement des applications de traitement du signal. Du point de vue de l'implantation, ces outils doivent assister le concepteur dans ses choix au niveau de son architecture et ceci, à un niveau d'abstraction le plus élevé possible afin d'explorer rapidement les différentes solutions envisagées. Ensuite, au niveau du développement matériel et logiciel, il est nécessaire d'utiliser des outils permettant d'automatiser certaines taches et de passer rapidement d'une description de haut niveau à une description de plus bas niveau. Pour situer notre travail de recherche et souligner son intérêt, le cycle de développement d'une application de traitement du signal est détaillé dans la section suivante.

Cycle de développement des applications de traitement du signal

Le cycle de développement d'une application de traitement numérique du signal est schématisé à la figure 1. La première étape, correspondant à la définition du cahier des charges, a pour but d'exprimer le besoin et de fixer les objectifs à atteindre. Ce cahier des charges permet de

spécifier le système à concevoir. Cette spécification décrit la vue externe du système et regroupe les différentes contraintes devant être respectées par le système. Trois aspects sont présents au sein de ces spécifications [17]. Les spécifications fonctionnelles décrivent l'ensemble des fonctions devant être réalisées par le système. Les spécifications technologiques définissent les contraintes sur la réalisation du système telles que la consommation d'énergie et les contraintes temporelles. Les spécifications opératoires, regroupant les critères de qualité associés à l'application, fixent les performances minimales devant être atteintes par le système. Ces critères de qualité sont variés et liés au domaine de l'application. Par exemple, dans le cadre d'un récepteur de communications numériques, les performances à atteindre sont définies à travers le taux d'erreur binaire.

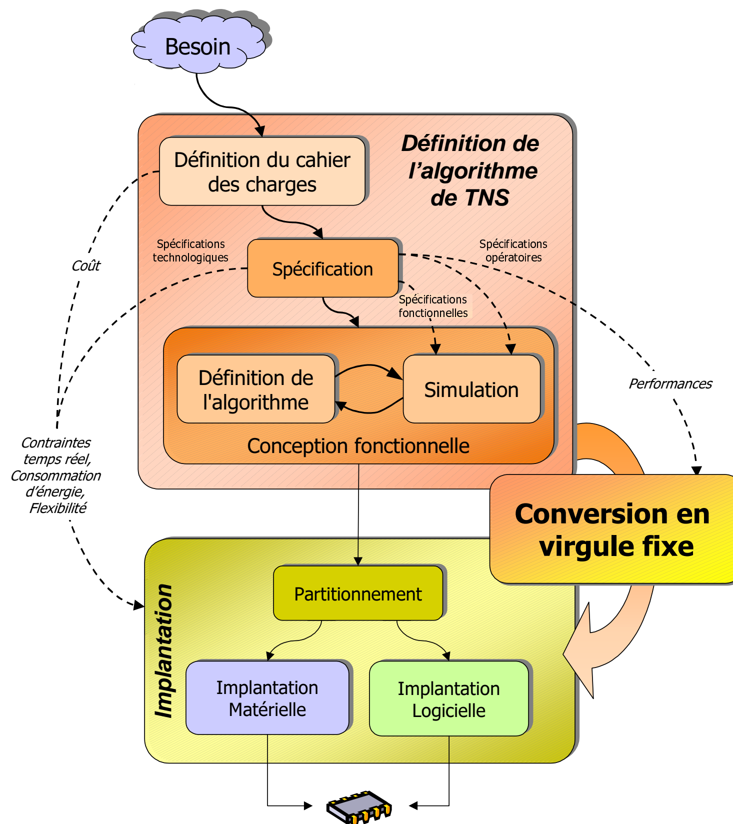


FIG. 1: Cycle de développement d'une application de TNS

L'étape suivante correspond à la conception fonctionnelle du système. Celle-ci aboutit à une description complète de l'algorithme de traitement numérique du signal. Cette phase de conception permet de définir l'algorithme puis, de valider celui-ci à travers une série de simulations. Ces simulations vont permettre de vérifier que l'algorithme réalise l'ensemble des fonctions souhaitées et que les différentes performances requises sont satisfaites. De nombreux outils d'aide à la conception d'algorithmes de TNS sont disponibles. Des outils tels que Ptolemy (Université de Californie à Berkley), SPW (Cadence), CoCentric (Synopsys) ou SIMULINK (Mathworks) mixent une représentation graphique de l'application et une description en langage de haut niveau de certains blocs. Ces outils proposent des librairies fournissant des fonctions ou des blocs permettant un niveau d'abstraction plus élevé. Les langages de haut niveau classiques tels que les langages C et C++ ou des extensions comme SystemC [115] ou des langages adaptés au traitement du signal tels que Matlab [96], Signal [43], Silage [47], permettent de simuler les algorithmes de TNS. De même, pour cette approche, le niveau d'abstraction est lié à la disponibilité de bibliothèques spécifiques au domaine de l'application. Au niveau de la conception

fonctionnelle, la description et la simulation de l'algorithme de TNS, utilisent l'arithmétique virgule flottante afin de s'affranchir des problèmes de précision des calculs.

Lorsque l'algorithme de TNS est figé, celui-ci est implanté au sein d'un système embarqué. Diverses solutions architecturales sont disponibles pour implanter cet algorithme. Ces solutions correspondent aux ASIC[1], à la logique reconfigurable, aux ASIP[2], aux DSP[3], aux AS-DSP[4] et aux micro-contrôleurs. Pour satisfaire les contraintes de performance, de coût, de consommation d'énergie et de flexibilité, plusieurs solutions peuvent être associées au sein d'une même plate-forme. Le choix des différents éléments composant ce système sur puce (SoC : System on Chip) dépend des contraintes associées à l'application en termes de coût, de consommation d'énergie, de flexibilité et de temps de développement. La combinaison de plusieurs cibles permet de satisfaire les différentes contraintes de l'application. Les processeurs programmables de traitement du signal représentent un compromis entre ces différentes contraintes. Ces processeurs permettent d'obtenir pour les applications de TNS de bonnes performances en termes de coût, de consommation d'énergie et de capacités de calcul. Les capacités de programmation de ces composants fournissent une flexibilité importante et conduisent à des temps de développement relativement faibles.

Les contraintes de coût, de consommation et temps réel inhérentes aux applications embarquées requièrent l'utilisation de l'arithmétique virgule fixe. En effet, la largeur des données traitées au sein des architectures en virgule fixe étant plus faible, le prix et la consommation d'énergie de ces architectures sont moins importants. L'arithmétique virgule flottante basée sur la norme IEEE-754 nécessite d'utiliser des données codées sur 32 bits. Dans le cadre des architectures en virgule fixe, la largeur des données est nettement plus faible. A titre d'exemple, la majorité des processeurs DSP manipule des données codées en mémoire sur 16 bits. De plus, les opérateurs utilisant l'arithmétique virgule fixe étant moins complexes, les architectures travaillant uniquement en virgule fixe sont plus rapides. Le surcoût en termes de temps d'exécution lié à l'émulation de l'arithmétique virgule flottante au sein d'architectures en virgule fixe étant prohibitif (facteur 10 à 500 [149]), il est nécessaire de coder l'ensemble des données de l'application en virgule fixe. Cependant, l'utilisation de l'arithmétique virgule fixe se traduit par des temps de développement plus importants. Pour chaque donnée, il est nécessaire de définir un format en virgule fixe. La dynamique de codage étant réduite, ce format doit garantir l'absence de débordement et fournir la précision requise. De plus, certaines taches telles que l'alignement des données sont à la charge du programmeur.

L'implantation des algorithmes de TNS au sein d'une architecture en virgule fixe nécessite de réaliser une conversion de la description de l'algorithme en virgule flottante en une spécification en virgule fixe. Cette conversion est une tâche fastidieuse, longue et source d'erreurs, si elle est effectuée manuellement. En effet, certaines expérimentations [42] ont montré que le temps consacré à cette phase de conversion en virgule fixe est relativement important. Lorsque la description de l'algorithme en virgule flottante est figée, la conversion manuelle en virgule fixe peut représenter jusqu'a 30% du temps global nécessaire à l'implantation de l'algorithme. Cette conversion en virgule fixe nécessite de déterminer le format de chaque donnée et de vérifier si la spécification en virgule fixe obtenue permet toujours de satisfaire les niveaux de performance requis par l'application.

La réduction du temps de mise sur le marché des applications exigeant l'utilisation d'outils

[1]Application Specific Integrated Circuit
[2]Application Specific Instruction-set Processor
[3]Digital Signal Processor
[4]Application Specific DSP

de développement de haut niveau permettant d'automatiser certaines tâches, des méthodologies de codage automatique des données en virgule fixe ont été proposées. Dans ce cas, la problématique est de rechercher la meilleure adéquation algorithme architecture d'un point de vue du codage des données. Dans le cadre d'une implantation matérielle, l'objectif est d'obtenir l'architecture satisfaisant les performances désirées et dont le prix, la surface et la consommation sont minimaux. Ainsi, la largeur des opérateurs est optimisée afin de minimiser la surface du circuit. Pour la conception logicielle, l'architecture du processeur est figée. En conséquence, la méthodologie doit optimiser l'implantation de l'algorithme d'un point de vue du codage des données, en utilisant au mieux les ressources offertes. Ce codage des données doit fournir une précision des calculs suffisante pour satisfaire les contraintes de qualité associées à l'application et doit permettre d'obtenir une implantation efficace au sein de l'architecture cible. Cette dernière contrainte implique de minimiser le temps d'exécution et la taille du code, ainsi que la consommation d'énergie du système.

Problématique de l'étude

L'utilisation de l'arithmétique virgule fixe conduit à la réalisation des calculs en précision finie. Ainsi, une erreur entre le résultat calculé en précision finie et celui obtenu en précision infinie, est présente. L'implantation d'une application en virgule fixe n'est viable que si malgré la présence de cette erreur, les critères de qualité associés à cette application sont toujours respectés. En conséquence, l'objectif de ce travail de recherche, est de définir et de mettre en œuvre une nouvelle méthodologie d'implantation automatique d'algorithmes spécifiés en virgule flottante au sein de processeurs programmables en virgule fixe sous contrainte de respect des critères de qualité associés à l'application. La classe d'architectures considérée au sein de cette méthodologie correspond aux composants DSP, aux cœurs de DSP et aux ASIP dédiés aux applications de TNS.

Cette méthodologie cible les applications embarquées pour lesquelles l'implantation doit être optimisée du point de vue du coût et de la consommation d'énergie. Ceci nécessite de minimiser le temps d'exécution et la taille du code généré par le compilateur. En conséquence, la méthodologie réalise la conversion de la description en virgule flottante en une spécification en virgule fixe et optimise certains critères tels que le temps d'exécution ou la taille du code généré.

Les méthodologies existantes [70, 149] réalisent une transformation de la représentation des données en virgule flottante en une représentation en virgule fixe au niveau du code source sans prendre réellement en considération l'architecture du processeur cible. En conséquence, au sein de notre travail de recherche l'influence de l'architecture sur la précision des calculs a été analysée. De plus, l'interaction entre les phases de génération de code et de conversion en virgule fixe a été étudiée. Suite aux résultats de ces études, en vue d'optimiser l'implantation de l'application, l'architecture du processeur cible est prise en compte lors de la conversion en virgule fixe et certaines phases de cette conversion sont couplées au processus de génération de code.

Le synoptique de la méthodologie développée est présenté à la figure 2. Les différentes parties de cette méthodologie sont définies au cours de la présentation, dans la partie suivante, du plan de cette étude.

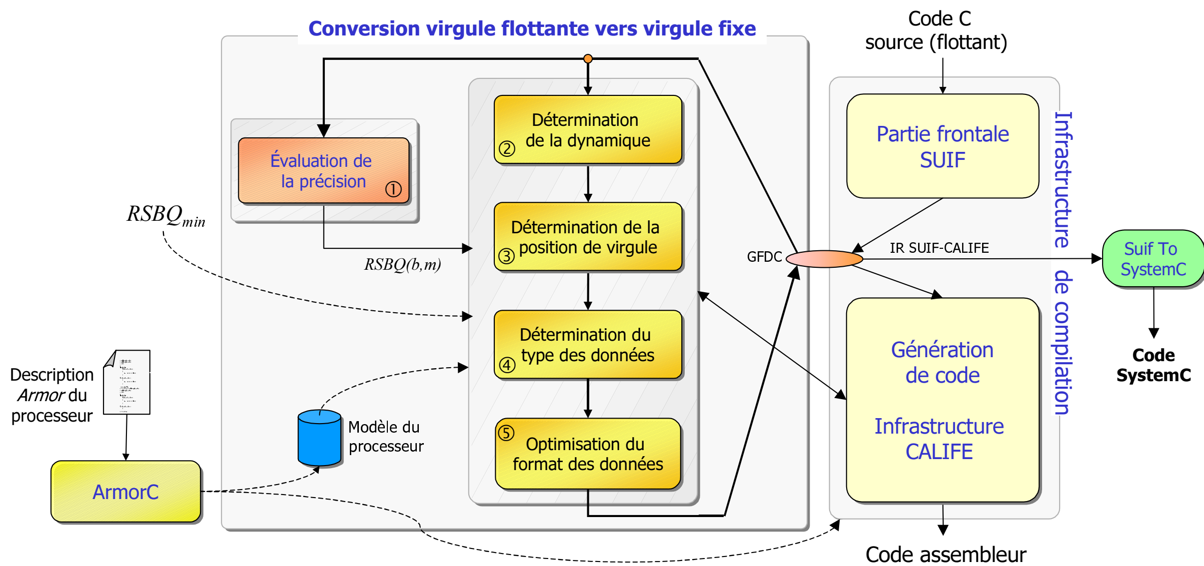


FIG. 2: Synoptique de la méthodologie proposée

Plan de l'étude

Le premier chapitre est consacré à un état de l'art des méthodologies d'implantation d'algorithmes spécifiés en virgule flottante au sein de processeurs programmables en virgule fixe. Dans la première partie, les différents aspects de l'arithmétique virgule fixe sont détaillés. Dans la seconde partie, les méthodologies de conversion en virgule fixe existantes sont présentées. Tout d'abord, les phases de détermination de la dynamique et d'évaluation de la précision sont détaillées. Ensuite, les méthodes de codage automatique en virgule fixe, présentes au sein de la littérature sont exposées.

Au sein du second chapitre, la méthodologie que nous proposons est définie. Cette définition est basée sur l'analyse de l'influence de l'architecture sur la précision des calculs et sur l'étude de l'interaction entre les différentes phases de compilation et le processus de codage des données. Dans un premier temps, les différents éléments influençant la précision des calculs sont détaillés et les résultats des expérimentations réalisées pour analyser l'influence de l'architecture sur la précision des calculs sont exposés [103]. Cette analyse montre la nécessité de tenir compte de l'architecture cible pour obtenir une implantation optimisée d'un point de vue du temps d'exécution et de la précision. Ensuite, les résultats de l'étude de l'interaction entre les phases de compilation et de codage des données sont présentés et illustrés à travers différentes expérimentations [100]. Cette analyse permet de définir les objectifs de certaines phases du processus de codage en virgule fixe et le couplage nécessaire entre ces phases et le processus de génération de code.

Le troisième chapitre traite de la méthode définie et mise en œuvre pour évaluer la précision au sein d'un système en virgule fixe à travers la détermination du Rapport Signal à Bruit de Quantification (RSBQ). Une méthodologie permettant de déterminer automatiquement l'expression analytique du RSBQ en fonction du format des données en virgule fixe est proposée. Cette phase d'évaluation de la précision est représentée au sein de la méthodologie globale détaillée à la figure 2, par le module ①. Dans un premier temps, les modèles de bruit présents au sein de la littérature sont analysés et un nouveau modèle pour le bruit issu de la quantification de la sortie d'un opérateur arithmétique est proposé. Ensuite, les concepts théoriques pour déterminer la puissance du bruit de quantification en sortie des systèmes linéaires [104] et

des systèmes non-linéaires et non-récursifs sont détaillés. La méthodologie mise en œuvre pour obtenir automatiquement l'expression analytique du RSBQ dans le cadre des systèmes linéaires est exposée [105]. Puis, les résultats des différentes expérimentations réalisées pour évaluer la précision des estimations et l'efficacité de l'outil développé pour implanter cette méthodologie, sont présentés. Finalement, la méthode proposée pour réaliser le lien entre les critères de qualité de l'application et la contrainte de précision utilisée au sein de la méthodologie est explicitée.

Le quatrième chapitre détaille les différentes phases de la méthodologie mise en œuvre pour coder les données en virgule fixe. Dans un premier temps, la dynamique des données est déterminée à l'aide d'une approche analytique combinant deux techniques différentes (module ②). Ainsi, la méthode mise en œuvre permet de traiter les structures linéaires et les structures non-linéaires et non récursives. Ces informations sur la dynamique permettent de déterminer la position de la virgule de chaque donnée (module ③). La technique proposée pour déterminer la position de la virgule, permet de prendre en compte la présence éventuelle de bits de garde au sein de l'architecture [101].

Pour obtenir un format des données en virgule fixe complet, la largeur de chaque donnée est déterminée (module ④) [99]. Cette phase doit prendre en compte l'ensemble des types des données manipulées au sein du processeur et en particulier au niveau des DSP récents qui proposent des capacités de traitement de données en parallèle. La méthode sélectionne la séquence d'instructions permettant de fournir une précision suffisante en sortie de l'algorithme et de minimiser le temps d'exécution du code. La technique d'optimisation basée sur un algorithme de type *branch and bound* est présentée et les solutions proposées pour limiter fortement l'espace de recherche sont détaillées.

La dernière phase du processus de codage correspond à l'optimisation du format des données en vue d'obtenir une implantation plus efficace (module ⑤). Les différentes opérations de recadrage sont déplacées afin de minimiser le temps d'exécution global des opérations de recadrage tant que la précision en sortie de l'algorithme est supérieure à la contrainte. Deux types de méthode ont été mis en œuvre en fonction de l'architecture ciblée. Pour les processeurs ne proposant pas de parallélisme au niveau instruction (ILP), le temps d'exécution des opérations de recadrage est déterminé à partir de la séquence d'instructions utilisée pour implanter celui-ci [102]. Pour les processeurs proposant du parallélisme au niveau instruction, le coût d'une opération de recadrage dépend de la manière dont les instructions sont ordonnancées. En conséquence, une technique permettant de coupler le déplacement des opérations de recadrage avec l'ordonnancement des instructions a été développée.

Le cinquième chapitre est consacré à la présentation des résultats de l'expérimentation de notre méthodologie sur des applications issues des systèmes de radio-communications de troisième génération. Ces applications réalisent la réception des données dans le cadre de transmissions à étalement de spectre par séquence directe. Dans un premier temps, la structure d'un émetteur et d'un récepteur WCDMA et les différents algorithmes de TNS utilisés sont détaillés. Ensuite, les différentes solutions architecturales envisagées sont présentées et les résultats obtenus avec l'outil développé pour implanter notre méthodologie sont exposés.

Dans le dernier chapitre, nous concluons cette étude en résumant les apports de ce travail de recherche. Ensuite, les différentes perspectives à cette étude sont exposées. Plus particulièrement, l'utilisation de différents aspects de ce travail dans le cadre du développement d'une méthodologie pour l'implantation d'algorithmes de TNS au sein d'architectures matérielles (ASIC, FPGA) est détaillée.

Chapitre 1

État de l'art

1.1 Codage des données

Dans cette première partie, nous présentons les caractéristiques du codage des données en virgule fixe, les conséquences de l'utilisation de données en précision finie et la modélisation du processus de codage. Tout d'abord, nous détaillons les différents types de codage des données et les paramètres associés et nous exposons les règles de l'arithmétique virgule fixe. Ensuite, nous présentons les modèles proposés pour l'erreur induite par la quantification d'un signal d'amplitude continue et nous exposons les conditions de validité de ceux-ci.

1.1.1 Les différents types de codage

Codage virgule fixe

Les données en virgule fixe sont composées d'une partie fractionnaire et d'une partie entière pour lesquelles le nombre de bits alloués reste figé au cours du traitement. L'exposant associé à chaque donnée est implicite et fixe. En conséquence, le facteur d'échelle attaché à la donnée est constant. La figure 1.1 représente une donnée en virgule fixe composée d'un bit de signe et de $b-1$ bits répartis entre la partie entière et la partie fractionnaire. Soit m le paramètre représentant la position de la virgule par rapport au bit le plus significatif (MSB) et n la position de la virgule par rapport au bit le moins significatif (LSB). Dans le cadre d'une donnée signée, les différents paramètres respectent la relation $b = m + n + 1$. Les paramètres m et n sont des entiers et représentent la distance, en nombre de bits, entre la virgule et les bits MSB et LSB. Si ces paramètres sont positifs, ils correspondent respectivement au nombre de bits pour la partie entière et pour la partie fractionnaire. Nous utilisons dans la suite du document la notation (b, m, n) pour définir le format d'une donnée. Nous trouvons aussi dans la littérature la notation $Q_n[b]$.

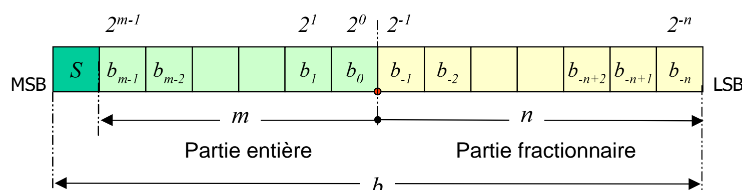


FIG. 1.1: Représentation des données en virgule fixe

Le format d'une donnée en virgule fixe est entièrement défini par la représentation choisie et la largeur de sa partie entière et de sa partie fractionnaire. Nous présentons dans les parties

suivantes les propriétés des représentations signe valeur absolue et complément à deux [72].

Représentation signe valeur absolue (SVA) : Pour cette représentation, la donnée x est composée d'un bit de signe S et de $b-1$ bits représentant le module de x. La valeur de cette donnée est la suivante :

$$x = (-1)^S \sum_{i=-n}^{m-1} b_i 2^i \tag{1.1}$$

Ce type de représentation possède deux représentations de la valeur zéro, ainsi le nombre de valeurs représentables N_c est égal à $2^b - 1$.

Le domaine de définition $\mathcal{D}_{\mathcal{R}}$ (dynamique) correspond à l'intervalle regroupant l'ensemble des valeurs représentables par le code. Les bornes minimales et maximales de cet intervalle sont respectivement X_{min} et X_{max}. Dans le cas d'une représentation signe valeur absolue, nous obtenons un domaine de définition symétrique par rapport à l'origine :

$$\mathcal{D}_{\mathcal{R}} = [X_{min}; X_{max}] = [-2^m + 2^{-n}; 2^m - 2^{-n}] \tag{1.2}$$

Le pas de quantification correspondant à la distance q entre deux valeurs successives, est fonction du domaine de définition du codage et du nombre de valeurs représentables N_c :

$$q = \frac{X_{max} - X_{min}}{N_c - 1} = 2^{-n} \tag{1.3}$$

Le niveau de dynamique correspond au rapport entre les valeurs absolues maximales et minimales représentables par le code. L'expression du niveau de dynamique exprimé en dB, est la suivante :

$$N_{D\ dB} = 20 \log\left(\frac{\max(|x|)}{\min(|x|)}\right) \simeq 20.(b-1).\log(2) = (b-1).6\ dB \tag{1.4}$$

Deux représentations particulières liées à la position de la virgule sont couramment utilisées. Lorsque la virgule est cadrée à droite la valeur codée est entière et lorsque celle-ci est cadrée à gauche la donnée est fractionnaire. Les caractéristiques de ces deux représentations sont présentées dans le tableau 1.1.

Représentation en complément à 2 (CA2) : La représentation en code complément à 2 est très utilisée car elle possède des propriétés arithmétiques très intéressantes pour l'addition et la soustraction. En effet, même si les résultats intermédiaires d'une série d'additions sont en dehors du domaine de définition du codage, le résultat final sera correct si celui-ci appartient au domaine de définition du codage. De plus, l'implantation dans les processeurs numériques des opérateurs traditionnels utilisant ce code est plus simple. Les valeurs négatives de r sont codées par $2^m - r$, ainsi la valeur de la donnée x est égale à :

$$x = -2^m S + \sum_{i=-n}^{m-1} b_i 2^i \tag{1.5}$$

Ce code a l'avantage de ne posséder qu'une seule représentation de la valeur zéro. Le domaine de définition de ce code n'est pas symétrique par rapport à l'origine, il est composé de $2^{b-1} - 1$ valeurs positives et de 2^{b-1} valeurs négatives :

$$\mathcal{D} = [-2^m; 2^m - 2^{-n}] \tag{1.6}$$

Le pas de quantification est identique à celui de la représentation précédente : $q = 2^{-n}$. Les caractéristiques des représentations cadrées à gauche et cadrées à droite sont présentées dans le tableau 1.1.

Représentation	Codage SVA cadrage à gauche	Codage SVA cadrage à droite	Codage CA2 cadrage à gauche	Codage CA2 cadrage à droite	Codage CA2 cadrage à n
Conditions	$m=0$	$n=0$	$m=0$	$n=0$	$n+m=b-1$
q	$2^{-(b-1)}$	1	$2^{-(b-1)}$	1	$2^{-(n)}$
$\mathcal{D}$	$[-1+q;1-q]$	$[-2^{b-1}+q;2^{b-1}-q]$	$[-1;1-q]$	$[-2^{b-1};2^{b-1}-q]$	$[-2^m;2^m-q]$

TAB. 1.1: Description des cas particuliers des représentations SVA et CA2

Codage virgule flottante

Les données en virgule flottante sont composées d'un exposant et d'une mantisse représentés à la figure 1.2. L'exposant permet d'obtenir un facteur d'échelle explicite et variable au cours du traitement, celui-ci est une puissance de 2. La mantisse représente la valeur de la donnée divisée par le facteur d'échelle. Afin d'éviter toute ambiguïté, le premier bit de la mantisse représente le coefficient $\frac{1}{2}$ et est fixé à 1. La valeur de ce bit restant fixe au cours du traitement, celui-ci n'est pas représenté dans le code. Nous présentons dans le paragraphe suivant uniquement le codage utilisant la représentation en signe valeur absolue.

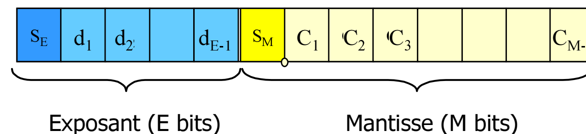


FIG. 1.2: Représentation des données en virgule flottante [72]

Représentation signe valeur absolue : La mantisse et l'exposant sont codés avec une représentation en signe valeur absolue, la valeur de la donnée x est la suivante :

$$x = 2^u.(-1)^{S_M}.\left(\frac{1}{2} + \sum_{i=1}^{M-1} C_i 2^{-i-1}\right) \quad \text{avec} \quad u = (-1)^{S_E}.\sum_{i=1}^{E-1} d_i 2^i \tag{1.7}$$

D'après l'équation 1.7 la valeur 0 n'est pas représentable, ainsi le domaine de définition est composé des 2 sous-intervalles suivants :

$$\mathcal{D}_R = \left[-2^K; -2^{-K-1}\right] \bigcup \left[2^{-K-1}; 2^K\right] \quad \text{avec} \quad K = 2^{E-1} - 1 \tag{1.8}$$

Le pas de quantification est fonction de la valeur représentée. Pour les valeurs de x comprises dans l'intervalle $[-2^u, -2^{u-1}] \bigcup [2^u, 2^{u-1}]$, le pas de quantification est égal à :

$$q = 2^u.2^{-(M+1)} \tag{1.9}$$

L'expression 1.10 détermine les bornes minimales et maximales du pas de quantification relatif. Nous pouvons considérer qu'il est pratiquement constant pour l'ensemble des valeurs de x.

$$2^{-(M+1)} < \frac{q}{\mid x \mid} < 2^{-M} \tag{1.10}$$

Le niveau de dynamique de cette représentation signe valeur absolue est égal à [161] :

$$N_D \simeq 20\log(2^{2K+1}) \quad \text{avec} \quad K = 2^{E-1} - 1 \tag{1.11}$$

La norme IEEE 754 utilise cette représentation en signe valeur absolue. Dans ce cas, la donnée est composée d'un exposant codé sur 8 bits et d'une mantisse sur 24 bits.

1.1.2 Définition des règles de l'arithmétique virgule fixe

Nous considérons dans la suite de ce document que les données sont codées en virgule fixe avec une représentation en CA2.

Addition

L'addition de deux opérandes a et b nécessite qu'ils possèdent un format commun. Le type de représentation, la longueur de la partie entière et la longueur de la partie fractionnaire doivent être identiques pour les deux opérandes. Si cette condition n'est pas respectée il est nécessaire de modifier le format des opérandes afin d'obtenir un format identique (b_c, m_c, n_c). Le format commun garantissant l'absence de perte d'information est le suivant :

$$\begin{aligned} m_c &= \max(m_a, m_b) \\ n_c &= \max(n_a, n_b) \\ b_c &= m_c + n_c + 1 \end{aligned} \tag{1.12}$$

Pour les données ayant un format différent du format commun, il est nécessaire d'étendre le nombre de bits des parties entières et fractionnaires en suivant certaines règles. Pour la partie fractionnaire, les $(n_c - n_a)$ bits supplémentaires sont mis à 0. Pour la partie entière, le bit de signe est étendu. Dans le cas du complément à 2, les $(m_c - m_a)$ nouveaux bits prennent la valeur du bit de signe.

Le format du résultat de l'addition de deux opérandes au format (b_c, m_c, n_c) est présenté à l'expression 1.13. Nous obtenons un débordement si le résultat de l'addition des deux opérandes n'appartient pas au domaine de définition $\mathcal{D}_c = [-2^{m_c}; 2^{m_c} - 2^{-n}]$. Dans ce cas, un bit supplémentaire est nécessaire pour coder la partie entière du résultat de l'addition.

$$\begin{aligned} n_{Add} &= n_c \\ m_{Add} &= \begin{cases} m_c + 1 & \text{si} \quad a + b \notin \mathcal{D}_c \\ m_c & \text{si} \quad a + b \in \mathcal{D}_c \end{cases} \end{aligned} \tag{1.13}$$

Multiplication

Pour une multiplication, les deux opérandes doivent posséder la même représentation mais le nombre de bits réservés pour chaque partie peut être différent. Néanmoins, il est nécessaire avant d'effectuer l'opération, d'étendre le bit de signe. La multiplication de deux nombres en virgule fixe entraîne le doublement du bit de signe, celui-ci peut être éliminé automatiquement à l'aide d'un décalage à gauche. Pour un code en complément à 2 nous pouvons considérer que ce

bit de signe redondant appartient à la partie entière. Le format du résultat de la multiplication de deux opérandes a et b est alors le suivant :

$$\begin{aligned} m_{Mult} &= m_a + m_b + 1 \\ n_{Mult} &= n_a + n_b \\ b_{Mult} &= b_a + b_b \end{aligned} \tag{1.14}$$

1.1.3 Processus de codage

Nous présentons dans ce paragraphe le processus de codage d'une donnée en virgule fixe. Nous détaillons les caractéristiques des différentes lois de quantification et de dépassement pouvant être utilisées. Soit x une valeur arbitraire appartenant au domaine $\mathcal{D}$ et y une valeur du domaine de définition $\mathcal{D}_{\mathcal{R}}$ du codage choisi. Le domaine $\mathcal{D}_{\mathcal{R}}$ est borné par les valeurs X_{min} et X_{max}. Nous définissons le sous-ensemble $\mathcal{D}_D$ de $\mathcal{D}$ regroupant l'ensemble des valeurs de $\mathcal{D}$ comprises dans l'intervalle $[X_{min}; X_{max}]$. Le processus de quantification correspond à l'opération de réduction d'une valeur arbitraire x à une valeur représentable y. Ce processus est régi par deux lois présentées ci-dessous.

La loi de dépassement permet d'associer à l'ensemble des valeurs x de $\mathcal{D}$ une valeur x appartenant au domaine $\mathcal{D}_{\mathcal{D}}$. Elle définit plus précisément le comportement pour les valeurs présentes en dehors du domaine $\mathcal{D}_{\mathcal{D}}$. Nous associons à cette loi une fonction de dépassement définie ci-dessous :

$$f_D(x) = \begin{cases} x & \forall\, x \in \mathcal{D}_{\mathcal{D}} \\ D(x) & \forall\, x \notin \mathcal{D}_{\mathcal{D}} \end{cases} \tag{1.15}$$

La loi de quantification définit les valeurs représentables y à associer à l'ensemble des valeurs x appartenant au domaine $\mathcal{D}_{\mathcal{D}}$. La fonction de quantification associée est la suivante :

$$f_Q(x) = Q(x) \quad \forall\, x \in \mathcal{D}_{\mathcal{D}} \tag{1.16}$$

Le processus de quantification global peut s'exprimer sous la forme suivante :

$$x \to f_Q(f_D(x)) = \begin{cases} Q(x) & \forall\, x \in \mathcal{D}_{\mathcal{D}} \\ D(x) & \forall\, x \notin \mathcal{D}_{\mathcal{D}} \end{cases} \tag{1.17}$$

Lois de dépassement

Arithmétique de saturation : Cette loi appelée *loi de saturation*, consiste à choisir la valeur du domaine $\mathcal{D}_{\mathcal{D}}$ la plus proche de la valeur à représenter x :

$$D(x) = \begin{cases} X_{min} & \forall\, x < X_{min} \\ X_{max} & \forall\, x > X_{max} \end{cases} \tag{1.18}$$

La caractéristique de cette fonction est représentée à la figure 1.3.a.

Arithmétique modulaire : Cette loi de dépassement modulaire substitue aux valeurs de x n'appartenant pas au domaine $\mathcal{D}_{\mathcal{D}}$, x *modulo* $(X_{max} - X_{min})$. La caractéristique de cette loi est présentée à la figure 1.3.b.

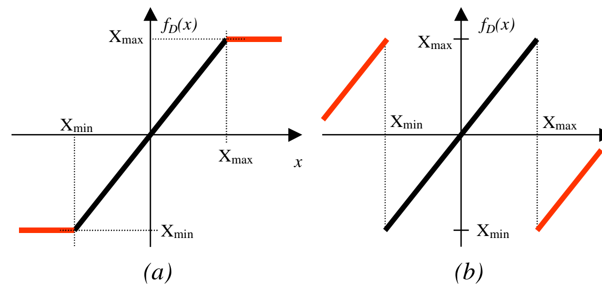


FIG. 1.3: Caractéristiques des lois de dépassement

Lois de quantification

Le domaine représentable $\mathcal{D}_{\mathcal{R}}$ est composé de N valeurs y_i avec $i = 1, 2, \ldots, N$ et le sous-domaine $\mathcal{D}_{\mathcal{D}}$ est subdivisé en N sous domaines juxtaposés Δ_i. La loi de quantification associe à tout x appartenant au domaine Δ_i la valeur y_i :

$$\forall\ x\ \in \Delta_i \quad Q(x) = y_i \tag{1.19}$$

Loi de quantification par arrondi : La loi de quantification par arrondi consiste à choisir la valeur représentable la plus proche de la valeur à quantifier en prenant la médiane de chaque intervalle Δ_i :

$$y_i = \frac{u_{i+1} - u_i}{2} = u_i + \frac{q}{2} \qquad \forall\ x\ \in \Delta_i = [u_i; u_{i+1}] \tag{1.20}$$

La caractéristique de cette loi est représentée à la figure 1.4.

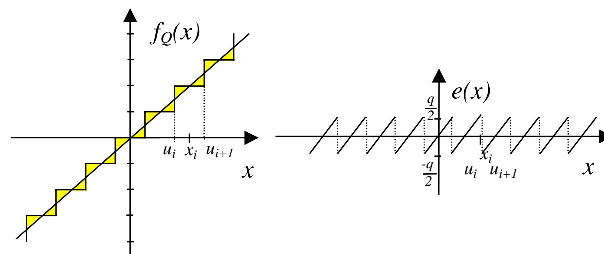


FIG. 1.4: Caractéristiques de la loi de quantification par arrondi

Loi de quantification par troncature : Cette loi de quantification consiste à tronquer un certain nombre de bits de poids faible. La quantification par troncature dans le cas d'une représentation en complément à 2 (voir figure 1.5) revient à prendre la valeur représentable immédiatement inférieure à la valeur à quantifier $y_i = u_i$.

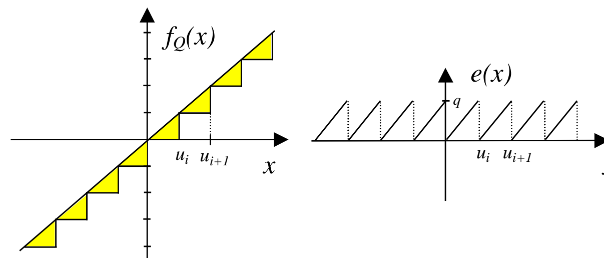


FIG. 1.5: Caractéristiques de la loi de quantification par troncature pour le codage CA2

1.1.4 Modélisation du processus de quantification

Dans cette partie, nous nous intéressons à la modélisation du processus de quantification d'un signal analogique. Tout d'abord, nous présentons l'analyse réalisée par Widrow qui permet de modéliser ce processus par un système linéaire où le signal quantifié est égal à la somme du signal d'origine et d'un bruit uniformément distribué [145] [146]. Ensuite, nous exposons les travaux de Sripad et Snyder [127] qui définissent les propriétés statistiques de l'erreur de quantification et les conditions de validité de cette approche. Les résultats sont présentés pour une loi de quantification par arrondi puis généralisés au cas de la quantification par troncature.

Méthode de Widrow

Modélisation du processus de quantification [145] [146] : La quantification d'un signal x de densité de probabilité continue $p_x(x)$ conduit à un signal y de densité de probabilité discrète $p_y(y)$ composée de N valeurs p_k. Chaque valeur p_k est égale à la probabilité que l'amplitude du signal x soit comprise dans l'intervalle $\Delta_k = [u_k, u_{k+1}]$. Elle correspond à l'aire de la densité de probabilité de x dans l'intervalle Δ_k. L'expression de la densité de probabilité $p_y(y)$ est la suivante :

$$p_y(y) = \sum_{k=1}^{N} p_k \delta(y - k.q) \quad \text{avec} \quad p_k = \int_{\Delta_k} p_x(x) dx \tag{1.21}$$

La fonction caractéristique d'une variable aléatoire est égale à la transformée de fourier inverse de sa densité de probabilité. En utilisant les différentes propriétés de la transformée de fourier, Widrow a démontré que la fonction caractéristique $\Phi_y(u)$ correspond à la duplication du produit des fonctions caractéristiques $\Phi_x(u)$ et $\Phi_q(u)$. A l'instar du théorème de Shannon pour l'échantillonnage des signaux analogiques, Widrow a proposé le théorème de quantification permettant de définir les conditions nécessaires pour reconstruire $p_x(x)$ à partir de $p_y(y)$ et vice-versa [147]. Si la condition proposée par Widrow [147] et présentée à l'équation 1.22 est respectée alors la variable aléatoire y est égale à la somme de deux variables aléatoires indépendantes x et e de densités de probabilité respectives $p_x(x)$ et $p_e(e)$. Ces résultats montrent que nous pouvons modéliser le processus de quantification par un système linéaire (figure 1.6) pour lequel la sortie est égale à la somme du signal d'entrée x avec une variable aléatoire e, appelée bruit de quantification ou erreur de quantification, dont la densité de probabilité est uniforme dans l'intervalle $[-q/2, q/2]$.

$$\Phi_x(u) = 0 \quad \text{pour } |u| > \frac{\Psi}{2} \tag{1.22}$$

FIG. 1.6: Modélisation du processus de quantification du signal x

Les expressions des moments du premier et du second ordre de l'erreur de quantification sont les suivantes :

$$\mu_e = \int_{-\infty}^{\infty} e\ p(e) de = \int_{-q/2}^{q/2} \frac{1}{q} e.de = 0 \tag{1.23}$$

$$\sigma_e^2 = \int_{-\infty}^{\infty} (e - \mu_e)^2\ p(e) de = \int_{-q/2}^{q/2} \frac{e^2}{q}.de = \frac{q^2}{12} \tag{1.24}$$

Méthode de Sripad et Snyder

Sripad et Snyder proposent une modélisation de l'erreur de quantification identique à celle de Widrow mais les conditions de validité de leur approche sont moins restrictives [127]. Le respect de ces conditions permet d'obtenir des propriétés intéressantes concernant le moment du second ordre de l'erreur de quantification et la corrélation entre le signal d'origine x et l'erreur de quantification e.

Modélisation de l'erreur de quantification : La densité de probabilité de l'erreur de quantification e est déterminée à partir de celle du signal d'entrée x de la manière suivante :

$$p_e(e) = \begin{cases} \sum_{k=-\infty}^{\infty} p_x(e - k.q) & -q/2 < e < q/2 \\ 0 & \text{ailleurs} \end{cases} \tag{1.25}$$

En calculant la fonction caractéristique de l'erreur de quantification, Sripad et Snyder ont montré qu'une condition nécessaire et suffisante pour que la densité de probabilité de l'erreur de quantification soit uniforme est que la fonction caractéristique de x soit nulle pour tout k entier différent de 0 :

$$\Phi_x(k\tfrac{2\pi}{q}) = 0 \quad \forall k \neq 0 \tag{1.26}$$

Cette condition est moins restrictive que celle proposée par Widrow. Ainsi, elle permet d'étendre la classe de signaux pour laquelle l'erreur de quantification est uniforme dans l'intervalle $[-q/2, q/2]$.

Statistique du second ordre de l'erreur de quantification : L'étude de la statistique du second ordre de l'erreur de quantification permet d'étudier les propriétés spectrales de celle-ci [161]. La densité de probabilité conjointe des variables aléatoires e_1 et e_2 représentant l'erreur de quantification aux instants t_1 et t_2, est égale à :

$$p_{e_1,e_2}(e_1, e_2) = \text{rect}\left(\frac{e_1}{q}\right)\text{rect}\left(\frac{e_2}{q}\right)\left[p_{x_1,x_2}(x_1, x_2) * \Big(f_d(e_1).f_d(e_2)\Big)\right] \tag{1.27}$$

En suivant une démarche similaire à celle utilisée dans les paragraphes précédents au travers du calcul de la fonction caractéristique associée à $p_{e_1,e_2}(e_1, e_2)$ les auteurs ont montré que la densité de probabilité conjointe $p_{e_1,e_2}(e_1, e_2)$ est égale à :

$$p_{e_1,e_2}(e_1, e_2) = \frac{1}{q}\text{rect}\left(\frac{e_1}{q}\right).\frac{1}{q}\text{rect}\left(\frac{e_2}{q}\right) = p_{e_1}(e_1).p_{e_2}(e_2) \tag{1.28}$$

Ainsi, les erreurs de quantification sont statistiquement indépendantes et le moment du premier ordre associé à la densité de probabilité conjointe des variables aléatoires e_1 et e_2 correspond au produit des moments du premier ordre associés à chaque variable aléatoire. Les moments d'ordre 1 des variables aléatoires e_1 et e_2 étant nuls, la fonction d'autocorrélation de l'erreur de quantification est la suivante :

$$\varphi_{ee}(\tau) = \begin{cases} q^2/12 & \tau = 0 \\ 0 & \text{ailleurs} \end{cases} \tag{1.29}$$

L'erreur de quantification est assimilable à un bruit blanc centré. Dans [67], l'auteur propose une condition plus simple à déterminer pour s'assurer que l'erreur de quantification est un bruit blanc.

Étude de la corrélation entre le signal d'entrée et l'erreur de quantification : La corrélation entre l'erreur de quantification et le signal d'entrée x peut être obtenue à partir de l'analyse du moment du second ordre de la variable aléatoire y représentant la somme de x et de e [161]. L'expression du moment du second ordre de cette somme est la suivante :

$$E(y^2) = E(x^2) + E(e^2) + 2E(x, e) \tag{1.30}$$

Le moment du second ordre de y correspond à la valeur à l'origine de la dérivée seconde de la fonction caractéristique de y. Le calcul de l'expression de cette dérivée seconde permet de déterminer l'expression de la corrélation entre les variables e et x :

$$E(xe) = \sum_{\substack{k=-\infty \\ k\neq 0}}^{\infty} q.\Phi_x'(-ku_0)\frac{(-1)^k}{\pi k} \tag{1.31}$$

Si la condition 1.32 est vérifiée alors $E(xe)$ est nulle [86]. Ainsi, l'erreur de quantification e n'est pas corrélée avec le signal d'entrée x.

$$\Phi_x'(k\tfrac{2\pi}{q}) = 0 \quad \forall k \neq 0 \tag{1.32}$$

L'expression 1.31 correspond à la déviation entre la corrélation réelle et la corrélation issue de la modélisation de l'erreur de quantification e par un bruit uniformément distribué. Dans le cas d'un signal distribué selon une loi normale, la condition 1.26 n'est pas respectée, mais la déviation issue de la modélisation est très faible dès que le pas de quantification est inférieur à l'écart type du signal [28] [161]. Ceci a été confirmé par les simulations présentées à la fin de cette section.

Extension à la quantification par troncature

L'analyse effectuée par Sripad et Snyder a été étendue au cas de la quantification par troncature dans le cas d'un codage en complément à 2. Nous présentons dans ce paragraphe les principaux résultats. L'expression de la densité de probabilité de l'erreur de quantification est la suivante :

$$p_{e_t}(e) = \mathrm{rect}\left(\frac{e - q/2}{q}\right)\Big[p_x(e) * f_d(e)\Big] \tag{1.33}$$

Si la condition 1.26 présentée à la page 14, est respectée alors la densité de probabilité de l'erreur de quantification est égale à :

$$p_{e_t}(e) = \frac{1}{q}\mathrm{rect}\left(\frac{e - q/2}{q}\right) \tag{1.34}$$

Les expressions des moments du premier et du second ordre et la fonction d'autocorrélation de l'erreur de quantification sont égales à :

$$\mu_{e_t} = \int_{-\infty}^{\infty} e\ p(e)\ de = \int_0^q \frac{1}{q}\ e\ de = \frac{q}{2} \tag{1.35}$$

$$\sigma_{e_t}^2 = \int_{-\infty}^{\infty} (e - \mu_e)^2\ p(e)de = \int_0^q \frac{1}{q}\Big(e - \frac{q}{2}\Big)^2\ de = \frac{q^2}{12} \tag{1.36}$$

$$\varphi_{ee_t}(\tau) = \frac{q^2}{12}\delta(\tau) + \frac{q^2}{4} \tag{1.37}$$

Ainsi, l'erreur de quantification peut être considérée comme un bruit blanc non centré. En suivant la démarche présentée précédemment, l'erreur de quantification e ne sera pas corrélée avec l'entrée x si la condition suivante est respectée :

$$\Phi'_x(k\frac{2\pi}{q}) = 0 \quad \forall k \tag{1.38}$$

Simulation

Pour illustrer les différents résultats présentés dans cette partie, nous avons simulé le processus de quantification et analysé les propriétés de l'erreur de quantification. Le signal d'entrée x est un bruit blanc gaussien d'écart type σ et non quantifié. En sortie de l'opérateur de quantification nous obtenons le signal y. L'erreur de quantification correspond à la différence entre les signaux x et y. Les simulations ont été réalisées pour les lois de quantification par arrondi et par troncature (codage CA2) et le nombre de bits utilisés pour coder y varie entre 1 et 24.

Nous avons comparé la variance σ^2_{sim} de l'erreur de quantification issue des simulations avec l'expression théorique σ^2_{theo}, en calculant C_σ, le rapport entre σ^2_{sim} et σ^2_{theo}. La figure 1.7 représente l'évolution de C_σ en fonction du rapport entre l'écart type σ_x du signal d'entrée et le pas de quantification q. Ces résultats sont quasiment identiques à ceux présents dans [127] et montrent la validité de l'expression de la variance pour $\sigma_x > q$. Cette condition est toujours vérifiée en pratique. De plus, nous avons mesuré le coefficient de corrélation entre l'entrée x et l'erreur de quantification e. Pour les deux lois de quantification, ce coefficient est inférieur à 0.02 si la condition $\sigma_x > q$ est vérifiée.

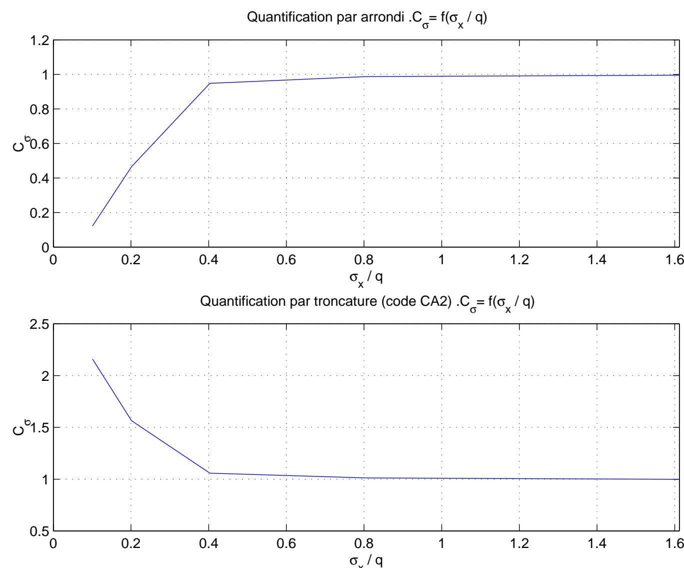


FIG. 1.7: Comparaison de la variance théorique σ^2_{theo} et simulée σ^2_{sim} de l'erreur de quantification en fonction du rapport entre la variance du signal x et le pas de quantification q. Le rapport $C_\sigma = \sigma^2_{sim}/\sigma^2_{theo}$ est présenté pour les lois de quantification par arrondi et par troncature

1.1.5 Comparaison des codages en virgule fixe et en virgule flottante

Dans cette partie, nous comparons la dynamique et le rapport signal à bruit de quantification des données codées en virgule fixe et en virgule flottante.

Analyse de la dynamique

Le niveau de dynamique exprimé en dB du codage en virgule fixe est linéaire par rapport au nombre de bits b utilisés par le codage :

$$D_{N\ dB} \simeq 20.(b-1).\log(2) \tag{1.39}$$

Le niveau de dynamique pour une représentation en virgule flottante est fonction du nombre de bits E alloués pour l'exposant :

$$D_{N\ dB} = 20\log(2^{2K+1}) \quad \text{avec} \quad K = 2^{E-1}-1 \tag{1.40}$$

Nous avons représenté à la figure 1.8 les expressions 1.39 et 1.40 en fonction du nombre de bits utilisés. Nous avons fixé pour le codage en virgule flottante la taille de l'exposant à 1/4 de la longueur totale. Lorsque le nombre de bits est inférieur à 16, le niveau de dynamique obtenu avec une représentation en virgule fixe est supérieur à celui d'une représentation en virgule flottante. Cette tendance s'inverse pour un nombre de bits supérieur à 16. Pour N=32, la représentation en virgule flottante montre tout son intérêt, la dynamique disponible permet d'utiliser ce codage dans la majorité des applications sans risque de débordement.

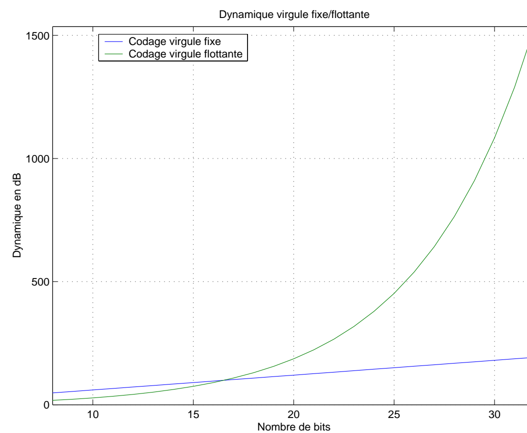


FIG. 1.8: Evolution du niveau de dynamique des codages en virgule fixe et en virgule flottante en fonction du nombre de bits utilisés

Analyse du Rapport Signal à Bruit de Quantification (RSBQ)

La puissance P_e du bruit de quantification correspond au moment du second ordre de l'erreur de quantification e :

$$P_e = \mu_e^2 + \sigma_e^2 \tag{1.41}$$

Soient P_x et D_x, respectivement la puissance et la dynamique du signal x. Nous définissons K_x le rapport entre la racine carrée de la puissance du signal x et sa dynamique. Ainsi, l'expression de la puissance P_x est la suivante :

$$P_x = \lim_{N\to\infty} \frac{1}{N} \sum_{k=0}^{N-1} x(k)^2 = (K_x D_x)^2 \tag{1.42}$$

L'expression du RSBQ exprimée en dB dans le cas d'un codage en virgule fixe est présentée ci-dessous. Cette expression montre que le RSBQ est linéaire par rapport à l'amplitude du signal d'entrée.

$$\rho_{dB} = 10\log(\frac{P_s}{P_e}) = 20\log(D_x) + 20\log(K_x) - 10\log(\mu_e^2 + \sigma_e^2) \tag{1.43}$$

Pour un codage en virgule flottante et une quantification par arrondi, l'expression du RSBQ est présentée dans [54]. Pour illustrer cette analyse nous avons représenté à la figure 1.9 un exemple de l'évolution du RSBQ en fonction de la dynamique du signal d'entrée. Cet exemple montre bien que le RSBQ est quasiment constant dans le cas de la virgule flottante. L'utilisation d'un exposant explicite dans le codage permet de s'adapter à la dynamique du signal et de maintenir un RSBQ constant et indépendant de la dynamique du signal. Pour les signaux de dynamique faible, très sensibles à l'erreur de quantification, la représentation en virgule flottante permet d'obtenir un meilleur RSBQ. Lorsque le nombre de bit est identique, le RSBQ du codage en virgule fixe est supérieur à celui en virgule flottante pour des signaux dont la dynamique d'entrée est élevée. Ceci correspond au cas où le pas de quantification de la représentation en virgule flottante devient supérieur à celui en virgule fixe. Pour le codage en virgule flottante, le choix du nombre de bits alloués à la mantisse et à l'exposant est un compromis entre une dynamique élevée et un RSBQ élevé.

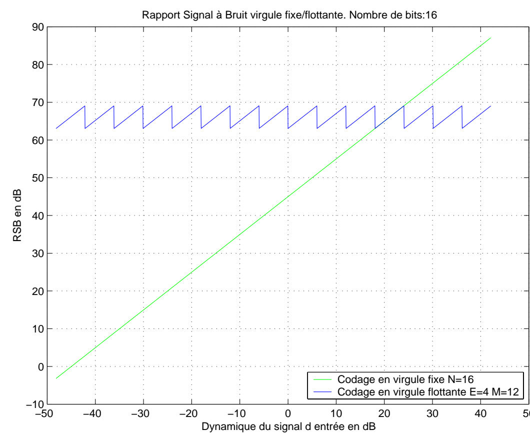


FIG. 1.9: Evolution du RSBQ en fonction de la dynamique du signal d'entrée

1.2 État de l'art des méthodologies

1.2.1 Introduction

Dans cette seconde partie, nous présentons les méthodologies existantes pour l'implantation automatique d'algorithmes spécifiés en virgule flottante au sein des processeurs de traitement du signal (DSP) en virgule fixe. La problématique est de rechercher la meilleure adéquation algorithme architecture d'un point de vue codage des données. Pour l'algorithme le facteur de qualité d'une implantation en virgule fixe correspond à la précision obtenue en sortie de l'algorithme. Concernant l'architecture, certains paramètres tels que le coût, la consommation d'énergie, le temps d'exécution et la taille du code doivent être optimisés.

La méthodologie de codage des données doit tenir compte des contraintes d'intégrité de l'algorithme et de précision des calculs. Ce codage doit respecter les règles de l'arithmétique virgule fixe et doit garantir l'absence de débordement. Le codage en virgule fixe ne permettant de représenter qu'un nombre limité de valeurs, il est nécessaire de s'assurer pour l'ensemble des données que les valeurs utilisées sont incluses dans le domaine de définition du codage choisi. De plus, l'utilisation du codage en virgule fixe implique que la précision des données est limitée et qu'une erreur est présente entre le signal obtenu et celui souhaité. Le codage des données devra permettre de respecter la précision souhaitée en sortie de l'algorithme.

L'analyse de ces deux contraintes conduit à une méthodologie de codage des données en plusieurs étapes. Tout d'abord, le domaine de définition des données présentes au sein de l'algorithme est déterminé. Ensuite, le format de chaque donnée est défini. Ce format doit permettre de représenter avec une précision suffisante l'ensemble des valeurs prises par cette donnée. En conséquence, la méthodologie doit réaliser une évaluation de la précision de la spécification en virgule fixe.

L'implantation d'algorithmes dans une architecture matérielle (ASIC, FPGA), nécessite de minimiser la largeur des données afin de diminuer la surface du circuit en vue de réduire les coûts. Dans le cadre des processeurs programmables (DSP), l'architecture étant figée, la méthodologie cherche à optimiser le *mapping* de l'algorithme dans l'architecture en utilisant au mieux les ressources offertes. La méthodologie doit déterminer le codage optimal permettant de maximiser la précision et de minimiser le temps d'exécution et la taille du code.

1.2.2 Détermination de la dynamique des données

Le but de cette première phase est de déterminer le domaine de définition de chaque donnée afin d'en déduire la position de la virgule par rapport au bit le plus significatif. Cette position est définie afin de garantir l'absence de débordement. Les conventions utilisées au niveau du format des données correspondent à celles présentées à la figure 1.1 de la page 7. La position de la virgule d'une donnée spécifiée à travers son paramètre m doit permettre de représenter l'ensemble des valeurs prises par cette donnée et d'éviter la présence de bits non utilisés au sein de celle-ci afin de maintenir une précision maximale. Ces deux contraintes permettent d'obtenir l'inégalité suivante :

$$2^{m_{x_i}-1} \leq \tilde{x}_{imax} < 2^{m_{x_i}} \tag{1.44}$$

$\tilde{x}_{imax}$ représente la valeur absolue maximale estimée de la donnée x_i. Elle est obtenue à partir de la connaissance du domaine de définition de chaque donnée selon la relation suivante :

$$\tilde{x}_{imax} = \max\Big(|\min(x_i)|, |\max(x_i)|\Big) = \max(|x_i|) \tag{1.45}$$

Deux types d'approche sont utilisés pour déterminer la valeur absolue maximale de chaque donnée. Pour la première approche, l'estimation $\tilde{x}_{imax}$ est déterminée à partir des paramètres

statistiques de chaque donnée obtenus par simulation de l'algorithme en virgule flottante. La seconde approche utilise une approche analytique. La valeur $\tilde{x}_{imax}$ en sortie d'un opérateur est obtenue à partir de la connaissance du domaine de définition des entrées de cet opérateur.

Méthode statistique

Ce type d'approche permet d'estimer la dynamique d'une donnée à partir de ses caractéristiques déterminées lors de la simulation de l'algorithme en virgule flottante. L'estimation de la dynamique la plus simple consiste à prendre la valeur absolue maximale des échantillons obtenus lors de la simulation [2]. Cependant, cette approche est très sensible au choix des stimuli d'entrée. Ainsi, dans [66] une méthode statistique est proposée et présentée ci-dessous. Elle estime la valeur absolue maximale à partir de la statistique du signal et de la fonction de distribution de celui-ci. La statistique des signaux est déterminée à partir de l'analyse des échantillons obtenus au cours de la simulation de l'algorithme en virgule flottante.

Chaque donnée est considérée comme une variable aléatoire x_i. L'expression du coefficient de dissymétrie (*skewness*) d'une variable aléatoire x_i, est définie de la manière suivante :

$$s_{x_i} = \frac{E\left[(x_i - \mu_{x_i})^3\right]}{\sigma_{x_i}^3} \tag{1.46}$$

Une fonction de distribution est symétrique si son coefficient de dissymétrie est nul. Le coefficient d'aplatissement (*kurtosis*) permet d'évaluer l'étalement d'une fonction de distribution par rapport à une loi normale. L'expression de ce coefficient normalisé afin d'obtenir une valeur nulle dans le cas d'une fonction de distribution gaussienne, est la suivante :

$$k_{x_i} = \frac{E\left[(x_i - \mu_{x_i})^4\right]}{\sigma_{x_i}^4} - 3 \tag{1.47}$$

L'analyse du nombre de maxima d'une fonction de distribution permet de définir le mode de celle-ci. Une fonction sera *uni-modale* si elle ne possède qu'un seul maximum et *multi-modale* si elle en possède plusieurs.

Cette méthode propose une expression permettant de calculer $\tilde{x}_{imax}$ si la fonction de distribution de x_i est symétrique et *uni-modale*. La symétrie est déterminée à partir de l'analyse du coefficient de dissymétrie. Kim et al. [62] proposent de discriminer les fonctions de distribution *uni-modales* et *multi-modales* à partir d'une méthode empirique basée sur l'étude du coefficient d'aplatissement. Une fonction est qualifiée d'*uni-modale* si son coefficient est compris dans l'interval $[-1.2, 5]$. Pour une distribution *uni-modale* et symétrique la valeur absolue maximale estimée est obtenue par la relation suivante :

$$\tilde{x}_{imax} = |\mu_{x_i}| + (4 + k_{x_i}).\sigma_{x_i} \tag{1.48}$$

Les informations sur la statistique des signaux sont obtenues à partir de simulations utilisant des stimuli d'entrée représentatifs. Pour obtenir une bonne estimation des différents paramètres, il est essentiel d'avoir un nombre d'échantillons élevé et différents jeux de stimuli d'entrée. Afin d'analyser la dispersion des paramètres statistiques au regard des différents jeux de stimuli j, un facteur de sensibilité est défini et intégré à l'estimation de chaque paramètre statistique.

Pour les distributions non symétriques et *multi-modales* la méthode précédente ne peut être utilisée, ainsi la dynamique est estimée directement à partir de l'analyse de la fonction de distribution de la manière suivante :

$$\tilde{x}_{imax} = x_{i_{100\%}} + g \quad \text{avec} \quad \begin{cases} x_{i_X} \text{ défini tel que } P(x_i < x_{i_X}) = X \\ g = \alpha \left(x_{i_{100\%}} - x_{i_{99\%}} \right) \end{cases} \tag{1.49}$$

g est une valeur de garde proportionnelle à la différence entre $x_{i_{100\%}}$ et $x_{i_{99\%}}$. Les valeurs de $x_{i_{100\%}}$ et de $x_{i_{99\%}}$ retenues sont celles du jeu de stimuli le plus défavorable.

L'outil développé pour estimer la dynamique des différents signaux est basé sur le principe de la surcharge des opérateurs en C++. Un nouveau type *fSig* est défini [61] [66], il permet de recueillir l'ensemble des informations nécessaires pour calculer les paramètres souhaités.

Méthode analytique

Le but de cette méthode est de déterminer le domaine de définition de chaque donnée à partir de la connaissance du domaine de définition d'un certain nombre de données et en particulier celles en entrée de l'algorithme. A partir d'une analyse statique du code source, un graphe flot de données et de contrôle (GFDC) est construit. Le domaine de définition des données en sortie d'un opérateur est déterminé à partir du domaine de définition des opérandes en entrée de celui-ci et de règles associées à chaque type d'opérateur.

Règles de propagation : Les règles pour déterminer le domaine de définition des données sont regroupées dans le tableau 1.2. Ces règles définissent le domaine de définition des données dans le pire cas et sont issues de la théorie de l'arithmétique d'intervalles [56]. Les extremums d'une donnée en sortie d'un opérateur sont calculés à partir des extremums des opérandes d'entrée. La probabilité d'obtenir ce type de situation pouvant être nulle, la méthode fait une estimation pessimiste du domaine de définition des données.

Ce type d'approche a été utilisé au sein de différents travaux [141, 148, 16]. Dans [16], l'utilisation des résultats de la théorie de l'algèbre (max, +) permet de déterminer la dynamique de données présentes au sein de boucles imbriquées pour lesquelles les bornes ne sont pas connues a priori.

z	$\min(z)$	$\max(z)$
$z = x + y$	$\min(x) + \min(y)$	$\max(x) + \max(y)$
$z = x - y$	$\min(x) - \max(y)$	$\max(x) - \min(y)$
$z = x * y$	$\min(E)$	$\max(E)$
$z = x >> n_s$	$\min(x).2^{n_s}$	$\max(x).2^{n_s}$

avec $E = \Big(\min(x)\min(y), \min(x)\max(y), \max(x)\min(y), \max(x)\max(y)\Big)$

TAB. 1.2: Règles de propagation de la dynamique pour les différentes opérations arithmétiques

Présentation de l'outil FRIDGE : Pour illustrer cette méthode de détermination du domaine de définition, nous présentons l'outil FRIDGE basé sur ce concept [150]. Cette phase d'évaluation du domaine de définition est couplée à l'étape de détermination du nombre de bits pour la partie fractionnaire. Le synoptique de cet outil développé à l'université de Aachen, est présenté à la figure 1.10. L'outil *CoCentric Fixed-point Designer* [133] commercialisé par la société Synopsys est basé sur cette méthodologie.

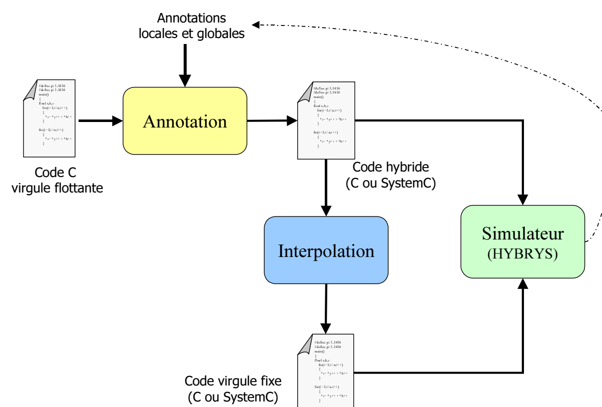


FIG. 1.10: Synoptique de la méthodologie associée à l'outil FRIDGE

Deux nouveaux types (*Fixed* et *fixed*) sont définis au sein de cet outil pour déclarer les données en virgule fixe [148] [58]. Ces deux types regroupent les mêmes informations sur le format des données (longueur totale de la donnée, longueur de la partie entière, signe) et sur les lois de quantification et de dépassement. Le type *fixed* utilise le concept ATI (*Assignement Time Instantiation*). Le format des variables est déterminé lors de l'instantiation et peut ainsi évoluer au sein de l'algorithme. Ce concept est nécessaire pour la phase d'*interpolation* car le format des variables n'est pas connu au moment de la déclaration. Il est lié au format des variables présentes en amont au sein du graphe flot de données et de contrôle. Pour le type *Fixed* le format des données est déterminé lors de la déclaration des variables. Lorsqu'un conflit de format intervient au moment de l'affectation entre deux variables, l'outil réalise soit un casting automatique ou demande à l'utilisateur une annotation permettant de résoudre le conflit.

L'utilisateur définit à l'aide des deux types *Fixed* et *fixed*, le format de quelques opérandes critiques et des données dont le format est connu. Cette première étape, appelée *Annotations locales*, conduit à une description de l'algorithme composée de données en virgule fixe et virgule flottante. Cette description peut être simulée afin de vérifier que celle-ci respecte toujours les contraintes souhaitées en termes de précision. De plus, cette simulation permet de collecter des informations supplémentaires (moyenne, variance, dynamique) en vue d'annoter de nouvelles données. Des annotations globales peuvent être ajoutées afin de spécifier des caractéristiques pour l'ensemble de l'algorithme telles que les restrictions (longueur maximale des données) ou les lois de quantification et de dépassement.

La seconde phase appelée *interpolation* [148][58] correspond à la détermination du domaine de définition et du nombre de bits pour la partie fractionnaire (paramètre n) des données non annotées. Ce paramètre n est déterminé afin de garantir l'absence de perte d'information en sortie de l'opérateur. Le format des données est obtenu à partir des règles de propagation définies ci-dessus et dans [148] et de l'analyse du flot de contrôle du programme. En sortie de cette phase, nous obtenons une description de l'algorithme source en virgule fixe, le format de chaque donnée est entièrement connu (type de codage, paramètres n et m). Cette description est simulée afin de vérifier l'adéquation avec les spécifications en termes de précision. Si les contraintes ne sont pas respectées, les annotations de départ sont modifiées et une nouvelle passe est réalisée.

Nous présentons ci-dessous, les solutions retenues pour le traitement des structures de contrôle. Pour les structures conditionnelles, l'approche analytique détermine le domaine de définition en se plaçant dans le cas le plus défavorable (estimation dans le pire cas). La méthode retient les valeurs maximales de m et n [148].

Pour les structures répétitives, il est nécessaire de distinguer les boucles pour lesquelles le nombre d'itérations est fixe ou borné par une valeur maximale et celles dont le nombre d'itéra-

tions n'est pas connu a priori. Dans le premier cas, la boucle peut être déroulée afin d'obtenir un code séquentiel. Le format des variables est déterminé à chaque itération de la boucle et stocké dans un tableau. De même, le principe d'analyse dans le pire cas est appliqué. Le format final retenu est celui pour lequel les valeurs de m et n sont maximales [58]. Dans le second cas, où le déroulage de boucle n'est pas possible, une technique itérative est utilisée [148]. Le format d'une donnée pour une itération i est déterminé en fonction du format des données en entrée de la boucle et du format des données obtenues à l'itération précédente $i-1$.

Conclusions

Comparaison des deux approches : Pour comparer ces deux approches nous avons analysé différents critères correspondant à la qualité et aux performances de l'estimateur, aux connaissances nécessaires a priori et aux informations fournies par l'estimateur.

Une approche est d'autant plus performante qu'elle minimise l'erreur entre l'estimation $\tilde{x_i}_{max}$ et la valeur absolue maximale ($\max(|x_i|)$). Une méthode est jugée de qualité si elle garantit que l'estimation $\tilde{x_i}_{max}$ est toujours supérieure aux valeurs réelles $|x_i|$, permettant d'assurer l'absence de débordement au sein de l'algorithme. La méthode analytique garantit l'absence de débordement au sein de l'algorithme si les valeurs des données en entrée sont comprises dans le domaine de définition utilisé par l'estimateur. En contrepartie les performances de cet estimateur sont faibles, car la méthode utilisée est conservatrice. L'estimation est basée sur une analyse dans le pire cas pouvant entraîner la présence de bits significatifs non utilisés dans la représentation des données réelles. La méthode statistique permet d'obtenir de meilleures performances en utilisant les informations liées à la statistique des données. La connaissance des moments des différentes données permet d'obtenir une meilleure estimation du domaine de définition des données. La méthode garantit que la probabilité de débordement est très faible pour les signaux dont les caractéristiques statistiques sont proches de celles des stimuli d'entrée utilisés pour déterminer le domaine de définition des données. Mais nous n'avons aucune information sur la probabilité de débordement pour les autres types de signaux. Cette méthode est intéressante pour les algorithmes pour lesquels nous avons une connaissance fiable des stimuli d'entrée. La qualité de l'estimation est fonction du nombre total d'échantillons utilisés et de la représentativité des signaux employés par l'estimateur en regard de ceux rencontrés en réalité.

La méthode analytique requiert juste la connaissance du domaine de définition des signaux d'entrée de l'algorithme. Pour la méthode statistique, il est nécessaire de posséder différents jeux de stimuli d'entrée représentatifs afin de pouvoir simuler l'algorithme.

Méthode utilisant les deux approches : La méthode proposée par Cmar et al. [23] se propose d'utiliser les résultats des estimations issues de la méthode analytique $\tilde{x_i}_{Ana}$ et de la méthode statistique $\tilde{x_i}_{Stat}$ afin de déterminer la position de la virgule et la loi de dépassement $D(x)$ à utiliser. Les règles associées à cette méthode sont présentées dans [23]

1.2.3 Evaluation de la précision

L'utilisation de l'arithmétique virgule fixe limite la précision des calculs et conduit à une erreur de quantification en sortie de l'algorithme correspondant à la différence entre le signal en précision infinie et en précision finie. Ainsi, il est nécessaire d'évaluer la précision des calculs afin de garantir que celle-ci est supérieure à un seuil minimal. Le critère le plus utilisé pour évaluer la précision est le Rapport Signal à Bruit de Quantification (RSBQ). Celui-ci correspond au rapport entre la puissance du signal et la puissance du bruit de quantification. Pour déterminer la puissance du bruit en sortie de l'algorithme deux types d'approche peuvent être utilisés. La

première approche consiste à déterminer les paramètres statistiques de l'erreur de quantification à partir de la simulation de l'algorithme en virgule fixe et en virgule flottante. La seconde détermine l'expression analytique de la puissance du bruit en propageant un modèle de bruit au sein du graphe flot de l'algorithme.

Méthode statistique par simulation

La précision de la spécification en virgule fixe est évaluée de manière statistique à partir des signaux obtenus après simulation de l'algorithme en virgule fixe et en virgule flottante. La puissance du bruit de quantification $P_{b\ sortie}$ est déterminée à travers le calcul du moment du second ordre de la différence entre la sortie obtenue en précision infinie et en précision finie.

Afin d'obtenir des résultats précis, il est nécessaire d'utiliser un nombre d'échantillons en entrée de l'algorithme (N_{ech}) élevé. Soit N_{ops} le nombre d'opérations présentes dans la description de l'algorithme et N_i le nombre de fois que la précision de cette description est évaluée. Une première estimation du nombre de points (N_{pts}) à calculer, présentée à l'équation 1.50 met en évidence la nécessité d'utiliser un simulateur efficace afin d'obtenir des temps de simulation raisonnables. En effet, dès que nous souhaitons optimiser la spécification en virgule fixe afin de minimiser un paramètre, le nombre d'itérations du processus d'évaluation de la précision (N_i) peut devenir élevé.

$$N_{pts} = N_{ech} * N_{ops} * N_i \tag{1.50}$$

Dans la partie suivante, les différentes méthodes utilisées pour émuler les mécanismes de l'arithmétique virgule fixe sont présentées. La première méthode est basée sur les propriétés de la surcharge des opérateurs au sein des langages objet. Les autres méthodes utilisent les types disponibles sur la machine hôte. Les concepts utilisés et les performances en termes de temps de simulation sont détaillés.

La méthode présentée dans [66] permet de simuler une spécification en virgule fixe, en utilisant les concepts de surcharge des opérateurs au niveau du langage C++. Un nouveau type *gFix* [61] [65] est introduit, il est composé de la valeur de la donnée, de sa largeur totale, de la largeur de sa partie entière et de différents attributs. Ces derniers représentent le type de codage et les lois de quantification et de débordement. Les différents opérateurs arithmétiques sont surchargés afin d'implanter les mécanismes de l'arithmétique virgule fixe. Lors d'une assignation, une conversion du format entre la donnée de droite et celle de gauche est réalisée en suivant les règles spécifiées au sein des attributs. La spécification en entrée du simulateur correspond au programme source écrit en C++, où le type des données en virgule flottante a été remplacé par le type *gFix*. Les différents membres de chaque donnée sont initialisés au cours des phases de détermination du format des données. La surcharge des opérateurs met en évidence que l'exécution de l'algorithme en virgule fixe sera nettement plus longue qu'une simulation en virgule flottante. Les mêmes concepts sont mis en œuvre pour la simulation en virgule fixe à l'aide de SystemC [115]. Dans [26], les temps de simulation en virgule flottante et en virgule fixe sont comparés pour quelques applications de traitement du signal. Les temps de simulation obtenus avec la librairie SystemC classique et celle en précision limitée sont respectivement en moyenne 540 et 120 fois supérieurs à ceux obtenus en virgule flottante.

Afin de diminuer le temps de simulation par rapport à la méthode présentée ci-dessus un nouveau type *pFix* a été proposé [66]. Il a pour but d'améliorer le temps de calcul des opérations en virgule fixe en utilisant au mieux, le chemin de données de l'unité de calcul en virgule flottante. Pour cela, le type *pFix* utilise la mantisse des données en virgule flottante pour stocker les données en virgule fixe. Ce type permet d'améliorer nettement le temps de simulation de l'algorithme, mais il est limité par la taille maximale de la mantisse (53 bits dans le cas d'un

double). Le temps de simulation en virgule fixe d'un filtre IIR d'ordre 4 avec le type *pFix* est 7,5 fois supérieur à celui obtenu en virgule flottante [66].

La méthode [27] présentée ci-dessous propose d'utiliser les différents types entiers présents sur la machine afin de coder plus efficacement les données de l'algorithme en vue de diminuer le temps d'exécution des simulations. Ce concept est aussi utilisé dans le simulateur HYBRIS associé à l'outil FRIDGE [58] (figure 1.10). La largeur des données à coder étant inférieure à celle des différents types supportés par la machine, il existe de nombreux degrés de liberté pour l'implantation de ces données. Ces degrés de liberté vont permettre d'optimiser le cadrage afin de minimiser le temps d'exécution de la simulation.

Pour optimiser le placement des données dans les mots de la machine hôte, la méthode minimise un coût reflétant le temps d'exécution de la simulation. Celui-ci est composé de deux éléments. Le premier représente le coût lié à l'alignement des données avant les opérations arithmétiques. L'évolution de ce coût est fonction de l'architecture des registres à décalage au sein de la machine (registres en barillet...). Le second coût est lié à la réalisation des mécanismes de quantification et de dépassement. Deux approches sont utilisées pour implanter ces mécanismes. La première teste la donnée puis la modifie si nécessaire. Dans le second cas, les mécanismes de quantification et de débordement sont réalisés en utilisant ceux présents dans la machine. Les données sont alignées sur une des extrémités (MSB ou LSB) du mot machine utilisé. Dans ce dernier cas, le coût est lié aux décalages réalisés pour aligner la donnée. Nous obtenons un problème d'optimisation combinatoire dont les variables sont les décalages entre les différentes données et dont le coût à minimiser représente le nombre de cycles du code obtenu. La fonction de coût étant non linéaire, les techniques de programmation linéaire en nombres entiers ne permettent pas d'obtenir de résultats satisfaisants. L'espace de recherche étant très large, différentes heuristiques ont été mises en œuvre afin de limiter celui-ci.

L'utilisation des types de la machine permet de diminuer nettement le surcoût de la simulation en virgule fixe par rapport aux méthodes basées sur la surcharge des opérateurs. Les temps de simulation avec la méthode implantée dans HYBRIS sont 3,6 fois supérieurs à ceux obtenus en virgule flottante [26]. Cependant, l'effort nécessaire pour obtenir ce code en virgule fixe optimisé pour la simulation n'a pas été quantifié. En effet, les techniques d'optimisation présentées ci-dessus sont relativement complexes et peuvent conduire à des temps d'optimisation importants par rapport au temps de simulation de l'algorithme. Ainsi, le gain lié à l'utilisation de ce code optimisé peut être annihilé par l'effort nécessaire à l'optimisation de celui-ci.

Méthode analytique

Dans [141, 92], une méthode de calcul analytique du bruit de quantification est proposée afin d'éviter une phase de simulation nécessitant des puissances de calcul élevées. Cette méthode est basée sur la propagation de modèles de bruit au sein de la description de l'algorithme. Chaque opérateur est modélisé par une source de bruit propagé et une source de bruit généré. Afin d'obtenir des expressions des bruits générés et propagés indépendantes de la statistique des signaux utilisés en entrée des opérateurs, l'auteur a posé plusieurs hypothèses. Les variables en entrée de chaque opérateur sont considérées comme indépendantes et centrées. Les bruits générés lors d'un changement de format sont assimilés à des variables aléatoires centrées. Ainsi, ce modèle n'est valide que pour les opérateurs utilisant une loi de quantification par arrondi. Chaque variable est codée en virgule fixe cadrée à gauche et afin d'être indépendant de la puissance de chaque donnée, celle-ci est bornée par la valeur 1. Ces différentes simplifications permettent d'obtenir une expression de la puissance du bruit en sortie, proportionnelle au carré du pas de quantification q.

L'application est modélisée par un graphe flot de données et celui-ci est parcouru des entrées vers la sortie et pour chaque nœud le bruit est calculé à partir des prédécesseurs de ce

nœud. Cette méthode permet de calculer plus rapidement le bruit de quantification en sortie de l'algorithme par rapport aux méthodes basées sur la simulation. Cependant, l'utilisation d'une technique de propagation d'un modèle de bruit ne permet pas de traiter les graphes possédant des cycles. Ainsi, la méthode ne peut pas être appliquée aux systèmes récursifs.

Conclusions

Les deux types d'approche permettant d'évaluer la précision d'une implantation en virgule fixe ont été détaillés. Les méthodes statistiques basées sur la simulation de l'algorithme en virgule fixe permettent de traiter tous les types d'algorithme de traitement numérique du signal. Cependant, ces techniques conduisent à des temps de simulation élevés et plus particulièrement si elles sont intégrées au sein d'un processus d'optimisation du format des données où de multiples simulations sont nécessaires. En effet, l'émulation des mécanismes de l'arithmétique virgule fixe augmente le temps de simulation par rapport à la virgule flottante. De plus, il est nécessaire d'utiliser de nombreux échantillons d'entrée afin d'obtenir des paramètres statistiques réalistes.
Le processus d'optimisation du format des données nécessite d'explorer l'espace de conception des différents formats. Pour les méthodes basées sur la simulation, une nouvelle simulation doit être réalisée dès que le format d'une donnée est modifié. Ce type de méthode a été utilisé dans le cadre de la conception matérielle (ASIC) afin de minimiser la surface du circuit tant que le RSBQ est supérieur à un seuil [62, 132, 57]. Pour limiter le nombre de simulations des heuristiques ont du être mises en œuvre afin de limiter l'espace de recherche [132]. En effet, des méthodes de recherche exhaustives conduisaient à des temps d'optimisation prohibitifs.

La seconde approche correspond aux méthodes analytiques qui permettent d'obtenir l'expression du RSBQ. Cette expression fournit plus d'information sur le comportement du bruit au sein de l'algorithme par rapport à une méthode basée sur la simulation. En effet, cette approche permet d'analyser l'influence de telle ou telle opération sur la précision globale des calculs. La détermination de l'expression du RSBQ est réalisée une seule fois et ensuite le temps de calcul du RSBQ correspond au temps d'évaluation de la valeur de l'expression du RSBQ. Ce temps de calcul est définitivement plus faible que le temps de simulation de l'application en virgule fixe. Ainsi, l'utilisation de cette technique au sein du processus d'optimisation du format des données conduit à des temps d'optimisation nettement plus faibles que ceux obtenus avec une méthode basée sur la simulation.
La méthode proposée par J.M. Toureilles [141] conduit à une estimation du RSQB peu précise en raison des différentes hypothèses posées. De plus, cette méthode ne permet de traiter que les structures non-récursives. Un des objectifs de nos travaux de recherche est de définir et de mettre en œuvre une méthode de détermination de l'expression analytique du RSBQ permettant aussi de traiter les structures linéaires récursives et fournissant une estimation précise du RSBQ.

1.2.4 Implantation des algorithmes dans les DSP

Les méthodologies d'implantation d'algorithmes de traitement du signal (TNS) dans les processeurs en virgule fixe doivent coder les données en tenant compte des contraintes liées à la virgule fixe et à la génération de code pour DSP. Le code généré doit être optimisé en termes de taille, de temps d'exécution et de précision en sortie de l'algorithme. De plus, l'implantation doit assurer l'intégrité de l'algorithme (absence de dépassement, alignement correct des données).

Différentes approches sont possibles pour intégrer la conversion en virgule fixe au sein du processus d'implantation d'algorithmes de TNS dans les DSP. Les premières méthodes présentées ci-dessous réalisent la conversion en virgule fixe au niveau du code source et fournissent à un compilateur classique un code de haut niveau modifié permettant de prendre en compte les

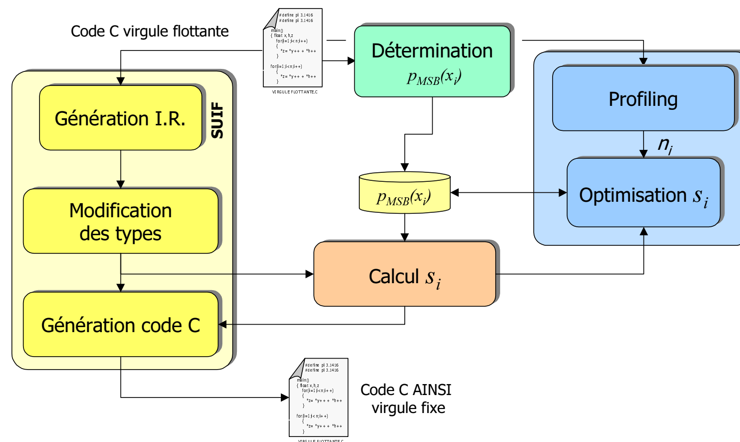


FIG. 1.11: Synoptique de la méthodologie *Autoscaler for C*

aspects virgule fixe. Dans la seconde approche, le passage en virgule fixe est mis en œuvre après la génération de code.

Conversion en virgule fixe avant la génération de code

Le but des méthodes présentées dans cette partie est de réaliser la conversion des données en virgule fixe au niveau du code source. Les deux premières méthodes présentées, gênèrent en sortie un code C ANSI pouvant être utilisé par un compilateur C classique.

Méthodologie *Autoscaler for C* : La méthodologie présentée dans [69] a pour objectif de minimiser le nombre d'opérations de recadrage présentes au sein du code C virgule fixe généré. Cette méthodologie est composée de deux parties. La première correspond à la conversion de la description en virgule flottante en un code C utilisant le type entier [68] et la seconde partie permet d'optimiser le codage des données afin de minimiser le nombre de décalages. Le synoptique de cette méthode est représenté à la figure 1.11.

La conversion de code est réalisée à l'aide de l'outil SUIF conçu pour la réalisation de l'ensemble des phases de la partie frontale d'un compilateur. La spécification en virgule flottante est convertie en une représentation intermédiaire pour laquelle chaque expression est représentée par un arbre (arbre d'expression). Pour chaque donnée en virgule flottante, l'outil modifie la table des symboles en substituant le type entier au type flottant. Les expressions permettant le cadrage des données sont ajoutées à la représentation intermédiaire en utilisant les résultats fournis par le module de calcul des décalages. La représentation intermédiaire est transformée en un code C ANSI n'utilisant que le type entier.

Pour faciliter le cadrage des données, les différents éléments d'un tableau et les données associées à un même pointeur sont codés avec le même format. Le format retenu correspond au format de la donnée possédant la dynamique la plus élevée.

La réduction du nombre de décalages est basée sur l'égalisation du format de données pertinentes. Cette réduction est réalisée de façon globale afin de ne pas effectuer d'optimisations locales susceptibles de dégrader le coût total. Cette partie est composée de trois phases correspondant à la détermination des décalages, à leur optimisation et au *profiling* du code. Ce *profiling* permet de déterminer la fréquence d'utilisation de chaque expression et de détecter les boucles nécessitant une attention plus particulière en vue de diminuer de façon significative le

temps d'exécution de l'algorithme.

Le module de calcul des décalages détermine les équations permettant d'obtenir la valeur des décalages à réaliser afin d'aligner correctement les données. Chaque expression en virgule flottante est mise sous la forme suivante :

$$x_i = \sum_{j,k} x_j * x_k + \sum_l x_l \tag{1.51}$$

Ces expressions sont transformées en virgule fixe en incluant les instructions de cadrage de la manière suivante :

$$x_i = (\sum_{j,k} (x_j * x_k) >> s_{jk} + \sum_l (x_l >> s_l)) << s_i \tag{1.52}$$

Pour chaque expression les décalages sont calculés en résolvant le système d'équations définissant la position de la virgule m_x de chaque variable x de l'expression 1.52 :

$$\begin{cases} m_{max} = \max_{j,k,l} \left(m_{x_j} + m_{x_k} + 1, m_{x_l}, m_{x_i} \right) \\ s_{jk} = m_{max} - (m_{x_j} + m_{x_k} + 1) \\ s_l = m_{max} - m_{x_l} \\ s_i = m_{max} - m_{x_i} \end{cases} \tag{1.53}$$

Une fonction de coût est calculée à partir de la valeur des décalages, des informations issues du *profiling* et du type d'architecture du DSP. Deux fonctions de coût sont utilisées en fonction de la présence ou non d'un registre à décalage en barillet au sein du processeur. Lorsqu'un registre de ce type est absent, le processeur réalise un décalage d'un bit par cycle, ainsi l'expression de la fonction de coût globale est la suivante :

$$C = \sum_i (s_i + \sum_j s_{ij}) n_i \tag{1.54}$$

avec

- s_{ij} : nombre de décalages du j ^{ème} terme de la i ^{ème} expression ;
- s_i : nombre de décalages pour l'assignation de la i ^{ème} expression ;
- n_i : nombre d'exécutions de la i ^{ème} expression, déterminé au sein de la phase de *profiling*.

Pour les architectures possédant un registre à décalage en barillet, l'expression de la fonction de coût globale est la suivante :

$$C = \sum_i \Big(f_B(s_i) + \sum_j f_B(s_{ij}) \Big) n_i \quad \text{avec} \quad f_B(x) = \begin{cases} 0 \text{ si } x = 0 \\ 1 \text{ sinon} \end{cases} \tag{1.55}$$

La fonction de coût est minimisée en fixant des limites de variation de la position de la virgule. Pour garantir l'absence de débordement, la position de la virgule (m) doit rester supérieure à celle estimée lors de la phase de détermination du domaine de définition des données. Afin de limiter la dégradation de la précision, une valeur maximale de m est fixée. L'algorithme de recuit simulé est utilisé pour minimiser la fonction de coût.

Méthodologie FRIDGE : Dans [149], l'auteur propose une méthode appelée *embedded approach* permettant de générer un code C-ANSI pouvant être utilisé par un compilateur pour DSP. Cette méthodologie implante au sein d'un DSP de manière exacte (*bit-true implementation*), la spécification en virgule fixe issue des phases d'*annotation* et d'*interpolation* de l'outil FRIDGE présenté au paragraphe 1.2.2. Elle reprend les concepts de la méthode de simulation implantée dans l'outil HYBRIS présenté au paragraphe 1.2.3. Chaque donnée (donnée source) dont le format a été déterminé au cours des phases d'*annotation* et d'*interpolation*, est insérée

au sein d'une donnée (donnée cible) dont le type est supporté par le processeur. Si la largeur de la donnée source est inférieure à celle de la donnée cible, alors des degrés de liberté apparaissent sur la position de la donnée source au sein de la donnée cible. Ces degrés de liberté vont permettre de minimiser les décalages nécessaires pour aligner les données. De plus, les opérations de saturation et de quantification par arrondi ou troncature présentes dans la spécification sont reproduites. Cette méthodologie a été développée en particulier pour cibler le processeur TMS320C62x de la société Texas. Afin de diminuer le temps d'exécution des opérations permettant d'implanter les lois de quantification et de dépassement, les fonctions intrinsèques associées à ce processeur sont utilisées.

Méthodologie MakeFixed (mkfixd) : L'objectif de cet outil [2] est de réaliser l'implémentation d'un code C virgule flottante au sein de processeurs spécialisés virgule fixe de type ASIP (*Application Specific Instruction-set Processor*). Cette conversion est réalisée en deux étapes au niveau d'une représentation intermédiaire obtenue à l'aide de l'infrastructure SUIF. La première phase correspond à l'instrumentation du code source afin de collecter les valeurs minimales et maximales des données pour déterminer leur dynamique. Dans un second temps, les opérations de décalage sont insérées lors du parcours de chaque arbre d'expression de la représentation intermédiaire. Les règles associées à chaque type d'opération sont définies afin que l'opérande en sortie de celle-ci soit codé au plus précis. Ensuite, les opérations de décalage redondantes sont éliminées si ce processus n'affecte pas la précision des calculs. De plus, cette méthodologie propose la mise en œuvre d'un nouveau type d'instructions dans le cadre de la conception conjointe matérielle-logicielle des ASIP. L'objectif de ce type d'instructions est d'optimiser la précision et le temps d'exécution des opérations de multiplication suivies d'un décalage [3].

Conversion en virgule fixe après la génération de code

Méthodologie de conversion pour le TMS320C25/50 : La méthode présentée dans ce paragraphe se propose d'intégrer la gestion des aspects virgule fixe après la phase de génération de code. La méthode consiste à générer du code à partir d'une représentation en C sans se soucier du cadrage des données et ensuite, à déterminer les décalages qui permettront d'éviter les débordements et de maximiser la précision en sortie de l'algorithme. Cette méthode a été développée uniquement pour les processeurs de la famille TMS320C25/50 de la société Texas Instruments (T.I.). Elle met en œuvre une stratégie d'initialisation des registres à décalage basée sur une modélisation de l'architecture par un système d'équations linéaires. Pour cette méthodologie, 2 compilateurs ont été utilisés : le compilateur C de la société T.I. [64] et le compilateur C GNU [55]. La méthodologie de génération de code et de cadrage des données varie en fonction du compilateur utilisé, mais nous pouvons en ressortir une démarche globale que nous allons décrire dans cette partie. Le synoptique de l'outil associé est présenté à la figure 1.12.

La première étape correspond à la détermination du domaine de définition des données de l'application au niveau du code source. La partie frontale du compilateur C est classique, elle génère une représentation intermédiaire à partir du code source. Cette représentation intermédiaire est fournie au générateur de code qui réalise les phases de sélection de code, d'allocation de registres et d'ordonnancement. Le code non cadré, issu de cette phase de génération de code, est segmenté en blocs afin qu'au sein de chaque bloc, le format de l'accumulateur soit unique. Un nouveau bloc est créé lors du chargement de l'accumulateur avec une variable mémoire ou lors de la remise à zéro de celui-ci. Pour chaque bloc, la stratégie de codage présentée dans la partie suivante est utilisée pour calculer les cadrages nécessaires en utilisant les informations issues de la phase de détermination du domaine de définition des données. Cette phase génère un code cadré sur lequel des post-optimisations sont réalisées afin d'obtenir le code assembleur final.

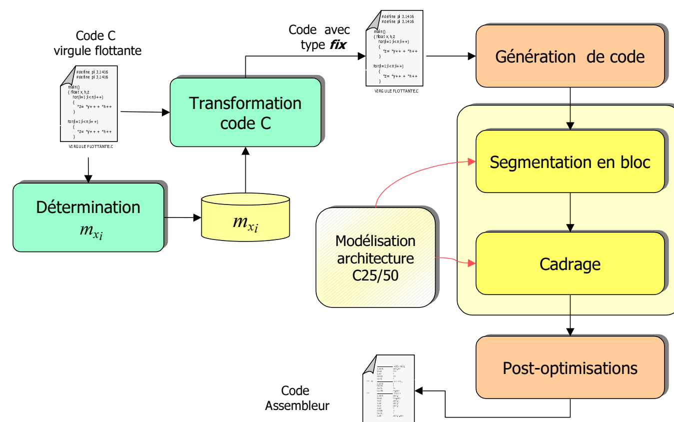


FIG. 1.12: Synoptique de la méthodologie de conversion pour le TMS320C25/50

Le chemin de données du TMS320C50 est représenté à la figure 1.13.a. Il est composé de 3 registres à décalage. Les registres à décalage A et O permettent de réaliser le cadrage des données lors des transferts entre l'unité de traitement et la mémoire. Le registre à décalage en sortie du multiplieur ne peut réaliser que 4 décalages prédéfinis. Les positions des virgules des différentes données présentes dans l'unité de traitement sont régies par le système d'équations suivant :

$$\begin{cases} m_{ACC} = m_x + 16 - s_a & \text{avec} \quad s_a = [0...16] \\ m_{ACC} = m_W + m_Y + 1 - s_m & \text{avec} \quad s_m = -6, 0, 1, 4 \\ m_z = m_{ACC} - s_z & \text{avec} \quad s_z = [1...7] \end{cases} \tag{1.56}$$

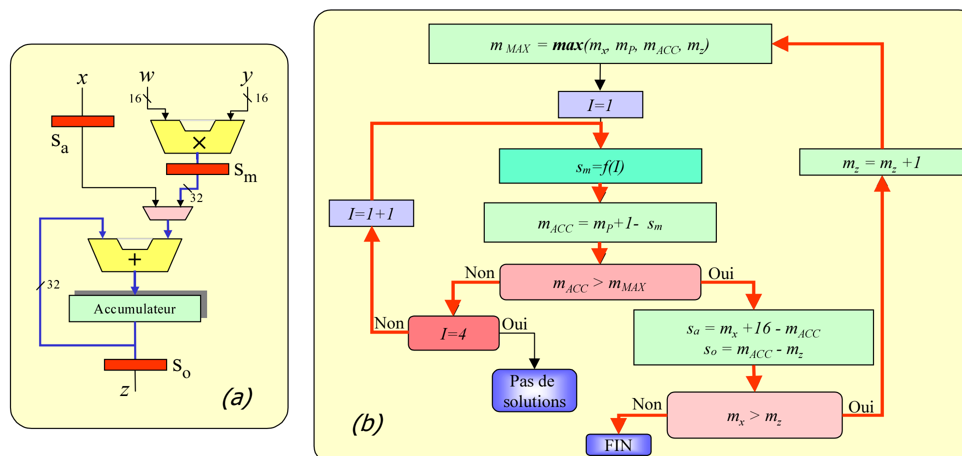


FIG. 1.13: Architecture du TMS320C25/C50 et stratégie de codage associée à la méthodologie

La stratégie permettant de déterminer les paramètres des différents registres à décalage est présentée à la figure 1.13.b [64] [63]. La première phase consiste à détecter la donnée induisant la contrainte la plus forte, c'est à dire celle dont le paramètre m représentant la position de la virgule est le plus élevé au sein du bloc. La seconde phase détermine le format de l'accumulateur. Ce format doit permettre de stocker la donnée possédant la dynamique la plus importante x_{max} et être compatible avec le format du résultat de la multiplication. En effet, le registre à décalage M ne peut réaliser que 4 décalages prédéfinis. La méthode recherche la valeur du décalage à gauche maximale s_m afin d'obtenir la meilleure précision possible et permettant aussi de représenter x_{max} dans l'accumulateur. La dernière phase paramètre les registres à décalage d'entrée A et de sortie O à l'aide des expressions 1.56. Si une solution ne peut être trouvée, le

format de la sortie est modifié et la procédure est recommencée.

Dans [131], l'auteur propose un compilateur C virgule fixe réalisé à partir du compilateur flexible LCC et basé sur le même concept. Mais à la différence de la méthode présentée dans cette partie, le cadrage des données est réalisé sur la représentation intermédiaire (DAG) avant la génération du code. Ce compilateur utilise une stratégie de détermination de la valeur des registres à décalage plus conservatrice car la position de la virgule de la donnée stockée dans l'accumulateur n'est pas connue au moment de la compilation (son contenu exact n'est connu qu'après la génération du code).

Conclusions

La méthodologie présentée au paragraphe 1.2.4 réalise la conversion en virgule fixe après la génération de code. Ainsi, elle permet de prendre en compte l'architecture du processeur lors du processus de codage. Cependant, la modélisation de l'unité de traitement du processeur sous la forme d'un système d'équations linéaires et la stratégie de codage ne peuvent pas être généralisées à d'autres classes d'architecture de DSP. De plus, la segmentation du code en bloc ne permet pas d'optimiser globalement l'algorithme.

La méthodologie associée à l'outil *makefixed* développée à l'Université de Toronto réalise la conversion d'une spécification en virgule flottante en virgule fixe au niveau de la représentation intermédiaire avant la génération de code. La méthodologie associée à l'outil FRIDGE permet d'implanter exactement une spécification en virgule fixe au sein d'un DSP. Cependant, les opérations permettant de réaliser les lois de quantification et de dépassement insérées dans le code afin d'obtenir une implantation exacte au niveau bit vont augmenter inutilement le temps d'exécution du code. De plus, ces deux méthodologies ne cherchent pas à optimiser le RSBQ en sortie de l'algorithme ou le temps d'exécution du code à travers la modification du format de certaines données afin d'obtenir une implantation de meilleure qualité.

La méthodologie *Autoscaler for C* développée à l'université de Séoul s'intègre bien dans la problématique de notre thème de recherche. Elle prend en compte la plupart des contraintes liées à l'implantation d'un algorithme en virgule flottante dans un processeur programmable en virgule fixe. Cette méthode respecte l'intégrité de l'algorithme en garantissant l'absence de débordement et en calculant les décalages nécessaires à l'alignement des données. De plus, elle cherche à optimiser les décalages afin de minimiser le temps d'exécution du code généré. Mais la méthode n'optimise pas le temps d'exécution du code sous contrainte d'un RSBQ souhaité en sortie de l'algorithme. Nous ne pouvons pas fournir à la méthode un objectif de RSBQ en sortie de l'algorithme. La contrainte de RSBQ est uniquement présente à travers la définition d'une dégradation maximale de la précision autorisée pour chaque donnée. De plus, le modèle d'architecture utilisé n'est pas assez réaliste. Les registres à décalage spécialisés permettant de réaliser certains décalages sans cycle supplémentaire ne sont pas pris en compte. Le coût d'un décalage réalisé à l'aide d'un registre en barillet au sein d'une architecture DSP traditionnelle n'est pas toujours égal à un cycle. Il dépend de la localisation de la donnée au sein de l'unité de traitement. Pour les processeurs possédant du parallélisme au niveau instruction, le surcoût lié au décalage d'une donnée est fonction de la phase d'ordonnancement des instructions et ne peut être déterminé sans tenir compte des différentes phases de la génération du code. Ces différents aspects sont détaillés dans le chapitre suivant.

Pour les différentes méthodologies réalisant la conversion en virgule fixe avant la génération de code, l'architecture du processeur n'est pas prise en compte ou que très partiellement lors du processus de codage des données en virgule fixe. Ainsi, au sein de nos travaux de recherche, les conséquences de l'absence de prise en compte de l'architecture lors du processus de codage, sont étudiées à travers l'analyse de l'influence de l'architecture sur la précision des calculs.

1.3 Conclusions

Dans la première partie de ce chapitre, nous avons présenté les caractéristiques et les propriétés des différents types de codage en virgule fixe et en virgule flottante. L'analyse de la dynamique du codage et du RSBQ permet de montrer la supériorité, d'un point de vue arithmétique, du codage en virgule flottante. Cependant, nous détaillerons dans le chapitre suivant les différentes raisons de la domination des processeurs en virgule fixe. De plus, nous avons présenté une modélisation du processus de quantification d'un signal d'amplitude continue et nous avons fourni les conditions de validité de ces modèles.

Dans la seconde partie de ce chapitre, nous avons présenté les différentes méthodologies existantes pour implanter un algorithme spécifié en virgule flottante dans une architecture en virgule fixe. Les différentes parties de ces méthodologies ont été détaillées. Tout d'abord, la dynamique de l'ensemble des données de l'algorithme est estimée. Les deux types d'approche permettant de déterminer le domaine de définition des données ont été exposés.
Les différentes techniques d'évaluation de la précision ont été détaillées. Les méthodes basées sur la simulation permettent de traiter tous les algorithmes, mais elles conduisent à des temps de simulation élevés en particulier si elles sont utilisées au sein du processus d'optimisation du format des données. Ainsi, l'alternative basée sur les approches analytiques sera viable si des estimateurs précis sont mis en œuvre pour un nombre important de types d'algorithme de traitement du signal.

Dans la dernière partie, les différentes méthodologies de codage en virgule fixe existantes dans le cas des processeurs de traitement du signal et leurs limites ont été présentées. La conception d'une nouvelle méthodologie nécessite tout d'abord de définir la localisation du processus de codage des données au sein du cycle de développement d'une application de traitement numérique du signal. Pour cela, nous avons analysé le couplage entre le processus de codage des données et de génération de code. Celui-ci est nécessaire à l'obtention d'une code de qualité en termes de précision et de temps d'exécution. Cette analyse permet de définir la stratégie de codage des données et son couplage avec le processus de génération de code. De plus, nous avons analysé l'influence de l'architecture sur la précision des calculs et montré la nécessité de prendre en compte l'architecture lors du codage des données afin de tirer partie des différents éléments de celle-ci.

Chapitre 2

Définition de la méthodologie

L'objectif de ce chapitre est de décrire la démarche suivie pour définir une nouvelle méthodologie d'implantation d'algorithmes spécifiés en virgule flottante au sein d'architectures programmables en virgule fixe sous contrainte de respect des critères de qualité associés à l'application. Cette méthodologie optimise l'implantation de l'application tant que la précision des calculs réalisés en virgule fixe permet de satisfaire les critères de qualité spécifiés.

Les méthodologies existantes présentées dans le chapitre précédent réalisent une transformation de la représentation des données en virgule flottante en une représentation en virgule fixe au niveau du code source sans prendre en considération l'architecture du processeur. Cependant, l'analyse de l'influence de l'architecture sur la précision des calculs et des différentes phases de la génération de code montre la nécessité de tenir compte de l'architecture et de coupler le processus de codage et de génération de code pour obtenir une implantation de qualité en termes de précision et de temps d'exécution.

Dans la première partie de ce chapitre, nous analysons l'influence de l'architecture sur la précision des calculs. Dans un premier temps, l'architecture des processeurs de traitement du signal est présentée et les différents éléments influençant la précision des calculs sont détaillés. Ensuite, les résultats des expérimentations réalisées pour analyser l'influence de l'architecture sur la précision des calculs sont exposés. Cette analyse montre la nécessité de tenir compte de l'architecture cible pour obtenir une implantation optimisée d'un point de vue du temps d'exécution et de la précision.

Dans la seconde partie, nous analysons l'interaction entre les différentes phases de compilation et le processus de codage des données. Dans un premier temps, les techniques de compilation sont exposées et les différentes phases de la génération de code sont détaillées. Ensuite, les résultats de l'analyse de l'interaction entre les phases de compilation et de codage des données sont présentés et illustrés à travers différentes expérimentations. Cette analyse permet de définir les objectifs de certaines phases du processus de codage et le couplage de ces phases avec le processus de génération de code.

Au sein de la troisième partie, la synthèse des différents éléments présentés dans les deux premières parties de ce chapitre est réalisée afin de définir la structure de la méthodologie de codage des données en virgule fixe et les objectifs des différentes phases de cette méthodologie.

2.1 Architecture des processeurs de traitement du signal

2.1.1 Introduction

Dans cette partie, nous analysons les caractéristiques des différentes unités des processeurs de traitement numérique du signal (DSP). Dans un premier temps, nous présentons les différents éléments de l'unité de traitement (chemin de données) et les structures utilisées pour effectuer

des calculs intensifs. Nous détaillons plus particulièrement les différents aspects concernant le traitement de la précision finie dans les DSP. Ensuite, les architectures des unités de mémoire et de contrôles sont décrites. Finalement, l'influence de l'architecture sur la précision des calculs est étudiée afin de déterminer le degré de prise en compte de l'architecture dans le processus de codage des données.

Les solutions architecturales

Dans ce paragraphe, les différentes solutions architecturales pour l'implantation des algorithmes de traitement numérique du signal (TNS) sont présentées. Pour chaque solution, les caractéristiques en termes de coût, de consommation d'énergie, de performances, de flexibilité et de temps de développement sont détaillées. De plus, les conséquences en termes de stratégie de codage des données en virgule fixe sont exposées. Pour illustrer cette analyse, certains des critères énumérés ci-dessus sont représentés à la figure 2.1.

Processeurs à usage général : L'objectif des processeurs à usage général est de pouvoir implanter une panoplie importante d'algorithmes. En conséquence, leur architecture n'est pas spécialisée pour un domaine particulier tel que celui du traitement numérique du signal (TNS). L'architecture homogène du processeur conduit à une consommation d'énergie et un coût élevés. Ainsi, les performances en termes d'efficacité énergétique et de ratio performance coût sont relativement faibles pour les algorithmes de TNS. Dans le cas des processeurs à usage général destinés aux applications embarquées, des efforts sont réalisés afin de diminuer la consommation du processeur. De plus, face à l'accroissement du nombre d'algorithmes de TNS au sein des nouvelles applications, de nombreux processeurs à usage général intègrent des extensions permettant d'accélérer les traitements associés au domaine du TNS. Cependant, les performances en termes d'efficacité énergétique et de ratio performance coût restent inférieures à celles obtenues avec les processeurs de traitement du signal.

L'architecture homogène de ce type de processeur permet d'obtenir des compilateurs de langage de haut niveau efficaces. L'obtention d'un code assembleur de qualité à partir d'une description en langage de haut niveau permet d'éviter de longues phases de développement de code en assembleur. Ainsi, les temps de développement de ces applications peuvent être relativement réduits.

Circuits dédiés à une application : Les circuits dédiés (ASIC[1]) sont conçus et optimisés pour une application précise. Cette spécialisation du circuit à l'application cible permet d'obtenir la solution fournissant les meilleures performances en termes de temps de calcul, d'efficacité énergétique et de ratio coût performance. En contrepartie, ces circuits offrent une flexibilité très réduite, voire nulle. La complexité des différentes phases de conception, de validation, de test et de réalisation de ces circuits conduit à des coûts et des temps de développement élevés. De plus, cette tendance s'est aggravée avec l'évolution des technologies utilisées au niveau du processus de fabrication des circuits intégrés. Pour réduire les coûts et les temps de développement de ces circuits, le concept de réutilisation de blocs existants s'est généralisé. Le concepteur réalise l'assemblage de différents blocs IP (Intellectual Properties) déjà testés et validés.

La spécialisation de l'architecture à l'application est réalisée en partie par le dimensionnement des différents opérateurs. Ainsi, le codage des données en virgule fixe doit minimiser la surface du circuit tant que la contrainte de précision souhaitée en sortie de l'algorithme est respectée.

[1] Application Specific Integrated Circuit

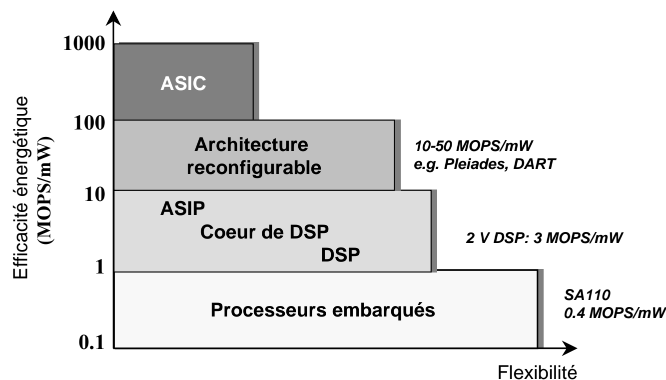


FIG. 2.1: Caractéristiques des solutions architecturales en termes d'efficacité énergétique et de flexibilité [124]

Processeurs de traitement du signal : Nous regroupons sous l'appellation processeurs de traitement du signal, les processeurs DSP disponibles sous forme de composant, les cœurs de DSP et les ASIP[2] dédiés aux applications de TNS.

Les processeurs DSP et les cœurs de DSP sont conçus pour implanter efficacement les algorithmes spécifiques au domaine du traitement du signal. Leur architecture est programmable et regroupe les différents éléments nécessaires aux opérations de TNS. Deux catégories de processeurs DSP sont présentes sur le marché. La première catégorie correspond aux processeurs DSP conçus afin d'obtenir une efficacité énergétique élevée et un coût relativement faible. Ces caractéristiques sont obtenues en spécialisant l'architecture du processeur. En contrepartie, les compilateurs de langage de haut niveau ne permettent pas de fournir un code assembleur de qualité par rapport à un code développé et optimisé à la main. Ainsi, les temps de développement des applications sont relativement importants en raison du développement manuel en assembleur des parties critiques du code. La seconde catégorie des processeurs DSP correspond aux processeurs conçus afin d'obtenir des performances élevées en termes de capacité de calcul. Pour cela, des architectures homogènes et intégrant de nombreuses unités fonctionnelles pouvant travailler en parallèle sont mises en œuvre. Les compilateurs de langage de haut niveau permettent d'obtenir un code de qualité si ceux-ci exploitent efficacement le parallélisme offert par l'architecture. En contrepartie, l'efficacité énergétique et le prix sont plus élevés par rapport aux processeurs DSP spécialisés. Des solutions réalisant un compromis entre les deux approches sont également proposées.

Les ASIP sont des processeurs programmables dédiés à une classe d'applications composée de quelques algorithmes. Les ASIP représentent un intermédiaire entre les processeurs DSP et les ASIC [93]. Deux familles d'ASIP correspondant à deux approches de conception différentes peuvent être définies. La première famille correspond à des processeurs proches des ASIC. Ils ont été développés afin d'obtenir une flexibilité plus importante par rapport aux ASIC à travers la possibilité de programmer le circuit. Ces processeurs vont permettre, pour les quelques applications cibles, d'obtenir des performances se rapprochant de celles des ASIC en termes d'efficacité énergétique et de ratio coût performance. La seconde famille d'ASIP correspond aux processeurs conçus à partir d'un cœur de DSP. Dans ce cas, le cœur est paramétré en fonction des applications cibles et certains accélérateurs sont intégrés afin d'obtenir de meilleures performances. La spécialisation de l'architecture à la classe d'algorithmes visés permet d'obtenir des performances en termes d'efficacité énergétique, supérieures à celles des processeurs DSP. Comme pour les processeurs et les cœurs de DSP, l'efficacité des compilateurs de langage de haut niveau dépend de la spécialisation du processeur et de l'adéquation entre les différentes phases de la génération de code et de l'architecture.

[2]Application Specific Instruction-set Processor

Dans le cadre des DSP, l'architecture étant figée, le processus de codage des données en virgule fixe doit permettre d'obtenir le meilleur compromis entre la précision des calculs et différents paramètres tels que le temps d'exécution, la taille du code ou la consommation d'énergie. De plus, pour les cœurs de DSP ou les ASIP dont la largeur du chemin de données est paramétrable, la méthodologie doit explorer les différentes solutions possibles.

Architectures reconfigurables : Afin d'obtenir des performances élevées et un niveau de flexibilité relativement important, différents types d'architectures reconfigurables ont été proposés.

La reconfiguration au niveau logique correspond à celle présente au niveau des composants programmables de type FPGA. Cette reconfiguration est réalisée au niveau des éléments logiques de l'architecture et de leurs interconnexions. Les configurations appliquées à ce type d'architecture nécessitent un volume d'information très important, ainsi elles sont maintenues pendant un nombre de cycles d'horloge élevé et elles ne sont pas modifiées au cours du traitement réalisé. La reconfiguration au niveau logique permet d'adapter les opérateurs et leurs interconnexions à l'application ciblée. Ainsi, les largeurs des différents opérateurs peuvent être optimisées en fonction de la contrainte de précision. Cependant, cette flexibilité au niveau des opérateurs est obtenue au détriment des performances en termes de temps d'exécution et de consommation d'énergie de ces opérateurs. En effet, l'utilisation de cellules standards ne permet pas d'optimiser les opérateurs arithmétiques par rapport à une solution *full-custom*. Ainsi, ce type d'architecture est préconisé pour les algorithmes manipulant des données dont la largeur est relativement faible. Les méthodologies proposées [25] pour le codage des données en virgule fixe sont très proches de celles utilisées pour les ASIC.

La reconfiguration au niveau fonctionnel permet de modifier le flot des données au sein de l'architecture. Les interconnexions entre les ressources et la fonctionnalité de ces ressources peuvent être reconfigurées. Ce type de reconfiguration permet d'adapter le modèle de programmation et les motifs de calcul à l'application ciblée. Si le volume d'information nécessaire à la reconfiguration de l'architecture n'est pas trop important, alors la reconfiguration peut être réalisée dynamiquement au cours de l'exécution du code.

Comme pour les DSP, la largeur des opérateurs est fixe et ne peut prendre que quelques valeurs prédéfinies. Ainsi, les méthodologies de codage des données en virgule fixe pour ce type d'architecture sont relativement proches de celles utilisées pour les DSP. La différence majeure réside dans le critère à optimiser au niveau de l'architecture pour obtenir une implantation de qualité.

Système sur une puce SoC : Pour répondre aux défis de l'implantation des systèmes complexes tels que les nouveaux algorithmes destinés aux télécommunications de troisième génération, des systèmes intégrant sur une même puce différentes solutions hétérogènes sont mis en œuvre. Ces systèmes peuvent regrouper les différents éléments présentés ci-dessus. L'objectif est d'obtenir une plate-forme relativement flexible et offrant des performances élevées à travers l'intégration d'accélérateurs adaptés aux parties critiques de l'application. La flexibilité est présente à travers les parties programmables ou reconfigurables.

La méthodologie de codage des données dans le cadre d'un système sur puce nécessite de mettre en œuvre une méthodologie globale permettant d'optimiser plus ou moins en parallèle le codage des données pour les différentes solutions architecturales.

Domaine d'application des processeurs de traitement du signal

Le marché des DSP a connu une forte augmentation en termes de volume de production à la fin des années 90 et représentait en 2001 un marché de 4,8 milliards de $ [129]. Cette forte croissance était liée à l'explosion du principal marché des DSP que représente le domaine des communications sans fil cellulaires. Ces processeurs sont aussi présents dans le domaine des modems et du contrôle moteur pour les disques durs. Malgré une stagnation des ventes au cours de ces deux dernières années, la société Forward Concepts, prévoit une croissance des ventes de DSP de 27% par an dans les cinq années à venir [129]. Cette prévision est liée à la reprise de la croissance au niveau mondial, du marché des communications sans fil avec les systèmes de troisième génération et le GPRS et à l'émergence de nouveaux marchés dans le domaine du multimédia et d'internet (voix sur IP [130], décodeurs MP3 [128], caméras numériques). Les ASIP sont utilisés pour des applications nécessitant des performances supérieures aux DSP en termes de puissance de calcul, de consommation d'énergie ou de coût. Ils sont présents dans le domaine du multimédia (télévision numérique, compression audio et vidéo et graphisme 3D) et des télécommunications (communications sans fil et par satellites, ATM) [120].

2.1.2 Architecture de l'unité de traitement

Dans cette partie, nous présentons l'architecture des unités de traitement des processeurs de traitement du signal. Dans une première partie, la représentation des données au sein des DSP est présentée. Ensuite, les différentes architectures des unités de traitement (U.T.) sont exposées et les différents éléments constituant celles-ci sont détaillés.

Représentation des données

Processeurs virgule fixe et virgule flottante : Les processeurs de traitement du signal sont divisés en deux grandes catégories en fonction du type d'arithmétique utilisé pour réaliser les traitements mathématiques. Les DSP virgule fixe et virgule flottante possèdent des caractéristiques et des propriétés propres qui les orientent vers des marchés différents. Différents critères tels que le coût, la consommation d'énergie, le temps de développement des applications sont analysés dans cette partie pour ces deux catégories de processeurs DSP.

Le coût et la consommation d'énergie des DSP virgule fixe sont plus faibles que ceux des DSP virgule flottante, car la taille des données qu'ils manipulent est plus faible et l'unité de traitement est moins complexe. Les figures 1.8 et 1.9 situées à la page 17, montrent la dynamique et la précision obtenues pour différentes largeurs de données dans le cas de l'utilisation d'un codage en virgule fixe et en virgule flottante. Lorsque la taille des données est relativement faible (16 bits) les caractéristiques en termes de précision et de dynamique des deux codages sont proches. La représentation des données en virgule flottante nécessite l'utilisation de deux parties : la mantisse et l'exposant. Ainsi, lorsque le nombre de bits utilisés est relativement faible, un compromis entre la dynamique et la précision doit être effectué et peu de bits peuvent être utilisés pour l'exposant. En conséquence, la dynamique obtenue n'est pas supérieure à celle des DSP virgule fixe. Ainsi, pour des largeurs de données relativement faibles, les DSP sont exclusivement réalisés avec un codage en virgule fixe. Le codage en virgule flottante ne montre de l'intérêt que pour des largeurs de données plus élevées (32 bits), permettant alors d'obtenir d'excellentes performances en termes de précision et de dynamique de codage.

Le prix d'un processeur est proportionnel à sa surface et la majorité de celle-ci est occupée par les bus et la mémoire. La largeur des données manipulées par les processeurs virgule fixe étant plus faible que celle des processeurs virgule flottante, le coût du circuit est moins important. De même, une grande partie de la consommation en énergie d'un circuit est liée au transfert des

données et des instructions entre la mémoire et les unités de traitement et de contrôle et cette consommation est proportionnelle à la largeur des éléments transférés. Dans les DSP virgule fixe, la largeur des données étant plus faible, la consommation en énergie est moins importante.

L'unité de calcul des processeurs virgule flottante est plus complexe car elle doit réaliser les opérations telles que l'alignement des données qui est à la charge du programmeur dans le cas de l'arithmétique virgule fixe. Ceci augmente le coût, la consommation et le temps de latence de ce type d'unité de traitement. Les expérimentations réalisées dans [141] montrent que le temps de latence d'un opérateur de type additionneur en virgule flottante est 2,3 fois supérieur à celui d'un opérateur en virgule fixe et que la surface occupée est 4 fois supérieure. La société Texas Instruments propose une architecture identique déclinée en une famille de processeurs en virgule fixe (TMS320C62x) et une famille de processeurs en virgule flottante (TMS320C67x). Les fréquences de fonctionnement des processeurs en virgule flottante sont nettement plus faibles que celles des processeurs en virgule fixe. Ainsi, l'utilisation de l'arithmétique virgule fixe permet d'obtenir de meilleures performances en termes de capacité de calcul.

Lors du développement d'applications pour les processeurs en virgule fixe, le concepteur doit réaliser le codage et le cadrage des données présentes au sein de l'algorithme. Il analyse la dynamique de l'ensemble des données, en déduit le codage et détermine les décalages nécessaires pour aligner ces données lors des opérations d'addition ou de soustraction. Le codage et le cadrage des données doivent être optimisés afin de garantir l'absence de débordement au sein de l'algorithme tout en maintenant la précision maximale.

Les processeurs en virgule flottante proposent une dynamique élevée qui permet de libérer l'utilisateur des tâches de codage des données et d'analyse des débordements éventuels. L'alignement des données est automatiquement réalisé par le matériel, au sein de l'unité de calcul. Ainsi, le temps de développement d'applications pour ce type de processeur est nettement plus faible par rapport à celui des processeurs virgule fixe.

Les DSP virgule fixe sont bien adaptés pour les applications destinées au grand public ou le coût de revient des composants constituant le système doit être le plus faible possible. Le surcoût du développement lié à la spécificité du codage en virgule fixe est rapidement amorti par les grands volumes de production. La faible consommation de ce type de composant permet de l'utiliser au sein des applications embarquées. Ces DSP sont très utilisés dans les secteurs tels que les télécommunications et le multimédia. Les processeurs en virgule flottante sont destinés aux applications nécessitant une dynamique importante ou une précision élevée pour lesquelles un processeur en virgule fixe ne peut répondre à ces exigences. Ces applications sont par exemple présentes dans le domaine de l'audionumérique. Les processeurs en virgule flottante sont utilisés pour des applications réalisées en petite série ou le coût du développement représente une part très importante du coût total du produit. Le marché des processeurs DSP est dominé par les processeurs en virgule fixe (95% des DSP en 1996) et plus particulièrement par les processeurs dont la largeur des données est de 16 bits (95% des DSP virgule fixe en 1996) [75].

Largeur naturelle du processeur : La largeur naturelle (b_{nat}) des données correspond à la largeur des données qui sont manipulées le plus efficacement par le bus et le chemin de données du processeur [76]. La majorité des opérations sur ces données est réalisée en un cycle.

La largeur naturelle des données est un paramètre fondamental au niveau du DSP car elle influence directement la précision des calculs et le prix de celui-ci en fixant la largeur minimale des opérateurs, des bus de données et de la mémoire. Afin d'adapter au mieux le DSP à l'application souhaitée, certains fabricants proposent des cœurs de DSP pouvant être synthétisés et dont la largeur naturelle est paramétrable. Ainsi, le concepteur va choisir la largeur minimale répondant aux critères de précision souhaités. Nous avons regroupé dans le tableau 2.1 les lar-

Fournisseur	Cœur	Largeurs naturelles (bits)
DSP Group	PalmCore [116]	16, 20, 24
DSP Group	CedarDSPCore	16, 24, 32
Philips	REAL [60]	16, 24
Philips	EPICS [108]	12,16,18,20,24
Clarkspur	DSP CD2450 [11]	16 à 24

TAB. 2.1: Largeur naturelle des données de quelques cœurs de DSP

geurs naturelles de différents cœurs. Pour le cœur SP5 de la société 3DSP, deux configurations de multiplieur sont proposées (16×24 ou 32×32). De même, le cœur de processeur paramétrable Xtensa [39], permet d'ajouter des extensions afin de réaliser les opérations de traitement du signal et différentes configurations de multiplieur sont disponibles [134]. Dans le cas de la conception des ASIP, il est nécessaire de définir la largeur naturelle du processeur. Pour les ASIP conçus à partir d'un cœur de DSP, la largeur naturelle de l'ASIP est fonction du choix du cœur de DSP. Ainsi, l'utilisation d'un cœur de DSP avec une largeur paramétrable nécessite de sélectionner la largeur naturelle la plus faible permettant de respecter les contraintes de précision sur la classe d'algorithmes à implanter. Pour les ASIP spécialisés proches des ASIC, la largeur du chemin de données est optimisée en fonction de la précision souhaitée au niveau des quelques algorithmes implantés avec ce processeur. Ainsi, pour la conception de ce processeur, les techniques d'optimisation de la largeur du chemin de données sous contrainte de précision pourront être utilisées.

Extension de la précision : Les DSP permettent d'utiliser au sein de l'unité de traitement une largeur de données étendue. La largeur des bus d'interconnexion entre les différentes unités fonctionnelles du chemin de données est supérieure à la largeur naturelle du processeur afin de conserver l'ensemble des bits issus des opérateurs. Ceci permet de maintenir une précision plus élevée sur une suite d'opérations dont les opérandes restent dans l'unité de traitement et ne sont pas renvoyés en mémoire. L'efficacité de cette méthode est fortement liée au nombre d'unités de stockage (registres) disponibles au sein du chemin de données. Les architectures DSP sont composées d'au moins un multiplieur et d'un additionneur. Ainsi, pour conserver une précision suffisante les largeurs de la sortie du multiplieur, de l'entrée et de la sortie de l'additionneur sont supérieures ou égales à $2.b_{nat}$.

L'arithmétique multi-précision permet de manipuler des données de largeur plus importante, obtenues par concaténation de plusieurs données de largeur naturelle. Une donnée x en double précision pour un processeur de largeur naturelle b_{nat} est représentée à la figure 2.2.a. Cette donnée de largeur $2.b_{nat}$ est composée d'une donnée signée x_H et d'une donnée non signée x_L dont les largeurs sont égales à b_{nat}.

Une opération utilisant des données en multi-précision est décomposée en une suite d'opérations manipulant des opérandes de largeur naturelle. Les figures 2.2.b et 2.2.c représentent les différentes opérations à réaliser pour effectuer respectivement une multiplication et une addition. Afin de pouvoir implanter ces opérations en double précision le processeur doit posséder des opérateurs capables de travailler sur des données signées et non signées et permettre de conserver la retenue d'une opération afin de l'injecter dans la suivante.

Dans le cadre d'une opération de type MAC (multiplication-accumulation) sur deux données en double précision, la multiplication nécessite quatre opérations de multiplication et trois opérations d'addition classiques et l'addition requiert deux additions classiques. Étant donné

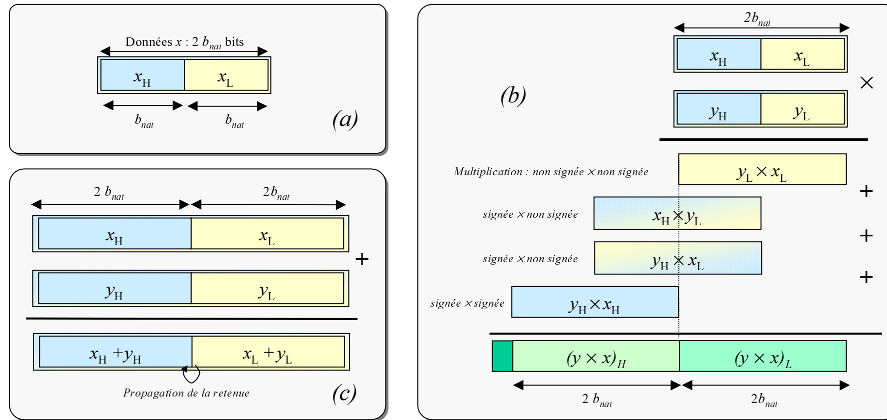


FIG. 2.2: Opérations arithmétiques double précision

le nombre d'opérations à réaliser pour le calcul d'une opération de base en double précision, le temps d'exécution de celle-ci est fortement augmenté par rapport à une arithmétique simple précision. Ceci limite l'utilisation de ce type d'opération à des portions de code critiques en termes de précision.

Exploitation du parallélisme au niveau des données : Afin de diminuer le temps d'exécution des opérations, certains DSP permettent l'exploitation du parallélisme au niveau des données. Ils intègrent des instructions SWP (*Sub-Word Parallelism*)[3] permettant de traiter en parallèle des données dont la largeur est inférieure à la largeur naturelle. Cette technique divise un opérateur manipulant des données de largeur N afin de pouvoir exécuter en parallèle k opérations sur des fractions de mot de largeur N/k [37] [8]. Ce concept est illustré à la figure 2.3 pour les opérations de multiplication et d'addition/soustraction. Pour un additionneur de largeur N, la réalisation de k additions en parallèle nécessite de ne pas propager la retenue entre deux fractions de mot. Le temps d'exécution d'une opération est diminué d'un facteur k, mais en contrepartie la précision des données est nettement plus faible. Cette technique peut être utilisée pour les opérations de multiplication, d'addition, de soustraction et de décalage.

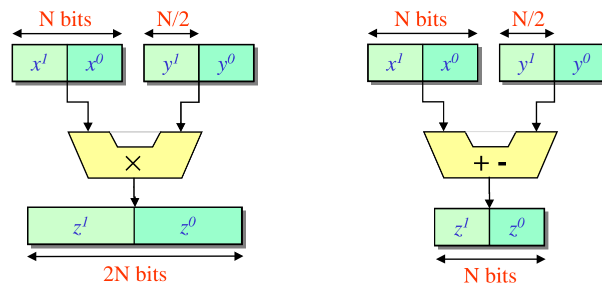


FIG. 2.3: Opérations arithmétiques exploitant le parallélisme au niveau des données

L'utilisation des instructions *SWP* nécessite de ranger les données en mémoire en fonction de l'ordre de traitement de celles-ci et de les aligner correctement les unes par rapport aux autres. Ainsi, les processeurs proposent des instructions de concaténation et d'extraction de fractions de mot. Les premières permettent de concaténer plusieurs fractions de mot de largeur N/k au sein d'un mot de largeur N et les secondes réalisent l'extraction d'une fraction de mot située au sein d'un mot de largeur plus importante. L'efficacité d'une solution utilisant les opérations SWP

[3] Le terme SIMD est aussi employé pour dénommer cette technique

Processeur	$N_{o/u}$	Opérations Multiplication b_{e_1}	b_{e_2}	b_s	Addition b_e	b_s	Décalage b_e	b_s
TMS320C64x	1	16	16	32	32/40	32/40	32/40	32/40
	1	16	32	32/64				
(T.I.)	2	16	16	32	16	16	16	16
[138]	4	8	8	16	8	8		
TigerSHARC	1				64	64	64	64
(A.D.)	2	32	32	32/64	32	32	32	32
[154] [45]	4	16	16	16/32	16	16	16	16
	8				8	8	8	8
SP5	1	24	16	40	32/48	32/48	32	32
(3DSP)	2	16	16	n.a.	16	16	16	16
[1]	4	8	16	n.a.	8	8	-	-
OneDSP*	1	32	32	32	32	32	32	32
(Siroyan)	2	16	16	16	16	16	16	16

*instructions manipulant des données codées en virgule fixe

TAB. 2.2: Présentation des capacités SWP de certains processeurs DSP

réside dans la limitation du surcoût lié aux instructions de concaténation ou d'extraction. En effet, ce surcoût ne doit pas annihiler le gain obtenu par la réalisation d'opérations en parallèle. Afin de pouvoir utiliser des données non-alignées, le DSP TigerSHARC [37] possède une unité permettant d'aligner des données présentes dans deux mots différents.

Cette technique a été initialement utilisée au sein des processeurs destinés aux applications de multimédia (compression vidéo, traitement d'image). En effet, ces dernières possèdent un fort parallélisme au niveau des données. Ces applications mettent en œuvre des traitements répétés pour chaque pixel de l'image et ne nécessitant pas une précision trop importante. Depuis quelques années cette technique est exploitée dans le cadre de l'implantation des applications de TNS dans les DSP. Dans [32], les auteurs ont montré l'apport de cette technique pour l'implantation dans le processeur TigerSHARC, d'une boucle de synchronisation du code d'un récepteur CDMA. Le code obtenu permet de réaliser en moyenne 6,6 MAC/cycle à l'aide de deux unités de traitement de type MAC (multiplication-accumulation). Cette technique est de plus en plus utilisée dans les nouveaux processeurs DSP destinés aux applications nécessitant des capacités de calcul importantes. Ainsi, les derniers processeurs ou cœurs de processeur tels que ceux présentés dans le tableau 2.2, conçus pour les systèmes de télécommunications de troisième génération intègrent des capacités SWP. Nous avons résumé dans le tableau 2.2, les capacités SWP des processeurs DSP proposant cette technique pour les opérations de multiplication, d'addition-soustraction et de décalage. Pour chaque opération nous avons spécifié la largeur des opérandes d'entrée (b_e) et de sortie (b_s). Certains processeurs ne proposent des capacités SWP que pour l'additionneur [14, 89, 136]. En effet, les modifications à apporter à l'opérateur sont simples et ceci permet d'accélérer le traitement ACS (Addition, Comparaison, Sélection) utilisé dans le cadre de l'algorithme de Viterbi.

Les processeurs proposant des capacités SWP sont amenés à manipuler divers types (largeurs) de données. Pour illustrer ceci nous avons résumé dans le tableau 2.3 les différentes largeurs de données traitées par ces DSP. En comparaison, les DSP classiques manipulent seulement deux à trois largeurs de données différentes.

Processeur	Types manipulés (bits)
TMS320C64x (T.I.)	8, 16, 32, 40, 64
TigerSHARC (A.D.)	8, 16, 32, 64
SP5 (3DSP)	8, 16, 24, 32, 48
OneDSP (Siroyan)	8, 16, 32, 44, 88
ManArray (BOPS)	8, 16, 32, 40, 64, 80

Tab. 2.3: Types des données manipulées par certains processeurs DSP

Architecture des chemins de données

Les premières générations de DSP possédaient une architecture du chemin de données basée sur une structure de type MAC [75]. Pour faire face à la demande croissante en puissance de calcul nécessaire pour les nouvelles applications de télécommunications et de multimédia, les concepteurs de DSP ont proposé deux types d'architecture [36]. La première est une extension de l'architecture MAC (multiplication-accumulation) traditionnelle et la seconde est une architecture orthogonale s'inspirant des processeurs RISC et basée sur la mise en parallèle d'unités fonctionnelles au sein du processeur.

Architecture traditionnelle (structure MAC) : Les premières générations de DSP ont été conçues afin d'implanter efficacement les opérations de type MAC correspondant à la structure de base de nombreux algorithmes tels que les filtres (FIR, IIR). La structure d'un filtre FIR est représentée à la figure 2.4, celui-ci est constitué de cellules élémentaires juxtaposées réalisant une opération de multiplication accumulation : le résultat en sortie de la cellule est égal à la somme du résultat précédent et celui de la multiplication du coefficient h_k et de l'opérande x_{n-k} issu de la ligne à retard. Les DSP possèdent une architecture permettant de calculer la sortie d'une cellule élémentaire par cycle.

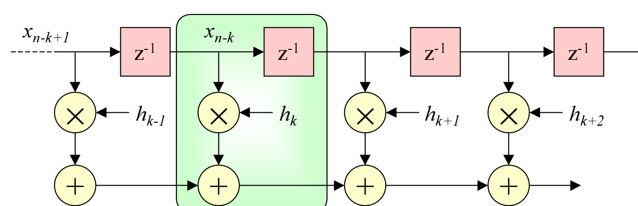


Fig. 2.4: Structure du filtre FIR

Le schéma d'une architecture traditionnelle de type MAC est présenté à la figure 2.5.a, les opérandes sont soit situés dans les registres Rx et Ry, soit placés directement en mémoire.

Deux variantes de la structure MAC sont utilisées dans les DSP. La première réalise les opérations de multiplication accumulation de façon pipelinée, le multiplieur réalise la multiplication des opérandes de la $k^{ème}$ cellule et stocke le résultat dans un registre intermédiaire (P), en parallèle l'accumulateur réalise l'accumulation associée à la cellule précédente. Cette structure est présente dans les DSP tels que le TMS320C5x [135], DSP16xx [88].

Dans la seconde variante, l'opération de multiplication est d'abord réalisée et le résultat est transmis à l'accumulateur afin d'effectuer l'addition. Ces deux opérations étant réalisées de

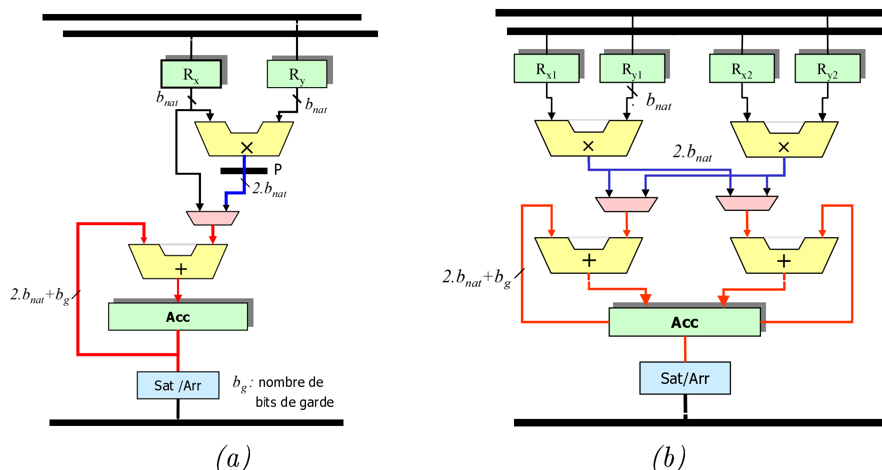


FIG. 2.5: Architecture traditionnelle de type MAC *(a)* et double-MAC *(b)*

manière séquentielle au cours d'un même cycle, les temps d'exécution du multiplieur et de l'accumulateur doivent être plus faibles par rapport à la solution précédente.

Les premières générations de DSP possédaient peu d'unités fonctionnelles, ainsi l'unité arithmétique et logique (UAL) du DSP était utilisée pour réaliser l'opération d'accumulation au sein du MAC. Les générations actuelles de DSP intégrent plus d'unités en parallèle et permettent d'utiliser un additionneur/soustracteur pour l'unité MAC, indépendant de l'UAL. De plus, ces DSP incorporent des unités de manipulation de bits incluant un registre en barillet.

Architecture traditionnelle étendue (structure multi-MAC) : Pour suivre l'augmentation de la complexité des algorithmes de TNS, la nouvelle génération de DSP doit fournir une puissance de calcul plus élevée. Afin de ne pas augmenter excessivement les fréquences d'horloge, des architectures utilisant plusieurs unités MAC en parallèle sont proposées. Le synoptique d'une architecture double MAC telle que celle du processeur DSP16xxx [14] ou du cœur de DSP PalmDSPCore [116] [121] est présenté à la figure 2.5.b.

Un additionneur/soustracteur à trois entrées est intégré au sein de l'unité de traitement afin de réaliser l'accumulation du résultat précédent et celui des deux multiplications. Cette structure permet d'évaluer deux étages d'un filtre FIR au cours d'un même cycle. Afin d'avoir une certaine souplesse au niveau de l'utilisation de cette architecture, les sorties des multiplieurs peuvent être utilisées par les deux UAL.

Architecture orthogonale : Les structures de DSP présentées précédemment, possèdent une architecture hétérogène destinée à optimiser les connexions entre les unités fonctionnelles et les opérateurs afin de réduire le coût et la consommation d'énergie du processeur. Cette spécialisation de l'architecture est l'une des principales causes de l'inefficacité des compilateurs de langages de haut niveau associés à ce type de processeurs [159, 160]. A l'opposé, les architectures orthogonales possèdent une structure homogène. Ces architectures sont composées d'une file de registres et d'unités fonctionnelles travaillant en parallèle ou n'importe lequel des registres est connecté à n'importe laquelle des unités fonctionnelles. Les processeurs de type VLIW (Very Long Instruction Word) ou super-scalaires sont basés sur ce type d'architecture orthogonale.

Le but de cette architecture est de mieux tirer profit du parallélisme inhérent aux algorithmes de traitement numérique du signal. L'architecture est composée de multiples unités fonctionnelles indépendantes pouvant fonctionner en parallèle. Le TMS320C62x [142] est un très bon exemple de ce type d'architecture, il est composé de deux chemins de données indépendants. Chaque chemin intègre une file de 16 registres, un multiplieur et trois UAL. Le schéma du cœur de traitement est représenté à la figure 2.6. La présence de nombreuses unités fonctionnelles pouvant travailler en parallèle nécessite un nombre important de registres au sein de la file de registres. Cet accroissement du nombre de registres entraîne une augmentation du temps d'accès aux registres, de la taille des instructions et rend la phase de décodage plus complexe. Pour diminuer ces effets, les unités fonctionnelles sont regroupées au sein de *clusters*. Chaque *cluster* est composé d'une file de registres et de plusieurs unités fonctionnelles. De plus, des mécanismes permettant les transferts de données entre les *clusters* sont présents. Nous pouvons considérer que le TMS320C62x présenté à la figure 2.6 est composé de deux *clusters*.

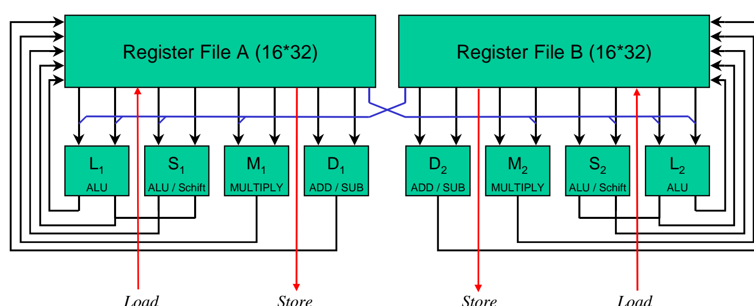


FIG. 2.6: Architecture du processeur DSP TMS320C62x [137]

Les différents éléments du chemin de données

Dans cette partie, les différents éléments constituant l'unité de traitement des processeurs de traitement du signal sont détaillés. Ces éléments correspondent aux opérateurs réalisant les traitements en virgule fixe et aux registres pour le stockage des données intermédiaires.

Registres : Avant de présenter les structures de registres, utilisées dans les unités de traitement, nous définissons les termes jeu de registres et classe de registres. Un jeu de registres correspond à l'ensemble des registres présents dans le chemin de données et destinés à stocker les opérandes et les résultats intermédiaires. Une classe de registres correspond à un sous-ensemble de registres du jeu de registres possédant les mêmes fonctionnalités.

Dans les processeurs, deux types de structure du jeu de registres sont présents. La première consiste à utiliser un jeu de registres homogène dont tous les registres le composant sont interchangeables. Cette technique nécessite d'utiliser un banc de registres intégrant plusieurs ports de communication permettant d'accéder en lecture et en écriture aux différents registres en parallèle. Si un opérateur utilise deux opérandes issus d'une file de registres alors cette dernière doit posséder 5 ports (2 pour la lecture des opérandes par l'opérateur, 2 pour l'écriture des opérandes issus de la mémoire et 1 pour l'écriture du résultat). Cette solution est très coûteuse en termes de surface de silicium, mais garantit l'homogénéité de la structure de registres. Le DSP TMS320C62xx [137] est composé de deux files de 16 registres de 32 bits. Les deux files de registres possèdent 9 ports de lecture (32 bits) et 6 ports d'écriture (32 bits). Cette approche est utilisée dans les processeurs à usage général de type RISC, les DSP virgule flottante et les nouveaux DSP virgule fixe.

La seconde approche consiste à spécialiser les registres et à les relier directement aux unités fonctionnelles, afin de réduire le nombre et la complexité des connexions. Une classe de registres est utilisée pour chaque opérande et le résultat de l'opération est stocké dans une troisième classe de registres. Cette spécialisation de la structure des registres permet de diminuer les coûts et d'augmenter la rapidité des transferts entre les différentes unités. Le jeu d'instructions associé à cette architecture est spécialisé car chaque instruction utilisant un opérateur, spécifie précisément le ou les registres qui peuvent être utilisés. Les DSP de première et de seconde génération [75] utilisent ce type d'approche afin d'optimiser le coût. Ces DSP possèdent de nombreuses classes de registres (2 à 8) composées de peu de registres (1 à 8). La classe de registres principale correspond aux registres d'accumulation qui reçoivent les résultats intermédiaires issus des différents calculs réalisés dans l'unité de traitement. Le nombre de registres d'accumulation influence la qualité du code et la souplesse de programmation du processeur. La présence d'un faible nombre de registres d'accumulation nécessite de renvoyer les résultats intermédiaires en mémoire. Ceci ralentit l'exécution du programme, augmente la taille du code et peut diminuer la précision du calcul.

Multiplieur : Les DSP possèdent un multiplieur câblé permettant de réaliser une multiplication par cycle. Le temps de latence entre la lecture des données et la disponibilité des résultats de certains multiplieurs étant supérieur à 1 cycle (TMS320C62x, ...), l'obtention d'une multiplication par cycle est réalisée à l'aide du pipeline interne.

La majorité des DSP possède un multiplieur dont la largeur des deux opérandes d'entrée est identique et dont la largeur de l'opérande de sortie est égale au double de celle des entrées. De plus, ces multiplieurs permettent d'utiliser des données signées et non signées afin de supporter l'arithmétique multi-précision. Cependant, la largeur des deux opérandes d'entrée de certains multiplieurs peut être différente. Les DSP TMS320C64x et SP5 proposent des instructions de multiplication non symétriques référencées dans le tableau 2.2 de la page 41. Afin de fournir un résultat sans perte d'information, la largeur de l'opérande de sortie d'un multiplieur est égale à la somme des largeurs des opérandes d'entrée de celui-ci. Cependant, certains DSP utilisent un multiplieur ne mettant à disposition que la partie la plus significative du résultat. Dans ce cas, la sortie du multiplieur est biaisée et un bruit de calcul est généré. Cette technique peut être utilisée pour différentes raisons. Elle permet de limiter la largeur du chemin de données du processeur. Le DSP Zilog Z893xx possède une largeur naturelle de 16 bits et un multiplieur dont la taille du résultat est de 24 bits [81]. Cette technique est mise en œuvre pour obtenir un résultat dont la largeur est compatible avec les autres opérations. Dans le cas du TMS 320C64x, la multiplication d'une donnée de largeur 16 bits avec une donnée de largeur 32 bits fournit un résultat sur 48 bits. Ainsi, celui-ci est soit stocké sur 64 bits, soit sur 32 bits. Certains DSP tels que le TigerSHARC (A.D.)[154, 45] et le SPXK5 [71] de la société NEC proposent de diminuer la largeur du résultat de la multiplication afin de faciliter les calculs suivants. La largeur du résultat de la multiplication est réduite à celle des entrées. Pour réduire l'influence du bruit généré par l'élimination de ces bits, une loi de quantification par arrondi est mise en œuvre au sein de ces deux processeurs.

Lorsque nous sommes en arithmétique virgule fixe, la multiplication entraîne le doublement du bit de signe. Certains DSP permettent d'éliminer automatiquement le bit redondant en réalisant un décalage à gauche d'un bit. Si les données d'entrée sont codées en virgule fixe cadrée à gauche, cette technique permet d'avoir un format du résultat identique au format en entrée.

Unité Arithmétique et Logique (UAL) : L'unité arithmétique et logique permet de réaliser les opérations logiques (AND, OR, NOT) et les opérations arithmétiques (addition, soustrac-

tion, incrémentation, décrémentation, valeur absolue) en un 1 cycle. Certaines UAL permettent de réaliser une division en plusieurs cycles. Les UAL sont conçues pour travailler sur des données dont la largeur est identique à celle des registres d'accumulation ou de la file de registres mais certains DSP tels que le ADSP21xx [7] possèdent une UAL ne pouvant manipuler que des données de largeur naturelle. Dans ce cas, une opération sur un registre d'accumulation nécessite plusieurs cycles.

Les architectures des DSP ont été conçues pour réaliser efficacement les opérations de type MAC (*multiplication accumulation*). La multiplication de deux opérandes de largeur b fournit un résultat dont la largeur est égale au double de b. Les accumulations successives des opérandes issus du multiplieur augmentent la dynamique du résultat intermédiaire et nécessitent des bits supplémentaires pour le coder. Dans certains DSP, les additionneurs proposent b_g bits de garde qui permettent de réaliser une suite de plusieurs accumulations sans provoquer de débordement. La largeur de ce type d'additionneur est égale à $2.b_{nat}+b_g$ bits. Ces bits de garde vont permettre de stocker le résultat de 2^{b_g} additions sans nécessiter de recadrer les données. Les DSP utilisant cette technique proposent 4, 8 ou 12 bits de garde permettant de réaliser respectivement au moins 16, 256 ou 4096 additions sans recadrage.

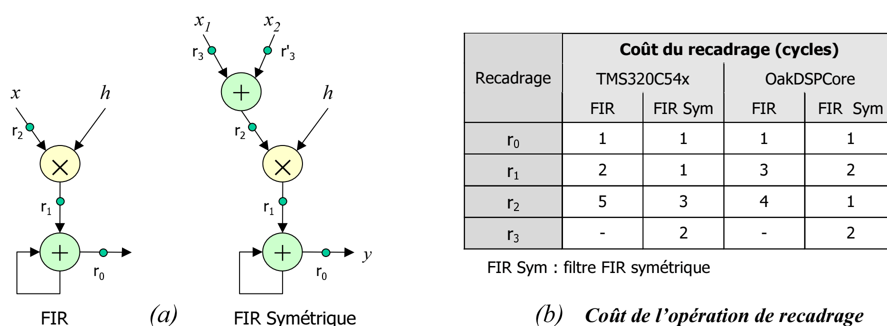
Registres à décalage : Le codage des données en virgule fixe nécessite de réaliser de nombreux décalages. Ces décalages permettent d'adapter le format d'une donnée à sa dynamique, d'aligner la position de la virgule des entrées d'un additionneur ou d'insérer des bits supplémentaires au niveau d'une donnée afin d'éviter des débordements lors des opérations suivantes. Nous trouvons au sein des unités de traitement des DSP, deux types de registres à décalage. La première catégorie correspond aux registres à décalage spécialisés qui ne peuvent réaliser que quelques décalages prédéfinis. Ces décalages sont réalisés en parallèle à l'exécution des autres opérations et ne nécessitent pas de cycle d'horloge supplémentaire. Ces registres sont affectés à la sortie ou à l'entrée d'un opérateur particulier afin de fournir une plus grande flexibilité pour le cadrage des données. La valeur du décalage à réaliser est spécifiée au sein d'un registre de mode ou directement au sein de l'instruction. L'utilisation de bits de mode permet de réduire la taille de l'instruction, mais nécessite d'initialiser le registre de mode avant de réaliser un décalage. Ce type de registres est par exemple présent en sortie des multiplieurs afin de pouvoir éliminer le second bit de signe présent avec une arithmétique fractionnaire et dans certains cas, pour modifier la position de la virgule afin d'éviter les débordements lors des accumulations réalisées par la suite.

Pour illustrer les différents registres à décalage spécialisés, les capacités de décalage du DSP TMS320C50 sont détaillées. L'architecture de l'unité de traitement de ce processeur est présentée à la figure 1.13.a de la page 30. Un registre à décalage spécialisé est présent en sortie du multiplieur. Il permet de réaliser un décalage à droite de 6 bits ou un décalage à gauche de 1 ou 4 bits. Ce registre est commandé à l'aide d'un registre de mode. Un registre à décalage nommé *postscaler* est présent en sortie du registre d'accumulation. Il permet de réaliser un décalage à gauche de 1 à 7 bits lors du renvoi de la donnée en mémoire. Un registre à décalage nommé *prescaler* est présent en entrée de l'accumulateur. Il permet de réaliser un décalage de 1 à 16 bits à gauche, lors du transfert des données de la mémoire vers l'accumulateur. De plus, ce *prescaler* peut être utilisé pour réaliser, à l'aide d'une instruction spécifique, un décalage de 1 à 16 bits à droite au niveau de la donnée située dans le registre d'accumulation. Dans ce dernier cas, la fonctionnalité de ce registre à décalage est proche de celle des registres à décalage en barillet présentés ci-dessous. Pour ces deux registres à décalage la valeur du décalage est spécifiée au sein de l'instruction utilisée pour réaliser le décalage. De plus, deux instructions spécifiques permettent de réaliser un décalage à gauche ou à droite de 1 bit de la donnée située dans le registre d'accumulation.

La seconde catégorie correspond aux registres en barillet, ils possèdent la capacité de réaliser l'ensemble des décalages à droite et à gauche possibles sur une donnée avec un temps d'exécution qui ne dépasse pas un cycle. Ils sont intégrés au sein d'une UAL ou constituent une unité spécifique en parallèle. Ce type de registres apporte une très grande souplesse dans le cadrage des données et permet de réaliser un décalage quelconque en un cycle d'instruction. L'absence de ce type de registres dans un DSP est un handicap important pour l'obtention d'un code optimisé. Le DSP56000 [111] ne possède que des registres permettant de réaliser le décalage de 1 bit à droite ou à gauche par cycle. Dans ce cas, la minimisation du nombre de décalages est un facteur primordial dans l'obtention d'un code efficace sachant qu'un décalage de N bits nécessite N cycles.

Le coût réel d'une opération de décalage utilisant un registre en barillet est fonction de l'architecture du processeur. Pour les processeurs basés sur une architecture MAC traditionnelle, le coût du recadrage dépend de la localisation de l'opérande source au sein de l'unité de traitement. Les registres en barillet permettent de réaliser un décalage sur des données stockées au sein des registres d'accumulation. La structure des registres n'étant pas homogène, si la donnée est stockée dans une autre classe de registres des opérations de transfert entre la classe de registres et les registres d'accumulation sont nécessaires pour faire l'opération de recadrage. Dans ce cas, le temps nécessaire au recadrage d'une donnée est supérieur à 1 cycle. De plus, la présence de l'opération de décalage modifie le motif de calcul et peut éliminer la possibilité d'utiliser des instructions complexes permettant de réaliser une suite d'opérations en un cycle.

Pour illustrer et quantifier ces phénomènes, nous avons implanté un filtre à réponse impulsionnelle finie (FIR) classique et un filtre FIR symétrique au sein des DSP TMS320C54x [136] et OakDSPCore [144]. Les graphes flot de données du cœur de traitement de chaque application sont représentés à la figure 2.7.a et sont annotés avec la position des différents recadrages testés. Nous avons regroupé dans le tableau présenté à la figure 2.7.b pour chaque application le temps d'exécution des différents recadrages en fonction de leur localisation. Pour évaluer le temps d'exécution de l'opération de recadrage nous avons mesuré les temps d'exécution t_{ar} et t_{sr} correspondant respectivement au temps d'exécution de l'application avec et sans l'opération de recadrage. Le temps d'exécution de l'opération de recadrage correspond à la différence entre les temps t_{ar} et t_{sr}. L'utilisation d'un registre à décalage en barillet conduit à des temps d'exécution de l'opération de recadrage indépendants de la valeur du décalage à réaliser.



Recadrage	Coût du recadrage (cycles)			
	TMS320C54x		OakDSPCore	
	FIR	FIR Sym	FIR	FIR Sym
r_0	1	1	1	1
r_1	2	1	3	2
r_2	5	3	4	1
r_3	-	2	-	2

FIR Sym : filtre FIR symétrique

(b) Coût de l'opération de recadrage

FIG. 2.7: Coût des différentes opérations de recadrage au sein des filtres FIR et FIR symétrique

Pour l'opération de recadrage r_0 située en sortie de l'additionneur, l'opérande étant stocké dans un des registres d'accumulation, le coût de l'opération de recadrage est de 1 cycle pour les deux DSP.

Le coût du recadrage en sortie de la multiplication (r_1) est égal à 1 cycle pour le filtre FIR symétrique implanté dans le DSP C54x. Le résultat de la multiplication étant stocké dans un des registres d'accumulation, le coût de l'opération de recadrage correspond au temps d'exécution de

l'instruction de décalage. Dans le cas du DSP Oak, le résultat de la multiplication étant stocké dans le registre P situé en sortie du multiplieur, un cycle supplémentaire correspondant au transfert du registre P vers un des registres d'accumulation, est nécessaire. Dans le cas du filtre FIR classique, l'opération de multiplication-accumulation (MAC) est réalisée en un seul cycle en l'absence de recadrage en sortie du multiplieur. La présence de l'opération de recadrage ne permet pas d'utiliser l'instruction complexe réalisant l'opération MAC en un cycle et nécessite d'utiliser une opération de multiplication, de décalage puis d'accumulation. Ainsi, le coût de ce recadrage est de 2 cycles pour le C54x et de 3 cycles pour le Oak.

Pour le filtre FIR classique, l'opération de recadrage (r_2) de l'entrée x du multiplieur nécessite de charger l'accumulateur avec la donnée x, de réaliser le décalage et de transférer la donnée recadrée vers le registre d'entrée du multiplieur. Ainsi, le coût de ce décalage est de 4 cycles pour le Oak et de 5 cycles pour le C54x car une opération NOP[4] est insérée entre le transfert de la donnée recadrée et la multiplication afin d'éviter un conflit au niveau du pipeline [136]. Pour le filtre symétrique, la donnée à recadrer étant située dans un registre d'accumulation, le coût du recadrage est de 1 cycle pour le Oak. Cependant, pour le C54X, le coût du recadrage est de 3 cycles. La donnée située dans le registre d'accumulation est décalée puis transférée dans le registre d'entrée du multiplieur. Comme pour le cas précédent, une opération NOP est insérée. La solution de référence sans recadrage réalise l'addition puis la multiplication directement sans réaliser de transfert entre le registre d'accumulation et le registre d'entrée du multiplieur.

Pour l'opération de recadrage r_3, 2 cycles sont nécessaires car la présence de cette opération ne permet plus d'utiliser l'opération d'addition entre une donnée située en mémoire et un registre d'accumulation. Ainsi, un transfert entre la mémoire et un registre, nécessitant un cycle, est nécessaire avant de réaliser l'opération de recadrage.

Pour les architectures orthogonales, les différentes données sont stockées au sein de la file de registres. Ainsi, aucune opération de transfert entre les registres n'est nécessaire pour réaliser l'opération de recadrage. Cependant, le coût réel de l'opération de décalage dépend de l'opportunité de mettre cette opération de décalage en parallèle avec d'autres instructions. Ce phénomène est analysé plus en détail dans la partie 2.2.4 de la page 70.

Implantation de la loi de dépassement : Les débordements peuvent se produire lors de différentes opérations arithmétiques au sein de l'unité de traitement ou lors d'un changement de format. Les architectures des DSP permettent d'utiliser une loi de dépassement modulaire ou de saturation. L'approche la plus simple en cas de débordement est de ne conserver que les N bits disponibles pour représenter la donnée et d'éliminer les bits les plus significatifs supplémentaires. Cette loi de dépassement modulaire possède des propriétés concernant l'erreur de codage peu intéressantes, ainsi de nombreux DSP possèdent des unités permettant d'implanter une loi de saturation. Elle permet d'obtenir une erreur de codage plus faible et de conserver le signe de la donnée à coder. Les unités de saturation détectent la présence d'un débordement et substituent la donnée par celle représentant la valeur minimale ou maximale la plus proche.

Ce type d'unités est présent en sortie des accumulateurs afin de saturer la valeur de ceux-ci avant le transfert de leur contenu en mémoire. Certains DSP [137] [14] [109] possèdent en sortie de l'additionneur ou de l'UAL une unité de saturation permettant de gérer les débordements lors des calculs intermédiaires. Une unité de saturation est présente au sein de quelques processeurs [14] [136] pour saturer le résultat de la multiplication. Dans le code complément à 2, lorsque les deux opérandes sont égaux à la valeur minimale représentable, le résultat de la multiplication ne fait pas partie du domaine de définition du code si le second bit de signe a été automatiquement éliminé. Dans ce cas, la sortie du multiplieur est saturée à la valeur maximale représentable.

[4]No Operation

L'utilisation des unités de saturation est spécifiée au sein de l'instruction [137] ou à l'aide de bits de mode stockés au sein de registres de contrôle [14] [88]. Dans ces deux cas, la saturation des données ne nécessite pas de cycle supplémentaire. Certains DSP proposent des instructions réalisant explicitement la saturation de l'accumulateur et nécessitant un cycle d'horloge [136] [7]. Cependant, l'utilisation d'unités de saturation modifie les valeurs des données et introduit des non-linéarités au sein du signal. Ainsi, la solution la plus efficace pour gérer les problèmes de débordement est de les éviter en réalisant un cadrage correct de l'ensemble des données.

Implantation de la loi de quantification : Le processus de quantification intervient dès qu'une donnée est transférée vers un registre ou un emplacement dont la largeur est inférieure à celle d'origine. De nombreux DSP permettent de choisir entre une loi de quantification par arrondi ou par troncature. La troncature consiste à éliminer les bits de poids faible qui ne peuvent être conservés dans le nouveau format. Cette quantification peut être facilement implantée car elle consiste à sélectionner les bits les plus significatifs et à les affecter à la nouvelle donnée. En contrepartie ce procédé entraîne un biais entre la valeur réelle et celle codée.

Afin d'obtenir une moyenne de l'erreur de quantification nulle, il est possible d'utiliser une loi de quantification par arrondi, celle-ci consiste à coder la donnée avec la valeur la plus proche appartenant au nouveau format. Nous trouvons au sein des DSP deux types d'arrondi, les arrondis conventionnels et convergents. Soit Δ, le pas de quantification associé au codage après quantification. Dans le cas d'une quantification par arrondi conventionnel, lorsque la donnée est égale à la valeur médiane de l'intervalle $[k\Delta, (k+1)\Delta]$, la valeur représentable la plus proche est choisie de la manière suivante :

$$Q(x) = \begin{cases} k\Delta & \text{si } k\Delta \leq x < (k+\frac{1}{2})\Delta \\ (k+1)\Delta & \text{si } (k+\frac{1}{2})\Delta < x \leq (k+1)\Delta \\ (k+1)\Delta & \text{si } x = (k+\frac{1}{2})\Delta \end{cases} \tag{2.1}$$

La valeur médiane de chaque intervalle $[k\Delta, (k+1)\Delta]$ étant codée systématiquement à la valeur supérieure, l'erreur de quantification n'est pas nulle. Cette erreur est néanmoins nettement plus faible que celle présente lors d'une quantification par troncature. Pour réaliser un arrondi sur le $k^{ème}$ bit de la donnée nous additionnons 2^{k-1} à la valeur à arrondir et nous réalisons ensuite une troncature au niveau du $k^{ème}$ bit. Certains processeurs [7] [111] [136] possèdent une instruction ou un mode qui supporte l'arrondi conventionnel et qui réalise automatiquement les opérations décrites ci-dessus. Sinon, il faut explicitement charger la valeur 2^{k-1} dans l'accumulateur avant de débuter les calculs. Ce type d'arrondi est le plus utilisé car son implantation est plus simple que celle de l'arrondi convergent présenté ci-dessous.

Afin d'éliminer le biais présent dans le cas de l'arrondi conventionnel, un arrondi convergent peut être utilisé [76]. Il affecte la valeur médiane de façon équiprobable à la valeur supérieure et inférieure :

$$Q(x) = \begin{cases} k\Delta & \text{si } k\Delta \leq x < (k+\frac{1}{2})\Delta \\ (k+1)\Delta & \text{si } (k+\frac{1}{2})\Delta < x \leq (k+1)\Delta \\ (k)\Delta \text{ ou } (k+1)\Delta & \text{si } x = (k+\frac{1}{2})\Delta \end{cases} \tag{2.2}$$

$$\text{avec } \mathrm{P}\Big(Q(x) = k\Delta \setminus x = (k+\frac{1}{2})\Delta\Big) = \mathrm{P}\Big(Q(x) = (k+1)\Delta \setminus x = (k+\frac{1}{2})\Delta\Big)$$

Cette technique permet de coder la valeur médiane de manière équiprobable à l'aide de l'analyse de la parité de la donnée. Les données impaires sont codées à la valeur supérieure et les données paires à la valeur inférieure. Pour réaliser ceci, le DSP DSP56000 [111] teste le $k^{ème}$ bit et ne réalise l'addition de cette donnée avec 2^{k-1} que si le bit est égal à 1. Pour sa part, le

DSP ADSP21xx [7] réalise d'abord l'addition et ensuite met à 0 le $k^{ème}$ bit. Ce type d'arrondi permet d'obtenir une moyenne de l'erreur de quantification nulle, mais il n'est présent que dans quelques DSP [89] [111] [7], car son implantation est plus complexe.

2.1.3 Architectures des unités de mémoire et de contrôle

Architecture de l'unité mémoire

Les applications de TNS requièrent la lecture et l'écriture de nombreuses données stockées en mémoire. En conséquence, il est nécessaire de posséder une structure de mémoire efficace permettant d'avoir une bande passante de communication entre les unités de calcul et la mémoire élevée afin de ne pas ralentir l'exécution des instructions. Les facteurs importants pour obtenir une architecture efficace sont l'organisation de la mémoire et l'interconnexion entre la mémoire et le chemin de données.

L'exemple le plus souvent utilisé pour montrer l'importance de l'architecture de la mémoire dans un processeur de traitement du signal est le filtre FIR présenté à la figure 2.4 de la page 42. Celui-ci est constitué de cellules de type MAC et d'une ligne à retards. Cette opération MAC étant réalisée en 1 cycle dans la majorité des DSP, l'architecture mémoire doit permettre d'effectuer l'ensemble des communications nécessaires entre l'unité de calcul et la mémoire en 1 cycle, afin de ne pas ralentir le traitement des données. Les différents accès à la mémoire sont la recherche de l'instruction, la lecture de la donnée x_{n-k}, la lecture du coefficient h_k et le "vieillissement" des données $x_{n-k-1} = x_{n-k}$. Afin d'éliminer le dernier accès, les processeurs utilisent un mode d'adressage particulier qui permet de "vieillir" automatiquement les données. Cette technique permet de réduire le nombre d'accès par cycle à 3.

L'architecture traditionnelle des processeurs à usage général est basée sur le modèle de Von Neuman pour lequel une mémoire unique est utilisée pour stocker les données et le programme. Pour obtenir des performances acceptables ces processeurs utilisent des mémoires caches. L'architecture des DSP est basée sur le modèle Harvard dont le principe est la séparation de la mémoire pour les données et le programme. Les deux espaces mémoire sont totalement indépendants, ils possèdent leurs propres bus de données et d'adresses et leur propre générateur d'adresses. Différentes variantes du modèle Harvard ont été proposées afin d'augmenter le nombre d'échanges entre la mémoire et les unités de calcul. Le choix des diverses variantes est fonction de la structure de localisation des opérandes [9] utilisée dans le processeur.

Modèles de localisation des opérandes : Les architectures [135] basées sur le modèle *registre-mémoire*, utilisent au sein d'une opération, un opérande présent en mémoire et un opérande stocké dans un registre [9]. Au cours d'un cycle, une seule lecture d'opérande en mémoire est réalisée, ainsi il est possible d'utiliser la structure Harvard de base possédant une seule mémoire pour les données [77]. Dans ce cas, le temps d'exécution d'une instruction est fonction du temps d'accès à la mémoire et du temps d'exécution de l'opération.

Afin de s'affranchir du temps d'accès à la mémoire une structure évitant la lecture directe d'un opérande en mémoire est utilisée. Pour ce modèle de type *load-store*, les opérandes sont stockés dans des registres et le chargement de ceux-ci est effectué en parallèle avec l'exécution d'une opération dans l'unité de calcul. Afin de respecter les contraintes de dépendances des données, l'opération courante s'effectue en même temps que le chargement des opérandes de l'instruction suivante. Ce modèle de localisation des opérandes nécessite pour un opérateur de lire deux données par cycle. Deux approches sont utilisées afin de réaliser ces deux lectures.

La première approche consiste à utiliser une mémoire multi-ports permettant de lire plusieurs données en parallèle au cours d'un même cycle. Le processeur possède une mémoire programme mono-port et une mémoire données double ports permettant de lire deux données en parallèle. L'aspect multi-ports de la mémoire va augmenter la surface et le coût du circuit. Cette structure est très utilisée au sein des architectures orthogonales telles que celle du TMS320C62x [137] [142]. L'association de ce modèle mémoire et d'une file de registres permet de garantir l'homogénéité de l'architecture.

La seconde approche utilise deux bancs de mémoires possédant chacun ses propres bus de données et d'adresses et sa propre unité de génération d'adresses. Cette structure permet de lire deux données en parallèle au cours d'un cycle sans utiliser de mémoires multi-ports. Cette solution est moins coûteuse en termes de surface de silicium par rapport à des mémoires multi-ports. Mais ce type d'architecture ne permet de lire qu'une seule donnée par banc de mémoire et par cycle, ainsi il est nécessaire de répartir efficacement les données dans les deux bancs de mémoires afin de pouvoir charger le maximum de données en parallèle.

Les nouveaux processeurs possèdent plusieurs unités fonctionnelles pouvant travailler en parallèle, ainsi il est nécessaire d'augmenter la bande passante de la mémoire afin de pouvoir alimenter l'ensemble de ces unités en opérandes et pour écrire les résultats en mémoire en un seul cycle. La première technique utilise le parallélisme au niveau des données en exploitant la régularité de la structure des données de nombreux algorithmes de TNS. Les DSP récents utilisent des bus de données dont la largeur est supérieure à la largeur naturelle des données, afin de pouvoir lire en parallèle plusieurs données adjacentes en mémoire [14] [142]. L'efficacité de cette technique est liée à l'organisation des données en mémoire. Cette augmentation de la largeur des bus de données permet de renvoyer en mémoire, en un cycle, des variables intermédiaires sans diminuer leur précision. La seconde approche consiste à utiliser plusieurs bus de données pour adresser la mémoire. Cette technique est plus coûteuse car elle nécessite d'associer un générateur d'adresses à chaque bus de données et d'utiliser des mémoires multi-ports.

Mode d'adressage : Afin de respecter les performances souhaitées au niveau de la bande passante de communication avec la mémoire, il est nécessaire d'avoir des unités spécialisées (unité de génération d'adresses) permettant de calculer les adresses des données rapidement et en parallèle avec l'exécution des autres opérations. En effet, la charge de calcul concernant les adresses est proche de celle des données [121].

Les algorithmes de TNS sont caractérisés par des accès réguliers et séquentiels à la mémoire. Les unités de génération d'adresses vont exploiter ces propriétés afin d'optimiser le calcul des adresses en proposant des solutions câblées permettant d'implanter les modes d'accès les plus couramment utilisés. Les DSP permettent d'utiliser de nombreux modes d'adressage tels que l'adressage immédiat, direct ou indirecte. Ce dernier est le plus adapté aux applications de TNS. Pour ce mode d'adressage, la donnée est pointée en mémoire via un registre d'adresse. Ceci permet de spécifier dans l'instruction le registre à utiliser et non l'adresse de la donnée et ainsi de réduire fortement le nombre de bits nécessaires pour encoder la localisation de la donnée. De plus, ce mode d'adressage permet de calculer l'adresse de la donnée suivante en parallèle à l'exécution de l'instruction courante. Ce type d'adressage correspond à l'adressage naturel des tableaux. Les unités de génération d'adresses proposent des opérations sur les registres d'adresse permettant de réaliser des adressages spécifiques tels que l'adressage linéaire, indexé, circulaire ou bit inversé.

Architecture de l'unité de contrôle

Pipeline : L'efficacité d'un DSP réside dans sa capacité à réaliser une instruction par cycle. Chaque instruction étant composée d'une séquence d'opérations, il est nécessaire d'exécuter ces opérations en parallèle. La technique du pipeline permet de décomposer les différentes phases d'une instruction et de les réaliser en parallèle [78]. Les principales phases du pipeline sont la recherche et le décodage de l'instruction, la lecture des opérandes, l'exécution de l'instruction et l'écriture du résultat. La profondeur du pipeline (nombre d'étages) est un compromis entre la facilité de programmer et de contrôler celui-ci et la rapidité du processeur. Les processeurs DSP traditionnels utilisent un pipeline dont la longueur varie entre 3 et 5 étages. Pour les processeurs de type VLIW, la longueur du pipeline est plus importante, ceci permet de simplifier chaque opération à réaliser dans celui-ci et ainsi d'utiliser des horloges de fréquence plus élevée, le TMS3320C62x possède un pipeline de 11 étages et permet d'utiliser une fréquence d'horloge de 300 MHz [137]. La mise en parallèle des différentes phases d'une instruction peut entraîner des aléas de ressources, de données ou de contrôle. La gestion de ces aléas dépend du type de codage utilisé au niveau des instructions [41, 78]. L'étude de ces aspects est important pour pouvoir estimer correctement le temps d'exécution d'une suite d'instructions.

Type de codage des instructions : Dans cette partie, nous présentons les deux types de codage d'instructions [41] [40] utilisés pour programmer les processeurs utilisant un pipeline. De plus, nous présentons les deux catégories de processeurs liées aux types de codage et les mécanismes associés pour gérer les aléas de données et de ressources. Le pipeline au niveau des données est présenté à la figure 2.8 et un exemple décrivant une opération de multiplication-accumulation au sein de l'architecture de la figure 2.5.a, est proposé.

Le premier type de codage correspond au codage des instructions stationnaires dans le temps. Chaque instruction du jeu d'instructions définit l'ensemble des opérations à réaliser pour chaque unité fonctionnelle, au cours du cycle [78]. Dans le cas de l'opération MAC, trois actions sont à réaliser, la première correspond à l'opération à exécuter et les deux dernières au chargement des deux registres avec les opérandes présents en mémoire. L'instruction spécifie l'ensemble des opérations à réaliser sur les données présentes dans les différentes parties du chemin de données. Le programmeur ou le compilateur doit assurer l'alimentation du pipeline et est responsable de la gestion des aléas liés à la dépendance des données et aux conflits de ressources, ceux-ci sont analysés à partir d'une table de réservation qui permet de visualiser l'utilisation des ressources et des données.

Le second type correspond au codage des instructions stationnaire au niveau des données. L'instruction spécifie une séquence complète d'opérations à réaliser sur un ensemble de données. Les différentes opérations nécessitent plusieurs cycles, ainsi le résultat n'est pas immédiatement disponible. Ce type de codage permet d'obtenir un code plus lisible et facilite la programmation. L'utilisateur ne doit pas spécifier l'ensemble des actions à réaliser au cours d'un cycle mais les opérations à effectuer sur les données. Cette approche est plus proche de l'algorithme à coder.

Les processeurs utilisant un codage d'instructions stationnaire dans le temps et réalisant l'ensemble des opérations spécifiées dans l'instruction en un cycle, sont appelés processeurs microcodés. L'ordonnancement des différentes tâches au sein du pipeline est réalisé par l'utilisateur ou le compilateur et est ainsi visible au niveau du programme assembleur.

Les processeurs dont une partie des instructions nécessite plusieurs cycles sont appelés processeurs macrocodés. Ainsi, les processeurs utilisant un code stationnaire au niveau des données font partie de cette catégorie. L'ordonnancement des opérations dans le pipeline est à la charge du contrôleur de pipeline, ceci permet de faciliter la programmation du processeur mais des

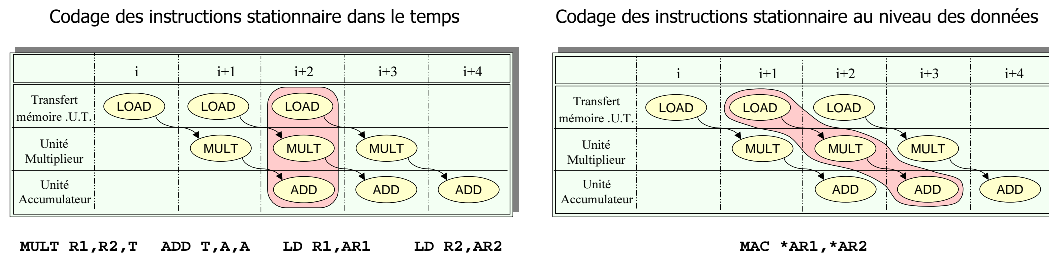


FIG. 2.8: Représentation du pipeline pour le codage des instructions stationnaire dans le temps et stationnaire au niveau des données

aléas de données et de contrôle peuvent se produire au sein de ce pipeline. En fonction du type de codage utilisé par le processeur, ces aléas peuvent être résolus soit de manière statique en modifiant le code assembleur ou dynamiquement en utilisant une méthode de verrouillage du pipeline. Dans ce dernier cas, le contrôleur de pipeline retarde la progression de la seconde instruction qui a provoqué le conflit. Cette méthode augmente le nombre de cycles pour exécuter la séquence d'instructions et accentue les difficultés pour estimer le temps d'exécution a priori de la séquence.

Format du jeu d'instructions : Pour satisfaire les contraintes liées aux applications embarquées, deux catégories de format d'instructions sont disponibles [41] [40]. Le choix entre ces deux catégories est fonction du domaine d'application du processeur. Pour le premier type de codage, l'objectif est de minimiser le coût et la consommation du processeur. En effet, le challenge pour les processeurs embarqués est de minimiser la taille du circuit afin de réduire les coûts, tout en maintenant des performances élevées en termes de puissance de calcul. Ainsi, le nombre d'instructions pour réaliser une opération et la largeur des instructions doivent être minimisés afin de diminuer la taille de la mémoire programme [121]. Pour obtenir des performances élevées, l'instruction doit encoder le maximum de parallélisme. Le second facteur important dans les applications embarquées est la consommation du circuit. Elle est proportionnelle à la taille des bus et de la mémoire. Ainsi, la minimisation de la taille des instructions va permettre de diminuer la consommation liée à la mémoire programme.

Le choix du format des instructions résulte d'un compromis entre le parallélisme offert par l'instruction et sa taille. Elle doit être compacte afin de minimiser la taille de la mémoire programme et elle doit offrir le maximum de parallélisme pour utiliser efficacement les ressources du processeur. Cette minimisation du nombre de bits est réalisée au travers de différentes restrictions au niveau du jeu d'instructions [76]. Ces restrictions vont sacrifier une partie du parallélisme disponible et rendre la structure du jeu d'instructions hétérogène à travers la spécialisation des registres et des modes d'adressage disponibles. Ce type de codage est utilisé pour les processeurs destinés aux applications pour le grand public et en particulier dans le domaine des télécommunications ou les facteurs prix et consommation sont primordiaux.

L'alternative au codage présenté ci-dessus correspond à l'utilisation d'un format de codage des instructions orthogonal tel que dans le cas des processeurs VLIW[5]. Pour les processeurs VLIW, l'instruction globale est composée de plusieurs champs de contrôle de taille fixe et indépendants les uns des autres [35]. Chaque champ représente une instruction élémentaire contrôlant une unité d'exécution du chemin de données. L'ordonnancement de ces instructions élémentaires est à la charge du programmeur ou du compilateur. Cette technique augmente nettement la taille de l'instruction globale par rapport à une instruction encodée. Une instruction globale peut ne

[5]Very Long Instruction Word

pas utiliser l'ensemble des unités fonctionnelles, ainsi des techniques sont mises en œuvre afin de regrouper les instructions élémentaires par paquets et d'adapter la longueur de l'instruction globale au nombre d'instructions élémentaires employées.

Ce type de format permet d'obtenir un fort parallélisme au niveau instruction en autorisant l'utilisation de l'ensemble des unités fonctionnelles en parallèle. En conséquence, le jeu d'instructions est plus régulier. Les instructions élémentaires possèdent peu de restrictions au niveau de l'utilisation des registres et des modes d'adressage. Les concepteurs utilisent une taille des instructions élémentaires proche de celle des instructions encodées (de 16 à 32 bits), mais à la différence de ces dernières, l'instruction élémentaire ne doit pas encoder le parallélisme au sein de l'instruction, ainsi plus de bits sont disponibles pour coder la localisation des opérandes source et destination. L'absence de restriction va permettre d'obtenir un jeu d'instructions plus homogène et ainsi de réaliser des compilateurs plus efficaces.

Les inconvénients majeurs de ce type de processeurs sont l'augmentation de la consommation d'énergie et de la taille de la mémoire programme. Comme nous l'avons décrit précédemment les instructions VLIW possèdent moins de restrictions sur l'utilisation des registres et des modes d'adressage. La largeur des instructions élémentaires est plus grande et la taille de la mémoire programme nécessaire plus élevée. La consommation des processeurs de type VLIW est supérieure à celle des DSP traditionnels. La taille de l'instruction étant plus élevée, le nombre de transferts de bits entre la mémoire et l'unité de traitement est plus important. De plus, la présence d'un nombre plus important d'unités d'exécution dans le chemin de données va augmenter la consommation de celui-ci.

Afin de diminuer la taille de la mémoire et la consommation par rapport aux processeurs de type VLIW, des DSP utilisant une approche mixte sont proposés [107]. Ces processeurs possèdent un chemin de données orthogonal permettant d'utiliser plusieurs unités fonctionnelles en parallèle. Le jeu d'instructions est composé d'instructions élémentaires courtes et d'instructions orthogonales longues de type VLIW. Les processeurs Carmel (Infineon)[34] et REAL (Philips) [59] [108] mettent en œuvre cette technique.

Ce type de format de codage orthogonal est destiné aux processeurs nécessitant l'utilisation de plusieurs unités fonctionnelles en parallèle afin d'atteindre des performances en termes de calcul élevées. Nous trouvons ce type d'architecture dans les processeurs DSP à usage général destinés aux calculs intensifs tel que le StareCore [153] ou le TMS320C62xx [137] et pour les processeurs de traitement d'images ou les performances doivent être élevées afin de respecter les contraintes temps réel imposées par ce type d'applications.

Structures de contrôle : La structure répétitive représente l'une des structures de contrôle les plus couramment utilisées dans les algorithmes de TNS. Dans la plupart des cas, les boucles permettent d'implanter un calcul numérique répétitif sur les éléments d'un vecteur, ainsi le nombre d'itérations est connu a priori. Le nombre d'instructions présentes dans le corps de la boucle pouvant être relativement faible, il est indispensable d'implanter efficacement ce type de structure et de minimiser le surcoût lié au contrôle de celle-ci. La plupart des DSP possède une unité de contrôle appelée boucle matérielle qui permet de coder la structure de boucle en une seule instruction et de ne pas utiliser de cycle supplémentaire pour réaliser le test de la valeur du compteur d'itérations, le branchement conditionnel et la décrémentation de ce compteur. Certains DSP proposent plusieurs supports de boucle matérielle afin de pouvoir imbriquer plusieurs boucles.

Les DSP possèdent des registres d'état ou de mode afin d'implanter un contrôle à l'aide de ce registre et non au sein de l'instruction. Ces registres de mode sont utilisés pour contrôler les parties du chemin de données qui varient peu au cours d'une séquence d'instructions. Ces registres sont initialisés ou modifiés à l'aide d'instructions spécifiques, avant l'utilisation de

l'unité fonctionnelle.

2.1.4 Analyse de l'influence de l'architecture sur la précision des calculs

Dans cette partie, les résultats de l'analyse de l'influence de l'architecture sur la précision des calculs sont exposés. L'influence de l'architecture a été étudiée à travers la comparaison du Rapport Signal à Bruit de Quantification (RSBQ) en sortie de différents algorithmes de TNS. Les applications retenues pour ces expérimentations correspondent aux filtres à réponse impulsionnelle finie (FIR) et à réponse impulsionnelle infinie (IIR) et à la transformée de Fourier rapide (FFT). Dans un premier temps, la démarche globale suivie pour coder les données est exposée. Ensuite, pour chaque application les différentes sources de bruit de quantification présentes au sein du système sont décrites. Les résultats de notre étude sont présentés sous deux angles différents. Nous avons détaillé l'influence de chaque paramètre de l'architecture et nous fournissons les résultats obtenus pour différentes implantations au sein de DSP. Les résultats de cette analyse ont été publiés dans [103, 100].

Les données codées en virgule fixe sont composées d'une partie entière et d'une partie fractionnaire. Le nombre de bits nécessaires pour coder la partie entière est défini à partir de la connaissance de la dynamique de la donnée, permettant ainsi de garantir l'absence de débordement. La dynamique des différentes données a été déterminée à partir de la norme l_1 [119]. Ces dynamiques étant variées, il est nécessaire de recadrer certaines données au sein de l'algorithme afin de conserver une précision maximale. Les recadrages vont permettre d'adapter le format d'une donnée en sortie d'une opération à sa dynamique, d'aligner la position de la virgule des entrées d'un additionneur ou d'insérer des bits supplémentaires au niveau de la partie entière d'une donnée afin de garantir l'absence de débordement au sein des opérations d'addition suivantes. Pour ce dernier cas, différentes alternatives sont possibles pour réaliser ce type de recadrage et sont présentées ci-dessous.

Certains processeurs DSP possèdent des bits de garde au niveau de l'additionneur. La présence de ces bits de garde permet de stocker les bits supplémentaires issus des additions. Cependant, si le nombre de bits de garde présents au sein de l'architecture n'est pas suffisant, alors cette première approche devra être couplée avec celle présentée ci-dessous. La seconde approche correspond au recadrage interne qui insère les bits supplémentaires avant les opérations d'addition. Cette technique est utilisée pour des opérations de type multiplication accumulation (MAC). Le résultat de chaque multiplication est recadré avant l'accumulation. Cette approche nécessite de posséder des capacités de recadrage efficaces en sortie du multiplieur. Ainsi, pour les processeurs ne possédant pas de bit de garde ou la possibilité de recadrer la sortie du multiplieur un recadrage externe est réalisé. Dans ce cas, l'entrée du système est recadrée afin d'insérer les bits supplémentaires nécessaires pour éviter les débordements lors des opérations d'addition au sein de l'application. Une alternative au recadrage externe consiste à recadrer les coefficients du système en intégrant les bits supplémentaires au niveau des coefficients du système. Ceci étant réalisé lors du codage des données, aucune opération de recadrage n'est insérée au sein du système. Cependant, cette technique modifie de façon plus importante la réponse fréquentielle du filtre par rapport à celle obtenue en précision infinie. Afin d'avoir une réponse fréquentielle identique pour chaque implantation, les résultats obtenus avec la technique de recadrage des coefficients ne sont pas présentés. Ainsi, pour chaque implantation, chaque coefficient est codé sur le même format.

La précision de l'implantation est évaluée à l'aide du Rapport Signal à Bruit de Quantification. Pour chaque application nous avons déterminé l'expression analytique de ce RSBQ. Le détail de ces expressions est présenté dans [97]. La détermination de cette métrique nécessite de déterminer la puissance du signal et du bruit de quantification en sortie du système. Le bruit de

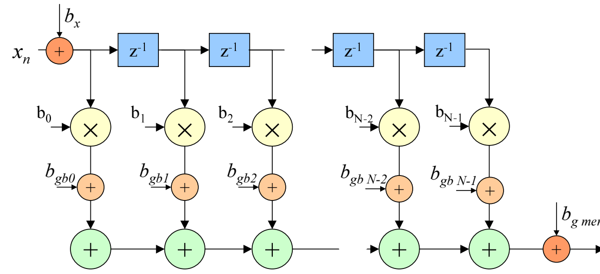


FIG. 2.9: Structure du filtre FIR et représentation des différentes sources de bruit potentielles

quantification en sortie est engendré par la propagation des différents bruits présents au sein du système. Ainsi, pour chaque application nous exposons les différentes sources de bruit situées au sein de celle-ci. Pour les différentes expérimentations, nous avons utilisé, comme signal d'entrée, un bruit blanc gaussien.

Filtre non récursif (FIR)

Nous avons représenté à la figure 2.9 et détaillé ci-après les différentes sources de bruit présentes au sein du filtre implanté. Le bruit b_x associé à l'entrée x est composé du bruit b_{qx} lié à la quantification de x et du bruit b_{gx} lié au recadrage effectué sur l'entrée dans le cas d'un cadrage externe. En sortie de chaque multiplieur nous avons un bruit b_{gbi} composé du bruit b_{gmd} lié au recadrage de la sortie du multiplieur dans le cas d'un recadrage interne et du bruit b_{gm} lié à l'utilisation d'un multiplieur réduit. Dans ce cas, la largeur de la sortie du multiplieur est inférieure à la somme de la largeur des entrées. Le dernier bruit $b_{g\ mem}$ correspond au bruit lié au changement de format de la sortie de l'additionneur lors du renvoi en mémoire de la sortie du filtre.

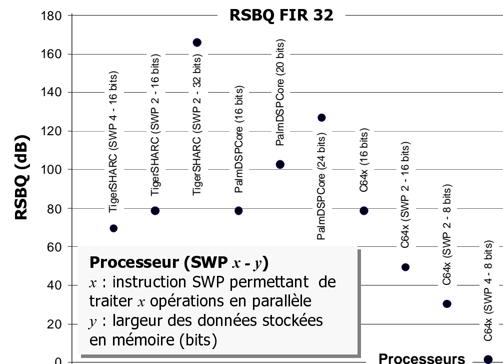


FIG. 2.10: Évolution du RSBQ en fonction de la largeur des données traitées

Le paramètre principal influençant la précision des calculs correspond à la largeur des opérandes traités. Cette largeur dépend de la largeur naturelle du processeur et des capacités SWP proposées par le processeur. Pour illustrer ce phénomène nous avons représenté à la figure 2.10 le RSBQ obtenu en sortie d'un filtre FIR composé de 32 cellules pour différents processeurs. Pour les différentes implantations représentées sur cette figure, la largeur des données en entrée et en sortie du filtre est égale à 8, 16, 20, 24 ou 32 bits. Pour un type d'implantation donné, le RSBQ exprimé en dB, est directement proportionnel à la largeur des opérandes traités.

Pour poursuivre notre étude, l'influence des autres éléments de l'architecture a été analysée en

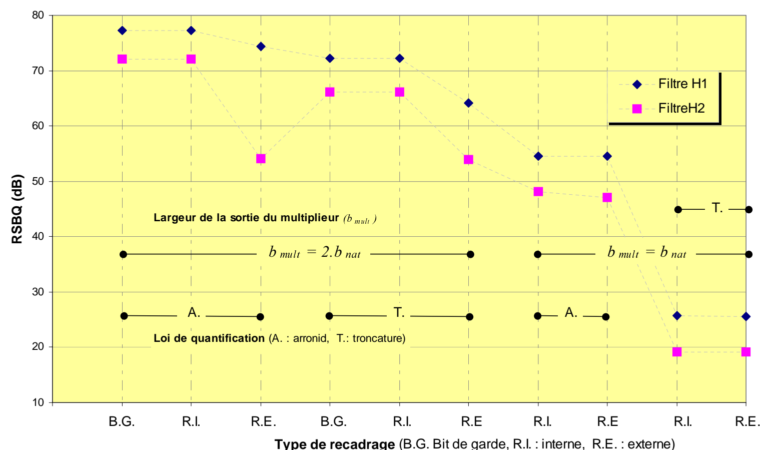


FIG. 2.11: Évolution du RSBQ en sortie de 2 filtres FIR en fonction des paramètres de l'architecture (bits de garde, capacités de recadrage, largeur de la sortie du multiplieur, loi de quantification)

considérant que les données en entrée et en sortie du filtre étaient codées sur 16 bits uniquement. Nous avons déterminé le RSBQ en sortie de deux filtres FIR $H_1(z)$ et $H_2(z)$. Le premier filtre $H_1(z)$ est une filtre passe-bas composé de 256 cellules. Pour le second filtre $H_2(z)$ les différents coefficients sont égaux à $+1$ ou -1, ce type de filtre FIR est équivalent à la corrélation d'un signal x correspondant à l'entrée du filtre avec un code bipolaire représenté à travers les coefficients h_i. Les résultats de cette analyse sont présentés à la figure 2.11.

Les différents paramètres étudiés sont le type de recadrage, la loi de quantification et la largeur de la sortie du multiplieur (b_{mult}). Cette largeur est soit égale au double de la largeur naturelle (double précision : $b_{mult} = 2.b_{nat}$) ou égale à la largeur naturelle (simple précision : $b_{mult} = b_{nat}$). En effet, certaines instructions SWP, nécessitent d'utiliser cette seconde alternative pour pouvoir réaliser le même nombre d'opérations de multiplication et d'addition (voir tableau 2.2 à la page 41). De même, si la sortie du multiplieur doit être sauvegardée en mémoire alors pour limiter les temps de transfert, cette donnée peut être stockée en simple précision.

Pour une architecture possédant un multiplieur classique ($b_{mult} = 2.b_{nat}$), la dégradation du RSBQ liée à l'utilisation d'une loi de quantification par troncature par rapport à une loi de quantification par arrondi, est comprise entre 0,2 dB et 10,72 dB. Cette dégradation du RSBQ est nettement plus importante si un multiplieur réduit est utilisé ($b_{mult} = b_{nat}$), elle est de l'ordre de 28 dB. Nous obtenons des résultats identiques si nous réalisons un recadrage interne ou si nous utilisons les bits de garde disponibles. Pour une architecture possédant un multiplieur classique l'utilisation d'un recadrage externe engendre une dégradation du RSBQ comprise entre 2,9 et 18 dB par rapport à un recadrage interne. Cette dégradation est quasiment nulle dans le cas d'un multiplieur réduit. La présence d'un multiplieur réduit utilisant une loi de quantification par arrondi, entraîne une dégradation du RSBQ de 7 à 24 dB par rapport à un multiplieur classique. Cette dégradation est nettement plus importante pour une loi de quantification par troncature, elle atteint des valeurs comprises entre 34,7 et 46,9 dB.

Les RSBQ obtenus pour différents DSP traitant des données mémorisées sur 16 bits sont présentés à la figure 2.12. Pour les processeurs de type VLIW (C64x, TigerSharc), une implantation sans recadrage en sortie du multiplieur a aussi été testée afin d'obtenir un code potentiellement plus rapide. La présence de bits de garde (TMS320C54x) au sein de l'accumulateur ou le recadrage des données en sortie du multiplieur (C64x (d), TigerSharc (S2-d)) permet d'obtenir de très bonnes performances lors des accumulations successives réalisées dans le filtre. Pour les processeurs ne possédant pas de bit de garde ou la possibilité de recadrer la

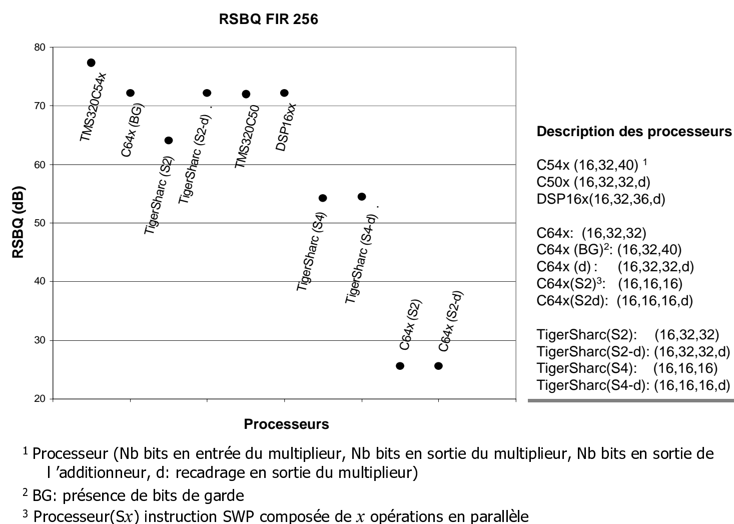


FIG. 2.12: Évolution du RSBQ en sortie d'un filtre FIR pour différents DSP 16 bits

sortie du multiplieur, un recadrage en entrée du filtre est réalisé (C64x, TigerSharc(S2)). Pour le recadrage externe, les performances obtenues avec un résultat de la multiplication codé en double précision, sont nettement plus faibles car le recadrage est réalisé avant la multiplication des données. La dégradation liée à la réalisation des calculs en simple précision, dans le cadre des instructions SWP, est relativement importante (C64x (S2)), TigerSharc (S4)), mais le gain sur le temps d'exécution peut atteindre un facteur 2. L'utilisation d'une loi de quantification par arrondi (C54x) permet d'améliorer le RSBQ et plus particulièrement si les calculs sont réalisés en simple précision (TigerSharc S4).

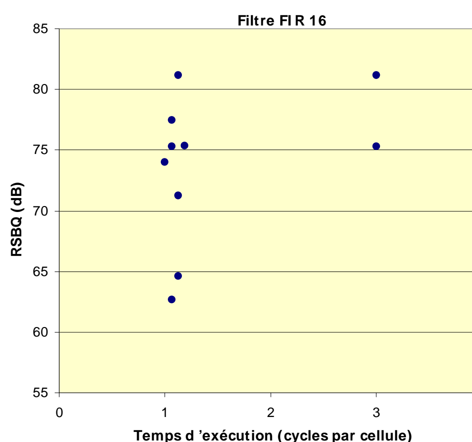


FIG. 2.13: Évolution du RSBQ en sortie d'un filtre FIR en fonction du temps d'exécution du filtre

Pour compléter cette analyse nous avons représenté à la figure 2.13, la caractéristique du RSBQ en fonction du temps d'exécution d'un filtre FIR composé de 16 cellules et implanté dans le DSP TMS320C54x. Chaque point correspond à une implantation particulière de ce filtre au sein du DSP. Cette caractéristique montre pour cette application simple, la diversité des implantations possibles au sein d'un DSP basé sur une architecture traditionnelle. Ainsi, elle souligne les opportunités d'optimisations possibles en termes de temps d'exécution et de précision si l'architecture est prise en compte. En effet, un codage des données en virgule fixe ne tenant pas compte de l'architecture aboutit à l'implantation représentée par le point de coordonnées (3, 75.3).

Filtre récursif (IIR)

Dans cette partie, nous analysons le bruit de calcul en sortie d'un filtre à réponse impulsionnelle infinie d'ordre 2. Les trois structures correspondant aux formes directe I et II et directe II transposée ont été étudiées. Les différentes sources de bruit présentes au sein de ces filtres sont détaillées à la figure 2.14. La dénomination des bruits est similaire à celle utilisée dans le cadre du filtre FIR présenté précédemment.

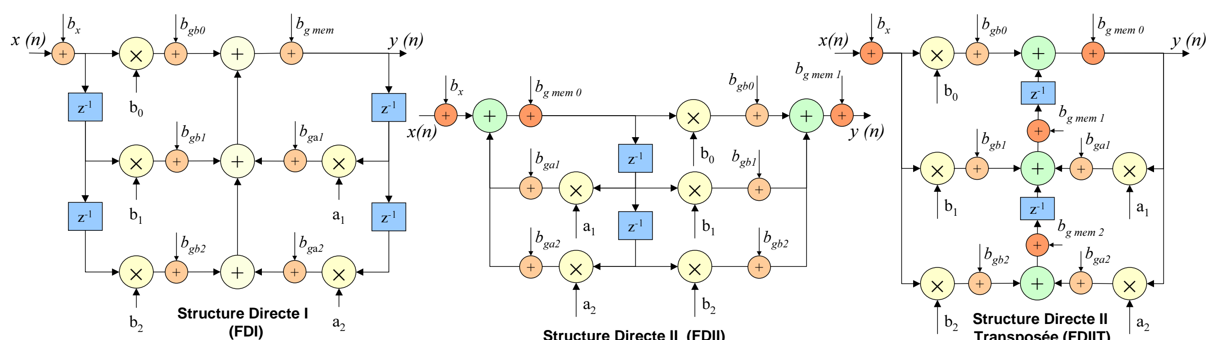


FIG. 2.14: Structures des filtres IIR et représentation des sources de bruit potentielles

Nous avons analysé le RSBQ en sortie de deux filtres IIR H_0 et H_1 pour les trois structures présentées ci-dessus et pour une largeur des données en entrée et en sortie du filtre égale à 16 bits. L'évolution du RSBQ en fonction des différents paramètres de l'architecture est présentée à la figure 2.15. Pour simplifier le codage, un format commun à tous les additionneurs a été choisi. Ainsi, pour les deux filtres considérés, la technique de codage des données utilisant les bits de garde au niveau de l'additionneur n'est pas réellement exploitable car pour la majorité des additionneurs, la dynamique de leur sortie est inférieure à la dynamique maximale des entrées de ces additionneurs.

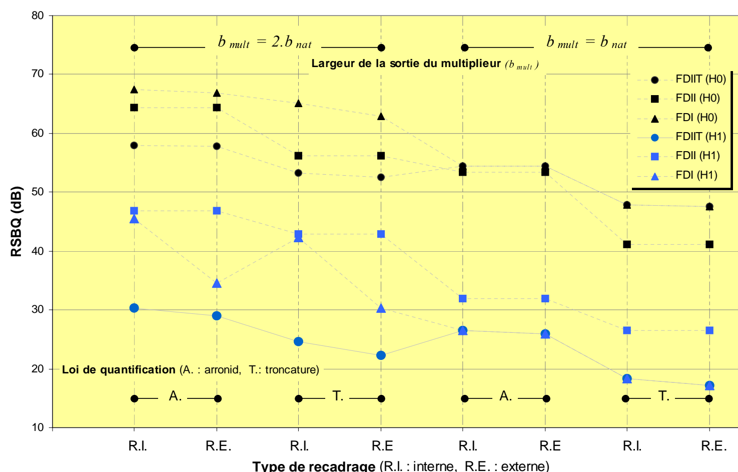


FIG. 2.15: Évolution du RSBQ en sortie de filtres IIR en fonction des paramètres de l'architecture (capacités de recadrage, largeur de la sortie du multiplieur, loi de quantification)

Pour les deux filtres utilisés la structure FDIIT possède dans la majorité des cas, des performances toujours inférieures à celles des deux autres structures. Ceci est lié au renvoi systématique en mémoire du résultat des additions pour réaliser le "vieillissement" des données. Pour les structures FDI et FDII, celle fournissant les meilleures performances est fonction du filtre et de l'architecture du processeur utilisé. La dégradation liée à l'utilisation d'une loi de quantification par troncature par rapport à une loi de quantification par arrondi dépend de la fonction

de transfert du filtre et de la structure utilisée. Cette dégradation est comprise entre 2,3 et 12,3 dB et elle est plus importante si un multiplieur réduit est utilisé. Pour la structure $FDII$, le format de l'entrée après recadrage est imposé par le format des entrées de l'additionneur, ainsi cette structure est insensible au type de recadrage. La structure $FDIIT$, est peu sensible au type de recadrage, car la puissance du bruit lié à ce recadrage est nettement plus faible que celle du bruit lié au renvoi des sorties des additionneurs en mémoire. Pour cette structure la dégradation liée à un recadrage externe est comprise entre 0 et 2,38 dB. Pour la structure FDI, la dégradation liée à un recadrage externe est nettement plus élevée, elle est peut atteindre 12 dB. Pour la structure $FDIIT$, la dégradation liée à l'utilisation d'un multiplieur réduit est peu importante (3,4 à 6,3 dB), car la structure impose un changement de format des données en sortie des différents additionneurs. La largeur de ces données est réduite à b_{nat}. Pour les deux autres structures cette dégradation est nettement plus élevée et en particulier si une loi de quantification par troncature est utilisée. Cette dégradation est comprise entre 11 et 24 dB.

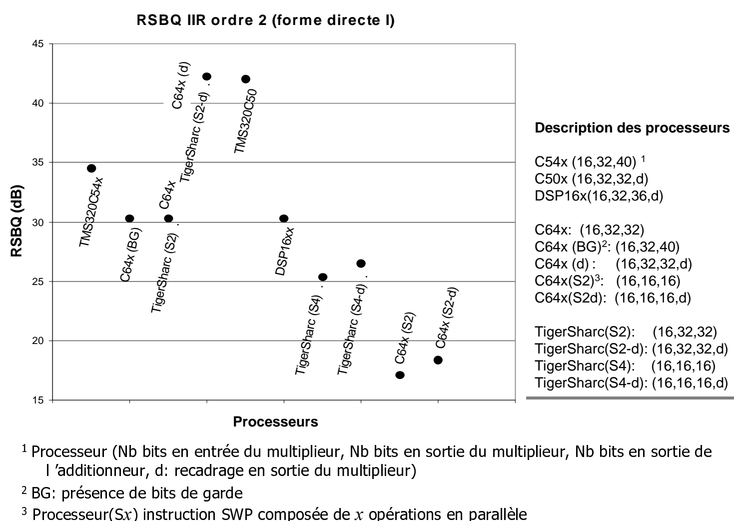


FIG. 2.16: Évolution du RSBQ en sortie d'un filtre IIR pour différents DSP 16 bits

Nous avons analysé le RSBQ en sortie de la forme directe I du filtre IIR H_1 pour différents DSP de largeur naturelle 16 bits. Les résultats sont présentés à la figure 2.16. Exceptée la technique de codage utilisant les bits de garde, nous retrouvons les mêmes conclusions que celles présentées pour le filtre FIR. Les meilleurs performances sont obtenues avec un recadrage interne. L'utilisation d'un recadrage externe conduit à une dégradation de 12 dB. Les performances sont nettement inférieures si un multiplieur réduit est utilisé.

Transformée de Fourier (FFT)

Dans cette partie, nous présentons l'analyse du bruit de calcul au sein de la transformée de Fourier rapide radix 2 et 4 basée sur une décimation dans le temps (DIT). Par rapport aux filtres FIR et IIR, cette application souligne l'influence du nombre de registres présents au sein de l'unité de traitement. En effet, pour les deux filtres FIR et IIR, le motif de calcul est du type multiplication accumulation et nécessite un seul registre pour stocker le résultat intermédiaire de l'accumulation.

Les structures des papillons des FFT radix 2 et 4 utilisant une décimation dans le temps sont présentées à la figure 2.17. L'accroissement de la dynamique de la sortie d'un papillon par rapport à son entrée nécessite un bit supplémentaire au niveau de la partie entière de la sortie par rapport à celle de l'entrée dans le cas d'une FFT radix 2. Pour une FFT radix 4, 2 bits supplémentaires sont nécessaires à chaque étage.

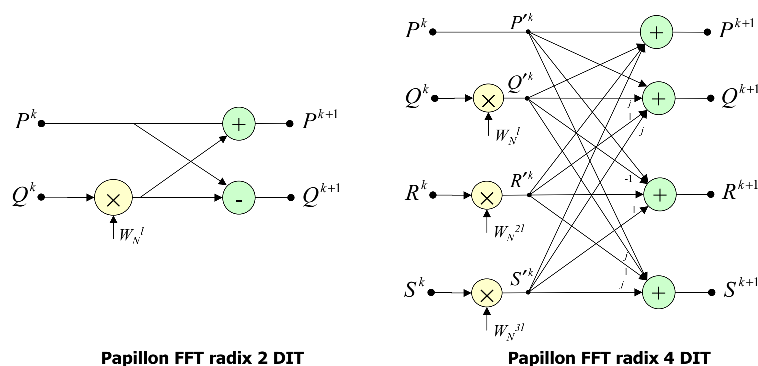


FIG. 2.17: Représentation des papillons des FFT DIT radix 2 et 4

Le nombre de registres disponibles pour stocker les résultats intermédiaires va influencer la précision des calculs au sein du papillon. Les parties réelles et imaginaires du résultat de la multiplication complexe $Q.W$ doivent être stockées pour réaliser ensuite l'addition complexe avec P. Ainsi, si l'unité de traitement ne possède pas assez de registres pour stocker ces deux résultats intermédiaires, ils doivent être renvoyés en mémoire. Pour limiter le temps de transfert vers la mémoire, ces données sont sauvegardées en mémoire en simple précision [117].

Nous résumons ci-dessous les principaux résultats obtenus dans le cadre de la FFT, une étude plus complète est présentée dans [97]. Dans le cadre d'un papillon d'une FFT de radix 2, le renvoi en mémoire des résultats intermédiaires entraîne une dégradation du RSBQ de 4 dB par rapport au premier cas. Cette dégradation est de 5,4 dB si nous utilisons un multiplieur dont la sortie est codée en simple précision. L'écart entre une solution utilisant un multiplieur classique et celle utilisant un multiplieur réduit n'est pas très élevé car dans les deux cas, la sortie du papillon est stockée en mémoire en simple précision. L'utilisation d'une loi de quantification par troncature entraîne une dégradation du RSBQ qui est comprise entre 9 et 11 dB. Pour un papillon d'une FFT radix 4, l'écart entre les différents cas est moins important. L'utilisation d'un multiplieur réduit apporte une dégradation de 2,75 dB et le renvoi des données intermédiaires en mémoire entraîne une diminution de la précision de 1,77 dB. Cependant, l'absence de renvoi de données intermédiaires en mémoire nécessite plus de registres de stockage dans l'unité de traitement par rapport à une FFT radix 2. Un papillon de FTT radix 4 utilise 7 registres intermédiaires contre 3 pour une FFT Radix 2.

2.1.5 Conclusions

Dans ce chapitre, nous avons résumé les caractéristiques principales des unités de contrôle et de mémoire des DSP et nous avons analysé plus en détail la structure des unités de traitement et les différents éléments qui la composent. Les résultats des expérimentations réalisées pour quantifier l'influence de ces différents éléments ont été détaillés. Les différents éléments de l'architecture pouvant influencer la précision des calculs sont la largeur naturelle du processeur, la présence de bits de garde au niveau de l'accumulateur, le nombre de registres présents dans l'unité de traitement, l'utilisation d'instructions SWP, les capacités de recadrage et la loi de quantification disponible. Cette analyse montre la nécessité de prendre en compte le modèle d'architecture dans le processus de codage des données afin d'obtenir une implantation de qualité en termes de précision et de temps d'exécution. De plus, les résultats obtenus permettent de comparer les différents modèles d'architecture des DSP. Ainsi, ce type d'analyse peut être utilisé pour compléter les métriques disponibles pour la comparaison des processeurs [12]. En effet, ces métriques évaluent essentiellement la vitesse de traitement des DSP.

2.2 Compilation pour les processeurs de traitement du signal

L'objectif de cette partie est d'analyser l'interaction entre les différentes phases de la compilation et le processus de codage des données. Cette étude permet de définir les objectifs de différentes phases du processus de codage et le couplage de ces phases avec celles de la génération de code. Après la définition des objectifs de la génération de code pour les DSP, les différentes techniques classiques de compilation sont exposées. Les différents traitements réalisés au sein de la partie frontale et finale d'un compilateur sont détaillés. Ensuite, les spécificités liées à la génération de code pour les DSP sont présentées. La dernière section de cette partie est consacrée à l'analyse de l'interaction entre les phases de compilation et de codage des données. Les résultats des différentes expérimentations réalisées sont présentés afin d'illustrer les concepts exposés.

2.2.1 Introduction

Les exigences de la compilation pour les processeurs de traitement du signal

La génération de code pour les processeurs de traitement du signal est soumise à des contraintes fortes parfois différentes de celles des processeurs à usage général. Elles sont liées aux applications cibles et aux spécificités des architectures des DSP.

L'utilisation des DSP pour des applications embarquées implique la génération d'un code de qualité en termes de temps d'exécution, de taille du code et de sûreté de fonctionnement. Les algorithmes de traitement du signal étant soumis à des contraintes temps réel sévères, il est essentiel de minimiser le temps d'exécution du code afin de respecter les contraintes de temps imposées. La génération d'un code inefficace nécessitera l'utilisation d'un processeur plus performant et entraînera l'augmentation du coût et de la consommation de l'application. Le coût des processeurs est fortement lié à la taille de la mémoire. Il est nécessaire de générer un code compact afin de minimiser la taille de la mémoire réservée pour le code de l'application. Les algorithmes générés sont implantés dans des applications embarquées placées dans un environnement réel. Ainsi, il est indispensable que les systèmes soient sûrs et garantissent l'absence de fautes. Pour cela, le développement à l'aide de langages de haut niveau est privilégié par rapport à un langage de bas niveau tel que l'assembleur. La génération d'un code de qualité étant primordiale, les concepteurs acceptent une augmentation des temps de compilation par rapport à ceux obtenus pour les processeurs à usage général.

Dans la partie précédente, l'architecture des DSP et leurs particularités par rapport aux processeurs à usage général ont été décrites. Afin d'obtenir un code performant, le flot de compilation doit prendre en compte ces différentes spécificités. Les compilateurs doivent supporter l'arithmétique virgule fixe et permettre d'utiliser l'ensemble des processus associés tels que la possibilité de spécifier une donnée en virgule fixe avec un format quelconque ou le choix des lois de quantification et de débordement. De plus, les compilateurs doivent prendre en compte les spécificités présentes au sein de l'unité de mémoire et de contrôle. Les architectures DSP proposent des modes d'adressage variés et parfois plusieurs bancs de mémoires pour stocker les données. De même des instructions spécifiques telles que les boucles matérielles ou les instructions de mode sont disponibles.

L'évolution rapide des architectures de traitement du signal rend le concept de compilateurs flexibles primordial. Le but de la compilation flexible est de concevoir des compilateurs efficaces pour une classe d'architecture et de les adapter à une cible avec le minimum d'effort. Le temps de vie des processeurs DSP étant de plus en plus réduit, il devient essentiel d'utiliser un compilateur flexible afin de minimiser le coût de développement de celui-ci. Ce concept de

flexibilité est encore plus important pour les ASIP, en effet il n'est pas envisageable de concevoir un compilateur spécifique pour chaque ASIP pour lequel uniquement une dizaine d'algorithmes sera implanté. Dans [106], l'auteur définit différents niveaux de flexibilité pouvant être mis en œuvre au sein des compilateurs.

2.2.2 Les techniques de compilation classiques

Selon Aho et al. [6] la définition d'un compilateur est la suivante : *"Un compilateur est un programme qui lit un programme écrit dans un langage source pour le traduire en un programme équivalent écrit dans un langage cible."*. Les compilateurs sont composés d'une partie frontale ou partie haute (*front end*) et d'une partie finale ou partie basse (*back end*). La partie frontale réalise l'identification des unités lexicales, la vérification du respect des règles associées au langage source et la création d'une représentation intermédiaire. La partie finale effectue la synthèse du programme dans le langage cible à partir de la représentation intermédiaire.

Les différentes phases de la partie frontale sont liées au langage source mais sont indépendantes du langage cible. De même pour la partie finale, les phases sont indépendantes du langage source. Ainsi, la représentation intermédiaire est indépendante des langages source et cible. Cette séparation du flot de compilation en deux parties permet théoriquement de ne changer qu'une seule des deux parties lors d'un changement de langage source ou de processeur cible.

Partie frontale (partie haute)

Dans cette partie, nous présentons les différentes phases de la partie frontale dont le synoptique est représenté à la figure 2.18.

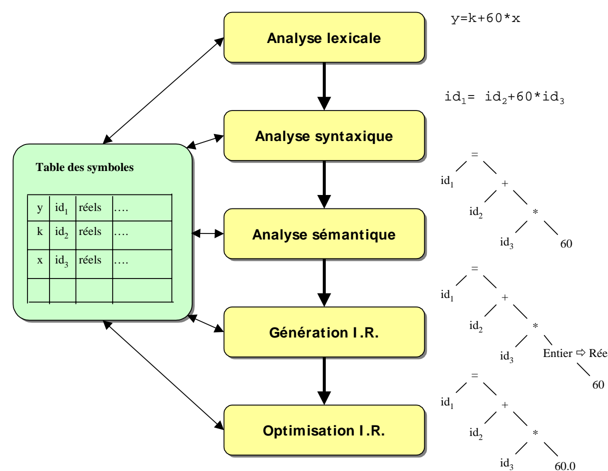


FIG. 2.18: Synoptique de la partie frontale d'un compilateur

Au cours de la phase d'analyse lexicale, le compilateur effectue le regroupement des caractères du programme source en unités lexicales cohérentes appelées lexèmes. Cette phase permet d'identifier les trois types d'unités lexicales possibles que sont les mots clés du langage, les séparateurs et les identificateurs. Les séparateurs correspondent aux opérateurs arithmétiques ou logiques, aux caractères de ponctuation et aux caractères spéciaux. Les identificateurs représentent les variables où les procédures. Les variables sont déclarées dans la table des symboles et associées à un identificateur.

La phase d'analyse syntaxique permet de vérifier que les chaînes lexicales peuvent être engendrées à partir de la grammaire du langage. Elle définit l'ensemble des règles décrivant la structure syntaxique du langage. Nous obtenons en sortie une structure hiérarchique représentée

sous forme d'un arbre d'expression ou les nœuds représentent les opérateurs et les prédécesseurs des nœuds les opérandes.

Au cours de la phase d'analyse sémantique, le compilateur vérifie la validité sémantique du programme source et collecte les informations concernant le type de chaque identificateur. Ensuite, un contrôle de type est réalisé afin de valider la cohérence de type entre les différents opérandes d'une opération. Lorsqu'une erreur de sémantique est détectée, le compilateur génère une erreur ou réalise lui même la conversion de type.

Pendant ces différentes phases le compilateur gère les différentes tables de symboles. Le compilateur enregistre dans une table des symboles les différents identificateurs et collecte les différentes informations les concernant (emplacement mémoire, type, porté). Pour les procédures il détermine le nombre, le type et le mode de passage des paramètres.

Les deux dernières phases de la partie frontale correspondent à la génération et à l'optimisation de la représentation intermédiaire (I.R.). Cette représentation peut être vue comme un programme pour une machine abstraite. Elle doit être facile à produire et facile à traduire. Différentes formes de représentation intermédiaire peuvent être utilisées. Nous pouvons citer les représentations sous forme de code à 3 adresses, d'arbres d'expression ou de graphe acyclique orienté (DAG). Pour la représentation intermédiaire sous forme d'arbres d'expression, chaque expression du code source est représentée par un arbre d'expression. La représentation sous forme de graphe acyclique orienté correspond à une représentation intermédiaire sous forme d'arbres d'expression pour laquelle les sous-expressions communes ont été éliminées. Dans un DAG, une sous-expression commune est modélisée par un nœud possédant plus d'un successeur. Cette représentation intermédiaire regroupe les traitements effectués sur les différentes données mais intègre aussi le flot de contrôle ainsi que les appels de procédure.

Les optimisations réalisées sur la représentation intermédiaire sont indépendantes de l'architecture de la machine cible. Elles ont pour but de rendre le code plus rapide ou plus compact tout en préservant l'intégrité du code. Les transformations réalisées sur le code intermédiaire sont issues de l'analyse du flot de contrôle et du flot de données. Elles sont effectuées au niveau local si le compilateur examine uniquement les instructions présentes au sein d'un bloc de base ou au niveau global si l'analyse des instructions dépasse les limites des blocs de base. Un bloc de base représente une séquence d'instructions consécutives dans laquelle le flot de contrôle est activé en début de celle-ci et est inhibé en fin de celle ci, sans possibilité d'arrêt ou de branchement autre qu'à la fin de la séquence. Les principales optimisations réalisées sur la représentation intermédiaire sont les suivantes :

- élimination des sous-expressions communes et du code inutile ;
- propagation des constantes et des copies ;
- simplification algébrique ;
- optimisation des boucles (déplacement de code, élimination des variables d'induction).

Partie finale (partie basse)

L'objectif de la partie finale est de générer à partir de la représentation intermédiaire un code exécutable par la machine cible. Ce processus se décompose en quatre parties correspondant à la sélection des instructions, à l'allocation des registres, à l'ordonnancement et à l'optimisation du code généré. Ce problème de génération de code optimal est très difficile à résoudre de manière exacte, ainsi il est nécessaire d'utiliser des heuristiques qui produisent un code de qualité mais pas toujours optimal.

Sélection d'instructions : Le but de la sélection de code est de choisir les instructions qui vont permettre de réaliser le plus efficacement possible l'ensemble des opérations de l'algorithme.

Cette phase associe une ou plusieurs instructions à une opération ou un ensemble d'opérations de l'algorithme en optimisant le temps d'exécution de celui-ci. Chaque instruction est modélisée par un motif décrivant l'instruction sous forme d'un arbre et par un coût représentant le temps d'exécution de l'instruction. Cette première phase utilise une représentation intermédiaire du code source sous forme d'arbres ou de graphes. Mais la sélection d'instructions est un problème NP-complet lorsque la représentation intermédiaire est un graphe acyclique dirigé (DAG), cependant un code vertical optimal peut être obtenu si certaines conditions sont satisfaites [5]. La représentation intermédiaire doit être un arbre d'expression et les motifs de base du processeur doivent être des arbres. De plus, la structure des registres doit être homogène.

Lorsque la représentation intermédiaire est un graphe acyclique dirigé, des heuristiques sont utilisées pour transformer les DAG en arbre d'expression. Les déconnexions sont réalisées au niveau des nœuds qui possèdent plusieurs successeurs. Ensuite, la sélection de code se décompose en une phase de reconnaissance de motifs (*Tree pattern matching*) et de couverture d'arbres (*Tree covering*). La première phase localise les parties de l'arbre d'expression qui correspondent à des motifs d'instructions. L'objectif de la seconde phase est de trouver la meilleure couverture complète de l'arbre d'expression à l'aide des motifs présélectionnés en retenant les instructions permettant de minimiser le coût global de la sélection. Ce problème est résolu par une méthode basée sur la programmation dynamique proposée par Aho et Johnson [5].

Un des challenges de la sélection d'instructions est de détecter la présence de motifs pouvant être implantés à l'aide d'instructions complexes afin de réduire le temps d'exécution global et la taille du code. Ces instructions complexes spécifient l'enchaînement de plusieurs instructions et permettent, ainsi, de réaliser en parallèle plusieurs opérations.

Allocation des registres : Cette phase correspond à l'affectation des variables de l'algorithme aux unités de stockage de la machine et a pour but de minimiser l'ajout d'instructions de transfert de variables entre la mémoire et les registres. Elle se décompose en deux sous-problèmes, l'allocation de registres et l'assignation de registres. L'allocation de registres permet de déterminer les variables qui seront gardées dans des registres et celles qui seront renvoyées en mémoire. L'assignation de registres associe un registre physique de la machine à chaque variable devant rester dans l'unité de traitement. Si l'allocation de registres est locale, les variables sont allouées au sein d'un bloc de base et ensuite renvoyées en mémoire à la fin de celui-ci. Afin d'éviter des renvois en mémoire inutiles une allocation globale est préférée. Les variables les plus fréquemment utilisées au sein de l'algorithme sont assignées à des registres et conservées dans ceux-ci au sein de plusieurs blocs de base.

Une formulation classique du problème d'allocation de registres correspond à la coloration optimale d'un graphe d'interférences. Ceci permet de réaliser en même temps l'allocation et l'assignation de registres. Cependant, certaines hypothèses doivent être posées pour pouvoir utiliser les techniques de coloration de graphe. La structure des registres du processeur doit être homogène et le nombre de registres N doit être prédéfini. De plus, ce type de technique suppose que la phase de sélection de code soit déjà réalisée et que la durée de vie des variables soit connue. Cette dernière hypothèse implique que l'ordre d'exécution du code soit connu. Ce problème de coloriage optimal d'un graphe d'interférences est un problème NP-complet, mais des heuristiques donnant des résultats satisfaisants ont été proposées par Chaitin et al. [19].

Ordonnancement : La phase d'ordonnancement détermine l'instant d'exécution des différentes instructions. Ces instants sont définis afin de minimiser le temps d'exécution du code tout en respectant la sémantique de celui-ci. L'ordre d'exécution des instructions doit respecter les contraintes sur les données et les ressources. Dans le premier cas, le résultat d'une opération

doit être disponible avant son utilisation par une autre opération. Deux instructions utilisant les mêmes ressources ne pourront s'exécuter en même temps.

Pour les processeurs possédant du parallélisme au niveau instruction, la phase d'ordonnancement nécessite de réaliser en plus un compactage du code. Il permet de regrouper des instructions partielles pouvant s'exécuter en parallèle. Le code vertical composé d'une suite ordonnée d'instructions partielles est transformé en un code horizontal utilisant des instructions globales (instruction composée de plusieurs instructions partielles exécutées en parallèle). Cette phase doit permettre d'exploiter au mieux les capacités de parallélisme du processeur. Cependant, pour les processeurs possédant des capacités de parallélisme, la limitation de l'ordonnancement aux limites du bloc de base ne permet pas toujours d'utiliser l'ensemble du parallélisme offert. Pour obtenir de meilleures performances, des ordonnancements globaux sont utilisés. Ils permettent de dépasser les limites des blocs de base dans la recherche des instructions pouvant être ordonnancées. Ces techniques déplacent certaines parties de code afin de pouvoir combiner des instructions appartenant à différents blocs de base. Les méthodes développées concernent plus particulièrement les structures répétitives et conditionnelles [46].

Pour les structures répétitives, l'objectif du dépliement de boucle est de favoriser le parallélisme au niveau instruction en augmentant la taille du bloc de base représentant le corps de la boucle. Les dépendances entre les données au sein d'une même itération ne permettent pas d'exécuter en parallèle les différentes opérations. Ainsi, le regroupement de plusieurs itérations au sein d'un même bloc de base permet d'exécuter en parallèle des opérations issues de ces différentes itérations. Cependant, la taille du code est augmentée, ainsi un compromis doit être réalisé entre la taille du code et les possibilités de parallélisme. La technique de pipeline logiciel est basée sur le même concept de mise en parallèle d'opérations issues d'itérations différentes afin de contourner les dépendances de données. Cette technique réalise un dépliement symbolique de la boucle, cependant le nombre d'itérations et la taille de la boucle ne sont pratiquement pas modifiés. Les opérations issues de différentes itérations sont entrelacées afin de séparer les opérations dépendantes et d'utiliser au maximum le parallélisme offert par l'architecture. Par rapport au dépliement de boucle, cette technique ne conserve au sein de la structure répétitive que le ou les cycles utilisant au maximum l'architecture. Pour respecter la sémantique du code un prologue et un épilogue sont ajoutés, afin d'amorcer le pipeline au départ puis de le vider à la fin de l'exécution de la boucle.

Optimisations à lucarne : La dernière phase du processus de génération de code correspond à l'optimisation du code généré afin d'obtenir un code de meilleure qualité. Les techniques d'optimisation à lucarne examinent une séquence courte d'instructions (lucarne) et transforment celle-ci afin d'obtenir une séquence plus rapide et plus courte. Ces techniques sont simples mais permettent d'obtenir de bons résultats. Chaque modification peut engendrer de nouvelles optimisations, ainsi il est nécessaire de réaliser les optimisations en plusieurs passes. Les optimisations les plus souvent utilisées correspondent à l'élimination des instructions redondantes, à l'optimisation du flot de contrôle et à la réduction de force (remplacement des opérations coûteuses en termes de temps de calcul).

2.2.3 Les spécificités de la compilation pour les DSP

Nous présentons dans cette partie les spécificités de la compilation pour les processeurs de traitement du signal. Tout d'abord, nous soulignons l'inefficacité des techniques classiques pour la génération de code pour les DSP. Ensuite, les différentes raisons de cette inefficacité sont présentées.

Différentes expérimentations ont montré et quantifié l'inefficacité des compilateurs de langage de haut niveau pour les DSP [80, 159, 151]. Les résultats obtenus dans le cadre d'applications de

TNS [159], montrent l'inefficacité des compilateurs C classiques pour les DSP conventionnels. Le temps d'exécution du code généré par le compilateur est 4 à 10 fois supérieur à celui du code assembleur et le surcoût de la taille du code est de l'ordre de 40% à 368%. Cette inefficacité rend impossible l'utilisation de ces techniques de compilation dans le cadre d'applications de TNS embarquées. Les résultats obtenus pour une architecture orthogonale de type VLIW sont nettement meilleurs.

L'inefficacité des compilateurs de langage de haut niveau est due à deux raisons majeures. Tout d'abord, le langage de haut niveau utilisé en entrée des compilateurs pour DSP n'est pas adapté au domaine du TNS. Ainsi, nous présentons ci-dessous les langages de description utilisés pour les applications de TNS et soulignons les faiblesses du langage C pour spécifier les applications de TNS. Ceci a une influence directe sur la définition de notre méthodologie et plus particulièrement sur la détermination du support utilisé pour décrire l'application en virgule fixe. En effet, le langage C ne permet pas de spécifier correctement les traitements utilisant l'arithmétique virgule fixe.

La seconde raison de l'inefficacité des compilateurs pour DSP réside dans l'inadéquation entre les techniques de compilation utilisées et l'architecture des DSP. Les techniques de compilation classiques destinées aux processeurs à usage général ne sont pas efficaces pour les architectures spécialisées. Les algorithmes de génération de code ne sont pas adaptés aux contraintes des architectures des DSP. De plus, ces techniques de compilation ne prennent pas en compte certaines spécificités des DSP et ainsi ne tirent pas profit de l'architecture.

Les techniques de compilation présentées dans le paragraphe 2.2.2 ont été développées pour des processeurs à usage général de type RISC. Ces compilateurs génèrent un code efficace pour les machines possédant une architecture homogène composée de nombreux registres généraux et interchangeables. Les DSP sont caractérisés par une architecture spécialisée intégrant un nombre de registres relativement faible et ceux-ci sont dédiés à des opérateurs précis. De plus, pour limiter la taille du code, le jeu d'instructions est irrégulier et des instructions complexes sont utilisées pour exprimer le parallélisme.

Les techniques de compilation classiques ne prennent pas en compte les spécificités des DSP qui permettent de traiter efficacement les algorithmes de TNS. Le placement efficace des données au sein des différents bancs mémoires permet de générer un code plus optimisé. De plus, il est nécessaire d'optimiser le placement des données les unes par rapport aux autres en fonction de leur utilisation afin de tirer profit des potentialités des unités de génération d'adresses (adressage indirect par registre et post-modifications). Les DSP possèdent souvent plusieurs registres d'état afin de limiter la taille des instructions. Les compilateurs ont tendance à systématiquement ajouter des instructions de changement de mode lors de l'utilisation de ces instructions sans analyser le mode courant. Ceci conduit à la présence d'instructions redondantes pour l'initialisation des registres de mode. Ainsi, la taille et le temps d'exécution du code sont augmentés.

L'inadéquation entre le compilateur et l'architecture est aussi liée à la méthode de conception de ces deux éléments. L'approche classique de conception du processeur et du compilateur est séquentielle. La première phase correspond à la conception et l'optimisation de l'architecture du DSP. Lorsque l'architecture est figée, le développement du compilateur est réalisé et les algorithmes de génération de code sont adaptés à l'architecture. Afin d'obtenir une meilleure adéquation compilateur/processeur, une conception conjointe du compilateur et de l'architecture du processeur doit être réalisée [158, 106].

Langage de description des applications de traitement du signal (TNS)

Un langage de haut niveau destiné à décrire des applications de TNS doit intégrer différentes caractéristiques. Ce langage doit permettre de représenter l'application à un niveau d'abstrac-

tion élevé afin de pouvoir réaliser une exploration architecturale rapide. Il doit faciliter la représentation du parallélisme de l'algorithme. Les types supportés par le langage doivent permettre de spécifier les données en virgule fixe avec un format arbitraire. De plus, la présence d'un type permettant de manipuler facilement les nombres complexes faciliterait la description de nombreux algorithmes de TNS.

Le langage C-ANSI : Des langages de haut niveau tels que SILAGE ont été spécifiquement conçus pour décrire des applications de TNS, mais le langage le plus utilisé reste le langage C. Il est employé par de nombreux organismes de standardisation pour décrire les normes associées aux applications de TNS. De plus, ce langage est très utilisé pour le développement logiciel. Le langage C est bien adapté pour décrire des applications orientées flot de contrôle mais il n'est pas conçu pour décrire des applications de TNS. Ce langage ne possède pas le support pour représenter les différentes spécificités des applications de TNS et de l'architecture des DSP.

Le langage C ne propose pas de support pour l'arithmétique virgule fixe. Il est composé uniquement de deux types de données numériques (*float* et *int*) ne permettant pas de décrire explicitement des données fractionnaires en virgule fixe. L'émulation des types virgule fixe à l'aide du type *int* ne fournit pas une description naturelle de l'algorithme et est source d'erreurs. De plus, ce langage ne permet pas de spécifier les lois de quantification et de saturation.

Le langage C n'intègre pas les différentes spécificités des unités mémoires des DSP. En effet, certains DSP possèdent différents bancs mémoires disposant de leur propre unité de génération d'adresses (UGA). Le langage C ne permet pas d'associer une donnée à un banc mémoire. Ceci permettrait de faciliter l'utilisation du parallélisme au niveau instruction. De même le langage C ne propose pas le support des modes d'adressage spécifiques aux DSP tels que les adressages modulo ou bit inversé qui permettent pour certains algorithmes de TNS de manipuler plus efficacement les données.

Les extensions du langage C-ANSI : Le langage C n'étant pas adapté pour décrire les types et opérations associés au domaine du TNS et des DSP, les constructeurs ont proposés des extensions à ce langage. Elles correspondent à la création de nouveaux types et à la mise en place de fonctions permettant d'exprimer des opérations spécifiques. Ces extensions permettent de faciliter la programmation des DSP à l'aide du langage C et d'améliorer les performances du code généré en fournissant plus d'informations aux compilateurs. Les résultats présentés dans [159] montrent l'amélioration de la qualité des compilateurs, si des extensions au code C sont utilisées.

Des extensions au langage C ANSI, dénommées *DSP-C* [4] [49], ont été proposées par l'ACE (Associated Compiler Experts) et la société Philips. Elles sont utilisées par certains constructeurs de DSP tels que Philips, NEC, Ericson et Zilog et dans les outils de développement de compilateurs de la société CoSy. Ces extensions définissent de nouveaux types associés à l'arithmétique virgule fixe et aux spécificités de l'architecture mémoire des DSP.

Deux nouveaux types ont été proposés afin de représenter des données en virgule fixe. Le premier type $_fixed$ permet de représenter les données codées en virgule fixe cadrée à gauche et le type *_accum* étend le domaine de définition du type $_fixed$ en intégrant une partie entière. Deux variables correspondant au facteur d'échelle et au nombre de bits pour coder la donnée permettent de paramétrer en fonction de la cible, les caractéristiques de ces deux types. Ces paramètres sont stockés dans le fichier de déclaration des types en virgule fixe. Le facteur d'échelle est commun au type $_fixed$ et *_accum*. Les modificateurs *long* et *short* sont utilisés pour déclarer des types possédant une largeur différente. L'association des deux types génériques $_fixed$ et *_accum* et des différents modificateurs permet de définir jusqu'a 6 types

en virgule fixe. Cependant, pour chaque type la position de la virgule étant définie a priori, un format figé est associé à chaque type. Ceci ne permet pas de définir une donnée en virgule fixe avec un format quelconque. L'association d'un type à une constante est réalisée à l'aide d'un suffixe représentant le type utilisé pour coder la constante. Pour compléter la description des données en virgule fixe le modificateur *_sat* est proposé afin de déclarer des variables utilisant une loi de saturation pour gérer les débordements.

L'extension du langage C++ à travers *SystemC* [115] permet de spécifier et de manipuler des données en virgule fixe à l'aide des types *sc_fixed* et *sc_fix*. Les différents mécanismes liés à l'arithmétique virgule fixe peuvent être spécifiés au sein de *SystemC*. Ainsi, *SystemC* permet de décrire une application de traitement du signal en virgule fixe mais à notre connaissance les compilateurs pour DSP ne supportent pas cette extension du langage C++.

Les fonctions intrinsèques permettent au programmeur de spécifier des instructions de bas niveau au sein du langage de haut niveau. Au cours de la compilation, une expression en C appelée fonction intrinsèque, est remplacée par une suite d'instructions de plus bas niveau (instructions intrinsèques). Ces dernières sont décrites directement en assembleur ou dans le langage de la représentation intermédiaire propre au compilateur. Ces fonctions intrinsèques vont permettre de spécifier certaines fonctionnalités non présentes au niveau du langage C telles que les lois de quantification ou de dépassement.

L'utilisateur peut inclure directement une séquence d'instructions en assembleur au sein du code C ou utiliser des macros assembleur. Ces dernières correspondent à des fonctions optimisées écrites en assembleur et pouvant être appelées en C. L'utilisation directe de l'assembleur permet d'obtenir un code plus efficace que celui généré par le compilateur. Mais ces techniques souffrent de nombreux désavantages. Ces fonctions intrinsèques ne sont pas portables d'une cible à l'autre, il est nécessaire de réécrire le code assembleur pour les nouvelles cibles. Pour les macros assembleur des instructions supplémentaires sont nécessaires pour réaliser l'appel de fonction et le passage de paramètres. Les séquences d'instructions intrinsèques sont vues par le compilateur comme une boite noire, ainsi il ne peut pas réaliser les optimisations de haut niveau (propagation des constantes, etc.) et de bas niveau. Au cours de l'ordonnancement le compilateur ne peut pas combiner les instructions du bloc d'instructions intrinsèques et celles présentes avant et après celui-ci. Ces contraintes diminuent la qualité du code en ne permettent pas le regroupement d'instructions pouvant s'exécuter en parallèle.

Langage de description orienté signal : Des langages de haut niveau tels que Signal [44, 13] ou SILAGE [47] ont été définis pour spécifier efficacement les applications de TNS. Nous résumons dans ce paragraphe les caractéristiques du langage SILAGE. Le langage C est un langage procédural qui impose un ordre partiel dans l'exécution des opérations. Il est basé sur le modèle d'exécution de Von Neuman qui considère une mémoire globale au sein de laquelle les variables stockées sont modifiées par l'exécution de commandes séquentielles. Le langage SILAGE [47] est un langage applicatif, il opère par l'application de fonctions sur des variables. Le but de ce langage est de représenter textuellement les graphes flot de données et d'incorporer les spécificités des opérateurs utilisés dans les algorithmes de TNS.

L'objet de base du langage est un train de valeurs appelé *signal* qui représente l'ensemble des échantillons associés à une variable. Un programme SILAGE est composé d'une séquence non ordonnée de définitions de signaux et de fonctions agissant sur ces signaux et produisant de nouveaux signaux. A chaque signal est associé un format. Dans le cas des signaux représentés en virgule fixe, le format est défini par le nombre total de bits alloués b et le nombre de bits n réservés pour la partie fractionnaire. Le format du membre de gauche d'une expression peut être obtenu à partir du format des membres de droite à l'aide de règles ou défini à l'aide d'un *casting*

si l'utilisateur souhaite fixer un format particulier. Le langage SILAGE permet de définir pour chaque opération les lois de débordement et de quantification à utiliser.

2.2.4 Analyse de l'interaction entre la compilation et le codage des données

L'interaction entre les différentes phases de compilation et le processus de codage des données en virgule fixe a été analysée afin de définir le couplage nécessaire entre notre méthodologie et les différentes phases de génération de code. Nous détaillons ci-dessous pour chaque phase, l'interaction avec le processus de codage des données.

Sélection d'instructions

L'objectif de la phase de sélection d'instructions est de choisir les instructions qui vont permettre de réaliser le plus efficacement possible l'ensemble des opérations de l'application. Cette phase associe une ou plusieurs instructions à une opération ou un ensemble d'opérations de l'application en optimisant le temps d'exécution de celle-ci. Une instruction est spécifiée par sa fonctionnalité et certains paramètres dont la largeur des opérandes d'entrée et de sortie dans le cadre des instructions arithmétiques. Ainsi, la phase de sélection d'instructions, nécessite que le type (largeur des données) des opérandes d'entrée et de sortie de chaque opération arithmétique soit défini.

Pour les processeurs basés sur une architecture traditionnelle, peu de degrés de liberté sont proposés au niveau de la largeur des opérandes d'entrée et de sortie. Cependant, pour les processeurs permettant de traiter divers types de données au travers des instructions SWP, plusieurs alternatives sont possibles pour chaque opération. La largeur de ces opérandes va directement influencer la précision des calculs, mais aussi le temps d'exécution du code. Comme pour les autres phases de notre méthodologie, l'objectif du processus de choix de la largeur des données est de minimiser le temps d'exécution et la taille du code tant que les critères de qualité associés à l'application sont respectés.

Si le type de chaque donnée n'est pas défini avant la phase de sélection d'instructions, alors le processus de génération de code doit prendre en compte pour chaque opération les différentes alternatives possibles. Les différentes phases de génération de code étant complexes, il ne semble pas envisageable de reporter le choix du type des opérandes après la phase de sélection d'instructions. Une solution fixant la largeur des opérandes avant la sélection d'instructions et permettant de remettre en cause ces choix si nécessaire lors du processus de génération de code, semble plus réaliste. Ainsi, les types des opérandes de chaque opération seront définis avant cette phase de sélection d'instructions.

Allocation de registres

L'allocation de registres correspond à l'affectation des variables de l'algorithme aux unités de stockage du processeur. Cette phase détermine les variables qui seront gardées dans des registres et celles qui seront renvoyées en mémoire. En effet, le nombre de registres présents au sein de l'unité de traitement étant limité, il est nécessaire de renvoyer certaines variables en mémoire lorsque tous les registres sont utilisés. Ce processus, dénommé *spilling*, augmente le temps d'exécution et la taille du code à travers l'insertion d'instructions d'écriture et de lecture en mémoire. Ces renvois en mémoire sont relativement fréquents au sein du code généré pour les DSP basés sur une architecture traditionnelle. Différentes raisons sont à l'origine de ces renvois en mémoire. Tout d'abord, le nombre de registres d'accumulation présents dans les DSP basés sur une architecture traditionnelle est relativement faible. Il est classiquement compris entre 1 et 8. De plus, les compilateurs ajoutent des renvois en mémoire supplémentaires lors de la phase d'allocation de registres. Ces renvois sont nombreux et sont en partie, à l'origine de la mauvaise qualité du code généré par les compilateurs C [149]. Dans [121], l'auteur a mesuré en termes

de temps d'exécution le surcoût lié à l'utilisation du compilateur C pour le DSP OakDSPCore [144] par rapport à un code assembleur optimisé manuellement. L'influence des différents facteurs responsables de l'inefficacité du compilateur C, a été mesurée. Pour la FFT, le surcoût lié à l'utilisation du compilateur C, représente 168% et les renvois en mémoire représentent 33% de ce surcoût. La part du surcoût lié aux renvois en mémoire peut atteindre 50% dans le cadre de certaines applications.

Pour la majorité des DSP, les résultats des calculs intermédiaires sont stockés au sein de l'unité de traitement en double précision sur $2.b_{nat}$ bits ou $2.b_{nat} + b_g$ bits si des bits de garde sont utilisés. Le renvoi en mémoire de ce type de données nécessite 4 à 6 accès pour l'écriture et la lecture. Afin de limiter le surcoût lié à ce renvoi en mémoire, le résultat peut être stocké en mémoire en simple précision (b_{nat}). Ce changement de format lié à l'élimination d'un certain nombre de bits, entraîne la présence d'un bruit de calcul supplémentaire.

Pour illustrer ce phénomène nous avons implanté un filtre FIR dont le code source est présenté à la figure 2.19.a au sein des DSP TMS320C54x et TMS320C5x. Le cœur de la structure répétitive correspondant à une cellule du filtre, réalise l'opération de multiplication-addition et le "vieillissement" des échantillons. Pour le TMS320C54x, le code généré par le compilateur pour le cœur est présenté à la figure 2.19.b. La première instruction réalise l'opération MAC et stocke le résultat au sein du registre d'accumulation A. Les deux autres opérations permettent de vieillir les échantillons en réalisant un transfert entre les deux données situées en mémoire à l'aide du registre d'accumulation B. Pour le TMS320C50, le code généré est présenté à la figure 2.19.c, les deux premières instructions et l'instruction *apac* réalisent l'opération de type MAC, Ensuite, les deux dernières instructions réalisent le "vieillissement" des échantillons, Ce transfert entre deux données situées en mémoire utilise le registre d'accumulation. Ce registre étant utilisé pour stocker le résultat intermédiaire de la suite d'accumulations, il est nécessaire de renvoyer le résultat intermédiaire de l'accumulation en mémoire avant de réaliser le transfert. Ce renvoi nécessite 4 cycles si ce résultat est stocké en double précision (32 bits) et 2 cycles si le résultat est stocké en simple précision (16 bits).

Ainsi, la phase d'allocation de registres va influencer sur la précision des calculs au travers du renvoi des résultats intermédiaires en mémoire. La méthodologie devra déterminer le format des données renvoyées en mémoire en fonction de la précision souhaitée en sortie de l'algorithme. L'objectif de la génération de code étant de minimiser le temps d'exécution et la taille du code, le surcoût des transferts des données entre l'unité de traitement et la mémoire sera minimisé tant que la précision des calculs en virgule fixe permet de satisfaire les critères de qualité associés à l'application. Ainsi, la méthodologie cherchera à renvoyer les données en mémoire avec une largeur correspondant à la largeur naturelle du processeur afin d'avoir un surcoût de transfert minimal. De plus, l'évaluation de la précision réelle d'une implantation doit tenir compte de cette phase d'allocation de registres. La précision évaluée avant cette phase ne correspondra à la précision réelle que si les données intermédiaires renvoyées en mémoire dans le cadre du phénomène de *spilling*, sont stockées en mémoire avec la même précision que dans l'unité de traitement.

Ordonnancement

La phase d'ordonnancement détermine l'instant d'exécution des différentes instructions. Ces instants sont définis afin de minimiser le temps d'exécution du code tout en respectant la sémantique de celui-ci. Pour les processeurs possédant du parallélisme au niveau instruction tels que les processeurs VLIW, la phase d'ordonnancement nécessite de réaliser en plus un compactage du code. Il permet de regrouper des instructions partielles pouvant s'exécuter en parallèle. L'objectif de notre méthodologie étant de minimiser le temps d'exécution du code sous contrainte de respect des critères de qualité associés à l'application, les opérations de recadrage

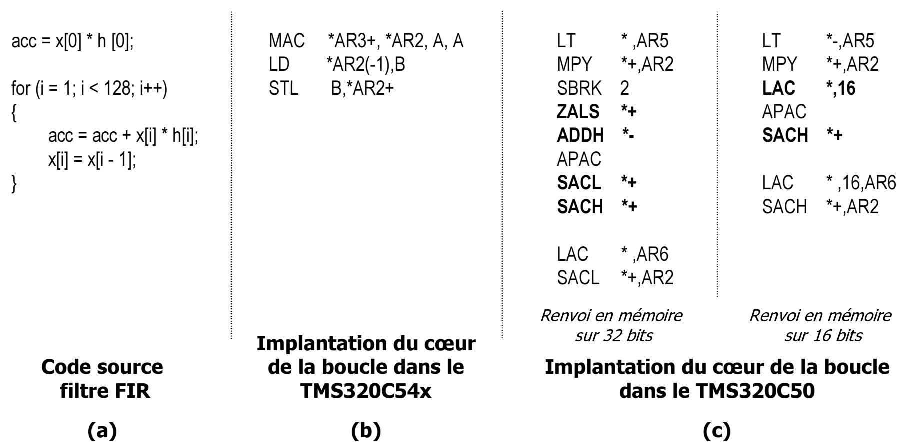


Fig. 2.19: Analyse de l'influence du *spilling* et de la largeur des données renvoyées en mémoire sur le temps d'exécution du code. Exemple d'implantation d'un filtre FIR au sein des DSP TMS320C50 et TMS320C54x

coûteuses en termes de temps d'exécution devront être déplacées. Pour les processeurs possédant du parallélisme au niveau instruction le coût d'un recadrage dépend de l'opportunité d'exécuter cette opération en parallèle avec les autres instructions. Ainsi, ce coût est lié à la manière dont les instructions sont mises en parallèle lors de la phase d'ordonnancement.

Pour illustrer et quantifier ce phénomène, différentes applications ont été implantées au sein du processeur VLIW TMS320C62x. Ces applications sont basées sur un cœur de traitement intégrant des opérations de type multiplication-accumulation (MAC). Cette suite d'opérations MAC, nécessite de recadrer les données avant l'accumulation afin d'éviter la présence d'un débordement. Nous avons évalué le coût d'une opération de recadrage située au sein du cœur de traitement de chaque application entre les opérations de multiplication et d'addition.

Pour évaluer le surcoût de cette opération de recadrage nous avons mesuré les temps d'exécution t_{ar} et t_{sr} correspondant respectivement aux temps d'exécution de l'application avec et sans opération de recadrage. La différence entre ces deux temps d'exécution représente le temps consacré à l'opération de recadrage. Afin de comparer les surcoûts entre les applications et pour évaluer le temps d'exécution de l'opération de recadrage en regard du temps d'exécution de l'application nous avons défini la métrique $\mathcal{C}_r$ dont l'expression est la suivante :

$$\mathcal{C}_r = \frac{t_{ar} - t_{sr}}{t_{sr}} \tag{2.3}$$

Cette métrique indique le pourcentage de temps d'exécution supplémentaire si une opération de recadrage est insérée. Les résultats obtenus pour les différentes applications implantées sont présentés dans le tableau 2.4 [100]. De plus, nous avons indiqué le niveau de parallélisme moyen (IPC) obtenu au niveau du code du cœur de la boucle implanté sans opération de recadrage. Cette seconde métrique indique le nombre moyen d'instructions exécutées en parallèle. Le TMS320C62 étant composé de deux *clusters* regroupant chacun 4 unités fonctionnelles, la valeur maximale de l'IPC est égale à 8.

Pour obtenir un niveau de parallélisme important, le compilateur a mis en œuvre des techniques basées sur le déroulage de boucle et le pipeline logiciel. Elles ont permis d'atteindre des valeurs de l'IPC proches de la valeur maximale. L'instruction de décalage peut être mise en parallèle aux autres instructions si l'unité fonctionnelle réalisant l'opération de décalage est libre et si les contraintes au niveau des dépendances entre les données ne sont pas trop restrictives. Les résultats montrent que plus l'IPC du code sans opération de décalage est élevé, plus le surcoût va être important. En effet, dans ce cas, les unités fonctionnelles étant fortement utilisées, peu d'opportunités sont présentes pour réaliser l'opération de décalage en parallèle aux

Applications	FIR		FIR symétrique		FIR complexe		IIR 2
	T.E.	T.B.	T.E.	T.B.	T.E.	T.B.	*T.B.*
IPC	7.5	6	7.2	7.4	6.5	4.875	2.8
$\mathcal{C}_r$ (%)	47	22	47	35	45	0	18

T.E. : Traitement par échantillon, T.B. : Traitement par bloc

TAB. 2.4: Surcoût du recadrage interne ($\mathcal{C}_r$) pour différentes applications

autres opérations et des cycles supplémentaires sont nécessaires pour exécuter les opérations de décalage. Lorsque l'IPC est plus faible le surcoût est diminué et il peut devenir nul.

Ces résultats montrent que pour les architectures permettant de réaliser du parallélisme au niveau instruction, le surcoût lié à une opération de recadrage est variable et dépend de la manière dont les instructions sont ordonnancées et regroupées en parallèle. En conséquence, l'optimisation des opérations de recadrage en vue de minimiser le temps d'exécution doit prendre en compte la manière dont les instructions sont ordonnancées.

La phase d'ordonnancement va influencer le temps d'exécution des opérations de recadrage réalisées à l'aide de certains registres à décalage spécialisés. Pour certains registres à décalage spécialisés, la valeur du décalage est spécifiée au travers d'un registre de mode. Ces registres sont situés en sortie d'un opérateur et ils permettent de décaler la sortie sans cycle supplémentaire. Cependant, il est nécessaire d'initialiser ce registre de mode avant d'exécuter l'instruction. Dans le pire cas, une instruction d'initialisation est requise avant chaque utilisation de l'opérateur. Si le même décalage en sortie de l'opérateur est utilisé pour plusieurs opérations alors une seule instruction d'initialisation est nécessaire et le coût des recadrages est diminué. L'obtention d'une suite d'opérations utilisant les mêmes valeurs de recadrage dépend de la manière dont les instructions sont ordonnancées. Ce phénomène a été mis en évidence par Liao dans [83, 84]. Dans le cas des architectures DSP traditionnelles possédant différentes fonctionnalités commandées par des bits de mode, des techniques d'ordonnancement permettant de minimiser le coût des instructions d'initialisation des registres de mode ont été proposées [85].

2.2.5 Conclusions

Dans cette partie, nous avons tout d'abord présenté la structure d'un compilateur et détaillé les différentes phases qui le composent. Ensuite, l'interaction entre les phases de compilation et le processus de codage des données en virgule fixe a été étudiée. Cette analyse permet de définir le couplage entre les phases de génération de code et ce processus de codage et les objectifs de certaines étapes de notre méthodologie.

Le type des différentes données de l'application doit être défini avant la phase de sélection d'instructions. Pour cela, la méthodologie sélectionne les instructions dont le type des opérandes permet de minimiser le temps d'exécution global et de fournir une précision suffisante en sortie de l'algorithme. La phase d'allocation de registres ajoute des instructions de renvoi de variables intermédiaires en mémoire. Ainsi, la méthodologie détermine le type de ces données renvoyées en mémoire selon le critère énoncé précédemment. Le coût global des transferts des données entre l'unité de traitement et la mémoire est minimisé tant que la précision en sortie de l'algorithme est suffisante. L'ordonnancement des instructions influence le coût des opérations de recadrage au sein des DSP proposant du parallélisme au niveau instruction. Ainsi, la manière dont les instructions sont ordonnancées doit être prise en compte pour déterminer le coût réel d'une opération de recadrage lors de l'optimisation du placement de ces opérations au sein de l'application.

2.3 Présentation de la méthodologie

2.3.1 Introduction

L'objectif de ce travail de recherche, est de définir une nouvelle méthodologie d'implantation d'algorithmes spécifiés en virgule flottante au sein d'architectures programmables en virgule fixe sous contrainte de respect des critères de qualité associés à l'application. Cette méthodologie cible les applications embarquées, ainsi l'implantation doit être optimisée du point de vue du coût et de la consommation d'énergie. Ceci nécessite de minimiser le temps d'exécution et la taille du code généré par le compilateur. En conséquence, la méthodologie réalise la conversion de la description en virgule flottante en une spécification en virgule fixe et optimise le temps d'exécution et la taille du code généré tant que l'implantation respecte les critères de qualité associés à l'application.

Les méthodologies existantes présentées dans le premier chapitre réalisent une transformation de la représentation des données en virgule flottante en une représentation en virgule fixe au niveau du code source. Ainsi, l'architecture du processeur cible n'est pas prise en compte au niveau de la détermination du codage des données. La minimisation du temps d'exécution des opérations de décalage prend en compte l'architecture cible à travers uniquement le temps d'exécution a priori des instructions de décalage associées à l'architecture. Les expérimentations présentées dans la partie précédente ont montré pour un processeur donné, la diversité des temps d'exécution obtenus. Ainsi, il est difficile de définir a priori pour une architecture, le temps d'exécution des opérations de recadrage. De plus, cette minimisation n'est pas réalisée sous une contrainte de précision globale de l'implantation. Elle est uniquement présente à travers la définition d'une dégradation maximale de la précision autorisée pour chaque donnée.

Par rapport aux méthodologies existantes [70, 149], la détermination et l'optimisation du codage au sein de notre méthodologie, sont réalisées sous contrainte de respect des critères de qualité en sortie de l'algorithme. De plus, l'architecture du processeur cible est entièrement prise en compte lors de ces deux phases. En effet, la description de l'architecture des DSP a permis d'énumérer les différents éléments influençant la précision des calculs. De plus, l'analyse de l'influence de l'architecture sur la précision des calculs a montré la nécessité de prendre en compte les différents éléments de l'architecture afin d'optimiser l'implantation au niveau du codage des données en virgule fixe. Cette étude permet de définir les différents éléments indispensables pour la modélisation du processeur. La présentation des différentes phases d'un compilateur et l'étude de l'interaction de celles-ci avec le processus de codage des données ont permis de définir la structure de notre méthodologie et les différents couplages avec l'infrastructure de compilation.

La structure de compilation utilisée pour mettre en œuvre notre méthodologie est composée d'une partie frontale SUIF [152] et d'un générateur de code CALIFE [21]. L'environnement SUIF inclut la partie frontale d'un compilateur permettant la transformation d'une application spécifiée en langage C vers une représentation intermédiaire (IR). L'environnement CALIFE est une structure de génération de code flexible destinée aux processeurs programmables spécialisés. Cet outil met à la disposition de l'utilisateur une bibliothèque de modules regroupant diverses passes de production et d'optimisation de code. L'utilisateur choisit et agence ces différentes passes afin d'adapter le flot de compilation à l'architecture cible. Le processeur est modélisé à l'aide du langage ARMOR.

Support de conversion

Les principales méthodologies présentées dans le premier chapitre réalisent la conversion de la description en virgule flottante en une spécification en virgule fixe au niveau du code source.

Ainsi, le support de la spécification en virgule fixe est le langage C. Ce langage ne permet pas de spécifier explicitement le format de chaque donnée en virgule fixe et les lois de quantification et de dépassement. Certaines fonctionnalités peuvent être intégrées au sein de certaines extensions du langage C, mais ces dernières ne sont pas standardisées. Ainsi, nous proposons de réaliser la conversion en virgule fixe au niveau de la représentation intermédiaire (IR). Ceci permet de spécifier directement l'application en virgule fixe au niveau de l'IR et ainsi de ne pas générer un nouveau code C pour transmettre l'application au générateur de code. De plus, la spécification de l'application au niveau de l'IR correspond exactement à la spécification qui sera implantée dans le DSP. En effet, celle-ci est obtenue après la phase d'optimisation de l'IR qui a pu modifier certaines parties de cette IR.

Les différentes phases de conversion en virgule fixe sont réalisées sur une représentation de type graphe flot de données et de contrôle (GFDC). Ainsi, la représentation intermédiaire du compilateur a été modifiée afin de pouvoir représenter l'application sous deux vues différentes. La première vue correspond à celle utilisée par le générateur de code et est issue de la partie frontale du compilateur. La seconde vue correspond à celle utilisée pour réaliser les différentes phases de conversion. Des liens entre les deux vues sont mis en œuvre au niveau des éléments représentant les structures de contrôle et le traitement des données.

La vue générale de notre méthodologie est présentée à la figure 2.20. La méthodologie se décompose en deux grandes parties. La partie correspondant à la conversion de la description en virgule flottante en une spécification en virgule fixe, regroupe les phases de détermination et d'optimisation du codage des données. L'implantation doit respecter les différents critères de qualité associés à l'application, ainsi, il est nécessaire d'évaluer la qualité de l'implantation au regard de ces critères.

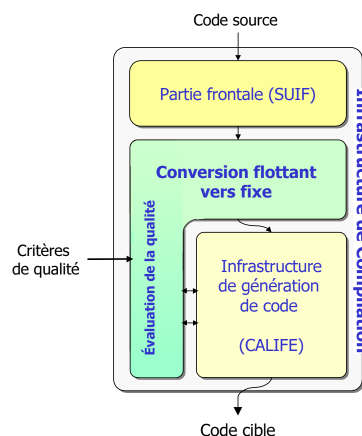


FIG. 2.20: Vue générale de la méthodologie de compilation d'algorithmes spécifiés en virgule flottante pour les processeurs en virgule fixe

2.3.2 Détermination et optimisation du codage des données

La première phase de la méthodologie détermine la dynamique des différentes données présentes au sein du graphe flot de données et de contrôle représentant l'application. Des méthodes analytiques ont été mises en œuvre afin de déterminer la dynamique de chaque donnée permettant de garantir l'absence de débordement. La méthode mise en œuvre permet de traiter les structures linéaires et les structures non-linéaires et non récursives.

La seconde phase correspondant à la détermination du format des données, est réalisée en deux parties. Dans un premier temps, la position de la virgule de chaque donnée est déterminée.

Ces positions sont définies à partir de la dynamique des données et des règles de l'arithmétique virgule fixe. Cette phase prend en compte la présence éventuelle de bits de garde au sein de l'architecture pour déterminer la position de la virgule des données présentes en entrée et en sortie des opérations d'addition ou de soustraction.

La seconde partie détermine ensuite, la largeur de chaque donnée et est réalisée en deux étapes. Ces largeurs sont déterminées en fonction du type des opérandes manipulés par les différentes instructions du processeur. La méthode sélectionne la séquence d'instructions permettant de fournir une précision suffisante en sortie de l'algorithme et de minimiser certains paramètres de l'implantation tels que le temps d'exécution et la taille du code. Cette première étape de la détermination de la largeur des données est réalisée avant la phase de sélection d'instructions afin que le type de chaque donnée soit défini avant de débuter le processus de génération de code. Le modèle du processeur utilisé regroupe les différentes instructions arithmétiques et de transfert de données du jeu d'instructions.

Les phases de sélection d'instructions et d'allocation de registres peuvent introduire des transferts de données entre l'unité de traitement et la mémoire. Les largeurs des données au sein de l'unité de traitement et de la mémoire étant différentes, il est nécessaire de définir si la donnée située dans l'unité de traitement doit être renvoyée en mémoire avec la même précision ou si une précision réduite peut être utilisée afin de diminuer le temps de transfert. Cette seconde étape est réalisée au cours du processus de génération de code après les phases de sélection d'instructions et d'allocation de registres.

La dernière phase du processus de codage correspond à l'optimisation du format des données en vue d'obtenir une implantation plus efficace. Les différentes opérations de recadrage sont déplacées afin de minimiser le coût global des opérations de recadrage tant que la précision en sortie de l'algorithme est satisfaisante. Cette phase nécessite de déterminer le coût de chaque opération de recadrage. Deux types de méthode sont mis en œuvre pour estimer le temps d'exécution de ces opérations de recadrage en fonction du type d'architecture. La distinction entre deux types d'architecture est réalisée en fonction des capacités de parallélisme au niveau instruction offertes par le processeur. Pour les processeurs ne proposant pas de parallélisme au niveau instruction, le temps d'exécution est déterminé à partir de la séquence d'instructions utilisée pour réaliser l'opération de recadrage. Pour les processeurs proposant du parallélisme au niveau instruction, l'estimation du temps d'exécution d'une opération de recadrage nécessite de prendre en compte la manière dont les instructions sont ordonnancées.

2.3.3 Évaluation de la qualité de l'implantation

L'objectif de notre méthodologie est d'implanter efficacement les algorithmes de traitement numérique du signal au sein des processeurs programmables. D'un point de vue de l'application, l'implantation doit satisfaire les contraintes de qualité propres à l'application. Ces critères de qualité sont étroitement liés au domaine de l'application. Par exemple, dans le domaine des transmissions numériques, le critère le plus utilisé est le taux d'erreur binaire et dans une moindre mesure, l'ouverture du diagramme de l'œil.

Les phases de détermination du type des données et d'optimisation du codage sont réalisées sous contrainte de respect des critères de qualité associés à l'application. Ces phases nécessitent d'explorer l'espace des formats des données afin de trouver la solution optimale permettant de minimiser certains paramètres de l'implantation et de respecter les contraintes de qualité. Ainsi, la vérification du respect de ces contraintes de qualité doit être réalisée de multiples fois. Certaines méthodologies permettent de vérifier si ces contraintes de qualité sont satisfaites à travers des simulations de l'implantation en virgule fixe. Cependant, coupler au sein des simulations la vérification de l'ensemble des critères de qualité de l'application et l'exploration

des différents formats virgule fixe conduit à des temps de simulation pouvant être prohibitifs.

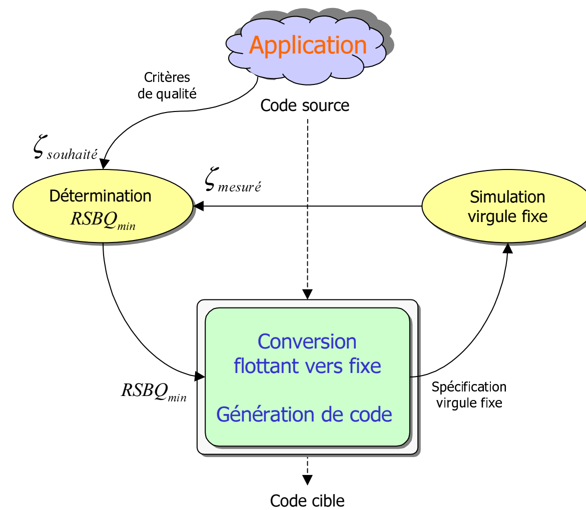


FIG. 2.21: Représentation du processus de détermination de la contrainte de précision ($RSBQ_{min}$)

Pour diminuer les temps d'optimisation des formats, nous mettons en œuvre une alternative aux méthodes basées sur la simulation en utilisant une méthode analytique. Cette dernière détermine tout d'abord l'expression analytique d'une métrique et utilise ensuite cette expression au sein du processus d'optimisation afin d'accélérer l'évaluation de cette métrique. Cette technique nécessite de mettre en œuvre une méthodologie permettant de déterminer automatiquement l'expression de cette métrique en fonction du format des données en virgule fixe. Cette méthodologie ne peut déterminer une métrique représentative, propre à chaque domaine d'application. Ainsi, une unique métrique est définie pour évaluer la qualité de l'implantation. La métrique la plus représentative de la dégradation de la précision liée à l'arithmétique virgule fixe est le rapport signal à bruit de quantification (RSBQ). Notre approche nécessite dans un premier temps de définir la valeur minimale du RSBQ ($RSBQ_{min}$) permettant de respecter les différents critères de qualité associés à l'application et ensuite les formats des données sont déterminés et optimisés tant que la précision de l'implantation est supérieure à la précision minimale ($RSBQ_{min}$). La spécification en virgule fixe obtenue peut être simulée afin de vérifier si réellement les critères de qualité sont vérifiés. Si la qualité de l'implantation ne correspond pas exactement aux critères requis, la valeur du $RSBQ_{min}$ est affinée et une nouvelle implantation est réalisée. Cette approche nécessite de ne réaliser uniquement que quelques simulations en virgule fixe. Le processus de détermination de la contrainte de précision ($RSBQ_{min}$) est schématisé est la figure 2.21.

Cependant, pour les systèmes linéaires, l'évaluation de la précision ne permet pas de vérifier l'ensemble des performances associées au système. En particulier, pour les filtres linéaires, le format des coefficients modifie la réponse fréquentielle du filtre. Ainsi, le codage de ces coefficients doit être réalisé en vue de respecter la contrainte sur la déviation maximale autorisée entre la fonction de transfert en précision infinie et en précision finie. En conséquence, cette contrainte doit être ajoutée à la contrainte sur la précision minimale lors de la détermination et l'optimisation du format des données du système. Dans le cadre de notre méthodologie, cette contrainte sur la déviation fréquentielle peut être intégrée aisément car la fonction de transfert en précision finie du système est obtenue lors de la phase de détermination de l'expression analytique du RSBQ.

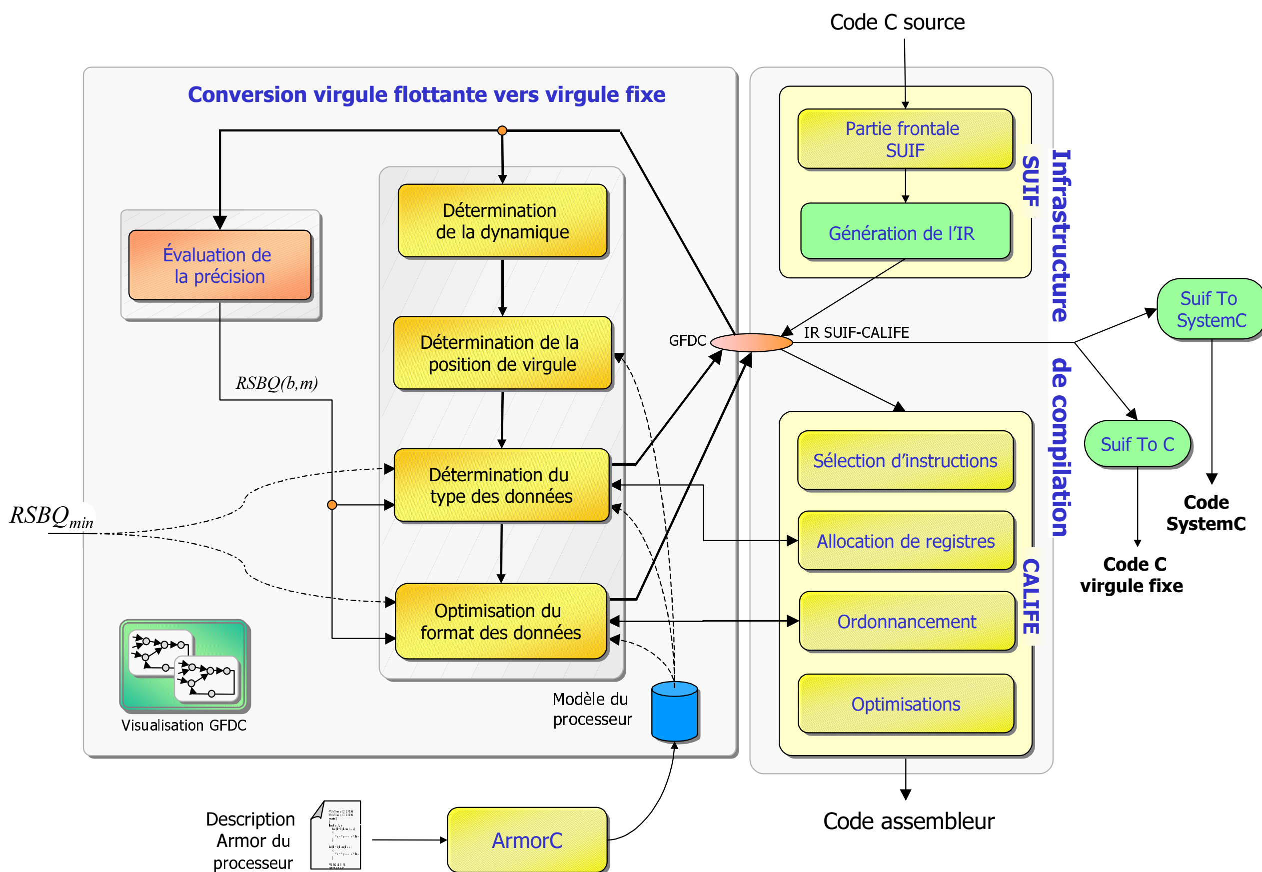


FIG. 2.22: Vue générale détaillée de la méthodologie de compilation d'algorithmes spécifiés en virgule flottante pour les processeurs en virgule fixe

2.3.4 Conclusions

Dans cette dernière partie, la structure de la méthodologie a été exposée. Le synoptique détaillé de celle-ci est présenté à la figure 2.22. Dans le chapitre suivant, la méthodologie proposée pour évaluer la précision de l'implantation est exposée. Les différentes phases du processus de conversion de la description en virgule flottante en une spécification en virgule fixe sont détaillées dans le chapitre 4.

Chapitre 3

Évaluation de la précision d'un système spécifié en virgule fixe

3.1 Introduction

L'utilisation de l'arithmétique virgule fixe se traduit par une erreur entre le résultat calculé en précision finie et celui obtenu en précision infinie. Ainsi, l'implantation d'une application en virgule fixe n'est viable que si malgré la présence de cette erreur, les critères de qualité associés à cette application sont respectés. Dans la dernière partie du chapitre précédent, nous avons défini la démarche utilisée pour vérifier le respect de ces critères de qualité. Cette vérification s'effectue en deux temps et utilise une métrique permettant d'évaluer la précision de l'implantation en précision finie. Cette métrique correspond au Rapport Signal à Bruit de Quantification (RSBQ). Dans un premier temps, la contrainte de précision minimale ($RSBQ_{min}$) permettant de respecter les critères de qualité est définie. Ensuite, le processus de codage optimise le format des données afin que la précision de l'implantation soit supérieure à la précision minimale. Cette précision est évaluée en déterminant le RSBQ. Cette métrique correspond au rapport entre la puissance du signal P_s et la puissance du bruit de quantification P_{b_s} associé à ce signal et lié à l'utilisation d'une arithmétique en précision finie :

$$RSBQ = \frac{P_s}{P_{b_s}} \tag{3.1}$$

L'évaluation du RSBQ en sortie de l'algorithme nécessite de déterminer la puissance P_y du signal en sortie et la puissance P_{b_y} du bruit de quantification présent en sortie de l'algorithme. La puissance du signal dépend de la nature des signaux présents en entrée et est indépendante de la manière dont les données de l'application sont codées en virgule fixe. En revanche, la puissance du bruit de quantification P_{b_y} est directement liée au format des données en virgule fixe. Ainsi, l'évaluation de la précision d'un système en virgule fixe consiste essentiellement à déterminer la puissance du bruit de quantification.

Un état de l'art des méthodologies disponibles pour évaluer la précision d'un système en virgule fixe a été présenté dans la partie 1.2.3 à la page 23. Deux types de méthode peuvent être considérés pour évaluer cette précision. La précision peut être évaluée à l'aide d'une simulation de l'algorithme en virgule fixe. Cependant, le nombre important d'échantillons utilisés pour la simulation, combiné à l'accroissement du temps d'exécution lié à l'émulation des mécanismes de l'arithmétique virgule fixe, conduit à des temps de simulation relativement longs. De plus, les processus de détermination et d'optimisation du format des données nécessitent d'explorer l'espace des différents formats de données. Pour les méthodes basées sur la simulation, l'exploration du format des données est réalisée par l'intermédiaire d'un processus itératif. Cette

technique nécessite de simuler une nouvelle fois l'algorithme en virgule fixe dès que le format d'une donnée est modifié. Ainsi, les temps d'optimisation peuvent devenir prohibitifs.

Les méthodes utilisant une approche analytique représentent une alternative aux méthodes basées sur la simulation. Ces méthodes permettent de déterminer l'expression analytique de la puissance du bruit de quantification en sortie d'un système. La connaissance de l'expression analytique du RSBQ fournit plus d'informations sur le comportement du bruit au sein de l'algorithme. Ainsi, ce type de méthode permet d'analyser plus efficacement l'influence de la structure utilisée pour implanter l'algorithme et facilite le choix d'une structure optimale. De plus, le temps requis pour l'évaluation de la puissance du bruit est nettement plus réduit, en particulier en ce qui concerne le processus d'optimisation du format des données. La détermination de l'expression du RSBQ en fonction du format des données n'est réalisée qu'une seule fois. Dans ce cas, la position de la virgule des données et la largeur des données sont considérées comme des variables. Ensuite, le temps requis pour l'évaluation de la précision est très faible. Il correspond au temps nécessaire pour évaluer une expression mathématique.

Au sein de notre méthodologie de conversion en virgule fixe, les processus de détermination de la largeur des données et d'optimisation du format des données requièrent de multiples évaluations de la précision de l'implantation. Ainsi, nous nous sommes orientés vers la définition d'une nouvelle méthodologie de détermination automatique de l'expression analytique du RSBQ. De plus, ce type de méthode pourra être utilisé avantageusement dans le cadre de la conception matérielle pour l'optimisation de la largeur des éléments du chemin de données.

Ce type de méthode nécessite de modéliser précisément le bruit généré lors de la quantification d'un signal à amplitude continue et à amplitude discrète. Nous présentons dans la partie 3.2 de ce chapitre les modèles de bruit présents dans la littérature. Nous avons proposé un nouveau modèle de bruit issu du raffinement du modèle du bruit de la quantification d'un signal à amplitude discrète. Ce raffinement est issu de l'étude précise de la probabilité de la valeur des bits composant une donnée.

Notre méthodologie permet de traiter les systèmes linéaires invariants dans le temps (coefficients constants) récursifs et non récursifs et les systèmes non-linéaires et non-récursifs. Les concepts théoriques associés à cette méthodologie sont présentés dans la troisième partie de ce chapitre. Dans le cadre des systèmes linéaires, la notion de fonction de transfert est utilisée afin de pouvoir traiter les différents types de structure.

La méthodologie mise en œuvre pour obtenir les différents éléments nécessaires à la détermination de l'expression analytique du RSBQ est décrite dans la quatrième partie. Cette méthodologie est basée sur une suite de transformations du graphe représentant l'application.

Nous avons évalué la qualité de l'estimation de la puissance du bruit basée sur cette méthodologie dans le cadre des systèmes linéaires et des systèmes non-récursifs et non-linéaires. Ces résultats sont présentés dans la cinquième partie. Ils permettent de vérifier que la précision obtenue avec ces estimateurs est nettement suffisante dans le cadre de notre application. De plus, ces résultats démontrent l'aptitude de notre méthodologie à résoudre le problème de la détermination automatique de l'expression du RSBQ. En outre, l'efficacité de notre méthodologie a été évaluée en mesurant le temps d'exécution de l'outil développé pour implanter celle-ci.

La dernière partie de ce chapitre est consacrée à la méthode mise en œuvre pour déterminer la contrainte de précision minimale en fonction des différents critères de qualité associés à l'application.

3.2 Modélisation du bruit

La quantification d'un signal génère une erreur qui est ensuite propagée au sein du système. Ainsi, l'objectif de cette partie est de modéliser le bruit généré lors de la quantification d'un signal et le processus de propagation de ces bruits au sein des opérateurs arithmétiques. Le signal quantifié peut être à amplitude continue, comme dans le cas du processus de conversion analogique-numérique ou à amplitude discrète comme lors d'une opération de changement de format (recadrage). Dans ce cas, le nombre de bits réservés pour la partie fractionnaire est réduit et certains bits sont éliminés.

Dans cette partie, les différentes modélisations du bruit issu de la quantification d'un signal à amplitude continue ou discrète, proposées dans la littérature, sont présentées. Un nouveau modèle pour la quantification d'un signal à amplitude discrète, prenant en compte le nombre de bits éliminés et leur probabilité, est proposé. Les résultats des simulations présentés dans la partie 3.5.1 soulignent l'apport de notre modèle pour obtenir des estimations plus précises. Enfin, les modèles de propagation du bruit au sein des différents opérateurs arithmétiques sont détaillés.

3.2.1 Modélisation du bruit généré

État de l'art de la modélisation du bruit de quantification

Quantification d'un signal à amplitude continue : Un modèle classique de quantification d'un signal à amplitude continue a été proposé par Widrow [145, 146, 147] puis affiné par Snyder et Sripad [127]. Ce modèle a été détaillé dans la partie 1.1.4 à la page 13. Le signal y issu de la quantification du signal x est modélisé par la somme de x et d'une variable aléatoire b. Ce bruit additif b est distribué uniformément sur son intervalle de définition et est non corrélé avec le signal d'entrée x et les autres bruits de quantification. De plus, le spectre du bruit b est blanc.

Widrow [147] a montré que ce modèle est valide si la fonction caractéristique du signal d'entrée est à bande limitée. Snyder et Sripad [127] ont proposé deux conditions moins restrictives sur la fonction caractéristique. Cependant, ces conditions ne sont pas respectées par certains types de signaux tels que le bruit gaussien. Mais il a été montré que ce modèle est correct dès que la dynamique du signal d'entrée est suffisamment grande devant le pas de quantification et si la largeur du spectre du signal est suffisamment grande.

Quantification d'un signal à amplitude discrète : Le modèle présenté ci-dessus a été étendu à la modélisation du bruit issu de la quantification d'un signal à amplitude discrète. Dans [10, 155, 156, 140], les auteurs montrent les limites du modèle et en particulier au niveau de l'erreur issue de la quantification du résultat de la multiplication d'un signal par une constante. Ils démontrent que pour certaines valeurs de la constante multiplicatrice, le comportement statistique de cette erreur est fonction de la valeur de la constante.

Dans [155], l'étude porte sur les propriétés de l'erreur issue de la quantification du résultat de la multiplication d'une constante $a = L/M$ par un signal quantifié, d'amplitude constante. La décomposition en série de Fourier de l'erreur de quantification est utilisée pour exprimer ses moments du premier et du second ordre. Les auteurs donnent les conditions nécessaires et suffisantes pour que l'erreur de quantification possède une distribution uniforme, un spectre blanc et qu'elle soit non corrélée avec l'erreur de quantification associé à l'entrée. Ces conditions sur la fonction caractéristique du signal d'entrée sont reliées au paramètre M associé au coefficient constant a. Dans ce cas, l'erreur est distribuée uniformément sur l'intervalle $[-\Lambda, \Lambda]$ défini de la manière suivante :

$$\Lambda = \frac{q}{2}\sqrt{1 - \frac{1}{M^2}} \tag{3.2}$$

Une approche similaire est utilisée par Barnes dans [10] pour analyser l'erreur issue de la quantification du résultat de la multiplication d'une constante $a = L/M$ par un signal d'amplitude discrète x. L'expression des différents moments de l'erreur de quantification peut se simplifier si la dynamique du signal d'entrée est suffisamment grande. Dans ce cas, les expressions des moments du premier et du second ordre de l'erreur sont présentées aux équations 3.3 et 3.4. De plus, l'erreur est non corrélée avec le signal d'entrée du multiplieur ou avec d'autres erreurs de quantification. La condition sur la dynamique dépend du coefficient a. La variance du signal d'entrée doit par conséquent respecter la condition $\sigma_x \geq M\frac{q}{2}$.

$$\mu_e = \frac{q}{2M} \tag{3.3}$$

$$\sigma_e^2 = \frac{q^2}{12}\left(1 - \frac{1}{M^2}\right) \tag{3.4}$$

Dans [24], un modèle basé sur une distribution discrète de l'erreur de quantification est proposé. En effet, les modèles classiques présentés ci-dessus et basés sur une distribution continue de l'erreur de quantification ne sont plus valides lorsque le nombre de bits éliminés devient faible. L'estimation de l'erreur de quantification basée sur une distribution continue de cette erreur est erronée si le nombre de bits éliminés est inférieur à 6 bits. Le nombre de bits éliminés lors d'un changement de format pouvant être faible, la modélisation basée sur une distribution discrète de l'erreur de quantification et intégrant le nombre de bits éliminés a été retenue. Cependant, le modèle proposé dans [24] se base sur l'hypothèse a priori que les différentes valeurs possibles de l'erreur de quantification résultant d'une opération de changement de format sont équiprobables. Ceci implique que tous les bits éliminés durant cette opération présentent une probabilité d'être égaux à 1 de 1/2. Cette hypothèse est étudiée dans la section qui suit, où nous démontrons qu'elle n'est pas systématiquement valide et en particulier lors de la quantification du résultat de la multiplication d'un signal par une constante. Ainsi, pour obtenir un modèle précis de l'erreur de quantification il est nécessaire de prendre en compte la nature de la donnée quantifiée et plus particulièrement le type de l'opération ayant générée cette donnée. Ainsi, pour chaque type d'opération un modèle de bruit est proposé. Celui-ci résulte de l'étude de la probabilité des différents bits de la donnée correspondant à la sortie de l'opérateur.

Tout d'abord, pour les différents types d'opération arithmétique, l'expression de la probabilité des bits composant la sortie de cet opérateur est définie. A partir d'un modèle de probabilité pour les signaux et les constantes, la probabilité des bits de la donnée représentant la sortie est déterminée. Ensuite, les moments du premier et du second ordre sont déterminés à partir des probabilités des bits éliminés.

Modélisation des opérateurs arithmétiques

L'objectif de cette section est de déterminer pour les différents opérateurs arithmétiques classiques utilisés au sein des processeurs de traitement numérique du signal, l'expression de la probabilité des bits présents en sortie. Ces opérateurs sont constitués d'opérateurs élémentaires dont la largeur des entrées et de la sortie est égale à 1 bit.

Opérateurs élémentaires : Les opérateurs arithmétiques élémentaires étudiés dans cette section sont le multiplieur, l'additionneur et le soustracteur. La représentation des opérateurs

et les équations logiques des sorties sont données à la figure 3.1. Nous définissons p_x la probabilité que le bit x soit égal à 1. La probabilité pour que x soit égal à 0 est alors $1 - p_x$.

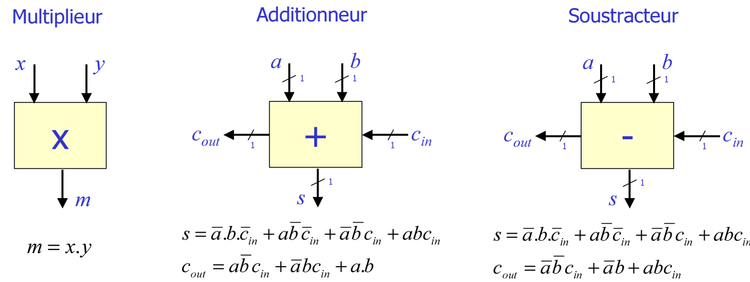


FIG. 3.1: Représentation des opérateurs arithmétiques élémentaires

Considérons un multiplieur, constitué de deux entrées x et y. Soit m la sortie du multiplieur. L'expression de la probabilité P_m de la sortie du multiplieur, obtenue à partir de l'expression logique de m, est définie comme suit :

$$P_m\left(p_x, p_y\right) = p_x p_y \tag{3.5}$$

Considérons un additionneur et un soustracteur constitués de 2 entrées a et b et d'une entrée c_{in} correspondant à la retenue. Soit s la sortie de l'opérateur et c_{out} la retenue sortante. Les équations logiques en sortie de l'additionneur et du soustracteur étant identiques, l'expression de la probabilité P_s en sortie de l'additionneur ou du soustracteur est définie comme suit :

$$P_s\left(p_a, p_b, p_{c_{in}}\right) = p_{c_{in}}\left(p_a p_b + (1-p_b)(1-p_a)\right) + \left(1 - p_{c_{in}}\right)\left(p_a(1-p_b) + p_b(1-p_a)\right) \tag{3.6}$$

Les expressions de la probabilité de la retenue sortante pour l'additionneur et le soustracteur sont les suivantes :

$$P_{c_{out.add}}\left(p_a, p_b, p_{c_{in}}\right) = p_{c_{in}}\left(p_a(1-p_b) + p_b(1-p_a)\right) + p_a p_b \tag{3.7}$$

$$P_{c_{out.sub}}\left(p_a, p_b, p_{c_{in}}\right) = p_{c_{in}}\left(p_a p_b + (1-p_b)(1-p_a)\right) + p_b(1-p_a) \tag{3.8}$$

Opérateurs arithmétiques : Considérons l'addition de 2 données en complément à deux $x = (x_N, \ldots, x_1, x_0)$ et $y = (y_N, \ldots, y_1, y_0)$ composées chacune de $N+1$ bits. La décomposition de cette opération au moyen des opérateurs arithmétiques élémentaires présentés ci-dessus est détaillée à la figure 3.2. L'expression de la probabilité des bits de sortie z_k est égale à :

$$p_{z_k} = P_s\left(p_{x_k}, p_{y_k}, p_{c_k}\right) \qquad \forall j < k \tag{3.9}$$

La probabilité de la retenue p_{c_k} est définie à partir de la retenue précédente à l'aide de la suite numérique suivante :

$$\begin{cases} p_{c_0} = 0 \\ p_{c_{k+1}} = P_{C_{out.add}}\left(p_{x_k}, p_{y_k}, p_{c_k}\right) \end{cases} \tag{3.10}$$

Considérons la multiplication de 2 données en complément à deux $x = (x_N, \ldots, x_1, x_0)$ et $y = (y_N, \ldots, y_1, y_0)$ composées chacune de $N+1$ bits. Cette multiplication fournit un résultat

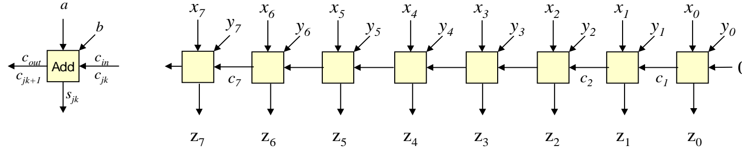


FIG. 3.2: Représentation à base d'opérateurs élémentaires d'une addition en CA2

$z = (z_{N_m}, \ldots, z_1, z_0)$ dont le nombre de bits est égal à $2(N+1)$. La décomposition de cette opération au moyen des opérateurs arithmétiques élémentaires présentés ci-dessus est détaillée à la figure 3.3 [113].

La probabilité du $k^{ème}$ bit de la sortie du multiplieur peut être définie de manière récursive à l'aide d'une suite numérique. Le premier élément $p_{s_{j_0 k}}$, l'élément intermédiaire $p_{s_{jk}}$ et le dernier élément p_{z_k} de la suite numérique sont définis comme suit pour les différentes valeurs de k :

$$
\begin{aligned}
&k = 0 \qquad p_{z_k} = P_m(p_{x_0} p_{y_0}) \\[1em]
&\forall\ k \in [1, N-1] \\
&\quad \begin{cases}
p_{s_{1k}} = P_s\left(P_m(p_{x_k} p_{y_0}), P_m(p_{x_{k-1}} p_{y_1}), p_{c_{1k}}\right) & \\
p_{s_{jk}} = P_s\left(p_{s_{j-1k}}, P_m(p_{x_{k-j}} p_{y_j}), p_{c_{jk}}\right) & 1 < j < k \\
p_{z_k} = P_s\left(p_{s_{k-1k}}, P_m(p_{x_0} p_{y_k}), 0\right) &
\end{cases} \\[1em]
&\forall\ k \in [N, 2(N-1)] \\
&\quad \begin{cases}
p_{s_{j_0 k}} = P_s(p_{c_{j_0-1k}}, P_m(p_{x_{k-j_0}} p_{y_{j_0}}), p_{c_{j_0 k}}) & \text{avec } j_0 = k - N + 1 \text{ et } p_{c_{0N}} = 0 \\
p_{s_{jk}} = P_s\left(p_{s_{j-1k}}, P_m(p_{x_{k-j}} p_{y_j}), p_{c_{jk}}\right) & j_0 < j < N+1 \\
p_{z_k} = P_s\left(p_{s_{Nk}}, P_m(p_{x_N} p_{y_{k-N}}), p_{c_{N+1k}}\right) &
\end{cases} \\[1em]
&k = 2N-1 \qquad p_{z_k} = P_s\left(p_{c_{N-1k}}, P_m(x_{N-1} y_N), p_{c_{Nk}}\right) + P_s\left(p_{s_{Nk}}, P_m(x_N y_{N-1}), p_{c_{N+1k}}\right)
\end{aligned}
\tag{3.11}
$$

Les expressions des différentes retenues peuvent être modélisées à l'aide des suites numériques en utilisant la même démarche.

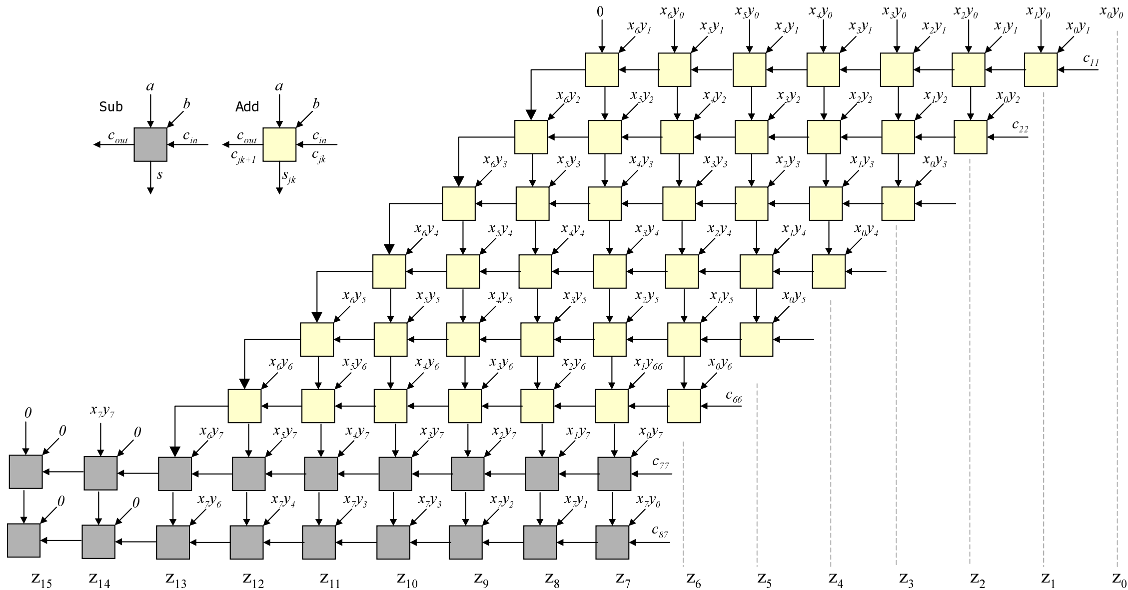


FIG. 3.3: Représentation à base d'opérateurs élémentaires d'une multiplication en CA2

D'après l'équation 3.6, la probabilité de la sortie d'un additionneur est nulle si la probabilité

des deux entrées et de la retenue en entrée est nulle. Ainsi, la probabilité p_{z_k} est égale à 0 si la probabilité de tous les produits $P_m(x_{k-j}y_j)$ et de toutes les retenues en entrée $p_{c_{jk}}$ est nulle :

$$\begin{cases} \forall i \in [1,\ k],\ \forall j \in [1,\ i] & p_{c_{ji}} = 0 \\ \forall i \in [0,\ k],\ \forall j \in [0,\ i] & P_m(x_{i-j}y_j) = 0 \end{cases} \quad \Rightarrow \quad p_{z_k} = 0 \tag{3.12}$$

Étant donné que $P_s\left(1/2, p_b, p_{c_{in}}\right) = 1/2$, alors $p_{z_k} = 1/2$ s'il existe une sortie d'additionneur ou de soustracteur s_{jk} dont la probabilité est égale à 1/2. Par conséquent, si la probabilité d'un produit $P_m(x_{k-j}y_j)$ est égale à 1/2, alors la probabilité de la sortie de l'additionneur ou du soustracteur dont une des entrées correspond au produit $x_{k-j}y_j$, est égale à 1/2. Ainsi, nous avons la relation suivante :

$$\exists\ P_m(x_{k-j}y_j) = 1/2 \quad \Rightarrow \quad p_{z_k} = 1/2 \tag{3.13}$$

Les différentes expressions définies dans ce paragraphe sont utilisées dans le paragraphe suivant pour déterminer la valeur de la probabilité des bits du résultat d'une opération arithmétique.

Modélisation du signal

L'objectif de cette section est de déterminer la probabilité des bits du résultat d'une opération arithmétique. Dans un premier temps, le modèle de probabilité des bits pour un signal d'entrée est défini. Ensuite, les cas de la multiplication d'un signal par une constante, de la multiplication et de l'addition de deux signaux sont traités.

Signal : L'objectif de ce paragraphe est de définir la probabilité des bits composant un signal quelconque. Un modèle de bruit blanc uniforme [73] est utilisé pour décrire l'activité des bits au sein d'une donnée. Ce modèle implique pour chaque bit que les probabilités pour que ce bit soit égal à 0 ou à 1 soient identiques. De plus, le modèle suppose que tous les bits sont indépendants d'un point de vue temporel et spatial. Dans [123], l'auteur montre la validité de ce type de modèle pour les bits les moins significatifs (LSB) d'un signal. Des expérimentations ont été menées pour évaluer les probabilités des bits de données pour différents types de signaux. Les résultats sont présentés à la figure 3.4. La probabilité du bit de signe p_{S_x} dépend de la fonction de distribution de la variable telle que définie ci-dessous :

$$p_{S_x} = \int_{-\infty}^{0} f_x(x)dx \tag{3.14}$$

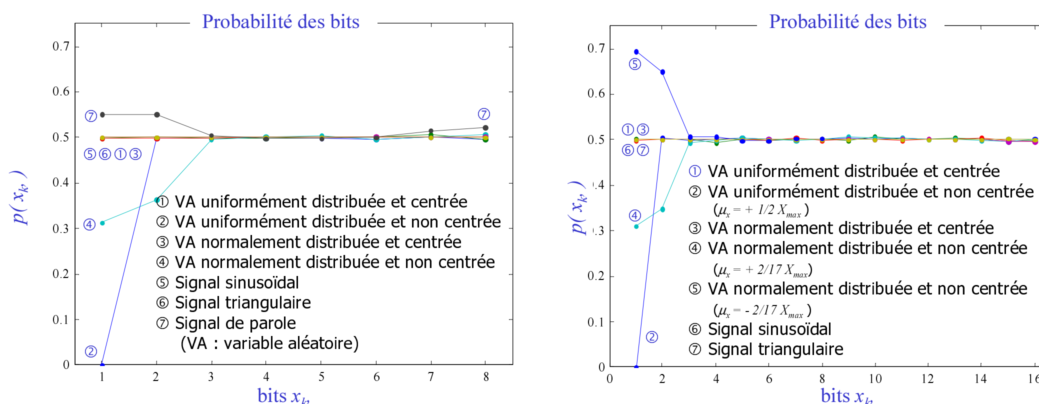


FIG. 3.4: Probabilités des bits d'une donnée pour différents signaux

Si la fonction de distribution de la variable est symétrique autour de 0, alors la probabilité du bit de signe est égale à 1/2. Pour tous les types de signaux testés, la probabilité des bits les moins significatifs est proche de 1/2. Ainsi, un paramètre l_x est introduit pour définir la position par rapport au bit le moins significatif (LSB), du dernier bit x_{l_x} dont la probabilité est égale à 1/2. Ainsi, le modèle de probabilité du signal est défini par l'équation 3.15 et sa représentation est donnée à la figure 3.5. Ce paramètre l_x dépend du type de signal. Les observations faites sur les probabilités des bits de données sont relativement similaires à celles associées au modèle DBT [74, 73].

$$\begin{array}{ll} p_{x_i} = 1/2 & \forall\, i \in [0,\ l_x] \\ p_{x_i} \neq 1/2 & \forall\, i \in [l_x+1,\ N] \end{array} \qquad (3.15)$$

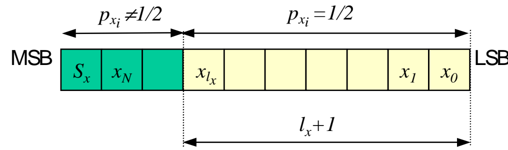


FIG. 3.5: Modélisation des données

Multiplication d'un signal x par une constante C : L'objectif est de déterminer la probabilité des bits du résultat de la multiplication d'un signal x par une constante C. Pour illustrer la démarche suivie, un exemple est présenté à la figure 3.7. La structure de la constante C de largeur $N+1$ bits est représentée à la figure 3.6. Soit l_1, la position par rapport au bit le moins significatif (LSB) du premier bit égal à 1 et l_2 la position par rapport au LSB du dernier bit égal à 1. Les bits $(C_{N+1}, \ldots, C_{l_2+1})$ et $(C_{l_1-1}, \ldots, C_0)$ sont alors égaux à 0. Dans le cas d'une constante négative, le paramètre l_2 est égal à N. La probabilité p_{C_i} des différents bits est résumée par l'équation 3.16.

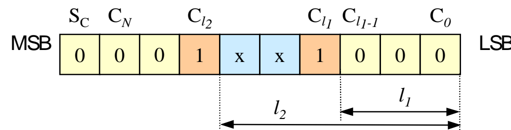


FIG. 3.6: Modélisation d'une constante C

$$p_{C_i} = \begin{cases} 0 & \text{si } i \in [0, l_1 - 1] \bigcup [l_2+1, N+1] \\ 1 & \text{si } i = l_1 \text{ ou } i = l_2 \\ p_i & \text{si } i \in [l_1+1, l_2-1] \quad \text{avec } p_i = 1 \text{ ou } 0 \end{cases} \qquad (3.16)$$

D'après l'équation 3.5 et l'expression de la probabilité des bits C_i, la probabilité du produit de x_j par C_i est égale à :

$$p_{x_j C_i} = \begin{cases} 0 & \text{si } i \in [0, l_1 - 1] \bigcup [l_2+1, N+1] \\ p_{x_j} & \text{si } i = l_1 \text{ ou } i = l_2 \\ p_{x_j}.p_i & \text{si } i \in [l_1+1, l_2-1] \quad \text{avec } p_i = 1 \text{ ou } 0 \end{cases} \qquad (3.17)$$

Étant donné que la condition exprimée dans l'équation 3.12 est respectée pour les l_1 premiers bits de sortie du multiplieur, la probabilité de ces bits est alors égale 0.

D'après l'équation 3.17 et la modélisation du signal donnée par l'équation 3.15, certains produits $x_j.C_i$ présentent une probabilité de 1/2 :

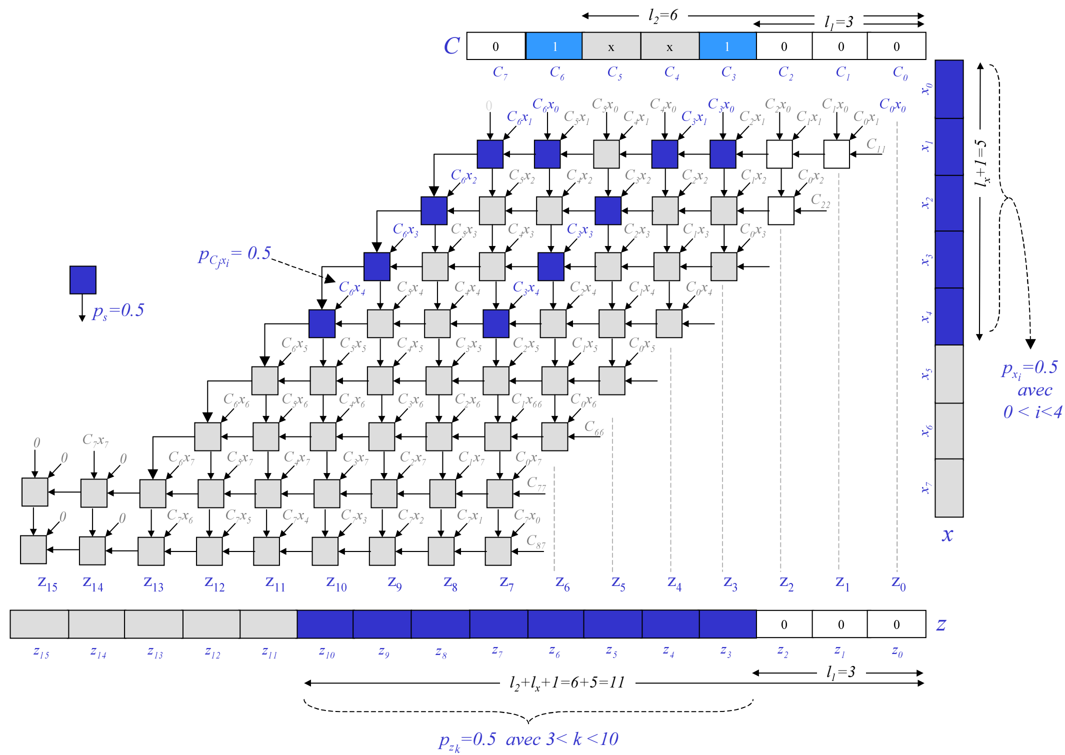


FIG. 3.7: Exemple de multiplication d'un signal par une constante

$$\begin{aligned} p_{x_j C_{l_1}} &= 1/2 \quad \forall\ 0 \leq j \leq l_x \\ p_{x_j C_{l_2}} &= 1/2 \quad \forall\ 0 \leq j \leq l_x \end{aligned} \tag{3.18}$$

Par conséquent, suite à la condition exprimée par l'équation 3.13, tous les bits z_k obtenus à partir du produit $x_j C_{l_1}$ et $x_j C_{l_2}$ auront une probabilité de 1/2. Ainsi, on peut déduire de l'équation 3.18 et 3.11 les relations suivantes :

$$\begin{aligned} p_{z_k} &= 1/2 \quad \forall\ l_1 \leq k \leq l_1 + l_x \\ p_{z_k} &= 1/2 \quad \forall\ l_2 \leq k \leq l_2 + l_x \end{aligned} \tag{3.19}$$

Par ailleurs, les bits C_i situés entre C_{l_1} et C_{l_2} et égaux à 1, conduiront à obtenir une probabilité p_{z_k} égale à 1/2 pour certaines valeurs de k :

$$p_{z_k} = 1/2 \qquad i \leq k \leq i + l_x \quad \forall\ i \text{ tel que } C_i = 1 \text{ et } l_1 < i < l_2 \tag{3.20}$$

Étant donné que l_x est proche de N, dans la plupart des cas, la probabilité des bits de sortie du multiplieur p_{z_k} est égale à

$$p_{z_k} = \begin{cases} 0 & \forall k \in [0, l_1 - 1] \\ 1/2 & \forall k \in [l_1, l_2 + l_x] \end{cases} \tag{3.21}$$

Multiplication de deux signaux x et y : Considérons la multiplication de deux signaux x et y. La probabilité des bits de sortie du multiplieur a été déterminée à l'aide des différentes expressions présentées dans cette section. Les probabilités p_{z_k} obtenues à partir de différents modèles probabilistes pour les bits d'entrée détaillés à la figure 3.8.a, sont présentées à la figure 3.8.b. D'après ces résultats, on peut considérer que la probabilité des bits médians en sortie du multiplieur est égale à 1/2.

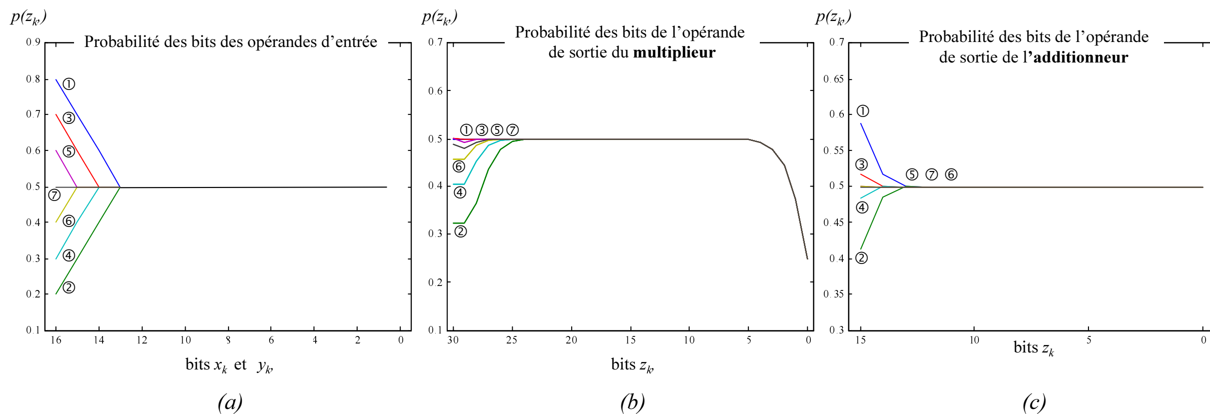


FIG. 3.8: Probabilités des bits en sortie d'un multiplieur et d'un additionneur

Addition de deux signaux x et y : Considérons l'addition de deux signaux x et y. Étant donné que $P_s\left(1/2, p_b, p_{c_{in}}\right) = 1/2$ et selon l'équation 3.9, la probabilité du bit de sortie de l'additionneur p_{z_k} est égale à $1/2$ si la probabilité de l'une des deux sorties est égale à $1/2$. Les probabilités p_{z_k} sont représentées à la figure 3.8.c, pour les différents modèles de probabilité pour les bits d'entrée (figure 3.8.a).

Paramètres statistiques de l'erreur de quantification

L'objectif de cette section est de déterminer l'expression des moments du premier et du second ordre de l'erreur de quantification résultant de l'élimination des certains bits en sortie d'un opérateur. Le cas de la multiplication d'un signal par une constante est détaillé ci-dessous et un nouveau modèle est proposé. Puis les résultats des erreurs de quantification associées aux opérations de multiplication et d'addition de deux signaux sont brièvement présentés. En effet, ces résultats sont similaires à ceux présentés dans [24].

Multiplication d'un signal par une constante : Considérons l'étude de l'erreur issue de la quantification du résultat de la multiplication d'un signal par une constante. Le résultat de la multiplication est composé de $N_m + 1$ bits dont n bits pour la partie fractionnaire. La valeur M_e en sortie du multiplieur est définie comme suit :

$$M_e = -2^{N_m - n}.S + \sum_{i=0}^{N_m - 1} m_i 2^{-n+i} \tag{3.22}$$

La valeur M_s de la variable résultant de la troncature de k bits en sortie du multiplieur est définie par l'équation 3.23. On considère que le nombre k de bits éliminés est inférieur au paramètre $l_2 + l_x$, défini dans la section précédente.

$$M_s = -2^{N_m - n}.S + \sum_{i=k}^{N_m - 1} m_i 2^{-n+i} \tag{3.23}$$

L'erreur de quantification, correspondant à la différence entre M_s et M_e est alors égale à :

$$b_g = M_e - M_s = \sum_{i=0}^{k-1} m_i 2^{-n+i} \tag{3.24}$$

D'après les résultats de l'équation 3.21, les l_1 bits les moins significatifs sont nuls, l'expression de l'erreur de quantification devient alors :

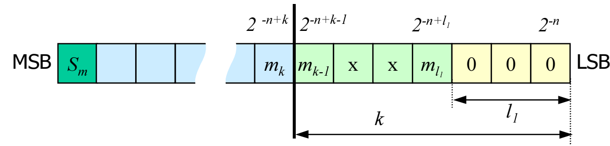


FIG. 3.9: Modélisation du résultat de la multiplication d'un signal par une constante

$$b_g = \sum_{i=l_1}^{k-1} m_i 2^{-n+i} = \sum_{i'=0}^{k-1-l_1} m_{i'+l_1} 2^{-n+i'+l_1} \tag{3.25}$$

soit

$$b_g = \sum_{i'=0}^{K-1} m_{i'+l_1} \Delta 2^{i'} \qquad \text{avec} \qquad \begin{cases} K = k - l_1 \\ \Delta = 2^{-n+l_1} \end{cases} \tag{3.26}$$

L'erreur de quantification b_g correspond à la somme de K variables aléatoires $e_{i'}$ définies de la manière suivante :

$$e_{i'} = m_{i'+l_1} \Delta 2^{i'} \tag{3.27}$$

La probabilité de chaque bit $m_{i'+l_1}$ étant égale à $1/2$, la densité de probabilité de la variable aléatoire $e_{i'}$ est alors définie comme suit :

$$p_{e_{i'}}(x) = (1 - p_{m_{i'+l_1}})\delta(x) + p_{m_{i'+l_1}} \delta(x - \Delta 2^{i'}) = \frac{1}{2}\left(\delta(x) + \delta(x - \Delta 2^{i'})\right) \tag{3.28}$$

Les variables aléatoires sont supposées être indépendantes ainsi la densité de probabilité de l'erreur de quantification est égale à :

$$p_{b_g}(x) = p_{e_0}(x) * p_{e_1}(x) * p_{e_2}(x) \ldots * p_{e_{K-1}}(x) \tag{3.29}$$

En substituant l'équation 3.28 dans l'équation 3.29, l'expression de la fonction de distribution de la variable b_g devient :

$$p_{b_g}(x) = \frac{1}{2^K} \left(\delta(x) + \delta(x - \Delta)\right) * \left(\delta(x) + \delta(x - 2\Delta)\right) * \ldots * \left(\delta(x) + \delta(x - 2^{K-1}\Delta)\right) \tag{3.30}$$

Après développement de l'expression 3.30, nous obtenons :

$$p_{b_g}(x) = \frac{1}{2^K} \sum_{j=0}^{2^K - 1} \delta(x - j.\Delta) \tag{3.31}$$

La moyenne μ_{b_g} de l'erreur de quantification est égale à :

$$\mu_{b_g} = \sum_{i=-\infty}^{+\infty} x_i p_{b_g}(x_i) = \sum_{i=0}^{2^K - 1} \frac{i\Delta}{2^K} \tag{3.32}$$

soit après développement,

$$\mu_{b_g} = 2^{k-n-1}(1 - 2^{l_1 - k}) = \frac{q}{2}(1 - 2^{-(k-l_1)}) \qquad \text{avec } q = 2^{-(n-k)} \tag{3.33}$$

La variance $\sigma_{b_g}^2$ de l'erreur de quantification est égale à :

$$\sigma_{b_g}^2 = \sum_{i=-\infty}^{+\infty} x_i^2 p_{b_g}(x_i) - \mu_{b_g}^2 \tag{3.34}$$

soit

$$\sigma_{b_g}^2 = \frac{2^{-2(n-k)}}{12}(1 - 2^{-2(k-l_1)}) = \frac{q^2}{12}(1 - 2^{-2(k-l_1)}) \quad (3.35)$$

Multiplication et addition de deux signaux : D'après les résultats présentés à la figure 3.8, la probabilité des bits les moins significatifs en sortie d'un additionneur est égale à 1/2. Les moments du premier et du second ordre de l'erreur de quantification correspondent à ceux obtenus précédemment en considérant que le paramètre l_1 est nul. Ainsi, nous obtenons les expressions suivantes [24] :

$$\mu_{b_g} = 2^{k-n-1}(1 - 2^{-k}) = \frac{q}{2}(1 - 2^{-k}) \quad (3.36)$$

$$\sigma_{b_g}^2 = \frac{2^{-2(n-k)}}{12}(1 - 2^{-2k}) = \frac{q^2}{12}(1 - 2^{-2k}) \quad (3.37)$$

Pour la multiplication de deux signaux les résultats sont présentés à la figure 3.8. Les bits médians du résultat de la multiplication possèdent une probabilité de 1/2. L'influence sur la valeur de l'erreur de quantification des quelques bits les moins significatifs dont la probabilité est différente de 1/2, est faible. Ainsi, les expressions des moments du premier et du second ordre de l'erreur de quantification sont équivalentes à celles présentées aux équations 3.36 et 3.37.

3.2.2 Modèle du bruit propagé

L'objectif de cette section est de décrire le modèle de propagation utilisé pour chaque type d'opérateur. Nous considérons successivement l'addition de deux signaux, la multiplication de deux signaux entre eux et la multiplication d'un signal par une constante.

Addition

Considérons l'addition de deux signaux x et y. Soient b_x et b_y les bruits de quantification associés respectivement à chaque signal x et y. La sortie de l'additionneur est composée d'un signal z et d'un bruit de quantification b_z. Les expressions respectives de z et de b_z sont les suivantes :

$$Z = X + Y \quad \Rightarrow \quad \begin{cases} z = x + y \\ b_z = b_x + b_y \end{cases} \quad (3.38)$$

Multiplication de deux signaux

Considérons la multiplication de deux signaux x et y. Soient b_x et b_y les bruits de quantification associés respectivement à chaque signal x et y. La sortie du multiplieur est composée d'un signal z et d'un bruit de quantification b_z. Les expressions respectives de z et de b_z sont les suivantes :

$$Z = X \times Y \quad \Rightarrow \quad \begin{cases} z = xy \\ b_z = b_x y + b_y x + b_x b_y \end{cases} \quad (3.39)$$

Le terme $b_x b_y$ représente le produit de deux bruits de quantification et est très inférieur aux autres termes, il sera négligé par la suite. Ainsi, l'expression du bruit b_z en sortie devient :

$$b_z = b_x y + b_y x \quad (3.40)$$

Paramètres statistiques	Amplitude-continue	Amplitude discrète $z = x \times y$, $z = x + y$	$z = x \times C$
Arrondi			
μ_{b_i}	0	$\frac{q}{2}(2^{-k})$	$\frac{q}{2}(2^{-(k-l_1)})$
$\sigma^2_{b_i}$	$\frac{q^2}{12}$	$\frac{q^2}{12}(1-2^{-2k})$	$\frac{q^2}{12}(1-2^{-2(k-l_1)})$
Troncature			
μ_{b_i}	$\frac{q}{2}$	$\frac{q}{2}(1-2^{-k})$	$\frac{q}{2}(1-2^{-(k-l_1)})$
$\sigma^2_{b_i}$	$\frac{q^2}{12}$	$\frac{q^2}{12}(1-2^{-2k})$	$\frac{q^2}{12}(1-2^{-2(k-l_1)})$

TAB. 3.1: Paramètres statistiques du bruit généré

Multiplication d'un signal par une constante

Considérons la multiplication d'un signal x par une constante C. Soient b_x le bruit de quantification associé au signal x et Δ_C le biais résultant de la quantification de la constante C. La sortie du multiplieur est composée du signal z et d'un bruit de quantification b_z. Les expressions respectives de z et de b_z sont

$$Z = X \times C \quad \Rightarrow \quad \begin{cases} z = xy \\ b_z = b_x C + \Delta_C x \end{cases} \tag{3.41}$$

3.2.3 Conclusions

D'après les résultats présentés dans cette partie, les propriétés suivantes peuvent être utilisées pour modéliser les erreurs de quantification. Le processus de quantification peut être modélisé par un système linéaire au sein duquel la sortie est égale à la somme de l'entrée et d'une variable aléatoire b_i (bruit de quantification). Cette source de bruit est un bruit blanc stationnaire et ergodique. La fonction d'autocorrélation de b_i est égale à :

$$\varphi_{b_i b_i}(\tau) = \mu^2_{b_i} + \sigma^2_{b_i}.\delta(\tau) \tag{3.42}$$

De plus, cette source de bruit n'est pas corrélée au signal d'entrée du processus de quantification, aux autres sources de bruit ou aux autres signaux présents au sein du système. Les paramètres statistiques du bruit de quantification b_i sont donnés dans le tableau 3.1 en fonction du type de donnée quantifiée.

3.3 Approche théorique pour l'évaluation de la précision

Dans cette partie, les concepts théoriques utilisés pour déterminer la puissance du bruit de quantification en sortie des systèmes linéaires et des systèmes non-linéaires sont présentés. La modélisation proposée permet de prendre en compte les différentes sources d'erreur liées à l'utilisation de l'arithmétique virgule fixe.

3.3.1 Systèmes linéaires

Présentation du système considéré

Considérons un système linéaire invariant dans le temps constitué de N_e entrées $x_j(n)$ et d'une sortie $y(n)$. Cette sortie est une fonction linéaire des entrées $x_j(n)$, des valeurs précédentes $x_j(n-k)$ et des valeurs précédentes de $y(n)$. Soit $H_j(z)$ la fonction de transfert partielle entre la sortie $Y(z)$ et chacune des entrées $X_j(z)$. Dans le cas d'un système multi-sorties (FFT, ...), le processus de calcul du RSBQ doit être répété pour chaque sortie. La sortie $y(n)$ est définie comme suit :

$$y(n) = \sum_{j=0}^{N_e-1} h_j(n) * x_j(n) \tag{3.43}$$

Considérons la version en virgule fixe de ce système. Soit $\widehat{x_j}(n)$ la $j^{ème}$ entrée quantifiée du système et $\widehat{H_j}(z)$ la fonction de transfert entre la sortie $Y(z)$ et l'entrée $X_j(z)$ et pour laquelle les coefficients sont quantifiés. L'utilisation de l'arithmétique virgule fixe conduit à trois types de sources d'erreur, dues à la propagation du bruit de quantification associé à chaque entrée du système, à la propagation des erreurs générées lorsque certains bits sont éliminés lors des opérations de changement de format et à la quantification des coefficients. Cependant, ce dernier type de sources d'erreur n'est pas présent si les coefficients du système ont été préalablement quantifiés avant de débuter le processus de conversion en virgule fixe.

Soit b'_{ej} le bruit de quantification associé à chaque entrée quantifiée du système $\widehat{x_j}(n)$ et défini dans l'équation 3.44. Soit b_{ej} le bruit en sortie dû à la propagation au sein du système du bruit d'entrée b'_{ej}.

$$\widehat{x_j}(n) = x_j(n) + b'_{ej}(n) \tag{3.44}$$

Soit $\Delta H_j(z)$ la fonction de transfert correspondant à la différence entre la fonction de transfert quantifiée $\widehat{H_j}(z)$ et la fonction de transfert réelle $H_j(z)$:

$$\Delta H_j(z) = \widehat{H_j}(z) - H_j(z) \tag{3.45}$$

L'élimination de certains bits au cours d'une opération de changement de format génère un bruit de quantification b'_{gi}. Soit b_{gi} le bruit en sortie dû à la propagation au sein du système du bruit ainsi généré b'_{gi}. La fonction de transfert entre la sortie du système et la source de bruit b'_{gi} est dénommée $H_{gi}(z)$. Soit b_g le bruit correspondant à la somme des N_g sources de bruit b_{gi} présentes au sein du système :

$$b_g(n) = \sum_{i=0}^{N_g-1} h_{gi}(n) * b'_{gi}(n) \tag{3.46}$$

L'expression de la sortie du système en précision finie $\hat{y}(n)$ est égale à :

$$\hat{y}(n) = \sum_{j=0}^{N_e-1} \widehat{h_j}(n) * \widehat{x_j}(n) + b_g(n) \tag{3.47}$$

En introduisant les équations 3.44, 3.45 et 3.46 dans l'équation 3.47, l'expression de la sortie $\hat{y}(n)$ devient :

$$\hat{y}(n) = \sum_{j=0}^{N_e-1} (h_j(n) + \Delta h_j(n)) * \left(x_j(n) + b'_{ej}(n)\right) + \sum_{i=0}^{N_g-1} h_{gi}(n) * b'_{gi}(n) \tag{3.48}$$

Comme expliqué à la section 3.2.2, le terme $\Delta h_j(n) * b'_{ej}$ correspond à la multiplication d'éléments représentant des bruits et des erreurs de quantification, ainsi, ils peuvent être négligés par rapport aux autres termes de l'expression. Ainsi, l'erreur b_y correspondant à la différence entre la sortie du système en précision infinie et la sortie en précision finie, est égale à :

$$b_y = \hat{y}(n) - y(n) = \sum_{j=0}^{N_e-1} h_j(n) * b'_{ej}(n) + \sum_{j=0}^{N_e-1} \Delta h_j(n) * x_j(n) + \sum_{i=0}^{N_g-1} h_{gi}(n) * b'_{gi}(n) \tag{3.49}$$

Pour simplifier la notation nous introduisons les variables définies de la manière suivante :

$$b_y(n) = b_q(n) + b_h(n) \qquad \text{avec} \qquad b_q(n) = b_g(n) + b_e(n) \tag{3.50}$$

$$b_e(n) = \sum_{j=0}^{N_e-1} h_j(n) * b'_{ej}(n) \tag{3.51}$$

$$b_h(n) = \sum_{j=0}^{N_e-1} \Delta h_j(n) * x_j(n) \tag{3.52}$$

Une représentation du modèle de bruit du système est donnée à la figure 3.10. Ce modèle est la généralisation de celui proposé dans [87]. Dans la suite de cette partie, nous détaillons les paramètres statistiques des différents bruits présentés ci-dessus afin de pouvoir déterminer la puissance du bruit de quantification $b_y(n)$.

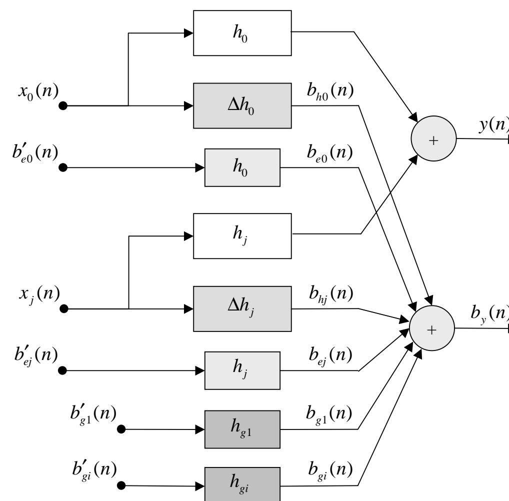


FIG. 3.10: Représentation du système linéaire au niveau bruit

Paramètres statistiques du bruit b_q

Moments du premier et du second ordre des bruits b_{ej} et b_{gi} : Les expressions des paramètres statistiques de b_{ej} et de b_{gi} sont identiques. En effet, ces bruits représentent la sortie d'un système linéaire excité par un bruit blanc (b'_{ej} ou b'_{gi}) comme défini précédemment. Pour simplifier, soit b_j la sortie du système, b'_j l'entrée et $H_j(z)$ la fonction de transfert de ce système. L'expression du bruit en sortie est par conséquent :

$$b_j(n) = b'_j(n) * h_j(n) \tag{3.53}$$

La moyenne μ_{b_j} du bruit b_j présent en sortie est égale à :

$$\mu_{b_j} = E\Big(b'_j(n) * h_j(n)\Big) = \mu_{b'_j} \sum_{m=-\infty}^{+\infty} h_j(m) = \mu_{b'_j} H_j(e^{j0}) \tag{3.54}$$

Le moment du second ordre de la sortie s'obtient en évaluant la fonction d'autocorrélation de b_j à l'origine ($\tau = 0$). L'expression de cette fonction d'autocorrélation $\varphi_{b_j b_j}(\tau)$ est la suivante :

$$\varphi_{b_j b_j}(\tau) = h_j(\tau) * h_j(-\tau) * \varphi_{b'_j b'_j}(\tau) \tag{3.55}$$

En substituant l'équation 3.42 dans l'équation 3.55, la fonction d'autocorrélation $\varphi_{b_j b_j}(\tau)$ devient :

$$\varphi_{b_j b_j}(\tau) = h_j(\tau) * h_j(-\tau) * \mu^2_{b'_j} + h_j(\tau) * h_j(-\tau) * \sigma^2_{b'_j} \delta(\tau) \tag{3.56}$$

$$= \mu^2_{b'_j} \sum_{m=-\infty}^{+\infty} h_j(m) \sum_{l=-\infty}^{+\infty} h_j(-l) + \sigma^2_{b'_j} \sum_{m=-\infty}^{+\infty} h_j(m) h_j(m-\tau) \tag{3.57}$$

par conséquent, le moment du second ordre de b_j est égal à :

$$E(b_j^2) = \varphi_{b_j b_j}(0) = \Big(\mu_{b'_j} \sum_{m=-\infty}^{+\infty} h_j(m)\Big)^2 + \sigma^2_{b'_j} \sum_{m=-\infty}^{+\infty} |h_j(m)|^2 \tag{3.58}$$

La variance de b_j s'obtient en substituant le carré de la moyenne de b_j dans l'équation 3.58 :

$$\sigma^2_{b_j} = \sigma^2_{b'_j} \sum_{m=-\infty}^{+\infty} |h_j(m)|^2 = \frac{\sigma^2_{b'_j}}{2\pi} \int_{-\pi}^{\pi} |H_j(e^{j\Omega})|^2 d\Omega \tag{3.59}$$

Moments du premier et du second ordre du bruit b_q : Le bruit b_q représente la somme des variables aléatoires $b_e(n)$ et $b_g(n)$. L'expression du moment du premier ordre de b_q est égale à :

$$\mu_{b_q} = E(b_q) = \sum_{j=0}^{N_e-1} \mu_{b_{ej}} + \sum_{i=0}^{N_g-1} \mu_{b_{gi}} \tag{3.60}$$

L'expression du moment centré de second ordre de la variable aléatoire b_q est égale à :

$$\sigma^2_{b_q} = E((b_q - \mu_{b_q})^2) = E\left(\left(\sum_{j=0}^{N_e-1} b_{ej} - \mu_{b_{ej}} + \sum_{i=0}^{N_g-1} b_{gi} - \mu_{b_{gi}}\right)^2\right) \tag{3.61}$$

Le bruit b_q correspond à la somme de $N_e + N_g$ bruits b_j définis précédemment. En considérant le modèle de bruit présenté à la section 3.2.3, chaque source de bruit b'_j est non corrélée avec un

signal quelconque ou une autre source de bruit. La sortie b_j du système linéaire h_j excité par le bruit b'_j sera donc non corrélée avec les autres sources de bruit. Le développement de l'équation 3.61 aboutit par conséquent à :

$$\sigma_{b_q}^2 = \sum_{j=0}^{N_e-1} \sigma_{b_{ej}}^2 + \sum_{i=0}^{N_g-1} \sigma_{b_{gi}}^2 \tag{3.62}$$

Le moment du second ordre de la variable aléatoire b_q est donc égal à :

$$E(b_q^2) = \sigma_{b_q}^2 + \mu_{b_q}^2 = \sum_{j=0}^{N_e-1} \sigma_{b_{ej}}^2 + \sum_{i=0}^{N_g-1} \sigma_{b_{gi}}^2 + \left(\sum_{j=0}^{N_e-1} \mu_{b_{ej}} + \sum_{i=0}^{N_g-1} \mu_{b_{gi}} \right)^2 \tag{3.63}$$

Paramètres statistiques du bruit b_h

Moments du premier et du second ordre des bruits b_{hj} : D'après les équations 3.45 et 3.49, l'expression de l'erreur en sortie b_{hj} résultant de la quantification des coefficients de la fonction de transfert $H_j(z)$ est la suivante :

$$b_{hj} = \Delta h_j(n) * x_j(n) \tag{3.64}$$

Chaque entrée est supposée être une variable aléatoire stationnaire et ergodique. Ainsi, le moment du premier ordre $\mu_{b_{hj}}$ de la sortie b_{hj} est égal à :

$$\mu_{b_{hj}} = \mu_{x_j} \sum_{m=-\infty}^{+\infty} \Delta h_j(m) = \mu_{x_j} \Delta H_j(e^{j0}) \tag{3.65}$$

Le moment du second ordre de b_{hj} s'obtient à partir de la fonction d'autocorrélation $\varphi_{x_j x_j}$ évaluée à l'origine. En utilisant le théorème de Parseval nous obtenons l'expression de $E(b_{hj}^2)$ suivante :

$$E(b_{hj}^2) = \frac{1}{2\pi} \int_{-\pi}^{\pi} \phi_{x_j x_j}(e^{j\Omega}) \left| \Delta H_j(e^{j\Omega}) \right|^2 d\Omega \tag{3.66}$$

Le calcul du moment du second ordre du bruit b_{hj} nécessite de connaître la densité spectrale de puissance $\phi_{x_j x_j}(e^{j\Omega})$ de chaque entrée x_j du système.

Intercorrélation entre les bruits b_{hj} : Considérons deux bruits b_{h_j} et b_{h_k} représentant la sortie des systèmes linéaires $\Delta h_j(n)$ et $\Delta h_k(n)$, excités respectivement par les signaux d'entrée x_j et x_k. L'intercorrélation entre les deux bruits b_{h_j} et b_{h_k} est obtenue à partir de l'intercorrélation entre les entrées x_j et x_k de la manière suivante :

$$\varphi_{b_{h_j} b_{h_k}}(\tau) = h_j(\tau) * h_k(-\tau) * \varphi_{x_j x_k}(\tau) \tag{3.67}$$

En suivant la même démarche que précédemment, l'intercorrélation entre les bruits b_{h_j} et b_{h_k} est la suivante :

$$E(b_{h_j} b_{h_k}) = \frac{1}{2\pi} \int_{-\pi}^{\pi} \phi_{x_j x_k}(e^{j\Omega}) \Delta H_j(e^{j\Omega}) \Delta H_k^*(e^{j\Omega}) d\Omega \tag{3.68}$$

L'intercorrélation entre deux bruits b_{h_j} et b_{h_k} n'est pas nulle si les entrées x_j et x_k sont corrélées. Par conséquent, le calcul de $E(b_{h_j} b_{h_k})$ nécessite de connaître la densité spectrale de puissance mutuelle $\phi_{x_j x_k}(e^{j\Omega})$ entre les différentes entrées x_j et x_k du système.

Moments du premier et du second ordre du bruit b_h : Le bruit b_h est la somme de N_e variables aléatoires. Les moments du premier et du second ordre du bruit b_h sont égaux à :

$$E(b_h) = \mu_{b_h} = \sum_{j=0}^{N_e-1} \mu_{x_j} \Delta H_j(e^{j0}) \tag{3.69}$$

$$E(b_h^2) = \sum_{j=0}^{N_e-1} E(b_{h_j}^2) + \sum_{j=0}^{N_e-1} \sum_{\substack{k=0 \\ k\neq j}}^{N_e-1} E(b_{h_j} b_{h_k}) \tag{3.70}$$

Les différents termes de cette expression sont définis aux équations 3.66 et 3.68.

Puissance du bruit b_y en sortie du système

La variable aléatoire b_y représentant le bruit de quantification en sortie du système, correspond à la somme des variables b_q et b_h :

$$b_y(n) = b_q(n) + b_h(n) \tag{3.71}$$

La puissance du bruit de quantification P_{b_y} en sortie correspondant au moment du second ordre de b_y est définie comme suit :

$$P_{b_y} = E(b_y^2) = E(b_q^2) + E(b_h^2) + 2E(b_q.b_h) \tag{3.72}$$

Étant donné que b_q est constitué de bruits non corrélés avec les autres signaux, l'expression de P_{b_y} devient :

$$P_{b_y} = E(b_q^2) + E(b_h^2) + 2\mu_{b_q}\mu_{b_h} \tag{3.73}$$

Les expressions du premier et du second terme de cette équation sont données par les équations 3.63 et 3.70. Le dernier terme de cette expression est obtenu à partir des équations 3.60 et 3.69. La précision de l'estimation de P_{b_y}, basée sur l'approche présentée dans cette section, a été évaluée. Les résultats sont présentés dans la section 3.5.2.

Conclusions

La technique proposée pour évaluer la puissance du bruit de quantification en sortie d'un système linéaire implique uniquement que les entrées du systèmes soient des variables aléatoires. Cette restriction ne s'applique que si le bruit lié au codage des coefficients est pris en compte.

L'évaluation de la puissance du bruit de quantification nécessite de déterminer différents éléments. Pour le bruit b_q, les moments du premier et du second ordre de chaque source de bruit et la réponse fréquentielle de la fonction de transfert entre la sortie et la source de bruit doivent être déterminés. Dans le cadre du bruit b_h, les différentes fonctions de transfert ΔH_j doivent être déterminées et la densité spectrale de puissance mutuelle entre les entrées et la densité spectrale de puissance de chaque entrée doivent être définies.

3.3.2 Systèmes non-linéaires

Dans le cas des systèmes non linéaires, le concept de fonction de transfert n'est plus valide. Ainsi, pour calculer le bruit en sortie, il est nécessaire d'analyser la propagation des différents bruits de quantification associés aux entrées ou générés au sein du système. Dans une première partie, les concepts utilisés pour déterminer le bruit de quantification en sortie d'un système non-linéaire composé d'opérations de multiplication et d'addition, sont présentés. De plus, l'évaluation de la précision dans les applications non-linéaires nécessite la définition de modèles de

bruit pour des opérateurs spécifiques. Le modèle de bruit de l'opération de calcul de la valeur absolue d'un signal est proposé dans une seconde partie.

Systèmes non-linéaires composés d'opérations d'addition et de multiplication

L'objectif de ce paragraphe est de déterminer l'expression du bruit de quantification présent en sortie d'un système non-linéaire et non-récursif et composé d'opérations de multiplication et d'addition. Le principe retenu consiste tout d'abord à déterminer l'expression du bruit de sortie b_y en fonction des différents bruits de quantification associés aux entrées ou générés au sein du système lors d'un changement de format. Ensuite, l'expression du moment du second ordre de ce bruit est calculée en fonction des différents paramètres statistiques des éléments composant l'expression du bruit b_y. Les bruits de quantification sont modélisés par un bruit blanc uniformément réparti et non-corrélé avec les autres bruits de quantification et les signaux. Pour obtenir l'expression du moment du second ordre du bruit b_y, nous avons posé l'hypothèse d'indépendance entre les différents bruits de quantification et entre un bruit de quantification et un signal.

Chaque source de bruit de quantification b_{q_i} présente au sein du système va contribuer au bruit global b_y présent en sortie. Soit b_i, le bruit présent en sortie du système et résultant de la propagation au sein du système du bruit de quantification b_{q_i}. Les modèles de propagation du bruit au sein des opérateurs arithmétiques sont présentés au paragraphe 3.2.2. Dans le cadre de l'addition de deux signaux, le bruit en sortie de l'opération correspond à la somme des bruits présents en entrée. Dans le cas de la multiplication, le bruit présent en sortie correspond à la somme des produits du bruit associé à une entrée par le signal associé à l'autre entrée. Ces modèles de propagation des bruits pour l'addition et la multiplication montrent que le bruit b_i présent en sortie est égal au produit du bruit b_{q_i} par différents signaux x_k. Ainsi, l'expression du bruit b_i issu de la propagation du bruit de quantification b_{q_i}, est la suivante :

$$b_i = b_{q_i} \times x_0 \ldots \times x_k = b_{q_i} \times X_i \qquad \text{avec} \quad X_i = x_0 \ldots \times x_k \tag{3.74}$$

En conséquence, le bruit b_y en sortie d'un système non-linéaire composé d'opérations d'addition et de multiplication, correspond à la somme des différents bruits b_i présents en sortie :

$$b_y = \sum_{i=0}^{N_s} b_i \tag{3.75}$$

L'expression du moment du second ordre du bruit b_y est la suivante :

$$E(b_y^2) = \sum_{i=0}^{N_s} E(b_i^2) + 2 \sum_{i=0}^{N_s} \sum_{\substack{j=0 \\ j>i}}^{N_s} E(b_i b_j) \tag{3.76}$$

En substituant l'expression 3.74 dans l'expression 3.76 et en supposant l'hypothèse d'indépendance entre les bruits de quantification et les signaux, l'expression 3.76 devient :

$$E(b_y^2) = \sum_{i=0}^{N_s} E(b_{q_i}^2).E(X_i^2) + 2 \sum_{i=0}^{N_s} \sum_{\substack{j=0 \\ j>i}}^{N_s} E(b_{q_i})E(b_{q_j}).E(X_i.X_j) \tag{3.77}$$

La première partie de l'expression nécessite la connaissance de la puissance du bruit b_{q_i} et celle du signal X_i. La seconde partie de l'expression nécessite de déterminer la moyenne des bruits de quantification et l'intercorrélation entre les signaux X_i et X_j. Cette seconde partie est nulle si une loi de quantification par arrondi est utilisée lors de chaque changement de format.

Ces résultats montrent que le calcul de l'expression du bruit de quantification en sortie du système nécessite de déterminer deux types d'information liés aux sources de bruit de quantification et aux signaux. L'expression de b_y est fonction des moments du premier et du second ordre des bruits de quantification générés et associés aux entrées. Ces paramètres sont déterminés à l'aide des expressions présentées dans le tableau 3.1 et sont fonction du codage des données en virgule fixe. Le second type d'information est lié aux différents signaux présents au sein du système. Les paramètres à déterminer correspondent à la puissance des signaux et à l'intercorrélation entre ces signaux. Ces paramètres sont indépendants de la manière dont les données sont codées en virgule fixe. Ainsi, ils interviennent sous forme de constantes au niveau de l'expression de b_y. Ces paramètres statistiques sont déterminés à l'aide d'une simulation de l'algorithme en précision infinie.

Modélisation d'opérations non-linéaires : exemple de la valeur absolue

Nous présentons dans ce paragraphe le modèle de bruit de l'opération de calcul de la valeur absolue d'un signal. Soit z, le signal correspondant à la valeur absolue du signal x affecté d'un bruit de quantification b_x. Soient $f_x(x)$ et $f_{b_x}(b_x)$, les densités de probabilité des variables aléatoires x et b_x. Soit α, le paramètre définissant la proportion d'échantillons positifs au sein du signal x d'entrée :

$$\alpha = \int_0^{+\infty} f_x(x)dx \tag{3.78}$$

Les échantillons du bruit b_x associés aux valeurs négatives de x vont conduire à un bruit b_z dont la valeur est l'opposé de celle de b_x. Les échantillons du bruit b_x associés aux valeurs positives de x vont conduire à un bruit b_z identique à b_x. Le bruit b_x n'étant pas corrélé au signal x, alors la densité de probabilité du bruit en sortie de l'opérateur de valeur absolue dépend du paramètre α selon l'expression présentée à l'équation 3.79. Ces différents aspects sont illustrés à la figure 3.11.

$$f_{b_z}(b_z) = (1-\alpha)f_{b_x}(-b_z) + \alpha f_{b_x}(b_z) \tag{3.79}$$

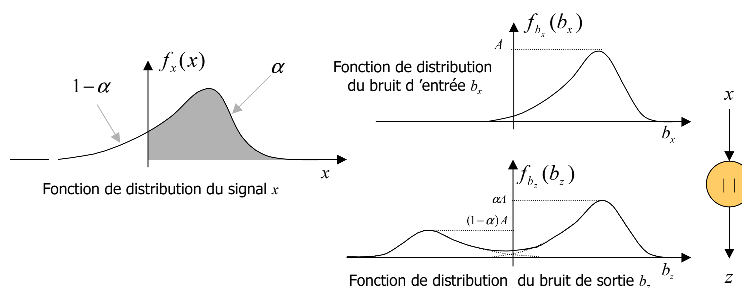


FIG. 3.11: Fonction de distribution des bruits dans le cas du calcul de la valeur absolue d'une donnée

Les moments du premier et du second ordre du bruit en sortie de l'opération de calcul de la valeur absolue d'un signal sont égaux à :

$$E(b_z) = \int_{-\infty}^{+\infty} (1-\alpha)uf_{b_x}(-u) + \alpha uf_{b_x}(u)du = \mu_{b_x}(2\alpha - 1) \tag{3.80}$$

$$E(b_z^2) = \int_{-\infty}^{+\infty} (1-\alpha)u^2 f_{b_x}(-u) + \alpha u^2 f_{b_x}(u)du = E(b_x^2) \tag{3.81}$$

Ainsi, la puissance du bruit en sortie de l'opération de calcul de la valeur absolue est égale à celle du bruit présent en entrée.

3.4 Méthodologie d'évaluation automatique de la précision

3.4.1 Introduction

L'objectif de cette partie est de présenter la méthodologie définie pour déterminer automatiquement l'expression analytique du RSBQ en sortie d'un système linéaire en virgule fixe. Cette méthodologie doit déterminer automatiquement à partir d'une description du système, les différents éléments nécessaires pour obtenir l'expression du RSBQ. Dans le cadre des systèmes linéaires, ces différents éléments correspondent aux sources de bruit et leur paramètres statistiques et aux différentes fonctions de transfert définissant le système.

La représentation de l'application en entrée de la méthodologie spécifie les traitements réalisés et les paramètres virgule fixe. Cette représentation correspond à un graphe flot de signal (GFS) G_s spécifiant le comportement de l'application en virgule fixe. Le GFS correspond à un graphe flot de données au sein duquel les relations temporelles entre les données sont spécifiées [90]. Pour cela, des opérations de délai (retard) correspondant au "vieillissement" des données sont insérées. Les nœuds N_s du graphe orienté $G_s = (N_s, E_s)$ représentent soit une opération soit une donnée et les arcs E_s spécifient les relations entre les données et les opérations. Chaque opération o_i du graphe est caractérisée par le type d'opération réalisée γ_i, le format de ses entrées et de sa sortie et les lois de quantification associées aux entrées et à la sortie. En effet, si un changement de format a lieu en entrée ou en sortie de l'opérateur, il est nécessaire de définir la loi de quantification utilisée. Soient Q_i^{e1}, Q_i^{e2} et Q_i^s, les lois de quantification associées respectivement aux entrées e_1 et e_2 et à la sortie s de l'opération o_i. Soient $(b_i^{e1}, m_i^{e1}, n_i^{e1})$ et $(b_i^{e2}, m_i^{e2}, n_i^{e2})$, les formats associés aux entrées et (b_i^s, m_i^s, n_i^s) le format associé à la sortie. Soient $b_i = \{b_i^{e1}, b_i^{e2}, b_i^s\}$ et $m_i = \{m_i^{e1}, m_i^{e2}, m_i^s\}$, les paramètres définissant respectivement la largeur des opérandes et la position de la virgule des entrées et de la sortie de l'opération o_i. Soit $Q_i = \{Q_i^{e1}, Q_i^{e2}, Q_i^s\}$, le paramètre spécifiant les lois de quantification associées à l'opération o_i. Soient b, m et Q, les vecteurs regroupant respectivement les paramètres b_i, m_i et Q_i de l'ensemble des opérations o_i du graphe. Ainsi, pour une application donnée définie par un GFS, la méthodologie détermine l'expression du RSBQ dont les variables sont les vecteurs b, m et Q.

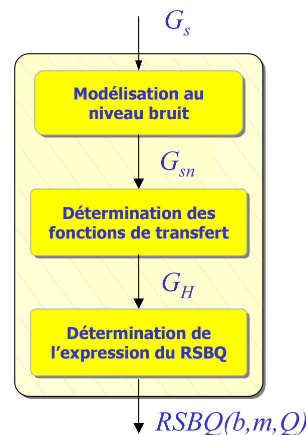


FIG. 3.12: Synoptique de la méthodologie de détermination automatique de l'expression du RSBQ

Les différents éléments nécessaires pour déterminer l'expression du RSBQ, sont obtenus par transformations successives du graphe représentant l'application. Tout d'abord, le GFS G_s est transformé en un graphe G_{sn} représentant le système au niveau bruit de quantification. Ce graphe prend en compte tous les bruits générés et modélise la manière dont les bruits sont propagés au sein de l'application. Ensuite, les fonctions de transfert entre la sortie et toutes

les entrées du système au niveau bruit, sont déterminées. Enfin, l'expression de la puissance du bruit est définie à partir de la réponse fréquentielle des différentes fonctions de transfert et des paramètres statistiques des sources de bruit et des entrées. Le synoptique de la méthodologie est représenté à la figure 3.12. En termes de terminologie, les nœuds du graphe ne possédant pas de successeurs sont les *racines* de celui-ci et les nœuds ne possédant pas de prédécesseurs sont les *sources* du graphe.

3.4.2 Représentation de l'application au niveau bruit

L'objectif de la première transformation (T_1) est d'obtenir le graphe G_{sn} représentant l'application au niveau bruit de quantification à partir du graphe d'entrée G_s. Ce graphe G_{sn} regroupe l'ensemble des sources de bruit de quantification et décrit la propagation du signal d'origine et du bruit de quantification dans l'algorithme. Ce graphe $G_{sn} = (N_{sn}, E_{sn})$ est constitué du sous-graphe G_s, représentant la partie signal et du sous-graphe $G_n = (N_n, E_n)$, correspondant à la partie bruit de quantification. Ainsi, la transformation T_1 génère le graphe G_{sn} en ajoutant à G_s les différents nœuds N_n associés aux bruits de quantification.

Insertion des sources d'erreur (T_{11})

L'objectif de cette première étape de la transformation T_1 est d'intégrer les différentes sources d'erreur au graphe. Les différents types de source d'erreur considérés dans notre modélisation ont été présentés dans la partie 3.3.1 à la page 92. Ces sources correspondent au bruit de quantification associé aux entrées, à l'erreur de quantification au niveau des coefficients et au bruit généré lors d'un changement de format.

Un bruit de quantification est associé à chaque entrée x_j du système. Ce bruit est modélisé par une source de bruit dont les propriétés sont présentées dans la partie 3.2.3. Les paramètres statistiques sont soit spécifiés par l'utilisateur soit déterminés à partir du format de l'entrée à l'aide du modèle de quantification d'un signal d'amplitude continue spécifié dans le tableau 3.1.

La quantification de chaque coefficient C_i conduit à une erreur constante ΔC_i. Cette erreur dépend du format associé à la constante codée en virgule fixe et correspond à la différence entre la valeur en précision infinie c_i et en précision finie $\widehat{c_i}$. Ainsi, chaque nœud représentant une constante est transformé en un nœud représentant la constante en virgule fixe et un nœud représentant le biais ΔC. Ce second nœud appartenant à l'ensemble des nœuds de type *bruit*, est un nœud de type constante dont la valeur est égale à $c_i - \widehat{c_i}$.

Insertion des sources de bruit généré : L'objectif de cette étape est d'insérer l'ensemble des sources de bruit potentielles au sein du graphe. Ces sources de bruit de quantification modélisent l'élimination de certains bits lors d'un changement de format. Ces changements de format sont liés à la modification de la position de la virgule ou à la diminution de la largeur d'une donnée. Considérons le graphe proposé à la figure 3.13. Le résultat de l'opération o_1 est utilisé en entrée des opérations o_2 et o_3.

La première source de bruit potentielle est liée à l'élimination de certains bits lors de l'opération o_1. En effet, le nombre de bits disponibles pour stocker le résultat de l'opération peut être insuffisant. Ainsi, il est nécessaire de définir pour chaque opération le format théorique de la sortie $\left(\widetilde{b}^s, \widetilde{m}^s, \widetilde{n}^s\right)$ représentant le format en sortie de l'opérateur dans le cas où aucun bit n'est

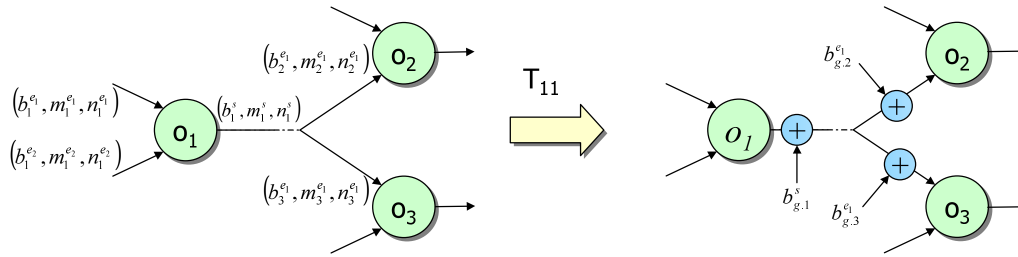


FIG. 3.13: Processus d'insertion des sources de bruit généré

éliminé. Ce format se déduit du format des données d'entrée en utilisant pour chaque paramètre du format, une fonction f_γ liée au type γ de l'opération :

$$\left(\widetilde{b}^s, \widetilde{m}^s, \widetilde{n}^s\right) = \left(f_\gamma^b\left(b^{e_1}, b^{e_2}\right), f_\gamma^m\left(m^{e_1}, m^{e_2}\right), f_\gamma^n\left(n^{e_1}, n^{e_2}\right)\right) \tag{3.82}$$

Ces fonctions f_γ sont obtenues à partir des règles de l'arithmétique virgule fixe présentées dans la partie 1.1.2 à la page 10. Le nombre de bits k_1^s éliminés en sortie de l'opération o_1 correspond à la différence entre le nombre de bits réservés pour la partie fractionnaire au niveau du format théorique ($\widetilde{n}_1^s$) et du format réel (n_1^s). L'expression de k_1^s en fonction du format des entrées et de la sortie de l'opération, est la suivante :

$$k_1^s = \widetilde{n}_1^s - n_1^s = f_{\gamma_1}^b\left(b_1^{e_1}, b_1^{e_2}\right) - f_{\gamma_1}^m\left(m_1^{e_1}, m_1^{e_2}\right) - b_1^s + m_1^s \tag{3.83}$$

Deux autres sources de bruit peuvent être présentes. Elles correspondent à l'élimination de certains bits entre le résultat de l'opération o_1 et les opérandes d'entrée des opérations o_2 et o_3. Ces bits sont éliminés si un recadrage de la donnée est réalisé ou si le nombre de bits utilisés pour stocker l'opérande d'entrée est inférieur à celui associé à la sortie de o_1. Le nombre de bits $k_j^{e_1}$ éliminés est égal à :

$$k_j^{e_1} = n_1^s - n_j^{e_1} = b_1^s - m_1^s - b_j^{e_1} + m_j^{e_1} \qquad \text{avec } j = 2 \text{ ou } 3 \tag{3.84}$$

Chaque source de bruit insérée au sein du graphe nécessite de déterminer la fonction de transfert entre la sortie du système et la source de bruit. Pour limiter les temps de traitement, certaines sources de bruit sont regroupées. Lorsque la sortie d'une opération o_1 n'est utilisée que par une seule opération o_2 , alors les deux sources de bruit potentielles peuvent être fusionnées. Dans ce cas, une seule source de bruit est insérée en entrée de l'opérateur o_2.

Les paramètres statistiques associés aux sources de bruit sont déterminés à partir des expressions présentées dans le tableau 3.1. Ils sont fonction du nombre de bits k éliminés, du nombre de bits n réservés pour la partie fractionnaire après quantification, de la loi de quantification Q utilisée et du paramètre l_1 spécifiant le nombre de bits présents au sein de la donnée avant quantification et dont la valeur est toujours égale à 0. Ce nombre de bits l_1 est obtenu en parcourant le graphe des sources [1] vers la racine [2] et en propageant le nombre de bits l_1 associé à chaque constante.

Modélisation des données et des opérations au niveau bruit (T_{12})

La seconde étape de cette transformation conduit à la représentation des données et des opérateurs au niveau du bruit. Chaque nœud de G_s représentant une donnée, excepté les sources de G_s, traitées dans l'étape T_{11}, est subdivisé en un nœud *signal* et en un nœud *bruit*. Puis,

[1]Nœud du graphe ne possédant pas de prédécesseurs
[2]Nœud du graphe ne possédant pas de successeurs

chaque opérateur est remplacé par son modèle de propagation du bruit. Ces modèles ont été définis à partir des expressions des équations 3.38 et 3.39. Une représentation de ces modèles est donnée à la figure 3.14.

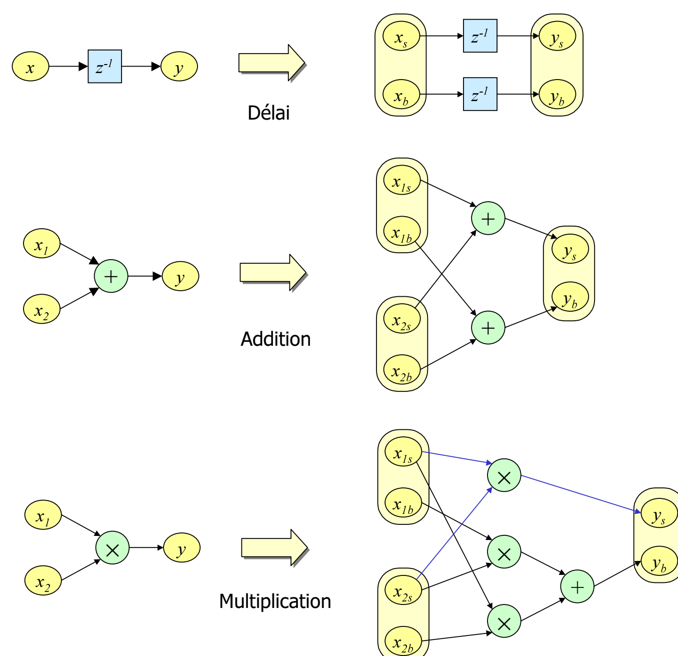


FIG. 3.14: Modèles de propagation du bruit des différents opérateurs

3.4.3 Détermination des fonctions de transfert

L'objectif de cette transformation est de déterminer les fonctions de transfert spécifiant le système. Selon nos connaissances, peu de méthodologies permettant de déterminer automatiquement la fonction de transfert d'un système spécifié par un graphe semblent disponibles. Néanmoins, dans le cadre de l'outil d'analyse de filtres numériques DIGICAP [143], une méthode permettant de déterminer la fonction de transfert d'un filtre a été mise en œuvre. Cette fonction de transfert est obtenue par inversion de la matrice d'état représentant le système. Cependant, selon l'auteur, l'approche proposée ne permet pas de traiter des structures possédant plusieurs pôles identiques. De plus, les structures telles que les filtres à réponse impulsionnelle finie (FIR) ne peuvent être traitées

Nous nous sommes orientés vers une solution déterminant les fonctions de transfert d'un système à partir des équations récurrentes définissant les dénominateurs et numérateurs du système. Ces équations récurrentes correspondent à des fonctions linéaires et sont construites en parcourant le graphe des sources vers la racine représentant la sortie du système. Cette technique est néanmoins inutilisable en présence de cycles au sein du graphe, comme dans le cas des structures récursives. Ainsi, l'utilisation de cette technique nécessite par conséquent de transformer d'abord le graphe en un graphe acyclique orienté. Pour expliquer la technique utilisée, prenons l'exemple simple de la figure 3.15. Ce système est composé d'une sortie y, d'une entrée x et d'un circuit représentant la partie récursive du système. Pour déterminer les fonctions linéaires de ce système, le circuit est coupé au niveau du nœud y et ce nœud est dupliqué à chaque extrémité de la branche. Le graphe acyclique qui en résulte est alors parcouru des sources vers la racine, permettant de construire les fonctions linéaires. Dans notre exemple, ce processus conduit pour la sortie à l'expression suivante $y(n) = b_0 * x(n) + a_1 * y(n-1)$. Ainsi, la fonction de transfert entre la sortie $Y(z)$ et l'entrée $X(z)$ peut être obtenue à partir de l'expression de $y(n)$ définie ci-dessus, en utilisant la transformation en $\mathcal{Z}$:

$$\frac{Y(z)}{X(z)} = \frac{b_0}{1 - a_1 z^{-1}} \tag{3.85}$$

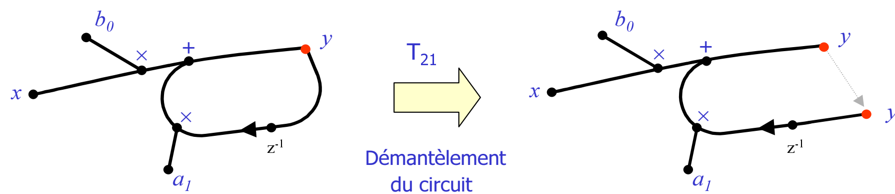


FIG. 3.15: Exemple de démantèlement d'un circuit

Cet exemple peut être généralisé afin d'établir les conditions nécessaires pour pouvoir déterminer directement les fonctions de transfert à partir des fonctions linéaires. Considérons un sous-système, constitué d'une sortie y et de plusieurs entrées u_k. La sortie y est une fonction linéaire des entrées u_k et des versions retardées de la sortie y comme définie ci-dessous :

$$y(n) = f_{yy}\Big(y(n-1)\dots y(n-i)\Big) + \sum_k f_{yu_k}\Big(u(n)\dots u(n-j)\Big) \quad \text{avec } i \geq 1,\ j \geq 0 \tag{3.86}$$

Propriété 1 : si un circuit C est présent dans le graphe G_{sn}, l'expression de la fonction linéaire correspond à l'équation 3.86 si la sortie considérée pour l'évaluation de la fonction linéaire appartient à ce circuit C et si au moins un opérateur retard est présent dans le circuit.

Propriété 2 : la fonction de transfert finale de ce sous-système peut être déterminée directement à partir de la transformée en $\mathcal{Z}$ de la fonction linéaire 3.86 si la fonction linéaire contient explicitement tous les termes associés aux versions retardées de la sortie y. Par conséquent, si une entrée quelconque u_k est une fonction linéaire de la sortie y, des substitutions de variables doivent être opérées avant la détermination de la fonction de transfert finale.

La transformation T_2 est décomposée en quatre parties différentes représentées à la figure 3.16. Les circuits sont tout d'abord identifiés puis démantelés pour obtenir différents graphes acycliques G_k. Les fonctions linéaires de chaque graphe acyclique sont définies en parcourant le graphe des sources vers la racine. La troisième étape consiste à déterminer les fonctions de transfert partielles associées à chaque graphe acyclique. Au cours de cette étape un ensemble de substitutions de variables entre les différentes fonctions linéaires est effectué pour respecter la propriété 2 présentée ci-dessus et afin d'aboutir à des fonctions linéaires telles que celle présentée à l'équation 3.86. La dernière étape permet d'obtenir les fonctions de transfert globales entre la sortie et les entrées du système à partie des différentes fonctions de transfert issues de l'étape précédentes.

Démantèlement des circuits (T_{21})

Le but de cette première étape est de transformer le graphe G_{sn} en plusieurs graphes acycliques G_k si G_{sn} contient des circuits. Ceci est réalisé en énumérant puis en démantelant les différents circuits présents au sein du graphe. Si G_{sn} ne contient aucun circuit, alors ce graphe G_{sn} est déjà un graphe acyclique et la transformation T_{21} n'a pas lieu d'être appliquée. Deux types de graphe correspondant aux graphes cycliques et aux graphes acycliques sont amenés à être traités. Les graphes acycliques vont correspondre aux structures non-récursives et ces graphes peuvent être composés de nombreux nœuds. En effet, par exemple, l'ordre d'un filtre FIR ou le nombre de points traités par une FFT peuvent être importants et conduire à des

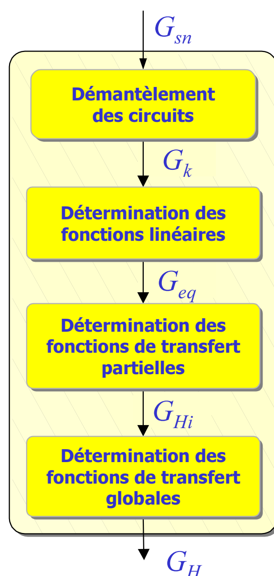


FIG. 3.16: Représentation des différentes étapes de la transformation T_2

graphes acycliques composés d'un nombre de nœuds élevé. Dans le cadre des structures récursives, le nombre de nœuds nécessaires pour représenter une application est plus faible que dans le cas des structures non-récursives. Les algorithmes d'énumération des circuits d'un graphe possèdent une complexité, dans le meilleur cas, polynomiale. Ainsi, l'utilisation de ces techniques sur des structures non-récursives conduit à des temps d'exécution pouvant être élevés alors qu'aucun circuit n'est présent. Ainsi, il est nécessaire de séparer les phases de détection et d'énumération des circuits au sein d'un graphe et de n'appliquer les phases d'énumération et de démantèlement que si le graphe contient des circuits.

Détection des circuits : Le but de cette première étape est de détecter rapidement la présence d'un circuit au sein du graphe et de décider si la transformation T_{21} doit être appliquée. Afin de minimiser la durée d'exécution de cette étape, la procédure de détection de circuits est interrompue dès qu'un circuit est détecté. Un algorithme de détection de circuits au sein d'un graphe basé sur un parcours en profondeur de celui-ci a été mis en œuvre.

Le parcours du graphe débute au niveau de la racine représentant la sortie du système et la détection d'un circuit se base sur le concept suivant : lors du parcours d'un graphe en profondeur, un nœud n_c membre d'un circuit est parcouru une seconde fois durant le parcours de ses prédécesseurs. En effet, le parcours d'un graphe permet de parcourir tous les chemins le composant et en particulier le chemin correspondant au circuit $C = \{n_c \ldots n_c\}$ et caractérisé par le fait que le nœud n_c est le nœud initial et terminal du chemin.

Lors du parcours classique en profondeur d'un graphe acyclique, un nœud peut être exploré plusieurs fois si celui-ci ou l'un de ses successeurs possède plus d'un successeur. Par conséquent, pour accélérer le parcours, chaque nœud n'est traité qu'une seule fois. En effet, si un nœud n_i a déjà été traité et si aucun circuit n'a été détecté durant l'exploration de ce nœud et de ses prédécesseurs, alors ce nœud et tous ses prédécesseurs n'appartiennent pas à un circuit. Ainsi, il n'est pas nécessaire de poursuivre l'exploration du sous graphe induit par le nœud n_i. La complexité de cet algorithme de détection de circuit au sein d'un graphe est linéaire par rapport au nombre de nœuds.

Lors du parcours du graphe, chaque nœud peut prendre 3 états différents. L'état initial de chaque nœud avant que celui-ci soit traité est *non traité*. Lors du traitement de ce nœud, son état passe à *en cours de traitement* puis à *traité* lorsque tous les prédécesseurs de ce nœud ont été traités. Un circuit est détecté si lors du parcours du nœud n_c, celui-ci se trouve déjà dans l'état *en cours de traitement*. Cet algorithme de détection de circuits, détaillé à la figure 3.17, se base sur une structure récursive et arrête l'exploration des nœuds dès qu'un circuit est détecté.

```
TraitementNoeud(Noeud N )
 {
    si EtatNoeud(N) ≠ TRAITÉ  - le nœud N n'a pas encore été traité
    {
       si EtatNoeud(N) = EN_TRAITEMENT   then
       {
          return TRUE    - le nœud N appartient à un circuit
       }
      else
       {
          EtatNoeud(N) = EN_TRAITEMENT

          Pour chaque Predécesseur(N) - exploration de chaque prédécesseur de N
          {
             ResPred = TraitementNoeud( Predécesseur(N) )

             si ResPred = TRUE  alors   return TRUE - un circuit a déjà été  détecté
          }

          EtatNoeud(N)=TRAITÉ       - fin du traitement du nœud N
       }
    }
   return FALSE
 }
```

FIG. 3.17: Description de l'algorithme de détection de circuits

Démantèlement des circuits : La transformation d'un graphe en un graphe acyclique nécessite de définir les nœuds où les circuits sont démantelés. Pour expliquer le choix de ces nœuds, prenons l'exemple proposé à la figure 3.18. Le graphe est constitué d'une sortie y, d'une entrée x et d'un circuit C ayant comme origine un nœud quelconque a . Il s'agit alors de déterminer la fonction de transfert $H_{YX}(z)$ entre la sortie $Y(z)$ et l'entrée $X(z)$. Étant donné que le nœud y n'appartient pas au circuit C, il est nécessaire, pour pouvoir déterminer correctement la fonction linéaire associée au circuit et pour respecter la *propriété 1* présentée au début de cette section, de définir une nouvelle variable intermédiaire $p(n)$. La fonction de transfert $H_{YX}(z)$ est alors le produit de deux fonctions de transfert définies ci-dessous :

$$H_{YX}(z) = \frac{Y(z)}{X(z)} = \frac{Y(z)}{P(z)}\frac{P(z)}{X(z)} = H_{YP}(z)H_{PX}(z) \tag{3.87}$$

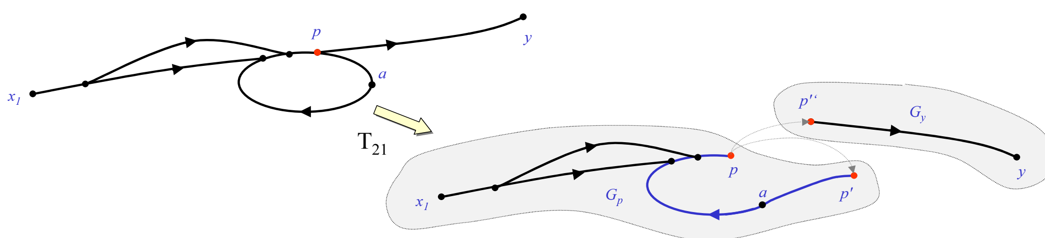


FIG. 3.18: Exemple pour la transformation T_{21} correspondant au démantèlement des circuits

Le nœud p doit appartenir au circuit, ainsi nous obtenons $C = \{a, \ldots, p, \ldots a\}$. Pour obtenir la fonction de transfert de l'équation 3.87, il faut déterminer deux fonctions linéaires. Elles s'obtiennent à partir des deux graphes acycliques G_y et G_p. La racine du graphe G_p est le nœud p et les sources sont le nœud x et le nœud p', duplication du nœud p obtenu après démantèlement du circuit au niveau de ce nœud. La racine du graphe G_y est le nœud y et ce graphe possède comme source le nœud p'', duplication du nœud p et représentant la variable $p(n)$.

Pour obtenir deux graphes distincts possédant chacun une seule racine, uniquement la variable $p(n)$ doit être commune aux deux graphes. Ainsi, si $L = \{a, \ldots, p, \ldots, y\}$ est le chemin entre les nœuds a et y, le point p est défini de telle manière que p soit le dernier point commun entre le circuit C et le chemin L. Ainsi, nous avons la relation suivante :

$$L \cup C = \{a, \ldots, p\} \tag{3.88}$$

Après détection de chaque circuit $C = \{a, \ldots, p_1, \ldots, p_i \ldots a\}$, tous les chemins L_i entre les nœuds a et y sont énumérés. Le graphe est démantelé au niveau des nœuds p_i correspondant au dernier nœud commun entre le circuit C et le $i^{ème}$ chemin L_i. Ces démantèlements permettent d'obtenir les différents graphes acycliques ne possédant qu'une seule racine. Un exemple avec deux chemins entre a et y est présenté à la figure 3.19. Cette transformation peut conduire à une répartition des éléments du circuit au sein de plusieurs graphes acycliques. La fonction linéaire associée à tous les éléments du circuit sera obtenue par un ensemble de substitutions de variables détaillé par la suite.

Énumération des circuits : Différentes classes d'algorithmes peuvent être utilisées pour énumérer l'ensemble des circuits présents au sein d'un graphe [95]. Nous avons retenu l'algorithme proposé par Johnson [52] qui permet d'énumérer les circuits en un temps polynomial. Cet algorithme est basé sur les techniques d'exploration de solutions au sein d'un ensemble. Cet algorithme détecte les circuits élémentaires d'un graphe orienté G_{sn} pour lequel les nœuds sont numérotés. Le graphe est représenté par une matrice définissant pour chaque nœud n_i, l'ensemble de ses successeurs. Les éléments de la $i^{ème}$ ligne de cette matrice représentent les successeurs du nœud n_i.

Cet algorithme utilise une méthode de retour arrière pour énumérer les chemins du graphe orienté puis pour identifier s'il s'agit de circuits. Pour traiter une seule fois chaque circuit, la technique utilisée au sein de l'algorithme de Tiernan [139] est mise en œuvre. Chaque circuit est construit à partir d'un nœud racine s dans un sous-graphe induit par s et par les nœuds plus grands que s. Dans le cadre de l'algorithme de Tiernan, le temps nécessaire pour détecter c circuits dans un graphe composé de n nœuds et de e arcs est borné par une limite supérieure en $O(n(k)^n)$ avec k constant [95]. L'algorithme de Johnson [52] basé sur les mêmes techniques améliore la durée d'exécution en réduisant l'espace de recherche des différents chemins. Ceci conduit à une durée d'exécution dont la limite est en $O((n+e)c)$. Cet algorithme a été modifié pour pouvoir énumérer l'ensemble des chemins entre deux nœuds.

Détermination des fonctions linéaires (T_{22})

Le graphe $G_{eq} = (N_{eq}, E_{eq})$ est un graphe orienté et annoté spécifiant l'algorithme par un ensemble de fonctions linéaires. Les nœuds de ce graphe correspondent au nœud de sortie y, à l'ensemble des variables intermédiaires définies lors du démantèlement d'un circuit et à l'ensemble des entrées du système spécifié au niveau bruit (entrées x_j, sources de bruit associées aux x_j et sources de bruit généré).

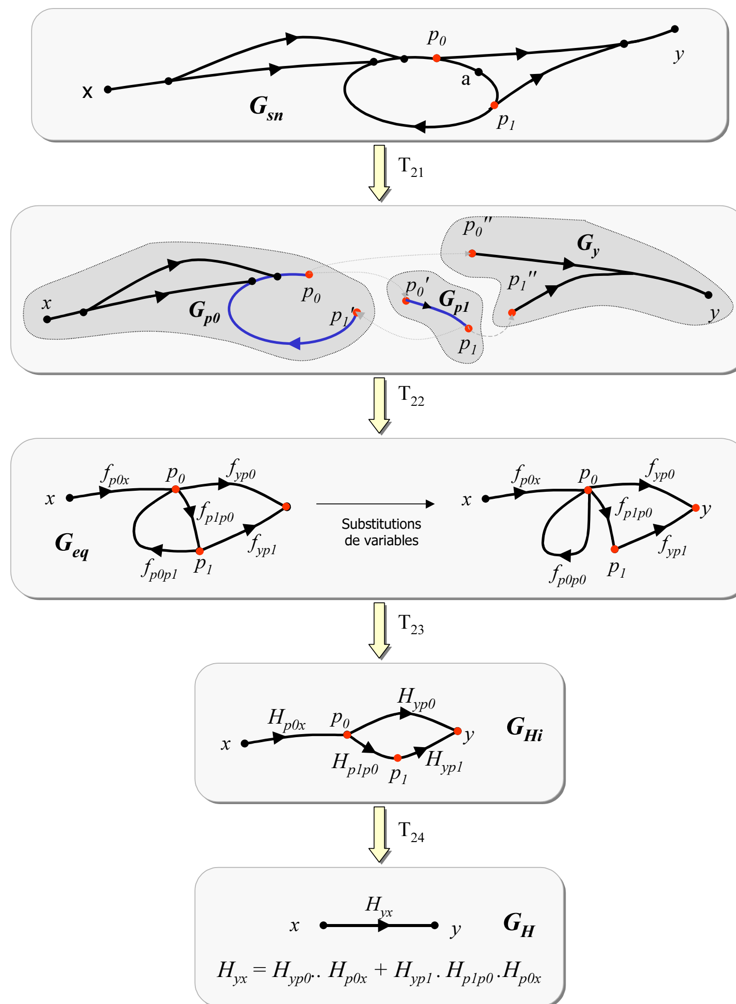


FIG. 3.19: Exemple pour la transformation T_2

Soit f_w une fonction linéaire des variables u, v et des valeurs précédentes de la variable $w(n)$. Cette fonction linéaire globale est la somme de trois autres fonctions linéaires partielles définies comme suit :

$$w(n) = f_w(\ldots u(n-i_u), \ldots v(n-i_v), \ldots w(n-1-i_w), \ldots) \quad \text{avec } i_u, i_v, i_w \geq 0 \quad (3.89)$$

$$w(n) = f_{wu}(\ldots u(n-i_u)\ldots) + f_{wv}(\ldots v(n-i_v)\ldots) + f_{ww}(\ldots w(n-1-i_w)\ldots) \quad (3.90)$$

Le graphe G_{eq} associé à cet exemple est donné à la figure 3.20. Un arc (u, w, f_{wu}) annoté par f_{wu} et orienté de u vers w signifie que la variable w est définie à partir de la variable u à l'aide de la fonction linéaire f_{wu}.

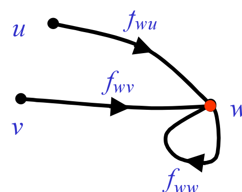


FIG. 3.20: Exemple de graphe G_{eq} représentant les différentes fonctions linéaires

L'objectif de la transformation T_{22} est de construire ce graphe G_{eq} à partir des différents

graphes acycliques G_k. La fonction linéaire globale f_w associée à un graphe G_w s'obtient par un parcours en profondeur de ce graphe. Un algorithme récursif post-traitement [48] est utilisé pour parcourir le graphe. Pour les nœuds n'appartenant pas aux sources du graphe, l'algorithme explore tout d'abord le nœud prédécesseur de gauche, puis de droite puis détermine ensuite la fonction linéaire à partir des résultats de chaque prédécesseur et de l'opérateur. Lors de l'exploration d'une source, le nom de la variable associée à ce nœud est renvoyé. Cet algorithme est décrit à la figure 3.21.

Cette transformation nécessite de définir une structure permettant de décrire efficacement une fonction linéaire. Chaque fonction linéaire globale f_w est représentée par une constante et une liste de fonctions linéaires partielles. Chaque fonction linéaire partielle f_{wv} est caractérisée par la variable d'entrée v associée et est représentée par une liste d'éléments. Chaque élément représente un terme du type $a_i z^{-r_i}$ et est caractérisé par un retard r_i et le terme a_i correspondant à une combinaison des différents coefficients présents au sein du graphe. Des méthodes permettant d'effectuer les opérations arithmétiques sur ces fonctions linéaires ont été mises en œuvre.

```
FonctionLinéaire( T )
{
  si T∈ N_source  - T est une source du graphe
  {
     return NomVariable(T)
  }
  sinon
  {  si  T∈ N_o    - T est un noeud opération
     {
        f_gauche = FonctionLinéaire( PrédécesseurGauche(T) )
        f_droite  = FonctionLinéaire( PrédécesseurDroite(T) )

        f = f_gauche Ops(T) f_droite
     }
     sinon  - T est une donnée
     {
        f = FonctionLinéaire( Prédécesseur(T) )
     }
  }

  return f;
}
```

FIG. 3.21: Algorithme récursif de détermination des fonctions linéaires

Détermination des fonctions de transfert partielles (T_{23})

Le graphe $G_{Hi} = (N_{G_{Hi}}, E_{G_{Hi}})$ est un graphe orienté et annoté spécifiant l'algorithme par un ensemble de fonctions de transfert intermédiaires. Les nœuds du graphe G_{Hi} appartiennent à l'ensemble N_{eq}. L'arc (u, w, H_{wu}) orienté de u vers w est annoté par la fonction de transfert H_{wu} entre les variables associées aux nœuds w et u.

Pour déterminer correctement les fonctions de transfert à partir des fonctions linéaires, il est nécessaire de respecter la *propriété 2* présentée à la page 103. Ceci implique de détecter et de démanteler au sein du graphe G_{eq}, tous les circuits d'une longueur supérieure à l'unité. Après ce traitement, le graphe G_{eq} ne doit contenir que des circuits de longueur unitaire. Un circuit de longueur unitaire associé à une variable $w(n)$ signifie que la variable est définie à partir de ses versions précédentes $w(n-k)$ avec $k > 0$. Un circuit C de longueur supérieure à l'unité est

transformé en un circuit de longueur unitaire en réalisant une suite de substitutions de variables au niveau de la fonction linéaire associée à l'une des variables du circuit C.

Pour une variable w_i appartenant à un circuit, le processus de substitution de variables consiste à parcourir en sens inverse l'ensemble des chemins L_i dont le nœud terminal est w_i et à substituer au sein de la fonction linéaire associée à un nœud les fonctions linéaires associées aux nœuds prédécesseurs. Ce processus de substitutions de variables est arrêté lorsque la fonction linéaire associée à la variable w_i ne contient plus de variables appartenant à un circuit. Ainsi, ce processus n'aboutit à une solution que si pour tous les chemins L_c de L_i ayant un nœud commun avec un circuit, les nœuds initiaux de L_c correspondent au nœud w_i. Ainsi, dans le cas de la présence de plusieurs circuits, il faut sélectionner pour chaque groupe de circuits ayant un point commun, l'un des nœuds commun à tous les circuits du groupe considéré, pour réaliser la suite de substitutions de variables.

Pour illustrer ce processus de substitution de variables, l'exemple d'un filtre basé sur une structure en treillis est présenté à la figure 3.22. Trois circuits $(w_1, w_2, w_3, w_5, w_1)$, (w_1, w_2, w_5, w_1) et (w_2, w_3, w_2) sont présents. Uniquement le nœud w_2 est commun aux trois circuits, ainsi il est possible d'exprimer la variable $w_2(n)$ en fonction des valeurs précédentes $w_2(n-k)$ et des autres variables du système n'appartenant pas à l'un des circuits. En effet, tous les chemins correspondant à des circuits et ayant pour nœud terminal w_2 possèdent comme nœud initial w_2. Les substitutions de variables effectuées sont représentées à l'aide d'un arbre à la figure 3.22.c. Après ce traitement, la variable $w_2(n)$ est fonction uniquement de ses valeurs précédentes et de $x(n)$. Le graphe G_{eq} résultant est présenté à la figure 3.22.d et il ne contient plus aucun circuit de longueur strictement supérieure à un.

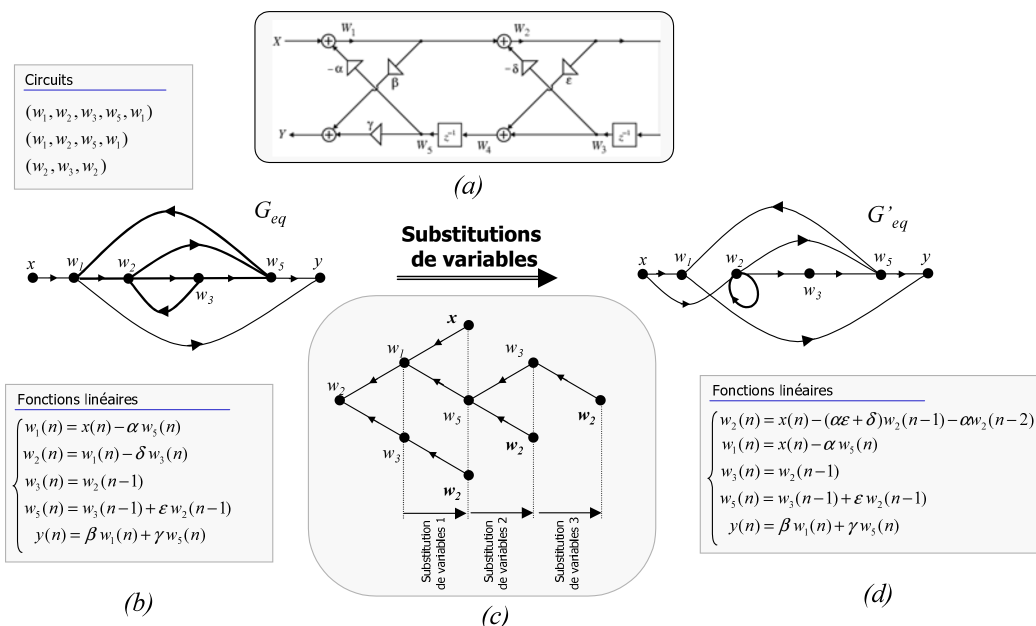


FIG. 3.22: Exemple de substitutions de variables

La seconde étape de cette transformation consiste à calculer la fonction de transfert à partir des fonctions linéaires à l'aide de la transformée en $\mathcal{Z}$. Considérons le graphe G_{eq} présenté à la figure 3.20. Ce graphe est composé de deux sources u et v et d'une racine w. Un circuit de longueur unitaire est associé au nœud w et annoté avec la fonction linéaire partielle f_{ww}. En effet, les différentes substitutions de variables ont permis de regrouper au sein de la fonction linéaire f_{ww}, l'ensemble des éléments faisant appel à une version retardée de la variable $w(n)$. Ainsi, la fonction linéaire f_w correspond à la somme de trois fonctions linéaires partielles f_{wu}, f_{wv} et f_{ww}. D'après la règle de Mason [94], les fonctions de transfert H_{wu} et H_{wv} entre la sortie w et les entrées u et v sont obtenues à l'aide de la transformée en $\mathcal{Z}$ de la manière suivante :

$$H_{wu}(z) = \frac{W(z)}{U(z)} = \frac{\mathcal{Z}(f_{wu})}{1-\mathcal{Z}(f_{ww})} \quad (3.91)$$

$$H_{wv}(z) = \frac{W(z)}{V(z)} = \frac{\mathcal{Z}(f_{wv})}{1-\mathcal{Z}(f_{ww})} \quad (3.92)$$

Détermination des fonctions de transfert globales (T_{24})

Le graphe $A_H = (N_{A_H}, E_{A_H})$ est un arbre annoté spécifiant le système avec un ensemble de fonctions de transfert globales entre la sortie et chacune des entrées du système. Cet arbre représente la modélisation du système donnée par la figure 3.10. La racine de cet arbre A_H est le nœud de sortie y et les feuilles correspondent aux nœuds d'entrée du système au niveau bruit. Chaque arc orienté est annoté par la fonction de transfert entre la sortie et l'entrée. L'objectif de cette transformation T_{24} est de calculer les fonctions de transfert globales entre la sortie et chaque entrée en éliminant les nœuds appartenant à l'ensemble N_p et représentant les variables intermédiaires.

Dans le graphe G_{H_i}, les chemins entre les entrées et la sortie sont énumérés. Soit L_{yx}, le chemin entre une entrée x et la sortie y défini de la manière suivante :

$$L_{yx} = \{x \ldots p_0 \ldots p_k \ldots p_K \ldots y\} \quad \text{avec } k = 0 \ldots K \quad \text{et } p_k \in N_p \quad (3.93)$$

La fonction de transfert globale H_{YX} entre la sortie $Y(z)$ et l'entrée $X(z)$ est déterminée de la manière suivante :

$$H_{YX}(z) = \frac{Y(z)}{X(z)} = H_{YP_K}(z) \times \ldots H_{P_k P_{k-1}}(z) \times \ldots \times H_{P_1 P_0}(z) \times H_{P_0 X}(z) \quad (3.94)$$

Cette transformation nécessite de définir une structure permettant de représenter une fonction de transfert et de mettre en œuvre des méthodes permettant de réaliser les opérations arithmétiques entre les fonctions de transfert. Pour accélérer le développement de notre méthodologie, le calcul des fonctions de transfert globales à partir des fonctions de transfert partielles est réalisé à l'aide de l'outil Matlab. En effet, ce dernier possède un objet permettant de représenter les fonctions de transfert. De plus, le concept de surcharge des opérateurs a été mis en œuvre dans le cas des opérations de type addition, soustraction et multiplication.

3.4.4 Détermination de l'expression du RSBQ

L'objectif de cette dernière transformation est de déterminer l'expression du rapport signal à bruit de quantification (RSBQ). La puissance du signal en sortie du système est constante car elle est indépendante de la manière dont les données sont codées en virgule fixe. Ainsi, cette puissance est soit définie par l'utilisateur ou soit calculée à partir des paramètres des signaux d'entrée et de la fonction de transfert entre la sortie et les entrées. Ces fonctions de transfert sont déterminées au cours de la transformation T_2.

L'expression globale de la puissance du bruit de quantification est définie selon l'équation 3.73 et fait appel aux expressions des paramètres statistiques des bruits b_q et b_h. Pour le bruit b_q lié aux bruits associés à chaque entrée x_j et aux bruits générés au sein du système, les paramètres statistiques de chaque source de bruit et le gain entre la sortie et la source de bruit doivent être définis. Ce gain est calculé à partir de la réponse fréquentielle de la fonction de transfert entre la sortie et la source de bruit. Sachant que ce gain est indépendant du codage des coefficients en virgule fixe, celui-ci peut être calculé qu'une seule fois.

Pour le bruit b_h lié à la quantification des coefficients, la fonction de transfert $\Delta Hj(z)$ dépend directement de la manière de coder les coefficients en virgule fixe. Ainsi, cette fonction de

transfert doit être calculée à chaque évaluation du RSBQ. La densité spectrale de puissance mutuelle entre les entrées et la densité spectrale de puissance de chaque entrée doivent être définies par l'utilisateur.

3.4.5 Conclusions

Nous avons détaillé dans cette partie, la méthodologie mise en ouvre pour déterminer l'expression du RSBQ dans le cadre des systèmes linéaires. Un outil permettant d'implanter celle-ci a été développé. Exceptées les transformations T_{24} et T_3 qui utilisent l'outil Matlab, les différentes transformations ont été développées en langage C en utilisant les structures de données présentes au sein de l'outil GAUT [91] pour représenter les graphes flot de données et les graphes flot de signal. Les résultats obtenus avec cet outil dans le cadre d'un filtre récursif sont présentés dans l'annexe A [98]. Les différentes fonctions de transfert déterminées par l'outil sont détaillées.

Dans le cadre des systèmes non-linéaires, actuellement, l'outil de détermination de l'expression du RSBQ n'est pas développé. Cependant, les différentes étapes nécessaires pour obtenir cette expression sont résumées ci-dessous. La première étape consiste à modéliser le système au niveau bruit de quantification et correspond à la transformation T_1. Ensuite, l'expression du bruit est déterminée en parcourant le graphe des sources vers la racine selon la technique présentée à la transformation T_{22}. La troisième étape détermine l'expression du moment du second ordre du bruit b_y à partir de son expression, obtenue dans l'étape précédente. Dans cette dernière étape, une simulation de l'algorithme en précision infinie est réalisée afin de collecter les différents signaux présents au sein de l'application en vue de déterminer les paramètres statistiques nécessaires à la détermination de P_{b_y}

3.5 Expérimentations

Différentes expérimentations ont été menées afin de valider cette méthodologie de détermination de l'expression analytique du RSBQ. La validation de cette méthode nécessite de mesurer la précision des estimations obtenues avec la modélisation proposée et de vérifier que la méthodologie fournit une expression du RSBQ correcte. De plus, le temps d'obtention de l'expression du RSBQ doit être évalué afin de montrer l'efficacité de notre approche par rapport aux méthodes basées sur la simulation. Dans un premier temps, la précision du modèle de bruit généré que nous avons proposé, est analysée. Ensuite, la qualité de l'estimation de la puissance du bruit de quantification en sortie d'un système est étudiée pour les systèmes linéaires et non linéaires. Dans la dernière section, le temps d'exécution de l'outil développé dans le cadre des systèmes linéaires est évalué.

3.5.1 Estimation des paramètres statistiques du bruit généré

La qualité du modèle de bruit généré, utilisé dans notre méthodologie, a été analysée. Pour cela, nous avons comparé les paramètres statistiques obtenus à l'aide des expressions présentées dans le tableau 3.1 et ceux mesurés à partir d'une simulation en virgule fixe. L'erreur relative entre la puissance du bruit de quantification estimée $P_{b_g est}$ et celle mesurée $P_{b_g mes}$ a été calculée de la manière suivante :

$$E_r = \frac{|P_{b_g mes} - P_{b_g est}|}{P_{b_g mes}} \tag{3.95}$$

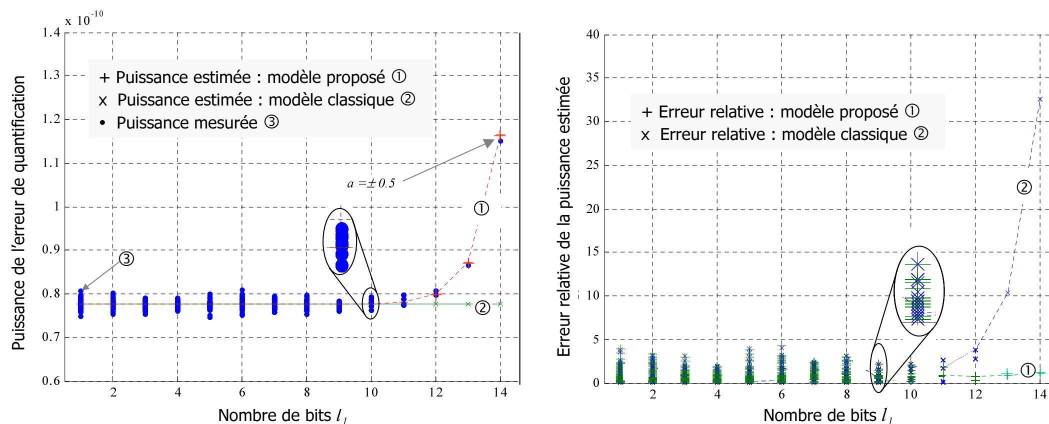


FIG. 3.23: Évolution de la puissance du bruit de quantification mesurée et estimée et des erreurs relatives des estimations en fonction du nombre de bits l_1 associé à la constante C dans le cas de la quantification par arrondi du résultat de la multiplication $x \times C$

Nous avons détaillé plus particulièrement dans cette section les résultats concernant le nouveau modèle proposé. Les paramètres statistiques du signal y issu de la quantification du résultat de la multiplication d'un signal x par une constante C ont été mesurés et estimés. Les signaux x, y et la constante C sont codés sur 16 bits. Pour la constante C, le nombre l_1 de bits égaux à 0 évolue entre 0 à 14. Les autres bits de la constante sont définis de manière aléatoire. Pour montrer l'apport de notre modèle, nous avons comparé les estimations obtenues à l'aide de notre modèle et celles issues du modèle classique proposé dans [24] et qui ne prend pas en compte le paramètre l_1. La puissance du bruit de quantification mesurée et estimée et les erreurs relatives des estimations sont présentées à la figure 3.23 pour une loi de quantification par arrondi et à la figure 3.24 pour une loi de quantification par troncature. Pour une valeur donnée de l_1, différentes valeurs de la puissance mesurée sont obtenues en fonction de la valeur de la constante C.

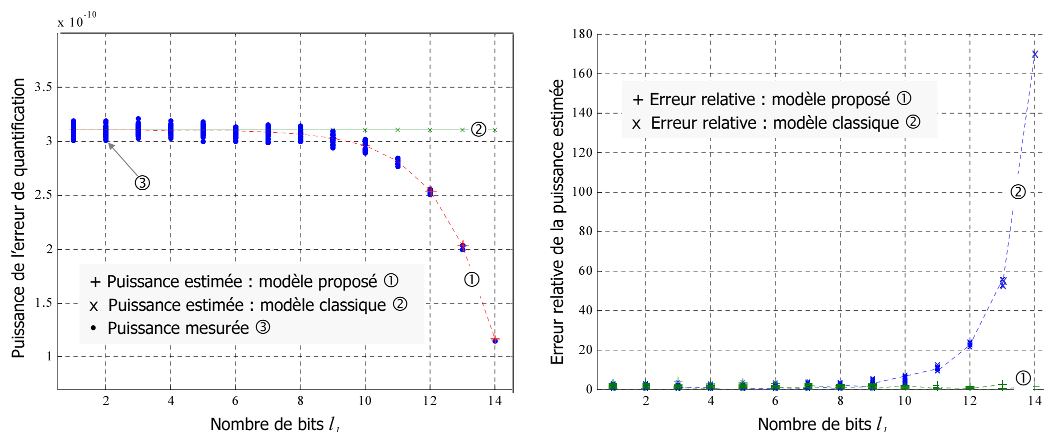


FIG. 3.24: Évolution de la puissance du bruit de quantification mesurée et estimée et des erreurs relatives des estimations en fonction du nombre de bits l_1 associé à la constante C dans le cas de la quantification par troncature du résultat de la multiplication $x \times C$

Lorsque le nombre de bits l_1 est faible, les résultats des deux estimations sont identiques et l'erreur relative est inférieure à 5%. Lorsque le nombre de bits l_1 est élevé, notre modèle permet toujours d'estimer précisément la puissance du bruit de quantification. En effet, l'erreur relative reste inférieure à 3.7%. En revanche, les estimations issues du modèle classique sont nettement moins précises. La valeur maximale de l'erreur relative est de 33% dans le cadre d'une quantification par arrondi et de 170% pour une loi de quantification par troncature. Ces résultats sont obtenus lorsque la constante est égale à ± 0.5 si nous considérons un codage en virgule fixe cadrée à gauche.

Notre modèle permet d'estimer précisément la puissance du bruit issu de la quantification du résultat de la multiplication d'un signal par une constante. Ce modèle permet d'obtenir une estimation nettement plus précise de la puissance du bruit de quantification en sortie de certaines applications utilisant des constantes ayant un paramètre l_1 élevé. Ce type d'application correspond par exemple à la corrélation d'un signal par un code bipolaire composé des valeurs 1 et -1. Cet exemple d'application est traité dans le paragraphe suivant.

Dans le cadre de la quantification du résultat de la multiplication ou de l'addition de deux signaux, les estimations obtenues à l'aide du modèle proposé dans [24] sont très précises. L'erreur relative de l'estimation est proche de celle obtenue dans le cas précédent avec un paramètre l_1 proche de 0.

3.5.2 Estimation de la puissance du bruit dans le cadre des systèmes linéaires

La qualité de l'estimation de la puissance du bruit de quantification en sortie d'un système linéaire a été analysée pour des structures récursives et non-récursives. Pour chaque application nous avons déterminé l'erreur relative entre la puissance du bruit estimée et celle mesurée en sortie de l'application. Différentes implantations de chaque application ont été effectuées afin de tester les différents aspects de la méthodologie. Ces implantations ont été obtenues en modifiant le type de recadrage des données, les lois de quantification utilisées et la largeur de la sortie du multiplieur. Différents types de signaux ont été utilisés en entrée des applications.

Afin d'apprécier la qualité de l'estimation, il est intéressant d'exprimer la précision de l'estimation en termes de bits. En effet, pour une puissance de bruit P donnée, il est possible de définir la largeur de la donnée qui génère, après quantification, un bruit dont la puissance de quantification est identique à P. Ainsi, une erreur d'estimation équivalente à 1/4 de bit se traduit par une erreur relative de l'ordre de 30 à 40% et une erreur de 1/10 de bit correspond

à une erreur relative de 14%.

Structures non-récursives

Dans le cadre des structures non-récursives, différents types d'application ont été testés et les résultats sont présentés à la figure 3.25. Pour une même application, différentes valeurs de l'erreur relative correspondant à différentes spécifications en virgule fixe de l'application, sont fournies. Les applications regroupées sous la dénomination *corrélateur* réalisent la corrélation entre un signal et un code de longueur N_{code} dont les valeurs c_i n'évoluent pas au cours du traitement. Dans le cadre de la corrélation complexe, une multiplication complexe entre le code complexe et le signal complexe est réalisée. Ensuite, les accumulations des parties réelles et imaginaires sont effectuées séparément. L'application *rake receiver* implante une technique de réception utilisant la diversité des trajets dans le cadre des transmissions à étalement de spectre par séquence directe. Cette application est présentée dans la partie 5.2.2 à la page 172.

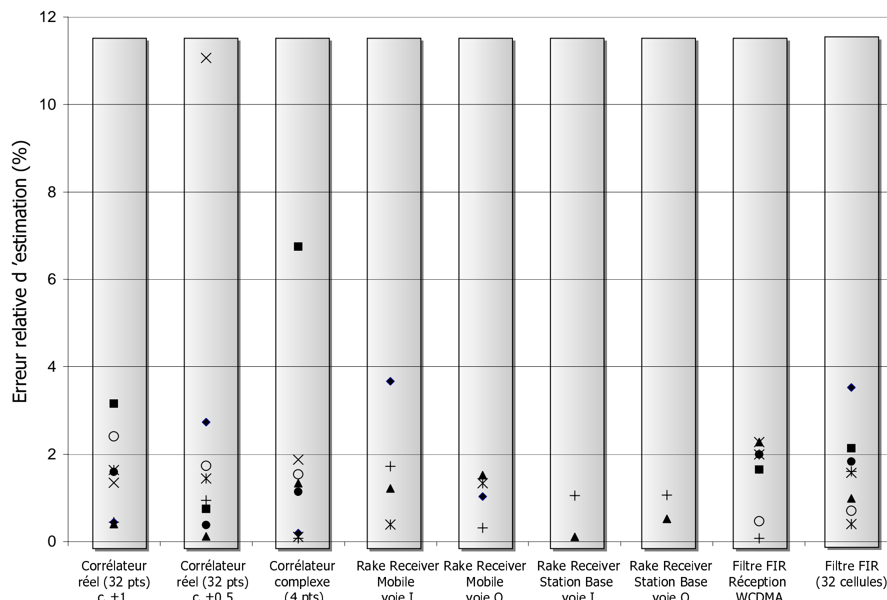


FIG. 3.25: Erreur relative sur l'estimation de P_{b_y} pour différentes applications non-récursives

Ces résultats montrent que les estimations obtenues pour différentes structures non récursives sont très précises. La valeur maximale de l'erreur relative est inférieure à 11% et sur les 44 estimations présentées uniquement 2 estimations conduisent à une erreur relative supérieure à 4%. Ces résultats montrent que la modélisation mise en œuvre au sein de cette méthodologie permet d'estimer très précisément la puissance du bruit de quantification en sortie d'une structure non-récursive.

Nous avons analysé l'influence du codage des coefficients d'un système linéaire sur le bruit de quantification en sortie de ce système. La puissance P_{b_y} du bruit de quantification en sortie d'un filtre FIR composé de 32 cellules a été évaluée pour différents formats de codage des coefficients. Le filtre FIR possède une entrée et une sortie codées sur 16 bits. La largeur des coefficients évolue entre 8 et 20 bits. Les résultats sont présentés à la figure 3.26.

L'estimation de la puissance P_{b_y} a été comparée avec la puissance mesurée à partir d'une simulation en virgule fixe. Pour les différentes largeurs de coefficient, l'erreur relative de l'estimation reste inférieure à 8%. Ainsi, notre modèle permet d'estimer précisément le bruit b_h lié à la quantification des coefficients. L'influence de ce bruit b_h est relativement faible lorsque la largeur des coefficients est supérieure à 16 bits. La différence entre les RSBQ obtenus pour

une largeur de 16 bits et de 24 bits est égale à 0.02 dB. En revanche lorsque la largeur des coefficients diminue, l'influence de ce bruit b_h n'est plus négligeable. L'utilisation de coefficients codés sur 8 bits se traduit par une dégradation du RSBQ de 32 dB par rapport à la valeur obtenue pour des coefficients codés sur 24 bits. Ce cas de figure peut se présenter pour certains processeurs tels que les DSP SP5 [1] et UniPHY de la société 3DSP. Ils permettent d'utiliser des instructions SWP réalisant des multiplications non symétriques avec un premier opérande de largeur 8 bits et un second opérande de largeur 16 bits (voir tableau 2.2, page 41). Cependant, cette dégradation est inférieure à celle obtenue si la largeur du signal est fixée à 8 bits et la largeur des coefficients est fixée 16 bits. Dans ce cas, la dégradation est de 48 dB.

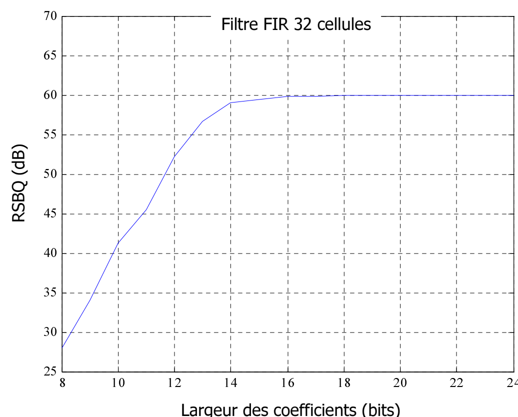


FIG. 3.26: Analyse de l'influence du codage des coefficients. Évolution du RSBQ en sortie d'un filtre FIR en fonction de la largeur des coefficients

Structures récursives

Dans le cadre des structures récursives, différents filtres à réponse impulsionnelle infinie (IIR) ont été testés. Les résultats obtenus pour deux filtres IIR d'ordre 8 et implantés sous la forme de cellules cascadées d'ordre 2 sont présentés à la figure 3.27. Pour chaque filtre, l'erreur d'estimation relative présente en sortie de chaque cellule est fournie.

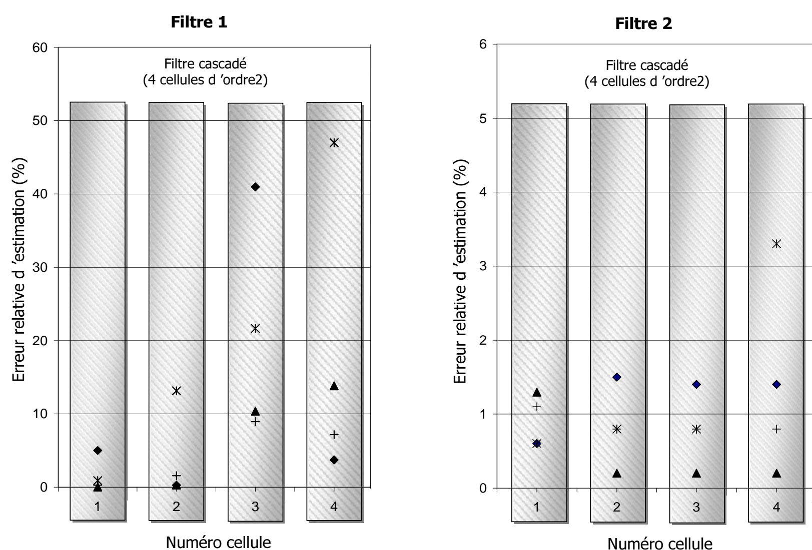


FIG. 3.27: Erreur relative sur l'estimation de P_{b_y} pour deux filtres IIR

Pour le second filtre, la valeur maximale de l'erreur relative d'estimation est inférieure à 3.3%

et sur les 16 estimations présentées une seule estimation conduit à une erreur relative supérieure à 1,5%. Ainsi, pour ce filtre, les estimations réalisées sont très précises. Cependant, pour le premier filtre, la qualité des résultats obtenus est inférieure. La valeur maximale de l'erreur relative est égale à 47%. Sur les 16 estimations présentées, trois estimations conduisent à une erreur relative supérieure à 15%. Pour les cas conduisant à une erreur d'estimation importante, une étude plus approfondie a montré que les paramètres statistiques des bruits générés étaient correctement estimés et que les propriétés des bruits de quantification étaient respectées. Cependant, pour les deux cellules (cellules 3 et 4) conduisant à une erreur d'estimation élevée, les pôles de ces deux cellules sont très proches du cercle unité. Ainsi, ces cellules sont en limite de stabilité.

Influence du modèle de bruit généré

Dans la section 3.5.1, la précision de l'estimation des paramètres statistiques du bruit issu de la quantification du résultat de la multiplication d'un signal par une constante a été évaluée et comparée avec celle obtenue avec un modèle classique. Dans cette section, nous montrons l'apport de ce nouveau modèle pour l'estimation de la puissance du bruit de quantification en sortie d'un système. La puissance du bruit de quantification en sortie du système a été estimée avec le modèle proposé et le modèle classique. Les résultats sont présentés à la figure 3.28 dans le cadre de structures non-récursives. Pour chaque application, l'erreur d'estimation relative est présentée pour les deux modèles.

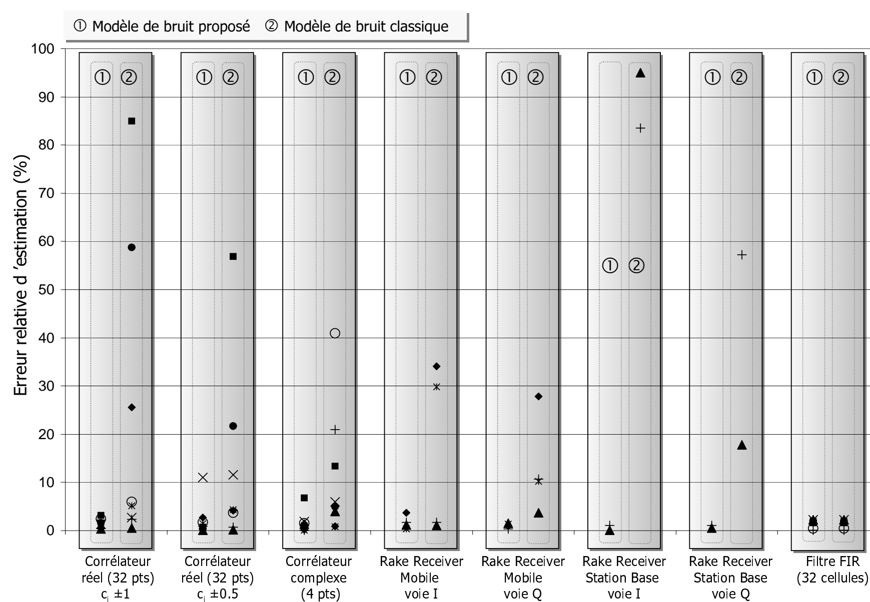


FIG. 3.28: Erreur relative sur l'estimation de P_{b_y}, comparaison des modèles de bruit

Pour certaines applications telles que le filtre FIR testé, les deux estimations conduisent à des résultats identiques. En effet, les coefficients de ce filtre possèdent un paramètre l_1 faible. Cependant, pour les applications dont les coefficients sont particuliers, l'utilisation du nouveau modèle permet d'estimer très précisément la puissance du bruit de sortie alors que le modèle classique fournit dans certains cas, une estimation très peu précise. En effet, l'erreur d'estimation relative maximale atteint 81% dans le cadre d'un corrélateur réel et 41% pour un corrélateur complexe. Ceci se traduit dans le cadre d'une application plus complexe telle que le *rake receiver* par une erreur d'estimation relative maximale de 95%. Pour ces différentes applications, les coefficients utilisés sont égaux à ± 1 ou ± 0.5.

Dans le cadre des filtres IIR, l'utilisation du modèle proposé permet de diminuer les erreurs

d'estimation en sortie de certaines cellules par rapport à un modèle classique. L'utilisation du modèle classique pour le premier filtre IIR présenté au paragraphe précédent, conduit à une erreur relative maximale de 61% et 8 estimations sur les 16 obtenues se traduisent par une erreur d'estimation relative supérieure à 15%.

Ces résultats montrent l'intérêt de notre modèle pour estimer la puissance du bruit de quantification en sortie d'un système. L'effort supplémentaire pour utiliser ce modèle étant peu important, celui-ci peut être mis en œuvre facilement au sein de la méthodologie.

3.5.3 Estimation de la puissance du bruit dans le cadre des systèmes non-linéaires

Différentes expérimentations ont été menées pour évaluer la précision de l'estimation pour différents systèmes non-linéaires. L'erreur relative entre la puissance mesurée et celle estimée est présentée à la figure 3.29, pour les différentes applications considérées. L'application d'évaluation de la puissance d'un signal complexe calcule le module au carré du signal. L'auto-corrélation d'un signal est calculée pour un retard donné. Cette application a été testée pour des signaux faiblement et fortement corrélés d'un point de vue temporel. L'écart entre la puissance du bruit obtenue pour un signal faiblement corrélé et un signal fortement corrélé est très élevé. Cet écart important montre la nécessité de prendre en compte la corrélation entre les signaux afin d'estimer correctement la puissance du bruit en sortie d'un système non-linéaire. Le filtre de Voltéra testé est un filtre non-linéaire du second ordre. Pour ces différentes applications, l'expression de la puissance du bruit a été obtenue à l'aide de la technique présentée dans la première partie de la section 3.3.2. L'application dénommée estimation de mouvement correspond à la mesure de la distance entre deux vecteurs u et v. Cette distance correspond à la somme des valeurs absolues des éléments du vecteur représentant la différence entre les vecteurs u et v (distance de Manhattan).

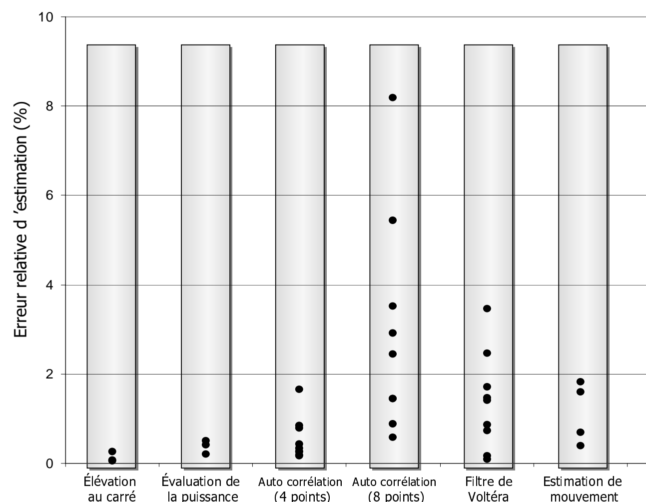


FIG. 3.29: Erreur relative sur l'estimation de P_{b_y} pour différentes applications non-linéaires

L'erreur relative obtenue pour les applications considérées est inférieure 8%. Cette erreur est relativement faible et est comparable à celle obtenue dans le cadre des systèmes linéaires non-récursifs. Ces résultats montrent que malgré les hypothèses posées, la modélisation proposée permet d'obtenir une estimation précise de la puissance du bruit en sortie d'un système non-linéaire.

3.5.4 Temps d'exécution de l'outil

Afin d'évaluer l'efficacité de notre méthodologie, le temps d'exécution de l'outil développé pour implanter cette méthodologie a été évalué. Deux versions de cet outil ont été testées. La première version (*v1*) implante la méthodologie décrite dans la partie précédente, elle permet d'obtenir en sortie l'expression du RSBQ en fonction de la largeur et de la position de la virgule des données. La seconde version *v2* de l'outil fournit la valeur du RSBQ obtenue pour une spécification en virgule fixe donnée. Ainsi, le nombre de sources de bruit présentes au sein du graphe G_{sn} est limité et correspond au nombre exact de sources de bruit pour la spécification considérée. Le format de chaque donnée est spécifié au sein d'un fichier d'entrée.

Structures non-récursives

Dans le cadre des structures non-récursives, la majorité du temps est consacrée à la transformation T_{22} qui correspond à la détermination de la fonction linéaire associée au graphe G_k. Les temps d'exécution de cette transformation T_{22}, obtenus dans le cadre de quelques applications basées sur des structures non-récursives sont détaillés pour les deux versions de l'outil dans le tableau 3.2.

Applications	N	Temps d'exécution (s)	
		version *v1*	version *v2*
FIR *(N cellules)*	32	0.27	0.01
	64	1.22	0.03
	128	6.46	0.2
	256	48.3	0.86
Corrélateur complexe *(N points)*	4	0.02	
	8	0.09	
	16	0.3	
	32	0.99	
	64	4.36	
	128	21.62	

TAB. 3.2: Temps d'exécution de la transformation T_{22}

Pour la seconde version de l'outil, les temps d'exécution obtenus sont très faibles même pour les graphes possédant un nombre de nœuds élevé tels que le filtre FIR composé de 256 cellules ou le corrélateur complexe sur 128 points. L'algorithme mis en œuvre pour déterminer la fonction linéaire associée à un graphe est basé sur l'exploration du graphe et possède une complexité qui est proportionnelle au nombre de nœuds présents au sein du graphe. De plus, pour que les nœuds ne soient pas explorés plusieurs fois, la fonction linéaire associée à un nœud est mémorisée si ce nœud est susceptible d'être visité une nouvelle fois. Ce cas se présente lorsqu'un nœud possède plusieurs successeurs.

Pour la première version de l'outil, les temps d'exécution de la transformation T_{22} sont plus élevés. Différentes raisons peuvent expliquer l'accroissement du temps d'exécution entre les deux versions de l'outil. Dans le cas de la première version, le nombre de nœuds présents au sein du graphe G_{sn} est plus élevé. En effet, ce graphe inclut toutes les sources de bruit généré potentielles. Nous avons constaté que pour une même application le nombre de nœuds de G_{sn} est 2.85 fois plus important lorsque l'ensemble des sources de bruit potentielles sont présentes au sein du graphe. Cet accroissement du nombre de nœuds se traduit par une augmentation du temps d'exécution des différentes transformations traitant ce graphe.

La structure utilisée actuellement pour spécifier les fonctions linéaires est composée d'un tableau de fonctions linéaires partielles. Une fonction linéaire partielle est affectée à chaque source de bruit. Ainsi, la taille de cette structure est proportionnelle au nombre de sources de

bruit. En conséquence, les traitements réalisés sur cette structure se traduisent par des temps d'exécution élevés lorsque le nombre de sources de bruit est important. Dans le cas du filtre FIR 256 ou du corrélateur sur 128 points, plus de 1200 sources de bruit sont présentes au sein du graphe. L'utilisation d'une structure dynamique ne prenant en compte que les sources de bruit réellement présentes au sein de la fonction linéaire permettrait de diminuer nettement la taille de la structure. Ainsi, nous pouvons supposer que la mise en œuvre d'une structure plus efficace pour spécifier les fonctions linéaires permettra de diminuer considérablement le temps d'exécution de cette transformation lorsque le nombre de sources de bruit est élevé.

Structures récursives

Dans le cadre des structures récursives, la majorité du temps est consacrée à l'énumération des circuits réalisée au sein de la transformation T_{21}. Les temps d'exécution mesurés pour différents filtres à réponse impulsionnelle infinie sont présentés dans le tableau 3.3 pour les deux versions de l'outil.

Applications	**Temps d'exécution (s)**	
	version *v1*	version *v2*
IIR 2	1.02	0.07
IIR 4	6.3	0.64
IIR 8	15.7	1.57

TAB. 3.3: Temps d'exécution de la transformation T_{21}

Le temps d'exécution de l'algorithme d'énumération de circuits dépend du nombre de nœuds, d'arcs et de circuits présents au sein du graphe. L'algorithme utilisé possède une complexité polynomiale. Le nombre de nœuds utilisés pour décrire une application récursive étant relativement faible, les temps d'exécution obtenus sont acceptables. Dans le cadre de la première version de l'outil, pour une même application, les temps d'exécution de l'outil sont plus importants. En effet, le nombre de nœuds présents au sein du graphe G_{sn} est plus important. L'ajout de toutes les sources de bruit potentielles, se traduit par une augmentation de 240% du nombre de nœuds au sein de G_{sn} et un accroissement du temps d'exécution de l'algorithme d'énumération des circuits.

A titre de comparaison, selon les expérimentations présentées dans [66], le temps de simulation en virgule fixe d'un filtre IIR d'ordre 4 est égal 16.3 s si une librairie optimisée est utilisée. L'utilisation de la technique de surcharge des types C++ aboutit à un temps de simulation de 340 s. Les expérimentations que nous avons réalisées avec la dernière version de l'outil de simulation en virgule fixe associé à Matlab conduisent à un temps de simulation de 8 s si 50000 échantillons sont utilisés en entrée du filtre IIR d'ordre 4.

Ainsi, le temps nécessaire à l'obtention de l'expression du RSBQ et le temps d'une simulation de l'algorithme en virgule fixe sont du même ordre de grandeur. En revanche, dans le cadre du processus de détermination et d'optimisation du format des données, notre méthodologie n'est utilisée qu'une seule fois alors que pour les techniques basées sur la simulation chaque évaluation du RSBQ nécessite une simulation. Ainsi, les temps d'optimisation obtenus avec notre méthode seront nettement plus faibles.

3.6 Détermination de la contrainte de précision minimale

3.6.1 Introduction

L'objectif de cette partie est de présenter la méthode proposée pour déterminer la valeur minimale de la précision ($RSBQ_{min}$) permettant de respecter les différents critères de qualité associés à l'application. Cette valeur minimale est utilisée par la suite au sein du processus de conversion en virgule fixe. Lors de la conversion en virgule fixe, le format des données est déterminé et optimisé afin que la précision des calculs soit supérieure à la contrainte de précision minimale ($RSBQ_{min}$). Ensuite, la spécification en virgule fixe obtenue en sortie du processus de conversion est simulée afin de vérifier si réellement les critères de qualité associés à l'application sont vérifiés. Dans le cas contraire, la contrainte de précision minimale est affinée et le processus est réitéré.

Le processus de vérification des contraintes de qualité associées à l'application est présenté à la figure 3.30.a. Celui-ci consiste simplement à vérifier que la sortie du système en précision finie $\widehat{y}$ permet de respecter les contraintes de qualité. Nous proposons de modéliser la sortie $\widehat{y}$ du système par la somme de la sortie y du système spécifié en précision infinie et b_y une source de bruit dont les caractéristiques sont présentées dans la section suivante. La modélisation proposée est présentée à la figure 3.30.b.

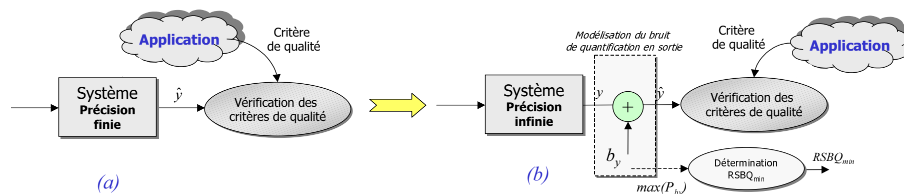


FIG. 3.30: Représentation du processus de détermination de la contrainte de précision minimale

Le processus de détermination de la valeur minimale du RSBQ, consiste à augmenter progressivement la puissance du bruit b_y tant que les critères de qualité associés à l'application sont respectés. La valeur du $RSBQ_{min}$ est directement obtenue à partir de la valeur maximale de la puissance du bruit de quantification. Cette technique nécessite pour l'implantation d'une application de réaliser une simulation de l'application en virgule flottante. Ensuite, une source de bruit dont la puissance est ajustée est sommée à la sortie de l'application obtenue en virgule flottante.

3.6.2 Modélisation du bruit de sortie

Propriétés du bruit de quantification de sortie

Le bruit de quantification présent en sortie de ce système résulte de la contribution des différentes sources de bruit présentes au sein du système. La contribution de chaque source dépend de sa puissance et du gain présent entre la sortie du système et la source. Pour définir les propriétés de la source de bruit modélisant le bruit de quantification présent en sortie du système, il est nécessaire de distinguer deux cas de figure. Le premier correspond au cas où une des sources de bruit présente en sortie prédomine et ainsi, le bruit de sortie est fortement lié à cette source de bruit. Dans ce cas, nous pouvons considérer que la source de bruit b_y est équivalente à la source de bruit prédominante. Les propriétés de cette source de bruit correspondent à celles du bruit issu de la quantification d'un signal et résumé dans la partie 3.2.3. En conséquence, la source de bruit b_y est modélisée par un bruit stationnaire et ergodique et uniformément réparti. Un exemple de ce type de bruit est présenté à la figure 3.31.b.1. La fonction de distribution et

la fonction d'autocorrélation de ce bruit sont représentées.

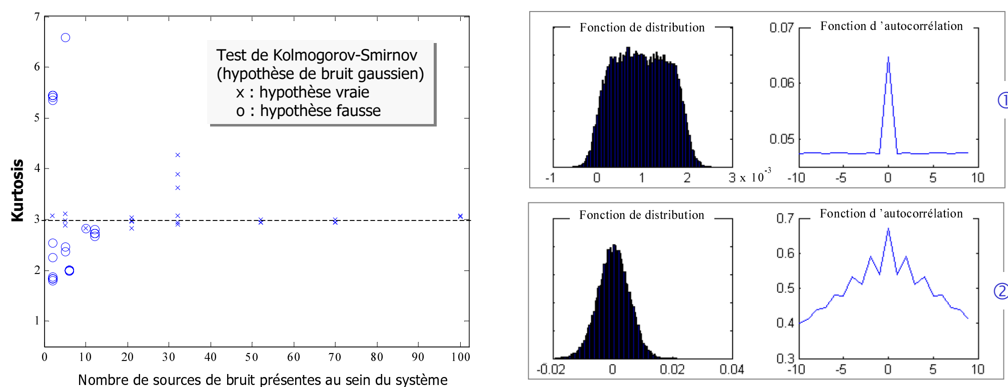


FIG. 3.31: Évaluation de l'hypothèse de bruit gaussien

Dans le second cas, les différentes sources de bruit contribuent toutes au bruit en sortie du système. Nous considérons qu'il n'existe pas de sources de bruit prédominantes en termes de puissance. Ainsi, si le nombre de sources présentes au sein de l'application est relativement élevé, alors la puissance de chaque source de bruit est faible par rapport à la puissance du bruit en sortie du système. En supposant que les différentes sources de bruit sont indépendantes, d'après le théorème de la limite centrale, la distribution du bruit en sortie du système peut être approximée par une distribution gaussienne lorsque le nombre de sources est élevé.

La validité de cette modélisation a été analysée à l'aide du test de Kolmogorov-Smirnov [125] sur différents bruits de quantification obtenus en sortie de systèmes en virgule fixe. Ce test permet de décider si une fonction de distribution est assimilable à une loi normale avec un seuil de confiance défini à 95% dans notre cas. De plus, le *kurtosis* de chaque bruit de quantification en sortie du système a été évalué. Cette métrique est égale à 3 dans le cas d'une distribution gaussienne. Les tests ont été réalisés sur 48 bruits de quantifications. Les résultats des tests et les valeurs du *kurtosis* sont représentés à la figure 3.31.a en fonction du nombre de sources de bruit présentes au sein du système.

Lorsque nous sommes dans le cas d'un nombre de sources de bruit faible, une majorité des bruits possède un kurtosis éloigné de 3 et le résultat du test est négatif. Ces bruits ont été obtenus dans le cas de système pour lesquels les résultats des calculs internes sont stockés en double précision et la sortie du système est stockée en simple précision. Ainsi, le bruit de quantification issu du passage en simple précision de la sortie du système prédomine et le modèle du bruit correspond à celui présenté dans le premier cas. L'analyse des fonctions de distribution obtenues confirme cette hypothèse. Lorsque le nombre de sources de bruit est plus important et qu'aucune source ne prédomine, l'hypothèse de bruit gaussien est vérifiée. Un exemple de ce type de bruit est présenté à la figure 3.31.b.2. Pour l'exemple considéré, les sources de bruits sont filtrées, ainsi le bruit de quantification en sortie n'est pas un bruit blanc.

Conclusions

La vérification des critères de qualité est réalisée avec deux types différents de source de bruit. Le premier type de bruit correspond à un bruit blanc uniformément réparti. Le second type de bruit correspond à un bruit gaussien. Deux types de bruit gaussien possédant des fonctions d'autocorrélation différentes sont testés. Les expérimentations sont réalisées avec des bruits pour lesquels les échantillons sont soit fortement soit faiblement corrélés d'un point de vue temporel. La vérification des critères de qualité est réalisée avec ces différents types de bruit et la valeur du $RSBQ_{min}$ sélectionnée correspond à celle ayant une valeur maximale.

3.7 Conclusions

Dans ce chapitre, nous avons présenté une nouvelle méthodologie de détermination automatique de l'expression analytique du RSBQ. Le premier aspect traité correspond à la définition des modèles du bruit de quantification généré lors d'un changement de format. Un nouveau modèle de bruit a été proposé dans le cas de la quantification du résultat de la multiplication d'un signal par une constante. Ensuite, l'expression de la puissance du bruit de quantification en sortie d'un système en virgule fixe a été détaillée pour les systèmes linéaires et les systèmes non-linéaires et non-récursifs. Dans le cadre des systèmes linéaires, la méthodologie mise en œuvre pour déterminer les différents éléments de l'expression du RSBQ a été exposée. Les résultats des expérimentations ont montré que les estimations obtenues étaient précises et en particulier si le modèle de bruit proposé était utilisé. De plus, l'efficacité de la méthode en termes de temps d'exécution a été démontrée.

Cette méthodologie ne permet pas actuellement de traiter tous les types d'applications tels que les filtres adaptatifs. Dans ce cas, les techniques basées sur la simulation doivent être utilisées. Mais pour les applications supportées, cette méthodologie permet d'accélérer significativement les traitements de conversion en virgule fixe en fournissant l'expression analytique du RSBQ. Pour ces applications, notre méthodologie permet de remplacer avantageusement les méthodes basées sur la simulation. Pour les filtres adaptatifs, les coefficients du filtre linéaire évoluent au cours du temps et la valeur de ces coefficients est fonction de la sortie du filtre. Ainsi, des circuits sont présents au sein du graphe représentant l'application et les techniques présentées dans ce chapitre ne peuvent être utilisées. Cependant, des expressions de la puissance du bruit ont été définies dans [18]. Ainsi, il est possible d'obtenir une expression analytique du RSBQ en sortie de ce type d'algorithme. A notre niveau, la problématique consiste à automatiser la détermination de cette expression du RSBQ dans le cadre de ces applications.

Chapitre 4

Codage des données en virgule fixe

Ce chapitre est consacré à la présentation des différentes phases de la méthodologie de codage des données en virgule fixe. L'objectif de cette méthodologie est de convertir la description en virgule flottante en une spécification en virgule fixe et d'optimiser l'implantation de l'application en fonction de la contrainte de précision minimale. Dans un premier temps, l'infrastructure de compilation utilisée est présentée. Celle-ci est composée d'une partie frontale SUIF et d'une partie finale CALIFE. De plus, les caractéristiques de la représentation intermédiaire définie pour réaliser le processus de conversion en virgule fixe, sont exposées. Ensuite, les différentes phases de transformation de la description en virgule flottante en une spécification en virgule fixe optimisée sont détaillées.

La première étape permet de déterminer la dynamique des données de l'application. Les techniques utilisées et la méthode mise en œuvre pour les implanter sont présentées. Ensuite, ces informations de dynamique sont utilisées pour déterminer la position de la virgule de chaque donnée. Dans cette partie, la technique proposée pour prendre en compte et traiter les bits de garde associés aux additionneurs est plus particulièrement détaillée. La troisième phase a pour objectif de déterminer la largeur des données. Dans ce cadre, les types des données permettant de minimiser le temps d'exécution du code et de respecter la contrainte de précision sont sélectionnés. Ainsi, la modélisation de ce problème d'optimisation et la technique utilisée pour résoudre celui-ci sont exposées. Au cours de la dernière étape, le format des données en virgule fixe est optimisé afin de minimiser le temps d'exécution global du code. Deux types de méthode ont été définis en fonction des capacités de parallélisme de l'architecture cible.

Pour les trois dernières étapes de ce processus de conversion, les résultats de différentes expérimentations sont fournis afin d'illustrer l'intérêt et les capacités de la méthodologie proposée.

4.1 Description de l'infrastructure de compilation

La méthodologie de conversion de la description en virgule flottante en une spécification en virgule fixe s'insère au sein du processus de compilation. La partie frontale du compilateur utilise l'outil SUIF et la partie finale est réalisée à l'aide de l'environnement CALIFE. Cet environnement, développé au sein de notre projet de recherche R2D2, permet de s'interfacer relativement aisément avec les différentes phases de la génération de code. De plus, cette partie finale du compilateur étant reciblable, celui-ci n'est pas dédié à une cible particulière.

Dans cette partie, l'infrastructure de compilation utilisée est présentée. Les parties frontales et finales de l'infrastructure et la représentation intermédiaire, support du processus de conversion en virgule fixe, sont décrites.

4.1.1 Description de la partie frontale SUIF

L'environnement SUIF [152] développé à l'Université de Stanford est une plate-forme permettant le développement et l'évaluation de nouvelles techniques de compilation. Cet environnement inclut la partie frontale d'un compilateur permettant la transformation d'une application spécifiée en langage C vers une représentation intermédiaire (IR). De plus, cet environnement est composé de différents outils permettant des transformations de l'IR et une bibliothèque de classes C++ permettant l'accès à cette IR.

La structure de la représentation intermédiaire de SUIF permet de spécifier l'application de manière hiérarchique. Une procédure est définie par une liste de nœuds de contrôle représentant soit une structure répétitive ou conditionnelle soit un lien vers un arbre d'expression. Pour les structures répétitives les bornes minimales et maximales, l'évolution de la boucle et le corps de celle-ci sont représentés par des listes de nœuds de contrôle. De même, pour les structures conditionnelles, la condition et les différentes alternatives sont spécifiées par des listes de nœuds de contrôle. Chaque arbre d'expression correspondant à une expression du code source est composé de nœuds représentant les opérations et de nœuds représentant les opérandes de ces opérations. Une table des symboles est associée à chaque fichier de l'application et à chaque procédure pour gérer respectivement les variables globales et locales.

4.1.2 Description de la partie finale CALIFE

L'environnement CALIFE est une structure de génération de code flexible destinée aux processeurs programmables spécialisés et utilisant le principe de la génération de code parametrée par la description du processeur [20]. Cet environnement est composé d'un langage de modélisation de processeurs dénommé ARMOR, d'une bibliothèque d'algorithmes de transformation et d'une infrastructure de liaison entre la description du processeur et les modules de transformation de code.

Langage de description de processeurs

Le processeur est modélisé à l'aide du langage ARMOR défini dans [106, 22]. L'objectif de ce langage est de permettre la description rapide d'un processeur à l'aide d'un langage simple afin de faciliter l'exploration architecturale. Le comportement du processeur est défini à travers la description de son jeu d'instructions.

Une description ARMOR est une grammaire dont toute dérivation est un comportement possible du jeu d'instructions du processeur. Cette description spécifie des éléments assimilables aux instructions. Ces éléments sont structurés en groupe afin de définir le parallélisme au niveau instruction du processeur. Une règle particulière permet de définir les instructions de manière hiérarchique afin de factoriser l'information et ainsi, d'obtenir une description plus compacte. Pour faciliter la description des architectures hétérogènes, des restrictions peuvent être spécifiées et en particulier au niveau du parallélisme. Le comportement de chaque instruction est défini soit par une règle correspondant au parcours du flot de données d'une unité fonctionnelle soit par une règle détaillant le flot de contrôle. Les ressources du processeur correspondant aux registres, aux files de registres et aux unités fonctionnelles sont définies à l'aide de règles spécifiques. Les différents modes d'adressage peuvent être spécifiés et plus particulièrement les modes d'adressage auto-modifiés.

Infrastructure de génération de code

L'infrastructure CALIFE met à la disposition de l'utilisateur une bibliothèque de modules regroupant diverses passes de production et d'optimisation de code. Ces modules intègrent les transformations traditionnelles présentes au sein des compilateurs telles que la sélection des instructions, l'allocation de ressources et l'ordonnancement. L'intérêt de cette bibliothèque est

de proposer, pour chaque catégorie de transformation de code, différents algorithmes adaptés à divers styles d'architectures. Ainsi, l'utilisateur choisit et agence ces différentes passes afin d'adapter le flot de compilation à l'architecture cible.

La représentation interne du programme contient plusieurs structures de données correspondant aux différents niveaux d'abstraction du programme source (programme, procédure, bloc, instruction). Cette représentation interne est issue de la représentation intermédiaire de SUIF.

La représentation interne du processeur contient l'ensemble des informations nécessaires pour paramétrer les différents modules de transformations de code. Ces différentes informations correspondent, par exemple, aux règles définies pour la sélection de code ou les tables de réservation pour l'ordonnancement des instructions. Ces différentes informations sont générées à l'aide du module *ArmorC* à partir de la description ARMOR du processeur.

4.1.3 Description de la représentation intermédiaire GFDC

Pour réaliser les différentes phases de conversion d'une description en virgule flottante en une spécification en virgule fixe, une nouvelle représentation intermédiaire a été définie. En effet, la représentation intermédiaire de SUIF n'est pas adaptée aux transformations mises en œuvre au sein du processus de conversion. Cette nouvelle représentation intermédiaire présentée dans cette partie, correspond à un Graphe Flot de Données et de Contrôle (GFDC).

Les structures de données utilisées pour représenter les structures répétitives et conditionnelles au sein de l'IR SUIF et du GFDC sont relativement proches. Par rapport à la représentation intermédiaire de SUIF, une structure de contrôle représentant un bloc de base a été définie. Cette structure regroupe au sein d'un même graphe l'ensemble des traitements des données correspondant au traitement du signal. Dans l'IR SUIF, les arbres d'expression regroupent au sein d'une même structure les traitements sur les données et les opérations nécessaires pour accéder à ces données. Au sein du GFDC, les traitements sur les données correspondant à la partie traitement du signal sont séparés des autres traitements tels que les calculs des indices d'un tableau par exemple.

Dans cette nouvelle représentation intermédiaire de type GFDC, une procédure est représentée par un graphe flot de contrôle et de données (GFDC). Ce GFDC est composé de nœuds représentant la partie contrôle et de nœuds représentant la partie traitement des données.

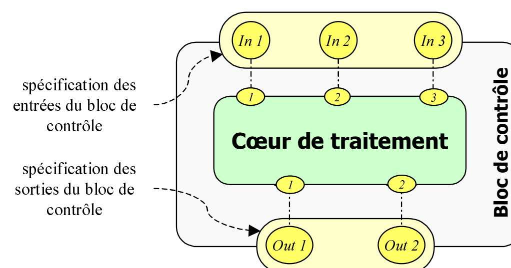


FIG. 4.1: Modélisation d'une structure de contrôle

Description du Graphe Flot de Contrôle (GFC)

Le graphe flot de contrôle (GFC) est un graphe acyclique orienté permettant de représenter la structure de contrôle d'une procédure. Chaque graphe flot de contrôle est composé d'une séquence de blocs de contrôle exécutés séquentiellement. Ces blocs représentent soit un bloc de base soit une structure répétitive ou conditionnelle. Chaque bloc de contrôle est composé d'un cœur décrivant le traitement réalisé au sein du bloc et d'une liste de ses entrées et de ses

sorties comme schématisé à la figure 4.1. Ces deux listes permettent de définir la vue externe du bloc de contrôle afin d'interfacer ce bloc avec l'extérieur. Les liens entre les nœuds représentant les entrées et les sorties des blocs de contrôle sont définis afin de spécifier le cheminement des données entre ces blocs de contrôle. En effet, certaines phases de conversion nécessitent de déplacer des opérations de recadrage au travers des frontières des blocs de contrôle. Ainsi, ces liens permettent de propager ces opérations de recadrage au sein du GFDC de l'application.

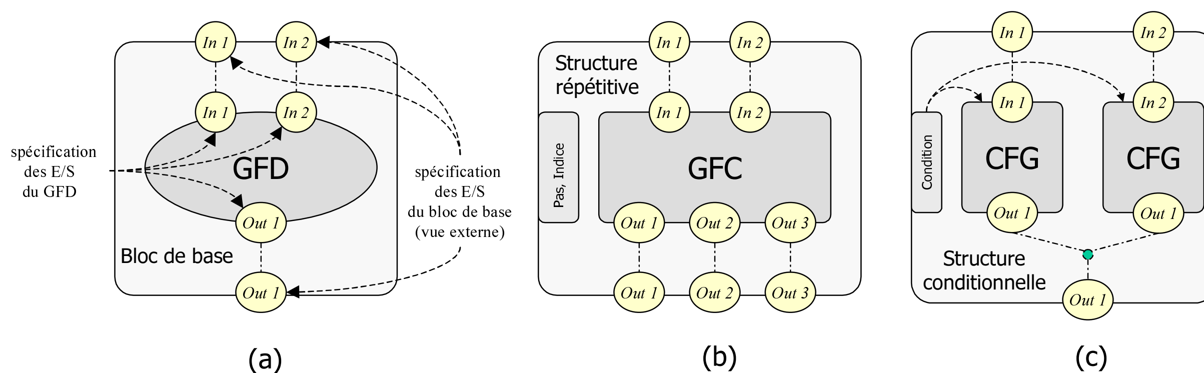


Fig. 4.2: Modélisation des structures de contrôle correspondant aux blocs de base et aux structures répétitives ou conditionnelles

Un bloc de base est une séquence d'opérations 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 [6]. Le cœur de traitement d'un bloc de base est modélisé par un Graphe Flot de Données (GFD) comme représenté à la figure 4.2-a. Les spécifications des entrées et des sorties d'un bloc de base correspondent à celles du GFD associé.

La structure répétitive implantée dans cette représentation permet de décrire les boucles dont le nombre d'itérations est borné. Le cœur de traitement de la structure répétitive est modélisé par un Graphe Flot de Contrôle (GFC) comme représenté à la figure 4.2-b. Cette structure regroupe l'ensemble des informations concernant la gestion de la boucle (incrément, valeur minimale et maximale de l'indice). Les entrées et sorties de ce type de bloc sont issues des entrées et sorties du GFC correspondant au cœur de traitement de la structure répétitive. Cependant, les différents paramètres tels que la dynamique et le format d'une entrée ou d'une sortie de la structure répétitive peuvent être différents des paramètres des entrées ou sorties équivalentes du GFC associé. En effet, les entrées et sorties du GFC spécifient celles d'une itération de la boucle alors que les entrées et sorties de la structure répétitive spécifient celles obtenues pour l'ensemble des itérations de la boucle. Pour illustrer ce concept, nous avons représenté à la figure 4.3, la modélisation de la Transformée de Fourier Rapide (FFT). La structure répétitive décrite permet d'appliquer le traitement spécifié au sein de la structure de contrôle dénommée GFC_i à chaque étage i de la FFT. La dynamique du vecteur représentant la sortie de l'étage i est égale au double de la dynamique du vecteur représentant l'entrée de cet étage i. A chaque étage, la dynamique du vecteur X est multipliée par deux. Notre modèle permet de spécifier au niveau de la structure répétitive la dynamique en entrée de la FFT ([-1,1]) et en sortie de celle-ci ([-N,N]). L'évolution de la dynamique du vecteur X au cours du traitement est spécifiée au niveau de la structure de contrôle représentant un étage i.

La structure conditionnelle permet de spécifier des alternatives de traitement en fonction d'un critère de sélection. Chaque alternative est spécifiée par un Graphe Flot de Contrôle (GFC) (Fig. 4.2-c). Les E/S de la structure conditionnelle correspondent à l'union des E/S de chaque alternative. Si plusieurs nœuds o_{ik} génèrent une même sortie o_i alors les paramètres de cette dernière sont issus de la fusion des paramètres des sorties o_{ik} selon des règles prédéfinies.

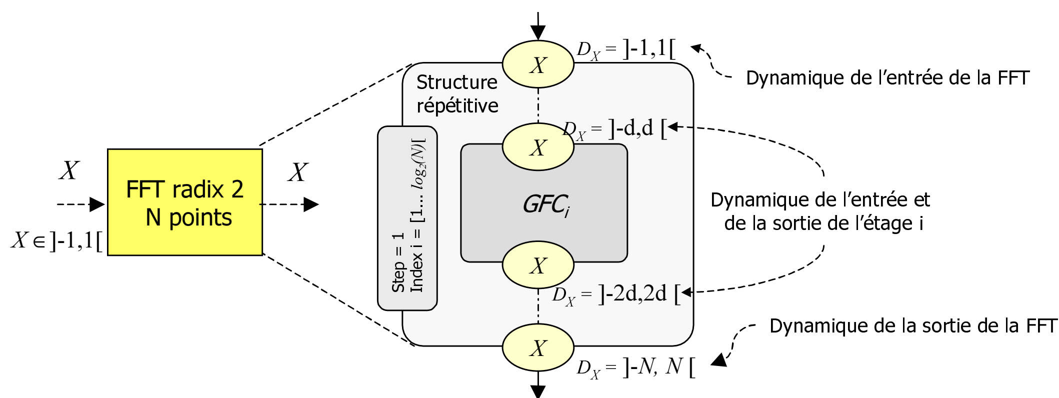


FIG. 4.3: Modélisation de l'algorithme de FFT

Description du Graphe Flot de Données (GFD)

Le graphe flot de donnée (GFD) permet de représenter l'ensemble des traitements sur les données correspondant à la partie du traitement du signal. Ainsi, les calculs d'indice ne sont pas spécifiés au sein de ce GFD, mais ils sont associés au nœud représentant la variable utilisant l'indice. Le GFD est composé d'un ensemble de nœuds représentant les données et les opérations sur ces données.

Génération de la représentation intermédiaire GFDC

La représentation intermédiaire utilisée par l'infrastructure de génération de code CALIFE est obtenue à l'aide de la partie frontale SUIF. Un module permettant de générer la représentation intermédiaire spécifiée ci-dessus à partir de l'IR obtenue avec SUIF a été développé. Les nœuds représentant une structure de contrôle au sein de l'IR SUIF sont représentés par leur équivalent dans le GFDC. Chaque arbre d'expression est représenté par un GFD. Les différents arbres d'expression formant un bloc de base sont identifiés et le GFD associé à un bloc de base est obtenu en fusionnant les différents GFD représentant les arbres d'expression associés à ce bloc.

Conversion GFDC vers SUIF

Un module permettant de mettre à jour la représentation intermédiaire de SUIF à partir de la spécification virgule fixe définie dans le GFDC a été développé. Cette transformation modifie le type des données et la valeur des constantes au sein de SUIF et insère les opérations de recadrage présentes au sein du GFDC. Cette mise à jour est nécessaire pour réaliser par la suite les phases de génération de code. De plus, un code C utilisant uniquement des types entiers peut être généré à l'aide de l'outil *s2c* présent au sein de l'environnement SUIF. Un exemple de code C obtenu après la phase de conversion en virgule fixe est présenté dans l'annexe B. De même, un outil similaire doit être développé pour obtenir une sortie en *SystemC* afin de pouvoir simuler la spécification virgule fixe obtenue au sein du GFDC.

Conclusions

Les caractéristiques de la représentation intermédiaire définie pour réaliser la conversion en virgule fixe de l'application ont été présentées. L'implantation de chaque structure de contrôle et de données fait appel à une classe C++ spécifique. Dans un premier temps, uniquement les structures de contrôle les plus utilisées dans le domaine du TNS ont été implantées. Ces structures correspondent aux blocs de base et aux structures répétitives.

4.2 Détermination de la dynamique des données

La première phase du processus de conversion en virgule fixe correspond à la détermination de la dynamique (domaine de définition) de chaque donnée du GFDC représentant l'application. La dynamique d'une donnée x permet de définir le paramètre m_x correspondant à la position de la virgule par rapport au bit le plus significatif. Cette position doit permettre de représenter la donnée avec le maximum de précision et garantir que toutes les valeurs prises par cette donnée pourront être représentées. La position optimale de la virgule d'une donnée x est alors obtenue à l'aide de l'inégalité présentée ci-dessous et nécessite de calculer à partir de la dynamique, la valeur absolue maximale de la donnée.

$$2^{m_x-1} \leq \tilde{x}_{max} < 2^{m_x} \quad \text{avec} \quad \tilde{x}_{max} = \max_n(|x(n)|) \tag{4.1}$$

Tout d'abord, les techniques utilisées pour déterminer la dynamique au sein des systèmes linéaires et des systèmes non-linéaires et non-récursifs sont détaillées. Ensuite, l'implantation de ces techniques au sein de notre méthodologie est présentée.

4.2.1 Définition d'une méthode d'estimation de la dynamique

Les différentes méthodologies disponibles pour déterminer la dynamique des données ont été présentées dans la partie 1.2.2 de la page 19. Les méthodes statistiques permettent d'obtenir une estimation de la dynamique précise mais elles ne garantissent pas l'absence de débordement. Les méthodes analytiques basées sur une estimation dans le pire cas garantissent l'absence de débordement mais l'estimation est plus pessimiste. Dans un premier temps, les techniques utilisées au sein de notre méthodologie correspondent aux méthodes analytiques afin de garantir l'absence de débordement. Les deux techniques utilisées pour déterminer la dynamique sont basées sur la théorie de l'arithmétique d'intervalles pour les systèmes non-récursifs et l'utilisation des normes L1 et de Chebychev dans le cas des systèmes linéaires.

L'arithmétique d'intervalles

L'arithmétique d'intervalles [56] correspond à une arithmétique définie sur un ensemble d'intervalles au lieu d'un ensemble de nombre réels. Un intervalle est assimilé au domaine de définition d'une variable x et est caractérisé par sa valeur limite inférieure $\underline{x}$ correspondant à la valeur minimale de x et sa valeur limite supérieure $\overline{x}$ correspondant à la valeur maximale de x.

L'arithmétique d'intervalles définit le domaine de définition des données issues des opérations arithmétiques classiques. Considérons la donnée z issue d'une opération arithmétique dont les entrées sont les deux variables x et y. Le domaine de définition de la donnée z pour les différents opérateurs arithmétiques est le suivant :

$$\begin{array}{ll} z = x + y & [\underline{z}, \overline{z}] = [\underline{x} + \underline{y},\ \overline{x} + \overline{y}] \\ z = x - y & [\underline{z}, \overline{z}] = [\underline{x} - \overline{y},\ \overline{x} - \underline{y}] \\ z = x * y & [\underline{z}, \overline{z}] = [\min(\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}),\ \max(\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y})] \end{array} \tag{4.2}$$

Considérons un système composé de N_e entrées $x_i(n)$. Soit $y_k(n)$ une donnée quelconque de ce système et définie en fonction des entrées de la manière suivante :

$$y_k(n) = f_k\left(x_0(n), x_1(n), \ldots x_i(n), \ldots x_{N_e-1}(n)\right) \tag{4.3}$$

Le domaine de définition de la donnée y_k peur être obtenu à partir de celui des entrées en appliquant à la fonction f_k, les différentes règles présentées à l'équation 4.2 :

$$[\underline{y_k}, \overline{y_k}] = f_k\left([\underline{x_0}, \overline{x_0}], \ldots [\underline{x_i}, \overline{x_i}], \ldots [\underline{x_{N_e-1}}, \overline{x_{N_e-1}}]\right) \tag{4.4}$$

Cette technique permet de définir le domaine de définition d'une donnée qui est fonction uniquement des entrées du système. En effet, dans le cas des systèmes récursifs, certaines données $y_k(n)$ sont fonction des valeurs précédentes $y_k(n-i)$ de ces données. Le domaine de définition de la donnée y_k étant fonction de son propre domaine de définition, l'arithmétique d'intervalles ne permet pas de déterminer directement le domaine de définition de ce type de donnée. Ainsi, cette technique ne peut être mise en œuvre de manière simple que pour les systèmes non-récursifs.

Pour déterminer la dynamique des données d'un système non-récursif, le graphe dirigé acyclique représentant le système est parcouru des sources vers les racines et les règles définies à la relation 4.2 sont appliquées à chaque opérateur.

Normes pour les systèmes linéaires

Dans cette partie, nous présentons les normes définies pour déterminer la dynamique des données dans le cadre des systèmes linéaires. Considérons un système linéaire invariant dans le temps composé de N_e entrées x_i. Soit y_k, une donnée quelconque de ce système définie de la manière suivante :

$$y_k(n) = \sum_{i=0}^{N_e-1} h_{ik}(n) * x_i(n) \tag{4.5}$$

Ces normes permettent de déterminer la valeur absolue maximale de la donnée y_k à partir de la valeur absolue maximale des différentes entrées x_i.

Norme L1 : D'après la relation 4.5, la valeur absolue de la donnée y_k est égale à :

$$|y_k(n)| = \left| \sum_{i=0}^{N_e-1} \sum_{m=-\infty}^{+\infty} h_{ik}(m) x_i(n-m) \right| \tag{4.6}$$

Sachant que la valeur absolue d'une somme de termes est inférieure à la somme des valeurs absolues de ces termes et que les entrées sont bornées par leur valeur absolue maximale l'expression 4.6 peut être bornée. Ainsi, la relation entre la valeur absolue maximale d'une donnée y_k et celle des entrées du système est la suivante :

$$\max_n (|y_k(n)|) \leq \sum_{i=0}^{N_e-1} \max_n (|x_i(n)|) \sum_{m=-\infty}^{+\infty} |h_{ik}(m)| = y_{k\ max} \tag{4.7}$$

Cette expression correspond à l'extension de la norme L1 pour un système linéaire à plusieurs entrées. Aucune hypothèse concernant la nature des signaux en entrée ayant été faite pour obtenir l'expression de $y_{k\ max}$, la relation 4.7 permet de borner la valeur absolue maximale de la donnée y_k pour tous types de signaux au niveau des entrées. D'après la relation 4.7, la valeur absolue maximale d'une donnée y_k est fonction de celle des entrées du système et de la réponse impulsionnelle des fonctions de transfert entre la donnée y_k et les différentes entrées x_i. L'utilisation de la notion de fonction de transfert permet de traiter aussi les structures récursives.

Cette méthode nécessite de déterminer pour chaque donnée y_k du système, la réponse impulsionnelle de la fonction de transfert entre cette donnée et chaque entrée du système. Afin d'accélérer la détermination de la dynamique, nous souhaitons intégrer dans notre méthode des résultats issus de la méthode présentée dans le paragraphe précédent. Pour cela, nous avons comparé pour les opérateurs classiques les résultats obtenus avec la norme L1 et ceux issus de la théorie de l'arithmétique d'intervalles.

Nous considérons une donnée y_s issue de la multiplication d'une donnée y_e par un coefficient b ne variant pas au cours du temps. L'expression de la valeur absolue maximale de cette donnée y_s en fonction de celle de y_e et du coefficient b est la suivante :

$$\max_n (|y_s|) = \max_n (|b.y_e(n)|) = |b|. \max_n (|y_e(n)|) \tag{4.8}$$

L'expression de la valeur absolue maximale d'une donnée issue de la multiplication d'une donnée par une constante correspond à celle obtenue dans le cadre de la théorie de l'arithmétique d'intervalles et définie avec la relation 4.2.

Considérons une donnée y_s correspondant à la somme de deux données y_{e_1} et y_{e_2}. Soient H_{is}, H_{ie_1} et H_{ie_2} les fonctions de transfert entre les données y_s ,y_{e_1} et y_{e_2} et les entrées du système. L'expression de la valeur absolue de y_s est la suivante :

$$|y_s(n)| = \left| \sum_{i=0}^{N_e-1} h_{is}(n) * x_i(n) \right| = \left| \sum_{i=0}^{N_e-1} (h_{ie_1}(n) + h_{ie_2}(n)) * x_i(n) \right| \tag{4.9}$$

En utilisant la propriété concernant la valeur absolue d'une somme, énoncée précédemment, la relation entre la dynamique de la sortie de l'additionneur et celle de ses entrées est la suivante :

$$\max_n (|y_s(n)|) \leq \max_n (|y_{e_1}(n)|) + \max_n (|y_{e_2}(n)|) \tag{4.10}$$

Le second terme de l'inégalité correspond à l'estimation de la valeur absolue maximale obtenue si l'arithmétique d'intervalles est utilisée. Ainsi, dans le cadre d'une addition de deux variables l'estimation obtenue avec la norme L1 est plus précise ou équivalente à celle obtenue avec la théorie de l'arithmétique d'intervalles.

Les deux estimations sont équivalentes dans le cadre des systèmes linéaires non récursifs. Pour les systèmes non-récursifs, la réponse impulsionnelle de la fonction de transfert H_{ik} est directement obtenue à partir des coefficients $b_{ik,l}$ composant cette fonction de transfert. L'expression de chaque fonction de transfert H_{ik} est la suivante

$$H_{ik}(z) = \sum_{l=0}^{N_{ik}} b_{ik,l}\ z^{-l} \tag{4.11}$$

Les deux méthodes d'estimation conduisent à la même expression de la valeur absolue maximale de la donnée y_k :

$$y_{k\ max} = \sum_{i=0}^{N_e-1} \max_n (|x_i(n)|) \sum_{l=0}^{N_{ik}} |b_{ik,l}| \tag{4.12}$$

Ces résultats montrent que les estimations de la dynamique des données au sein d'un système linéaire invariant dans le temps non-récursif sont identiques pour les deux estimateurs. De même, dans le cas d'une donnée issue de la multiplication d'un coefficient et d'une donnée, les estimations conduisent aux mêmes résultats.

Norme de Chebychev : L'estimation de la dynamique dans un système linéaire basé sur la norme L1 permet de garantir l'absence de débordement pour tous les types de signaux au niveau des entrées. Cette estimation étant relativement conservatrice, d'autres normes ont été proposées afin de tenir compte de la nature des signaux en entrée du système. La norme de Chebychev permet de déterminer la valeur absolue maximale d'une donnée pour des signaux

d'entrée à bande étroite. Dans ce cas, chaque signal d'entrée est modélisé par une sinusoïde définie de la manière suivante :

$$x_i(n) = A_j \cos(\Omega_j.n) \qquad \text{avec } 0 \leq \Omega_j < \pi \tag{4.13}$$

D'après les relations 4.5 et 4.13, l'expression de la valeur absolue de y_k est égale à :

$$|y_k(n)| = \left| \sum_{i=0}^{N_e-1} A_j.|H_{ik}(e^{j\Omega})| \cos\left(\Omega_j.n + \arg\left(H_{ik}(e^{j\Omega})\right)\right) \right| \quad \text{avec } 0 \leq \Omega_j < \pi \tag{4.14}$$

L'expression 4.14 peur être bornée en utilisant la valeur maximale du module de la réponse fréquentielle de H_{ik}. Ainsi, pour la norme de Chebychev, la relation entre la valeur absolue maximale d'une donnée y_k et celle des entrées du système est la suivante :

$$\max_n (|y_k(n)|) \leq \sum_{i=0}^{N_e-1} \max_n (|x_i(n)|) \max_\Omega \left(|H_{ik}(e^{j\Omega})|\right) \tag{4.15}$$

Cette méthode nécessite de calculer les réponses fréquentielles des fonctions de transfert entre la donnée considérée et les entrées du système.

Comme pour la norme L1, la dynamique d'une donnée y_s issue de la multiplication d'une constante b et d'une donnée y_e peut être obtenue à partir de la valeur absolue maximale de la donnée y_e à l'aide de la relation 4.8.

4.2.2 Implantation de la méthode d'estimation de la dynamique

Pour la méthode basée sur les normes L1 et de Chebychev, la détermination de la dynamique d'une donnée nécessite de déterminer les fonctions de transfert entre cette donnée et les différentes entrées du système. La méthode utilisée pour déterminer les fonctions de transfert correspond à celle présentée dans le chapitre précédent. Celle-ci nécessite de représenter l'application par un Graphe Flot de Signal (GFS). Pour la technique basée sur l'arithmétique d'intervalles, la dynamique d'une donnée est obtenue en propageant au sein du graphe représentant l'application la dynamique des entrées. Ce graphe unique doit représenter l'ensemble de l'application et les sources de ce graphe doivent correspondre uniquement aux entrées ou aux coefficients. En conséquence, les éventuelles opérations de "vieillissement" de données doivent être présentes au sein du graphe. Ainsi, pour les deux types d'estimation, l'application doit être représentée sous la forme d'un graphe flot de signal.

Le processus de détermination de la dynamique au sein du GFDC est schématisé à la figure 4.4. La première étape correspond à la transformation du Graphe Flot de Contrôle et de Données (GFDC) représentant l'application en un Graphe Flot de Signal (GFS). Ensuite, les techniques présentées ci-dessus sont appliquées en fonction du type de système et des choix de l'utilisateur. La dernière étape va permettre de reporter les informations de dynamique au sein du GFDC de départ. En effet, les différentes structures de contrôle présentes au sein du code original ne doivent pas être modifiées afin de ne pas altérer les performances en termes de temps d'exécution et de taille du code.

Transformation du GFDC en un GFS

L'objectif de cette partie est de transformer un Graphe Flot de Données et de Contrôle (GFDC) en un Graphe Flot de Signal (GFS) sur lequel les techniques présentées dans la section précédente, seront appliquées. L'entrée de ce module est le GFDC G_{app} représentant l'application et la sortie est le GFS G'_{app} représentant cette même application au niveau traitement du

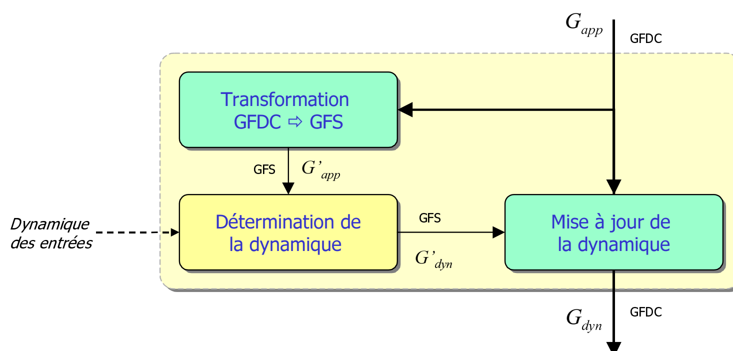


Fig. 4.4: Synoptique du processus de détermination de la dynamique des données

signal.

Transformation du GFDC en GFD : La première étape correspond à la transformation du GFDC en un Graphe Flot de Données (GFD). Ce GFD permet de représenter l'ensemble des traitements réalisés sur les données au sein de l'application sans utiliser de structures de contrôle. Ainsi, cette transformation correspond à l'élimination de chaque structure de contrôle présente au sein du GFDC. Pour chaque structure, une règle spécifiant la conversion de cette structure en un GFD est définie.

Pour les blocs de base, le GFD issu de cette transformation correspond à celui représentant le cœur de traitement du bloc. Pour les structures répétitives, cette transformation réalise un déroulage complet de la boucle. Ainsi, elle nécessite de connaître lors de la compilation du code les bornes inférieures et supérieures et le pas d'évolution de la structure répétitive. Si les valeurs de ces bornes ne sont pas définies au moment de la compilation, il est nécessaire que la valeur minimale de la borne inférieure et la valeur maximale de la borne supérieure soient connues. La transformation est appliquée au GFC représentant le cœur de la structure répétitive pour chaque itération de celle-ci. Ensuite, les GFD issus des différentes itérations sont fusionnés afin d'obtenir un unique GFD correspondant à la transformation de cette structure répétitive. Pour les GFC, la transformation est appliquée à chaque bloc de contrôle du GFC. Ensuite, les GFD issus de la transformation des différents blocs sont fusionnés afin d'obtenir un unique GFD correspondant à la transformation de ce GFC.

Transformation du GFD en GFS : La seconde étape correspond à la transformation du Graphe flot de données en un Graphe Flot de Signal (GFS). Ce GFS permet de spécifier les relations temporelles entre les données à travers la présence des opérations de retard. Ainsi, cette transformation correspond à la détection puis à l'insertion de ce type d'opération. Ces dernières sont insérées dans le graphe lorsqu'une opération de "vieillissement" est détectée. Cette opération de "vieillissement" correspond à l'affectation d'une donnée x_0 à une donnée x_1. L'opération d'affectation de x_0 à x_1 est une opération de vieillissement si la donnée x_1 a été utilisée (i.e lue) avant d'avoir été affectée (i.e. écrite).

Détermination de la dynamique au sein du GFS

Dans cette partie, le domaine de définition des données présentes au sein du GFS obtenu à l'étape précédente, est déterminé. Les entrées de ce module sont le GFS et le domaine de définition de l'ensemble des entrées du système défini par l'utilisateur. La sortie de ce module

correspond au GFS pour lequel chaque donnée est annotée avec la valeur de sa dynamique. Les deux techniques présentées précédemment ont été couplées afin de déterminer de manière analytique la dynamique des données dans le cadre des systèmes linéaires invariant dans le temps et des systèmes non-linéaires et non-récursifs.

Les résultats de la théorie de l'arithmétique d'intervalles ont été utilisés afin de déterminer la dynamique des données dans le cadre des systèmes non-récursifs. Dans ce cas, l'application est représentée par un GFS correspondant à un graphe dirigé acyclique. Ce graphe est parcouru des sources vers les racines et une règle de propagation de la dynamique est appliquée à chaque opération. Ces règles définissent le domaine de définition de la sortie d'un opérateur en fonction du domaine de définition de ses entrées.

Les normes L1 et de Chebychev ont été utilisées afin de déterminer la dynamique des données dans le cadre des systèmes linéaires. Cependant, l'obtention de la dynamique d'une donnée à l'aide d'une norme est plus complexe qu'à l'aide de la technique basée sur la propagation de la dynamique. Ainsi, l'utilisation de cette dernière technique est privilégiée si elle conduit à des résultats identiques à ceux obtenus avec une norme afin de diminuer les temps de calcul de l'outil. D'après la relation 4.12, dans le cadre des systèmes linéaires invariant dans le temps et non-récursifs, les résultats obtenus avec la norme L1 et la technique de propagation de la dynamique sont identiques. De plus, d'après la relation 4.8, pour une multiplication d'un signal par une constante, la dynamique de la sortie peut être obtenue à l'aide de la technique de propagation de la dynamique.

La première étape de cette méthode correspond à la sélection des données dont la dynamique est obtenue à partir du calcul d'une norme. Soit E_{norm}, l'ensemble regroupant ces données. Pour ces données, les fonctions de transfert entre chaque donnée et les entrées du système sont déterminées à partir de la technique présentée dans le chapitre précédent. Ensuite, la dynamique de ces données est déterminée à l'aide de l'expression des fonctions de transfert obtenues précédemment et de la dynamique des entrées du système. Selon la norme choisie, la relation 4.7 ou 4.15 est utilisée. Lorsque la dynamique de l'ensemble des données appartenant à E_{norm} a été définie, la dynamique des autres données est déterminée selon la technique de propagation présentée ci-dessus.

Annotation du GFDC avec les informations de dynamique

L'objectif de cette dernière partie est d'annoter l'ensemble des données du GFDC avec la valeur de leur dynamique. Cette dynamique est obtenue à partir de la dynamique des données présentes au sein du GFS. Soit E_{GFDC} et E_{GFS}, les ensembles regroupant les données appartenant respectivement aux graphes GFDC et GFS. Chaque donnée contenue dans le GFDC est représentée par une ou plusieurs données au sein du GFS. Soit $f : E_{GFDC} \to E_{GFS}$ la relation définissant pour chaque élément de E_{GFDC} ses images au sein du domaine E_{GFS}. Pour chaque donnée du GFDC, une ou plusieurs valeurs de la dynamique sont disponibles. Ainsi, il est nécessaire de mettre en œuvre une méthode permettant de définir la valeur de la dynamique d'un élément de E_{GFDC} en fonction de la dynamique des éléments images présents dans E_{GFS}.

Pour chaque donnée d_i possédant un seul élément image d_i' dans E_{GFS}, la dynamique de d_i correspond à celle de d_i'. Pour les données d_i possédant plusieurs éléments images dans E_{GFS}, il est nécessaire d'analyser l'évolution de leur dynamique. Nous pouvons distinguer deux types de données correspondant aux données à affectation unique et multiple. Chaque donnée présente au sein du GFS fait référence à un emplacement mémoire unique. Ainsi, une donnée d_i appartenant à E_{GFDC} est à affectation unique, si tous les emplacements mémoires associés aux éléments images de d_i contenus dans E_{GFS} ne sont affectés qu'une seule fois.

Une donnée d_k à affectation unique et possédant plusieurs éléments images dans E_{GFS}, correspond à une donnée accédant à différents éléments d'un tableau au travers d'un pointeur ou d'un indice évoluant au sein d'une structure répétitive. Dans un premier temps, pour simplifier le codage des données, nous imposons un format unique aux différents éléments d'un même tableau, ainsi, la dynamique $\mathcal{D}(d_k)$ de la donnée d_k est obtenue à partir de la valeur maximale de la dynamique des éléments images de d_k :

$$\mathcal{D}(d_k) = \max_{d \in f(d_k)} (\mathcal{D}(d)) \tag{4.16}$$

Cette restriction est utilisée dans les autres méthodologies de codage des données en virgule fixe [70, 148]. Elle peut conduire dans certains cas à un codage sous optimal. Dans ce cas, une technique similaire à celle mise en œuvre pour les données à affectation multiple doit être utilisée.

Pour une donnée d_l à affectation multiple, il est nécessaire d'analyser l'évolution de la dynamique de cette donnée afin d'avoir un codage optimal de celle-ci. La dynamique de la donnée étant utilisée par la suite pour déterminer la position de la virgule par rapport au bit le plus significatif, il est préférable de travailler directement sur ce paramètre m. Le problème consiste à déterminer la fonction f_d permettant de définir le paramètre m associé à la donnée d_l en fonction des paramètres m des données (d_{s_i}) représentant les sources du GFD contenant cette donnée d_l. Étant données les opérations arithmétiques mises en œuvre, la fonction f_d est linéaire par rapport au paramètre m_{s_i} des données représentant les sources du GFD :

$$f_d(m_{s_0}, \ldots, m_{s_N}) = \sum_{i=0}^{N+1} a_i . m_{s_i} \tag{4.17}$$

Le paramètre m représentant la position de la virgule exprimée en bit, doit être un entier. Ainsi, les coefficients a_i doivent être des entiers. Ainsi, le problème consiste à trouver la fonction linéaire f_d à coefficients entiers minimisant l'erreur entre la courbe réelle et la courbe associée à la fonction f_d. Actuellement, cette technique n'a pas encore été mise en œuvre. Ainsi, pour les données d_l à affectation multiple la dynamique est fixée à une valeur constante égale à la valeur maximale de la dynamique des éléments images de d_l.

4.3 Détermination de la position de la virgule

L'objectif de ce module est de déterminer la position de la virgule pour chaque donnée du GFDC afin d'aboutir à une spécification correcte d'un point de vue de l'arithmétique virgule fixe. Cette transformation doit garantir l'absence de débordement et respecter les règles de l'arithmétique virgule fixe. L'entrée de ce module est le GFDC G_{Dyn} au sein duquel chaque donnée est annotée avec sa dynamique. Cette transformation conduit au GFDC G_{pv} pour lequel la position de la virgule de chaque donnée est spécifiée. De plus, les opérations de recadrage nécessaires à l'obtention d'une spécification correcte d'un point de vue de l'arithmétique virgule fixe sont insérées. Elles permettent entre autre, d'adapter le format de codage d'une donnée à sa dynamique ou d'adapter le format des entrées d'un additionneur en vue d'aligner la position de leur virgule.

4.3.1 Détermination de la position de la virgule au sein d'un GFDC

La détermination de la position de la virgule de chaque donnée au sein du GFDC G_{Dyn} est réalisée de manière hiérarchique. Tout d'abord, chaque GFD de G_{Dyn} est traité indépendamment et ensuite un traitement global est réalisé afin d'assurer la cohérence de la position des virgules entre les données.

Lorsque chaque GFD a été traité, les informations concernant la position de la virgule des entrées et sorties de chaque structure de contrôle sont mises à jour. Ces paramètres sont obtenus à partir des paramètres des entrées et sorties des structures de niveau inférieur. Ensuite, la cohérence de la position de la virgule d'une donnée partagée entre plusieurs structures est vérifiée. Si cette cohérence n'est pas respectée alors une opération de recadrage est insérée afin d'obtenir un traitement correct. Les opérations de recadrage sont insérées au sein des blocs de base. Ainsi, si les structures de contrôle associées à une entrée et une sortie ne sont pas des blocs de base alors un bloc de base est inséré entre les deux structures de contrôle.

4.3.2 Détermination de la position de la virgule au sein d'un GFD

La position de la virgule des données et des opérateurs d'un GFD est déterminée à l'aide d'un parcours ascendant du GFD. Pour chaque nœud du graphe, une règle de propagation de la position de la virgule est appliquée afin de déterminer cette position en fonction des valeurs obtenues au niveau des nœuds prédécesseurs. Ce type de technique basé sur un parcours de graphe nécessite que celui-ci ne contienne aucun cycle. Ainsi, pour chaque GFD, les cycles sont démantelés afin d'obtenir un graphe dirigé acyclique. La technique pour définir les nœuds où les circuits sont démantelés est similaire à celle présentée dans la section 3.4.3.

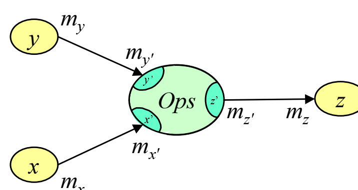


FIG. 4.5: Modélisation de la position de la virgule pour un opérateur

Une position de la virgule est associée à chaque donnée du GFD et à chaque entrée et sortie d'un opérateur. Le modèle d'un opérateur est présenté à la figure 4.5. Pour une donnée x, la position de sa virgule m_x est directement obtenue à partir de sa dynamique selon la relation suivante :

$$m_x = \left\lceil \log_2 \left(\max_n \left(|x(n)| \right) \right) \right\rceil \quad (4.18)$$

Une règle de propagation de la position de la virgule est définie pour chaque type d'opérateur. Cette règle définit la position de la virgule de ses entrées et de sa sortie $(m_{x'}, m_{y'}, m_{z'})$ en fonction de celle des données présentes en entrée et en sortie (m_x, m_y, m_z).

La position de la virgule des entrées du multiplieur $(m_{x'}, m_{y'})$ correspond à celle des données présentes en entrée de l'opérateur (m_x, m_y). La position de la virgule de la sortie est directement obtenue à partir de celle des entrées. En intégrant le bit de signe supplémentaire à la partie entière du résultat, les règles définies pour le multiplieur sont les suivantes :

$$\begin{cases} m_{x'} = m_x \\ m_{y'} = m_y \\ m_{z'} = m_{x'} + m_{y'} + 1 \end{cases} \quad (4.19)$$

Pour les opérations d'addition ou de soustraction, il est nécessaire de définir une position de la virgule commune aux entrées afin d'aligner la position de leur virgule. La position de la virgule de la sortie correspond à la position commune au niveau des entrées. Cette position commune doit garantir l'absence de débordement, ainsi elle est définie à partir de la contrainte la plus forte au niveau des entrées. Si l'opérateur ne possède pas de bit de garde permettant d'accueillir le bit supplémentaire issu d'un éventuel débordement il est nécessaire d'intégrer cette contrainte dans la détermination du format commun. Ainsi, pour un additionneur ne possédant pas de bit de garde, la position de la virgule commune est définie de la manière suivante :

$$\begin{cases} m_c = \max(m_x, m_y, m_z) \\ m_{x'} = m_c \\ m_{y'} = m_c \\ m_{z'} = m_c \end{cases} \quad (4.20)$$

Pour les additionneurs possédant des bits de garde, les largeurs des opérandes d'entrée et de sortie sont différentes et les bits les plus significatifs de ces données ne sont plus alignés. Ainsi, les positions des virgules entre les données ne sont plus cohérentes et un référentiel commun doit être utilisé pour analyser les contraintes au niveau des entrées et de la sortie.

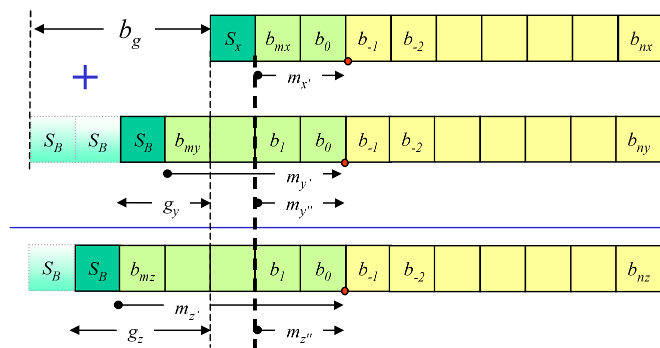


FIG. 4.6: Spécification de la position de la virgule en présence de bits de garde

Un paramètre g_x est associé à chaque donnée x pour représenter le nombre de bits de garde utilisés au sein de la donnée. Soient $m_{x''}, m_{y''}, m_{z''}$, les positions des virgules des données référencées par rapport au bit le plus significatif de l'entrée dont la largeur est la plus faible. Ces différents paramètres sont représentés à la figure 4.6. Le passage du référentiel associé à chaque donnée au référentiel commun nécessite de retrancher aux positions de la virgule le nombre

de bits de garde utilisés. Ainsi, les expressions de la position des virgules dans le référentiel commun sont les suivantes :

$$\begin{cases} m_{x''} = m_{x'} - g_x \\ m_{y''} = m_{y'} - g_y \\ m_{z''} = m_{z'} - g_z \end{cases} \tag{4.21}$$

Une position de la virgule commune aux entrées et à la sortie de l'additionneur dans le référentiel commun est définie. Cette position est déterminée à partir de la contrainte maximale sur les entrées et la sortie. Cependant, le calcul de la contrainte sur la sortie nécessite de connaître le nombre de bits de garde utilisés au niveau de la sortie. Cette valeur n'étant pas connue avant de déterminer le format commun, celle-ci est fixée à sa valeur maximale b_g, correspondant au nombre de bits de garde disponibles au niveau de l'additionneur. La position de la virgule commune est définie à partir de la relation suivante :

$$m_c = \max\left(m_x - g_x, m_y - g_y, m_z - b_g\right) \tag{4.22}$$

En conséquence, les positions des virgules des entrées sont égales à :

$$\begin{cases} m_{x'} = m_c + g_x \\ m_{y'} = m_c + g_y \end{cases} \tag{4.23}$$

Le nombre de bits de garde réellement utilisés par la sortie de l'additionneur est égal à :

$$\begin{cases} g_z = m_z - m_c & \text{si} \quad m_z > m_c \\ g_z = 0 & \text{si} \quad m_z \leq m_c \end{cases} \tag{4.24}$$

Ainsi, la position de la virgule de la sortie de l'additionneur est égale à :

$$m_{z'} = m_c + g_z \tag{4.25}$$

Si g_z est égal à 0, alors aucun bit de garde n'est utilisé et dans ce cas la position de la virgule de la sortie correspond à celle du format commun. Sinon, la position de la virgule de la sortie est obtenue à partir de la dynamique du résultat de l'opération ($m_{z'} = m_z$). Si la contrainte la plus forte au niveau de la relation 4.22 correspond à celle sur l'entrée alors, l'ensemble des bits de garde est utilisé pour coder la sortie et la valeur de g_z est égale au nombre de bits de garde disponibles au niveau de l'additionneur.

La relation 4.22 peut être utilisée pour déterminer la position de la virgule commune dans le cas des additionneurs avec et sans bits de garde. Si l'additionneur ne possède pas de bit de garde, les paramètres g et b_g sont nuls et nous retrouvons les résultats obtenus à l'équation 4.20.

Lorsque la position de la virgule de l'ensemble des données de chaque GFD a été définie, les opérations de recadrage nécessaires à l'obtention d'une spécification en virgule fixe correcte sont insérées. Pour chaque opérateur, la position de la virgule $m_{x'}, m_{y'}$ associée à ses entrées est comparée avec celle des données situées en entrée de l'opérateur m_x, m_y. Si les positions sont différentes, alors une opération de recadrage est insérée afin d'assurer le déplacement de la virgule. De même, au niveau de la sortie si les positions $m_{z'}$ et m_z sont différentes une opération de recadrage est insérée.

4.3.3 Expérimentations

Pour illustrer cette phase de détermination de la position de la virgule, la méthode proposée a été testée sur différentes applications. Pour chaque application, le cas d'une architecture ne possédant pas de bit de garde et le cas d'une architecture possédant un nombre de bits de

Applications	**Nombre d'exécutions des opérations de recadrage**	
	$b_g = 0$	$b_g = 8$
FIR (N cellules)	N	1
IIR 2	2	2
Corrélateur complexe *(N points)*	2.N	2
Rake Receiver - MS *(M fingers, $SF = N$)*	2.M.N + 2M	2.M+2
Rake Receiver - BS *(M fingers, $SF = N$)*	2.M.N + 2M	2.M+2

Conditions d'expérimentation : $M \in [1, 6]$ et $N \in [1, 256]$

TAB. 4.1: Nombre d'exécutions des opérations de décalage en fonction du nombre de bits de garde b_g

garde b_g égal à 8, sont considérés. Pour chaque cas, le nombre d'exécutions des opérations de recadrage est reporté au sein du tableau 4.1. Les applications dénommées *Rake Receiver* sont décrites dans le chapitre 5.

Dans le cas du filtre FIR composé de N cellules, la présence de 8 bits de garde permet de réaliser un recadrage uniquement après la série d'accumulations. L'absence de bit de garde conduit à un codage des données nécessitant de réaliser pour chaque cellule 1 recadrage entre l'opération de multiplication et d'accumulation. Ainsi, le nombre d'exécutions des opérations de recadrage pour les N cellules avec ce type de codage est égal à N. Ces résultats montrent la nécessité de mettre en œuvre une phase de déplacement des opérations de recadrage afin de minimiser le surcoût de ces opérations en termes de temps d'exécution. La présence de bits de garde permet de réduire significativement le nombre d'opérations de décalage à exécuter pour certaines applications. Ces applications intègrent des séries d'accumulations conduisant à une dynamique du résultat de l'accumulation supérieure à la dynamique des entrées. Dans le cas du filtre récursif du second ordre, les bits de garde n'apportent pas de gain au niveau du nombre d'opérations de décalage à exécuter. En effet, pour la partie récursive du filtre, la dynamique des entrées des additionneurs est supérieure à celle de la sortie, ainsi, les bits de garde ne peuvent être utilisés pour cette partie. La dynamique des données issues de la partie non récursive et de la partie récursive étant différente il est nécessaire de recadrer ces données avant de les additionner.

4.4 Détermination du type des données

L'objectif de cette partie est de présenter la méthode proposée pour déterminer et optimiser le type des données présentes au sein de l'application. Cette partie recouvre deux aspects. Dans un premier temps, la largeur de l'ensemble des données de l'application est définie avant de débuter le processus de génération de code. Ensuite, les phases de sélection d'instructions et d'allocation de registres pouvant introduire des renvois de variables intermédiaires en mémoire, il est nécessaire d'optimiser sous contrainte de précision le type de ces données afin de minimiser le temps d'exécution du code.

4.4.1 Détermination de la largeur des données

L'objectif de cette section est de définir la largeur de chaque donnée afin d'obtenir un format complet pour chacune d'elle avant de débuter le processus de génération de code. Cette transformation permet d'aboutir à une spécification de l'application en virgule fixe complète. Le choix de la largeur des données doit prendre en compte la diversité des types des données manipulées au sein du DSP telle que présentée dans le paragraphe 2.1.2 de la page 37. Nous dénommons opérations classiques, les opérations réalisées à l'aide d'une seule instruction et ne réalisant qu'une seule opération arithmétique ou de transfert de données entre l'unité de traitement et la mémoire. Certains DSP permettent d'accélérer les calculs à travers l'utilisation d'instructions SWP permettant de réaliser en parallèle au sein d'un même opérateur plusieurs opérations (opérations SWP). En contrepartie, la précision de ces opérations SWP est plus faible que celle obtenue avec les opérations classiques. A l'opposé, la majorité des DSP permet d'accroître la précision des calculs à travers l'utilisation d'opérations manipulant des données stockées en mémoire avec une précision plus importante (opérations multi-précision). Ces opérations utilisant l'arithmétique multi-précision sont composées d'une suite d'opérations classiques. En conséquence, le temps d'exécution de ces opérations est plus élevé. L'objectif principal de la génération de code étant de minimiser le temps d'exécution du code généré, la méthode présentée dans cette partie, sélectionne la séquence d'instructions dont la largeur des opérandes permet de respecter la contrainte de précision imposée et de minimiser le temps d'exécution global du code.

L'entrée de ce module est le GFDC G_{PV} au sein duquel la position de la virgule de chaque donnée est spécifiée. Ce GFDC est un graphe orienté et composé de nœuds représentant les données et les opérations sur ces données. Soit N_o le nombre d'opérations présentes au sein de ce GFDC. Chaque opération o_i est caractérisée par l'élément γ_i correspondant à la fonction réalisée, b_i la largeur de ses opérandes et t_i le temps d'exécution de l'opération. Le paramètre b_i regroupe la largeur des opérandes en entrée et en sortie de l'opération :

$$b_i = \{b_i^{e_1}, b_i^{e_2}, b_i^s\} \tag{4.26}$$

La méthode de détermination du type des données nécessite de réaliser certaines étapes avant de débuter le processus d'optimisation. Le synoptique de la méthode est présenté à la figure 4.7. Pour chaque opération du GFDC, les différentes instructions permettant de réaliser l'opération sont sélectionnées. Un module permettant d'estimer le temps d'exécution global du code en fonction des instructions choisies est mis en œuvre. La précision de la spécification en virgule fixe est évaluée à l'aide du Rapport Signal à Bruit de Quantification (RSBQ). Afin de minimiser le temps d'évaluation de cette métrique, une méthode analytique permet de déterminer l'expression du RSBQ qui sera utilisée dans le processus d'optimisation.

Évaluation de la contrainte de précision

L'objectif de cette partie est de déterminer l'expression du RSBQ associée à ce GFDC à l'aide de la méthode présentée dans le chapitre précédent. L'entrée de ce module est le GFDC pour

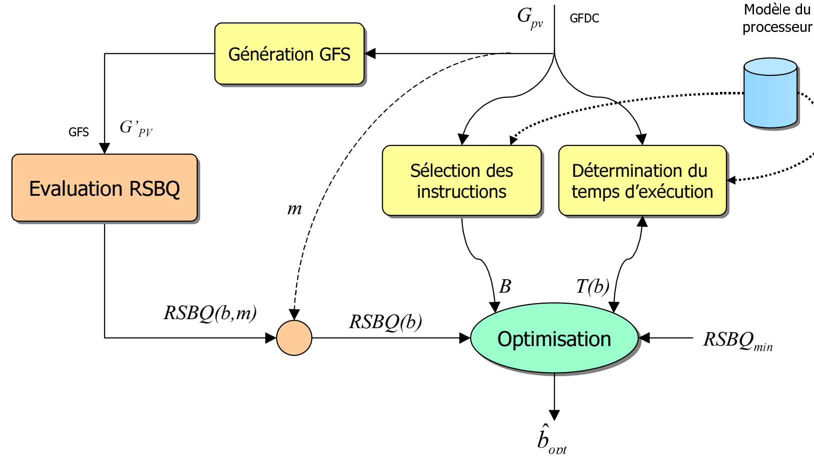


FIG. 4.7: Synoptique de la méthode de détermination de la largeur des données

lequel, les largeurs b_i des opérandes d'entrée et de sortie des opérations o_i sont des variables. Afin d'être plus général et de réutiliser cette expression pour le module suivant, la position de la virgule associée à chaque format est aussi considérée comme une variable. Soient $b = [b_1, b_2, \ldots, b_i, \ldots b_{N_o}]$ et $m = [m_1, m_2, \ldots, m_i, \ldots m_{N_o}]$, les vecteurs représentant respectivement la largeur et la position de la virgule des opérandes de l'ensemble des opérations du GFDC.

Le module d'évaluation de la précision permet de déterminer l'expression du RSBQ d'une application spécifiée à l'aide d'un graphe flot de signal (GFS). Ainsi, le GFDC représentant l'application est transformé en un GFS selon les principes exposés dans le paragraphe 4.2.2. Le GFDC utilisé pour déterminer l'expression du RSBQ est le GFDC de départ G_{app}. Les raisons de ce choix sont liées à l'étape suivante et sont explicitées dans le paragraphe 4.5.1. Chaque opération o_i du GFDC est représentée par une ou plusieurs opérations $o_{i'}$ au sein du GFS. Soit f la relation définissant les correspondances entre les opérations $o_{i'}$ et o_i :

$$\begin{array}{rlcl} f: & G'_{PV} & \mapsto & G_{PV} \\ & o_{i'} & \mapsto & o_i \end{array} \tag{4.27}$$

Soient $b' = [b'_1, b'_2, \ldots, b'_i, \ldots b'_{N_o}]$ et $m' = [m'_1, m'_2, \ldots, m'_i, \ldots m'_{N_o}]$, les vecteurs représentant respectivement la largeur et la position de la virgule des opérandes de l'ensemble des opérations du GFS. Chaque élément b'_i et m'_i des vecteurs b' et m' est défini à partir des éléments des vecteurs b et m à l'aide de la relation f de la manière suivante :

$$\begin{array}{l} b'_i = b_{f(o_{i'})} \\ m'_i = m_{f(o_{i'})} \end{array} \tag{4.28}$$

Le module d'évaluation de la précision fournit l'expression du RSBQ en fonction des vecteurs b' et m' représentant respectivement la largeur et la position de la virgule des opérandes de l'ensemble des opérations du GFS. En utilisant la relation f, l'expression du RSBQ est exprimée en fonction des vecteurs b et m associés au GFDC.

Détermination du temps d'exécution global

Modélisation du processeur : Pour cette partie, l'architecture est modélisée par un jeu d'instructions au niveau traitement de données. Ces instructions réalisent les opérations arithmétiques ou les transferts entre l'unité de traitement et la mémoire. Ces instructions sont obtenues à l'aide d'une instruction ou d'une séquence d'instructions du jeu d'instructions du processeur. Chaque instruction j_k est caractérisée par γ_k sa fonction, b_k la largeur de ses opérandes et t_k le temps d'exécution de l'opération réalisée. Ce temps d'exécution de l'opération

est obtenu à partir du temps d'exécution des instructions du processeur permettant de réaliser cette opération.

Actuellement, la description de ce jeu d'instructions est réalisée manuellement, mais par la suite cette modélisation pourra être obtenue automatiquement à partir de la description *Armor* du processeur qui décrit de manière synthétique le jeu d'instructions du processeur.

Évaluation du temps d'exécution global : L'objectif de cet estimateur est d'évaluer le temps d'exécution global de l'application en fonction des différents types d'instruction utilisés. Cependant, le but de cet estimateur n'est pas de déterminer précisément le temps d'exécution de l'application, mais il doit seulement permettre de comparer puis de sélectionner correctement deux séquences d'instructions différentes. De plus, le temps d'évaluation doit être le plus faible possible afin d'avoir des temps d'optimisation raisonnables. Ceci nécessite la mise en œuvre d'estimateurs peu complexes.

Un modèle d'évaluation du temps d'exécution global relativement simple a été retenu. Le temps d'exécution global T correspond à la somme des produits du temps d'exécution c_k de chaque bloc de base par le nombre d'exécutions n_k de ce bloc. Si le GFDC est composé de N_b blocs de base alors l'expression du temps d'exécution global T de ce GFDC est la suivante :

$$T = \sum_{k=1}^{N_b} n_k c_k \tag{4.29}$$

Le nombre d'exécutions n_k de chaque bloc de base est déterminé de manière statique à partir de l'analyse du flot de contrôle de l'application. Dans un premier temps, les structures conditionnelles ne sont pas prises en compte et les boucles ont un nombre fini d'itérations. Ainsi, une estimation statique du nombre d'exécutions de chaque bloc de base permet de déterminer exactement chaque valeur de n_k. Pour les structures de contrôle plus complexes, où le nombre d'exécutions des blocs de base dépend de la valeur de certaines variables, des méthodes statiques plus avancées ou des méthodes basées sur la simulation doivent être mises en œuvre [121].

Le temps d'exécution c_k du $k^{ème}$ bloc de base correspond à la somme du temps d'exécution de chaque opération présente au sein de ce bloc de base. Le temps d'exécution de chaque opération est fonction du type d'instruction utilisé et ainsi de la largeur des opérandes d'entrée et de sortie des opérations. Pour les opérations classiques, permettant de ne réaliser qu'une seule opération par instruction, le temps d'exécution de l'opération est égal au temps d'exécution de l'instruction utilisée pour réaliser cette opération. Pour les instructions SWP permettant de réaliser plusieurs opérations en parallèle au niveau d'un opérateur, le temps d'exécution de l'opération correspond au temps d'exécution de l'instruction divisé par le nombre d'opérations réalisées en parallèle par cette instruction SWP.

Dans le cas des processeurs ne possédant pas de parallélisme au niveau instruction, l'estimation du temps d'exécution d'un bloc de base, basée sur la somme des temps d'exécution des opérations de ce bloc fournit des résultats corrects et proches de la réalité. Pour les processeurs possédant du parallélisme au niveau instruction, cette estimation ne permet pas de prendre en compte l'exécution en parallèle de certaines instructions. Cette estimation permet de déterminer correctement le temps d'exécution du code vertical mais pas du code horizontal obtenu après la phase de compactage de code. Cependant, cette estimation fournit des résultats suffisants pour comparer correctement deux séquences d'instructions dans le cas des processeurs possédant du parallélisme au niveau instruction. Les différentes raisons justifiant cette affirmation sont présentées ci-dessous.

Pour les opérations classiques et les différentes opérations SWP, les gains en termes de temps d'exécution obtenus lors du passage du code vertical au code horizontal sont relativement

proches pour ces deux types d'opérations. En effet, les différentes séquences d'opérations classiques ou SWP utilisent les mêmes unités fonctionnelles et dans la majorité des cas pendant les mêmes cycles d'horloge. La différence réside dans la fonctionnalité des unités fonctionnelles. Dans le cas des instructions SWP, les unités traitent des fractions de mot au lieu de traiter des mots complets. Entre autre, les différentes instructions SWP de multiplication, d'addition et de décalage du DSP TMS320C64x présentées dans le tableau 2.2 à la page 41, satisfont l'hypothèse présentée ci-dessus. Nous pouvons considérer que les relations d'ordre entre deux séquences d'opérations classiques ou SWP avant et après les phases de compactage sont conservées.

Cependant, l'utilisation d'opérations SWP nécessite parfois des instructions de mise en paquet ou d'extraction des fractions de mots. Dans ce cas, la comparaison entre les opérations classiques et SWP, basée sur une estimation du temps d'exécution du code vertical privilégie les instructions classiques au détriment des instructions SWP. En effet, la phase de compactage pourrait permettre d'exécuter ces instructions de mise en paquet ou d'extraction, en parallèle aux autres instructions et ainsi réduire leur surcoût.

Les opérations multi-précision correspondent à une séquence d'opérations classiques. Ainsi, dans le cas le plus favorable après la phase de compactage, le temps d'exécution de l'opération classique est égal au temps d'exécution de l'opération multi-précision. Cependant, il est préférable dans ce cas de privilégier les opérations classiques si elles permettent d'obtenir la précision souhaitée car la taille nécessaire pour stocker les données est plus faible. Ainsi, pour n'utiliser les opérations multi-précision que si la contrainte de précision l'exige, le temps d'exécution des opérations multi-précision est fixé à la valeur maximale obtenue sans parallélisme d'instruction.

Cette surestimation possible du temps d'exécution des opérations multi-précision est sans conséquence pour ce processus d'optimisation car ces opérations ne sont pas sélectionnées sur le critère du temps d'exécution. En effet, elles ne sont utilisées que si les instructions classiques ne permettent pas d'atteindre la précision souhaitée.

Pour obtenir des estimations plus précises du temps d'exécution du code, des techniques d'estimation logicielle avancées telles que celles présentées dans [121, 112], peuvent être mises en œuvre.

Optimisation de la largeur des données

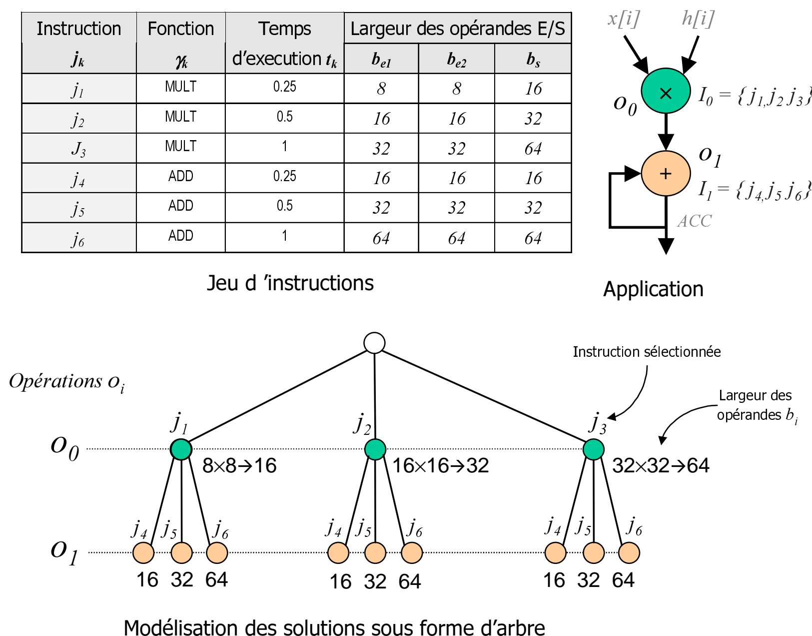
Pour chaque opération o_i du GFDC, les différentes instructions I_i permettant de réaliser cette fonction sont sélectionnées. Ainsi, pour chaque opération o_i, une des instructions de cet ensemble I_i devra être choisie pour réaliser cette opération. Les opérations SWP permettent d'accélérer les calculs en exploitant le parallélisme au niveau des données. Cependant, le surcoût de mise en paquet ou d'extraction des fractions de mots peut annihiler le gain lié à l'exploitation du parallélisme au niveau des données. Ainsi, cette technique doit être réservée aux structures de données régulières sur laquelle des traitements en parallèle peuvent être réalisés. Ainsi, les opérations SWP ne sont utilisées que pour les opérations manipulant des données stockées sous la forme de vecteur et présentes au sein des structures répétitives.

Soit B_i l'ensemble des largeurs des opérandes associés à l'opération o_i. Ainsi, pour chaque opération o_i, la largeur optimale $\widehat{b_i}$ appartenant à l'ensemble B_i, permettant de minimiser le temps d'exécution global $T(b)$ et de respecter la contrainte de précision ($RSBQ_{min}$) doit être déterminée :

$$\min_{b \in B} \Big(T(b) \Big) \quad \text{tel que} \quad RSBQ(b) \geq RSBQ_{min} \tag{4.30}$$

Les variables de ce problème d'optimisation correspondent aux largeurs b_i des opérandes des opérations o_i du GFDC. Chaque opération ne peut être réalisée que par quelques instructions correspondant aux instructions classiques, SWP ou multi-précision. Ainsi, les variables b_i ne

peuvent prendre que quelques valeurs entières prédéfinies. Le domaine de définition très restreint des variables b_i ne permet pas d'utiliser uniquement les techniques classiques de programmation en nombres entiers. En effet, ces techniques peuvent fournir des solutions relativement éloignées des différents éléments du domaine de définition des variables b_i. Cependant, le domaine de définition très restreint des variables b_i permet de modéliser ce problème de minimisation sous la forme de la recherche d'un chemin au sein d'un arbre. Chaque niveau k correspond au traitement de l'opération o_i (variable b_i). Chaque nœud de ce niveau i associe à l'opération o_i une instruction donnée j_k de I_i. Un exemple de modélisation sous forme d'aune instruction rbre de problème d'optimisation est présentée à la figure 4.8



Instruction j_k	Fonction γ_k	Temps d'execution t_k	Largeur des opérandes E/S b_{e1}	b_{e2}	b_s
j_1	MULT	0.25	8	8	16
j_2	MULT	0.5	16	16	32
J_3	MULT	1	32	32	64
j_4	ADD	0.25	16	16	16
j_5	ADD	0.5	32	32	32
j_6	ADD	1	64	64	64

FIG. 4.8: Exemple de modélisation sous forme d'arbre du problème de sélection de la séquence d'instructions permettant d'optimiser la largeur des données

La modélisation sous forme d'arbre permet de modéliser de manière exhaustive l'ensemble des solutions possibles de ce problème de minimisation. Le succès de ce type de méthode réside dans la possibilité de limiter fortement l'espace de recherche des solutions. En effet, la complexité de l'algorithme d'exploration de l'arbre est exponentielle. Soit N_o, le nombre de variables présentes au sein de ce problème d'optimisation et N_i, le nombre de valeurs possibles pour chaque variable. L'exploration exhaustive de l'arbre nécessite de traiter $N_i^{N_o}$ solutions possibles. La limitation de l'espace de recherche, nécessite de définir des critères permettant d'éliminer les solutions non-viables sans écarter des solutions correctes. Les différentes techniques permettant de limiter l'espace de recherche sont présentées ci-dessous.

Domaine de validité des variables : La modélisation du problème sous forme d'arbre permet de modéliser de manière exhaustive l'ensemble des solutions. Cependant, toutes les combinaisons d'instructions ne sont pas possibles. Pour illustrer ce problème, considérons l'exemple présenté à la figure 4.9. L'entrée et la sortie des opérations sont annotées avec leur format. Pour simplifier la présentation de cet étude nous considérons que chaque opération possède une seule entrée. Cependant, les résultats peuvent être facilement étendus aux opérations à deux entrées.

Sachant que la donnée correspondant à l'entrée de l'opération o_l est le résultat de l'opération o_k, le nombre de bits n_l^e réservés pour la partie fractionnaire de l'entrée de o_l ne peut être

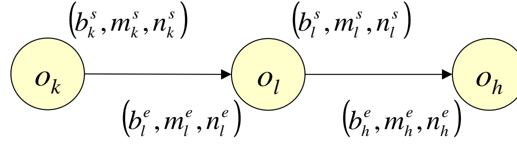


FIG. 4.9: Exemple de traitement au sein du GFD

strictement supérieur au nombre de bits n_k^s associé à la partie fractionnaire du résultat de o_k. Ainsi, pour l'opération o_l nous devons respecter les deux conditions suivantes :

$$\begin{array}{c} n_k^s \geq n_l^e \\ n_l^s \geq n_h^e \end{array} \tag{4.31}$$

Deux types d'approche peuvent être mis en œuvre pour vérifier que ces deux conditions sont respectées pour chaque opération o_l. La première approche suppose que la solution optimale est telle que pour chaque opération tous les bits les moins significatifs des opérandes d'entrée et de sortie sont utilisés. Ainsi, si pour l'opération o_l, les deux conditions présentées à l'équation 4.31 ne sont pas respectées alors l'instruction associée à o_l est déclarée non valide. En conséquence, l'exploration de cette branche de l'arbre est interrompue et une nouvelle valeur de b_i est testée. Cette technique permet de limiter fortement l'espace de recherche et correspond à celle utilisée dans l'outil développé pour implanter cette méthodologie.

Pour la seconde approche lorsque les conditions ne sont pas respectées, les formats des entrées et de la sortie des opérations sont modifiés. Nous considérons le traitement de la $j^{ème}$ instruction possible pour l'opération o_l. La largeur des opérandes d'entrée et de sortie de l'opération o_l pour cette solution correspond à $B(l,j)$. Les différents cas à traiter sont présentés ci-dessous :

- *Vérification de la première condition de l'équation 4.31 ($o_k \to o_l$) :*
  - Si la largeur de la sortie de l'opération o_k, n'a pas encore été définie alors la vérification de la première condition présentée à l'équation 4.31 sera effectuée lors du traitement du nœud o_k et la largeur de l'entrée de l'opération o_l est initialisée avec la valeur $B^e(l,j)$.

  - Si la largeur de la sortie de l'opération o_k est définie et si la valeur $B^e(l,j)$ ne permet pas de respecter la première condition de l'équation 4.31, alors le paramètre n_l^e est initialisé avec n_k^s afin que les nombres de bits pour les parties fractionnaires de la sortie de l'opération o_k et de l'entrée de l'opération o_l soient identiques. Cette modification de la valeur de n_l^e peut entraîner une modification du paramètre n_l^s définissant le nombre de bits réellement utilisés pour coder la partie fractionnaire de la sortie. Ce paramètre n_l^s est soit lié au paramètre n_l^e de l'entrée ou à la largeur associée à l'instruction testée $B^s(l,j)$ selon la relation 4.32. Les fonctions f_{γ_l} déterminent les paramètres du format de l'opérande de sortie en fonction de ceux des opérandes d'entrée et du type d'opération réalisée. Cette fonction est définie à partir des informations présentées dans le paragraphe 1.1.2 à la page 10.

$$\left\{ \begin{array}{l} n_l^s = \min\left(f_{\gamma_l}^n(n_l^e), B^s(l,j) - f_{\gamma_l}^m(m_l^e) - 1\right) \\ b_l^s = m_l^s + n_l^s + 1 \end{array} \right. \tag{4.32}$$

  Si la largeur b_l des opérandes d'entrée et de sortie est différente de celle de la solution testée $B(l,j)$, alors il est nécessaire d'analyser si cette largeur b_l correspond à une solution appartenant à $B(l)$. Le cas échéant, l'exploration est arrêtée car la valeur actuelle de b_l sera testée par la suite. Sinon, l'exploration est poursuivie avec la nouvelle valeur de b_l. Les largeurs des opérandes des différentes opérations du jeu d'instructions étant

homogènes, si les positions de la virgule de la sortie de l'opération o_k et de l'entrée de l'opération o_l sont identiques, alors dans la majorité des cas la largeur b_l correspond à une solution appartenant à $B(l)$ et l'exploration peut être stoppée.

- *Vérification de la seconde condition de l'équation 4.31 ($o_l \rightarrow o_h$)*
  - Si la largeur de l'entrée de l'opération o_h, n'a pas encore été définie alors la vérification de la seconde condition présentée à l'équation 4.31 sera effectuée lors du traitement du nœud o_h.
  - Si la largeur de l'entrée de l'opération o_h est définie et si la valeur b_l^s calculée à l'équation 4.32 ne permet pas de respecter la seconde condition de l'équation 4.31, alors la largeur de l'entrée b_h^e doit être modifiée afin de permettre de vérifier cette seconde condition. Dans ce cas, si la nouvelle largeur b_h correspond à une solution appartenant à $B(l)$ alors l'exploration est arrêtée car cette valeur de b_h sera testée par la suite. Sinon l'exploration est poursuivie avec cette nouvelle valeur de b_h. Cependant, suite à cette modification il est nécessaire de vérifier si les conditions sont vérifiées pour les autres opérations.

Évaluation des solutions partielles : La technique utilisée au sein des algorithmes de type *branch and bound*, pour limiter l'espace de recherche consiste à arrêter l'exploration d'une branche de l'arbre si celle-ci ne peut plus conduire à la meilleure solution. Ceci nécessite de pouvoir évaluer une solution partielle à un niveau quelconque de l'arbre, d'un point de vue du RSBQ et du temps d'exécution global.

Au niveau l de l'arbre, l'exploration du sous-arbre induit par le nœud représentant $\widehat{b}_l$ est arrêtée si le temps d'exécution minimal pouvant être obtenu lors de l'exploration du sous-arbre est supérieur au temps d'exécution global minimal déjà obtenu. Sachant qu'uniquement les largeurs b_0 à b_l ont été définies, ce temps d'exécution minimal est obtenu en sélectionnant pour chaque opération o_j pour laquelle la largeur des opérandes n'est pas encore définie, l'instruction ayant un temps d'exécution t_j minimal. Dans la majorité des cas, ceci équivaut à sélectionner l'instruction dont la largeur des opérandes est minimale ($\underline{b}_j$). Dans ce cas, à tous niveaux l de l'arbre nous avons la relation suivante :

$$T([\widehat{b}_0, \ldots, \widehat{b}_{l-1}, \widehat{b}_l, \underline{b}_{l+1}, \ldots, \underline{b}_{N_o}]) \leq T([\widehat{b}_0, \ldots, \widehat{b}_{l-1}, \widehat{b}_l, b_{l+1}, \ldots, b_{N_o}]) \qquad (4.33)$$

Au niveau l de l'arbre, l'exploration du sous-arbre induit par le nœud représentant $\widehat{b}_l$ est arrêtée si la contrainte sur la précision ne peut plus être respectée. Ceci est le cas si la valeur maximale du RSBQ global pouvant être obtenue lors de l'exploration du sous-arbre est inférieure à la valeur minimale du RSBQ souhaité. Le majorant du RSBQ est obtenu en fixant les largeurs b_i non définies à leur valeur maximale ($\overline{b}_i$) permettant de respecter les conditions présentées à l'équation 4.31. En effet, le RSBQ étant une fonction monotone et croissante, la valeur maximale du RSBQ est obtenue lorsque les largeurs des opérandes sont maximales. Ainsi, à tous niveaux l de l'arbre nous avons la relation suivante :

$$RSBQ([\widehat{b}_0, \ldots, \widehat{b}_{l-1}, \widehat{b}_l, \overline{b}_{l+1}, \ldots, \overline{b}_{N_o}]) \geq RSBQ([\widehat{b}_0, \ldots, \widehat{b}_{l-1}, \widehat{b}_l, b_{l+1}, \ldots, b_{N_o}]) \qquad (4.34)$$

Ordre d'évaluation des nœuds : Cette méthode de minimisation basée sur l'exploration d'arbre est sensible à l'ordre d'évaluation des variables. Afin de trouver rapidement une bonne solution en vue de réduire l'espace de recherche, il est nécessaire de traiter en premier les variables ayant le plus d'influence sur le processus d'optimisation. Ce processus évalue les deux paramètres que sont le temps d'exécution global de l'application et le RSBQ en sortie.

Le temps d'exécution d'une opération dépend du temps d'exécution de l'instruction choisie pour réaliser celle-ci et du nombre d'exécutions du bloc de base contenant cette opération. Le premier paramètre n'est pas connu avant le traitement de l'opération. En revanche, le second paramètre est connu avant de débuter le processus d'optimisation. Ainsi, les variables sont ordonnées afin de traiter en premier les opérations appartenant aux blocs de base ayant un nombre d'exécutions maximal. Ensuite, au sein de chaque bloc de base, les sources de bruit sont ordonnées en fonction de leur influence sur le RSBQ en sortie.

La puissance du bruit en sortie de l'application est liée à la contribution de chaque source de bruit. Cette contribution dépend du gain présent entre la sortie et la source de bruit et de la puissance associée à cette source. La puissance du bruit généré lors d'un changement de format dépend du nombre de bits éliminés et de la position de la virgule par rapport au bit le moins significatif. Cependant, d'une spécification virgule fixe à une autre, le nombre de bits éliminés est très variable. Ainsi, pour définir l'influence potentielle de chaque source de bruit, les largeurs de toutes les données sont fixées à une même valeur et pour chaque source de bruit, le nombre de bits éliminés est fixé à une même valeur. Les sources de bruit les plus influentes seront celles pour lesquelles le gain entre la source et la sortie est élevé ou celles associées à une donnée pour laquelle la position de la virgule est élevée. L'influence d'une opération dépend de l'influence des sources de bruit présentes au niveau de ses entrées et de sa sortie.

4.4.2 Optimisation de la largeur des données transférées en mémoire

L'objectif de cette section est d'optimiser la largeur des données issues du renvoi des variables intermédiaires en mémoire. En effet, lorsque le nombre de registres présents au sein de l'unité de traitement n'est pas assez élevé pour stocker l'ensemble des variables intermédiaires, il est nécessaire de renvoyer certaines de ces variables en mémoire. Ce type d'opération de renvoi en mémoire est présent au sein des architectures spécialisées possédant peu de registres. Comme nous l'avons présenté dans la partie 2.2.4, les données sont stockées en mémoire et au sein de l'unité de traitement avec des précisions différentes. Ainsi, un surcoût en termes de temps d'exécution peut être présent si la donnée est renvoyée avec une précision maximale. Ainsi, la méthodologie doit minimiser le temps d'exécution de ces opérations de renvoi de données intermédiaires en mémoire.

Le synoptique de la méthode de détermination de la largeur des données renvoyées en mémoire est présenté à la figure 4.10. Soit o_{t,i,e_k} l'opération de renvoi en mémoire de la variable intermédiaire présente au niveau de l'entrée e_k de l'opération o_i. Les différentes opérations de renvoi en mémoire sont signalées par la phase d'allocation de registres. Pour chaque opération $o_{t,i,,e_k}$, les différentes instructions de transfert entre l'unité de traitement et la mémoire sont sélectionnées. Chaque instruction de transfert est caractérisée par la largeur des opérandes manipulés et ainsi elle fixe la largeur $b_{t,i}$ des opérandes de l'opération o_i. Soit $B_{t,i}$ l'ensemble des largeurs possibles des opérandes associés à l'opération o_i. L'objectif de cette méthode est de minimiser le temps d'exécution global $T_t(b_t)$ des opérations de renvoi en mémoire tant que la contrainte de précision est respectée :

$$\min_{b_t \in B_t} \Big(T_t(b_t) \Big) \quad \text{tel que} \quad RSBQ(b_t) \geq RSBQ_{min} \tag{4.35}$$

Le temps d'exécution global des opérations de transfert en mémoire $T_t(b_t)$ est obtenu à partir des temps d'exécution des différentes opérations de transfert selon la même technique que celle présentée dans la section précédente. De même, ce problème d'optimisation est résolu en utilisant la méthode présentée précédemment et basée sur une modélisation du problème sous forme d'arbre.

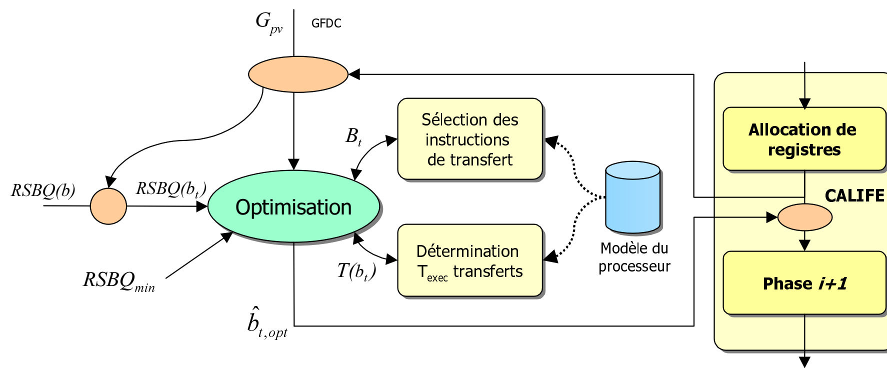


FIG. 4.10: Synoptique de la méthode de détermination du type des données dans le cadre de l'optimisation de la largeur des données transférées en mémoire

4.4.3 Expérimentations

Dans cette partie, les résultats des expérimentations réalisées avec l'outil développé pour implanter la méthode présentée dans le paragraphe 4.4.1, sont exposés. Ces résultats permettent à travers deux applications de montrer les capacités de notre méthodologie pour optimiser la largeur des données. Dans un premier temps, le modèle d'architecture d'un processeur possédant des capacités SWP est considéré. Ensuite, nous montrons l'intérêt de notre méthode pour coder certaines parties d'algorithme en double précision afin d'obtenir une précision plus importante. Les exemples choisis sont relativement simples afin de détailler le codage des données obtenu pour différents cas particuliers. Pour ces deux applications, les temps d'exécution présentés correspondent aux temps estimés à l'aide de cette méthodologie. Dans le dernier paragraphe, les temps d'optimisation obtenus sur différentes applications sont présentés.

Implantation d'un corrélateur complexe dans le TMS320C64x

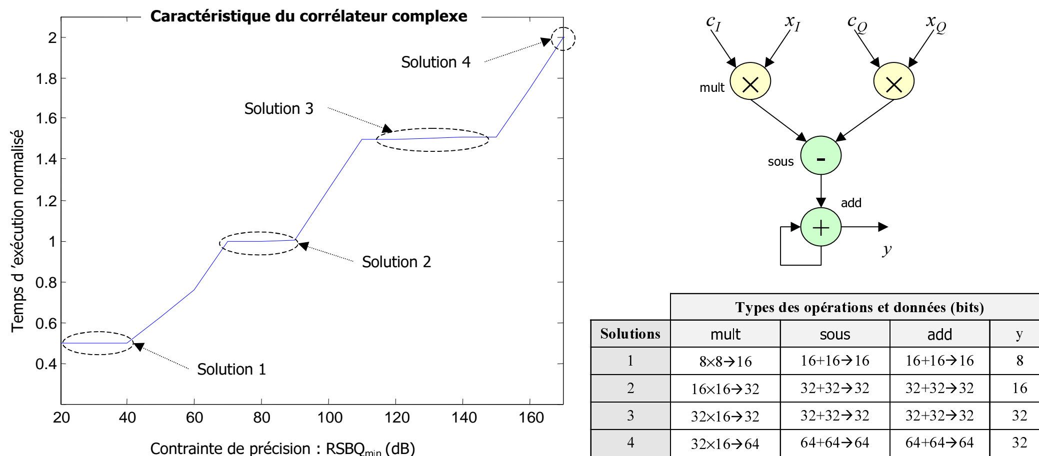
La première application testée est un corrélateur complexe réalisant la corrélation entre un signal x et un code bipolaire c composé de 32 éléments. Cette application est utilisée par exemple pour réaliser l'opération de décodage des données au sein d'un récepteur WCDMA. Le signal d'entrée et le code sont représentés sous la forme complexe. L'application effectue une multiplication complexe entre le signal et le code et réalise ensuite une accumulation sur les voies réelles et imaginaires. Dans l'application considérée, la précision est évaluée au niveau de la voie réelle de la sortie du corrélateur.

Notre méthodologie permet de déterminer l'évolution du temps d'exécution optimal estimé (T_{opt}) en fonction de la contrainte de RSBQ souhaité ($RSBQ_{min}$) pour l'implantation d'une application donnée dans un DSP donné. Chaque point de la courbe $T_{opt} = f\left(RSBQ_{min}\right)$, d'abscisse ρ_o et d'ordonnée T_o a été obtenu en optimisant le temps d'exécution de l'application sous une contrainte de RSBQ égale à ρ_o. Le temps d'exécution minimal obtenu est égal à T_o. Cette caractéristique $T_{opt} = f\left(RSBQ_{min}\right)$ permet par exemple à l'utilisateur d'évaluer le gain en termes de temps d'exécution d'une modification de la contrainte de précision.

La caractéristique $T_{opt} = f\left(RSBQ_{min}\right)$ a été déterminée pour le corrélateur complexe dans le cadre du modèle d'architecture du DSP TMS320C64x. L'évolution du temps d'exécution estimé en fonction de la contrainte de RSBQ souhaité est présentée à la figure 4.11. Afin d'explorer les différentes solutions possibles, aucune contrainte sur la largeur des entrées et de la sortie de cette application n'a été définie. De plus, pour comparer les différentes implantations, le temps d'exécution a été normalisé par rapport à la solution 2. Cette solution réalise les multiplications sur des données codées sur 16 bits et fournit un résultat sur 32 bits, les opérations d'addition et de soustraction sont réalisées sur des données sur 32 bits. La caractéristique $T = f\left(RSBQ_{min}\right)$

évolue par paliers. Les quatre paliers présents sur la courbe correspondent à des spécifications virgule fixe particulières pour lesquelles la largeur des différentes données présentes au sein du cœur de la boucle a été détaillée.

Les solutions 1, 2 et 4 sont basées sur la même structure de codage des données. Elles traitent respectivement des données stockées en mémoire sur b_n bits, le résultat de la multiplication fournit un résultat sur $2.b_n$ bits et les additions et soustractions sont réalisées sur des opérandes de largeur $2.b_n$. Pour les solutions 1, 2 et 4, cette largeur est respectivement égale à 8, 16 et 32 bits. La solution 3 utilise des données codées en mémoire sur 32 bits mais à la différence de la solution 4, les opérations d'addition et de soustraction sont réalisées sur 32 bits et permettent ainsi d'obtenir un temps d'exécution plus faible que celui obtenu avec la solution 4.



Solutions	Types des opérations et données (bits) mult	sous	add	y
1	8×8→16	16+16→16	16+16→16	8
2	16×16→32	32+32→32	32+32→32	16
3	32×16→32	32+32→32	32+32→32	32
4	32×16→64	64+64→64	64+64→64	32

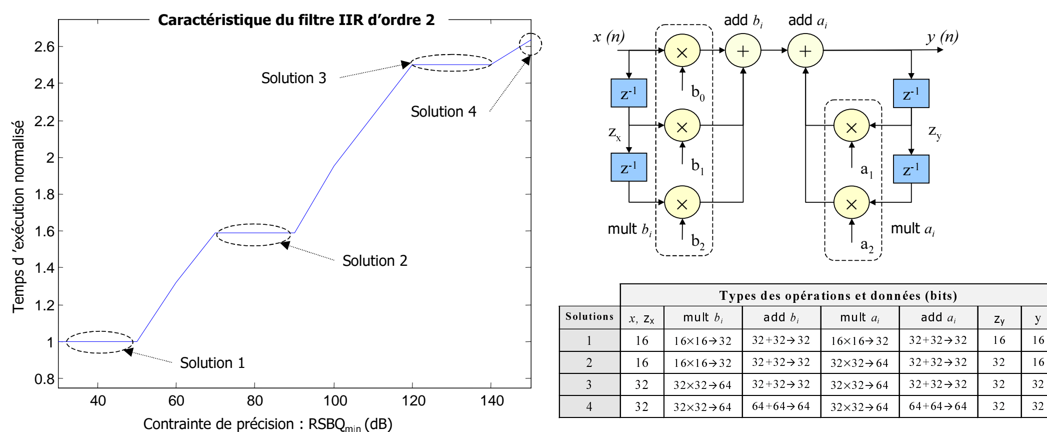
FIG. 4.11: Évolution du temps d'exécution minimal en fonction de la contrainte de précision dans le cas d'un corrélateur complexe

Implantation d'un filtre IIR d'ordre 2 dans le TMS320C54x

La méthode présentée dans cette partie montre tout son intérêt pour la détermination du type des données lors de l'implantation d'applications au sein de processeurs proposant des instructions SWP. Cependant, cette méthode se révèle être très intéressante pour l'implantation dans des DSP traditionnels, d'applications nécessitant une précision importante. Dans ce cas, la méthode permet de sélectionner les opérations classiques ou multi-précision. Pour illustrer ceci nous avons implanté un filtre à réponse impulsionnelle infinie dans le DSP TMS320C54.

Ce filtre possède des contraintes de codage en virgule fixe assez sévères. Ainsi, le RSBQ obtenu pour une solution utilisant uniquement des opérations classiques est relativement faible (50 dB). Pour améliorer la précision, les solutions utilisant les opérations multi-précision ont été testées. Le modèle d'architecture utilisé pour le TMS320C54x intègre les opérations classiques et double-précision. La caractéristique $T = f\left(RSBQ_{min}\right)$ obtenue pour le filtre IIR d'ordre 2 est présentée à la figure 4.12. Les temps d'exécution estimés ont été normalisés avec le temps d'exécution obtenu dans le cadre de la solution 1 utilisant exclusivement des opérations classiques. Ceci permet de déterminer directement le surcoût en termes de temps d'exécution lié à l'utilisation de l'arithmétique multi-précision.

Pour la solution 4, l'entrée et la sortie du filtre sont codées sur 32 bits et tous les calculs sont réalisés au sein du filtre en double précision. La solution 3 reprend le codage de la solution 4, excepté pour les additions des sorties des multiplieurs. Ces dernières sont réalisées avec des opérations classiques traitant des données sur 32 bits. Ainsi, les sorties des multiplieurs codées sur 64 bits sont tronquées sur 32 bits. Le temps d'exécution de ces deux solutions est égal au moins à 2.5 fois le temps d'exécution de référence.



Solutions	Types des opérations et données (bits)						
	x, z_x	mult b_i	add b_i	mult a_i	add a_i	z_y	y
1	16	16×16→32	32+32→32	16×16→32	32+32→32	16	16
2	16	16×16→32	32+32→32	32×32→64	32+32→32	32	16
3	32	32×32→64	32+32→32	32×32→64	32+32→32	32	32
4	32	32×32→64	64+64→64	32×32→64	64+64→64	32	32

FIG. 4.12: Évolution du temps d'exécution minimal en fonction de la contrainte de précision dans le cas d'un filtre IIR d'ordre 2

La solution 2 fournit des résultats très intéressants. L'entrée et la sortie du filtre sont codées sur un nombre de bits identique à celui de la première solution (16 bits). Ainsi, les spécifications virgule fixe de l'entrée et de la sortie des deux solutions sont identiques. La solution 2 permet d'augmenter le RSBQ en sortie du filtre de 40 dB avec un surcoût en termes de temps d'exécution de seulement 60%. Pour cette solution, la partie récursive du filtre utilise des opérations double-précision. Les sorties retardées du filtre $(y(n-1), y(n-2))$ sont stockées sur 32 bits afin de limiter la perte de précision dans cette partie récursive.

Temps d'optimisation

Le temps d'exécution de notre outil a été mesuré au cours des différentes expérimentations réalisées, afin de vérifier l'efficacité de notre méthodologie. Pour chaque application, différentes contraintes de précision ($RSBQ_{min}$) ont été testées afin d'étudier les variations des temps d'exécution en fonction de la contrainte $RSBQ_{min}$. Les résultats montrent que plus la contrainte de précision est élevée, plus le temps d'optimisation est faible. En effet, le nombre de solutions vérifiant la contrainte de précision étant moins important, l'espace de recherche est réduit et l'algorithme d'optimisation permet d'obtenir une solution plus rapidement.

Dans un premier temps, aucune contrainte n'a été fixée sur la largeur des données en entrée et en sortie des applications testées. Les résultats obtenus sont fournis à la figure 4.13.a. Cette figure représente les temps d'optimisation mesurés en fonction du nombre de variables à optimiser. Les expérimentations ont été réalisées avec un modèle d'architecture proposant 4 instructions par type d'opération (i.e. chaque variable peut prendre 4 valeurs différentes). Ces résultats montrent que le temps d'optimisation dépend fortement de l'application testée et de la contrainte de précision choisie. Cependant, pour l'application possédant 30 variables à traiter, le temps d'optimisation est de l'ordre de quelques minutes. Dans le cas de cette application, les temps d'optimisation sont inférieurs à 10 secondes, pour une architecture proposant deux instructions par opération.

Dans un second temps, nous avons mesuré les temps d'exécution en fixant la largeur des entrées et de la sortie des applications. Nous avons reporté à la figure 4.13.b, uniquement les résultats obtenus avec les applications composées de 30 et 33 variables. Pour les autres applications les temps d'optimisation étaient insignifiants. Pour ces deux applications composées d'une trentaine de variables, le temps d'optimisation est raisonnable il reste inférieur à 4 minutes et 10 secondes. Ainsi, les différentes techniques mises en œuvre pour réduire l'espace de recherche permettent de limiter efficacement celui-ci et d'obtenir des temps d'exploration acceptables. De plus, pour les résultats proposés, l'ordre d'évaluation des nœuds est quelconque. Ainsi, la mise

en œuvre de la technique proposée pour traiter cet aspect permettrait de diminuer les temps d'optimisation.

L'efficacité de cette méthode en termes de temps d'optimisation nécessite de mettre en œuvre des techniques d'estimation des temps d'exécution et d'évaluation de la précision efficaces. En effet, pour les applications composées d'une trentaine de variables, la précision de l'implantation a été évaluée entre 50000 et 100000 fois.

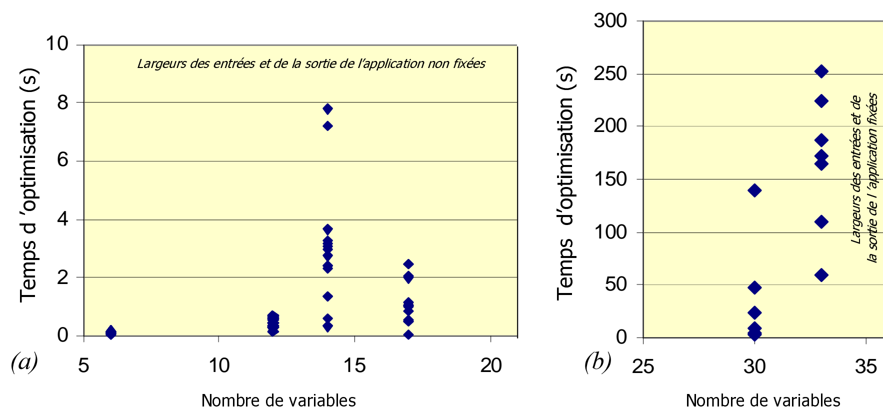


FIG. 4.13: Évolution du temps d'exécution du processus d'optimisation en fonction du nombre de variables à optimiser

4.4.4 Conclusions

Dans cette partie, la méthode de détermination de la largeur des données a été exposée. Tout d'abord, le problème d'optimisation a été modélisé et un algorithme d'optimisation basé sur l'exploration d'arbre a été utilisé. L'efficacité de cette technique étant basée sur la limitation de l'espace de recherche des solutions, les différentes techniques utilisées pour réduire cet espace de recherche (domaines de validité des variables, évaluation des solutions partielles, ordre d'évaluation des noeuds) ont été détaillées.

Les expérimentations ont montré l'intérêt de cette méthodologie pour explorer les différentes instructions proposées par les DSP en vue d'obtenir une implantation optimisée et respectant les critères de précision spécifiés par l'utilisateur. De plus, l'efficacité de notre méthodologie en termes de temps d'optimisation est satisfaisante. Cependant, nous pouvons nous heurter à des temps d'optimisation élevés pour des problèmes comportant un nombre de variables et de valeurs par variable élevé. Dans ce cas, il peut être intéressant de scinder ce problème d'optimisation en deux phases. La première phase réaliserait l'optimisation en considérant les variables comme des nombres entiers positifs. Dans ce cas, des méthodes de programmation en nombres entiers peuvent être utilisées. Cette solution permettrait de diminuer le domaine de définition en ne retenant que les valeurs situées autour de la solution obtenue au cours de la première phase. Ensuite, la méthode que nous avons exposée ci-dessus serait utilisée pour obtenir la solution optimale appartenant au domaine de définition des variables.

Condition de passage d'une spécification à une implantation optimisée

Cette phase d'optimisation de la largeur des données fournit une spécification en virgule fixe permettant de respecter la contrainte de précision et de minimiser le temps d'exécution des calculs à l'aide d'instructions SWP. Cette spécification est utilisée pour définir le type de chaque donnée au niveau de la représentation intermédiaire avant de débuter la phase de génération de code. Nous avons fait l'hypothèse que l'obtention de cette spécification optimisée

conduit à une implantation optimisée de l'application. Cette hypothèse implique que la phase de sélection d'instructions puisse regrouper efficacement les opérations SWP afin de pouvoir les exécuter en parallèle. Malgré les restrictions imposées au niveau de l'utilisation des instructions SWP, il est peu probable que les techniques de sélection d'instructions classiques puissent détecter les opportunités présentes au niveau de la spécification. Ainsi, cette phase d'optimisation de la largeur des données souligne les opportunités de réduction du temps d'exécution liées à l'utilisation des instructions SWP mais il est nécessaire de mettre en œuvre une technique permettant d'utiliser réellement ces instructions SWP au cours du processus de génération de code.

Ce problème de prise en compte des opérations SWP au niveau des compilateurs de langage de haut niveau est abordé par Leupers dans [79]. Pour contourner ce problème et tirer profit des capacités SWP, différentes approches sont utilisées [79]. La première technique consiste à utiliser des bibliothèques en assembleur proposant des routines utilisant les capacités SWP des opérateurs. La seconde technique correspond à l'utilisation de fonctions intrinsèques permettant de spécifier à l'aide de fonctions C, les différentes instructions SWP du processeur. Ces deux techniques souffrent de défauts majeurs. En effet, elles représentent un obstacle à l'optimisation du code et à la portabilité de celui-ci. Dans les deux cas, le travail de mise en vectorisation des calculs est à la charge du programmeur.

Dans la dernière version de son compilateur pour le Pentium MMX, la société Intel intègre un outil permettant la vectorisation automatique de boucles [15]. De même, Leupers propose une technique permettant de prendre en compte au niveau de la phase de sélection d'instructions, les capacités SWP du processeur [79]. La phase de sélection d'instructions est réalisée à l'aide d'une version modifiée de l'outil OLIVE et la couverture de code est mise sous la forme d'un problème de programmation linéaire en nombres entiers. Les résultats obtenus pour deux processeurs montrent l'aptitude de cette technique à prendre en compte les instructions SWP.

4.5 Optimisation du codage des données

4.5.1 Présentation du problème

Les phases de détermination de la position de la virgule et du type des données conduisent à l'obtention d'une spécification en virgule fixe optimale au niveau précision. En effet, les opérations de recadrage sont insérées afin de coder les différentes données au plus précis. Cependant, cette spécification peut nécessiter l'exécution de nombreuses opérations de recadrage comme l'indiquent les résultats présentés dans le paragraphe 4.3.3 situé à la page 137. Ces différentes opérations de recadrage augmentent le temps d'exécution et la taille du code. Pour diminuer le surcoût lié à ces opérations de recadrage, celles-ci sont déplacées. Ces déplacements peuvent permettre de fusionner différentes opérations de recadrage, de diminuer le nombre d'exécutions de l'opération ou de diminuer le temps d'exécution de celle-ci. En contrepartie, la précision des calculs va être diminuée car certaines données ne seront plus codées aussi précisément. La solution optimale est un compromis entre la précision des calculs et le temps d'exécution de l'implantation. Ainsi, l'objectif de cette partie est de rechercher cette solution optimale en fonction de la contrainte de précision souhaitée. Pour cela, le temps d'exécution du code est minimisé tant que la contrainte de précision est satisfaite. Cette réduction du temps d'exécution du code est obtenue en déplaçant les différentes opérations de recadrage.

L'entrée de ce module est le GFDC G_{VF} au sein duquel le format de chaque donnée est spécifié. La sortie du module correspond au GFDC G_{opt} pour lequel la localisation des différentes opérations de recadrage a permis de minimiser le temps d'exécution de ces opérations. Ce processus d'optimisation nécessite de définir une stratégie de déplacement des opérations de recadrage, d'estimer le temps d'exécution de chacune d'elles et d'évaluer la précision de l'implantation pour une configuration quelconque. L'estimation du temps d'exécution du code dépend du type des registres à décalage utilisés et du type d'architecture cible.

Deux types d'architectures sont distingués au sein de cette méthodologie. Le premier type correspond aux processeurs pour lesquels le parallélisme est encodé au sein d'instructions complexes. La phase de sélection d'instructions permet de détecter ces différentes instructions complexes. Le second type d'architectures correspond aux processeurs pour lesquels le parallélisme au niveau instruction est spécifié par le regroupement d'instructions partielles au sein d'une instruction globale. La phase de sélection d'instructions génère une liste d'instructions partielles correspondant à un code vertical. Ces instructions partielles commandent une seule unité fonctionnelle et ne spécifient pas de parallélisme. Le parallélisme est obtenu en regroupant différentes instructions partielles au cours de la phase d'ordonnancement. Ce compactage du code aboutit à un code horizontal.

Déplacement des opérations de recadrage

Dans cette section, la signification des différentes opérations de recadrage et la manière dont celles-ci sont déplacées sont exposées. Les opérations de décalage à gauche permettent d'adapter le format d'une donnée x à sa dynamique. Le nombre de bits m_x utilisés pour coder la partie entière est diminué car celui est trop important par rapport à la dynamique réelle de la donnée. L'objectif de ce type de décalage est d'éliminer les bits non-utilisés au sein de la partie entière afin de pouvoir consacrer plus de bits pour la partie fractionnaire. L'exécution de cette opération de diminution du nombre de bits pour la partie entière peut être retardée. Ainsi, une opération de décalage à gauche peut être déplacée vers les racines du graphe représentant l'application afin que celle-ci soit située en aval de la position courante.

Les décalages à droite permettent d'insérer des bits supplémentaires au niveau de la partie

entière de la donnée. Ces bits sont insérés afin d'aligner la position de la virgule des entrées d'un additionneur ou pour éviter la présence d'un débordement lors des opérations suivantes. L'exécution de cette opération d'insertion de bits supplémentaires peut être avancée. Ainsi, les opérations de décalage à droite peuvent être déplacées vers les sources du graphe flot de données afin que celles-ci soient situées en amont de la position courante.

Des opérations de décalage à droite sont aussi insérées en sortie de séries d'accumulations utilisant des bits de garde. Les bits les plus significatifs du résultat de l'accumulation, présents au sein des bits de garde doivent être transférés au sein de la partie du registre ne contenant pas les bits de garde. Dans ce cas, il est nécessaire d'aligner le bit le plus significatif de la donnée sur le bit le plus significatif du registre d'accumulation sans les bits de garde. Ce processus d'alignement des bits de garde est représenté à la figure 4.14. La valeur du décalage à réaliser correspond au nombre de bits de garde réellement utilisés. Ce déplacement de la donnée au sein du registre d'accumulation est indispensable car ces bits de garde ne peuvent être utilisés que pour les opérations d'accumulation. En effet, les autres opérateurs ou les bus pour le transfert en mémoire ne permettent pas de traiter ces bits de garde. Ainsi, si ce décalage n'est pas réalisé immédiatement après la série d'accumulations, les bits de garde sont perdus et le contenu de la donnée est erroné. En conséquence, ces opérations de décalage utilisées pour aligner les bits de garde ne peuvent pas être déplacées.

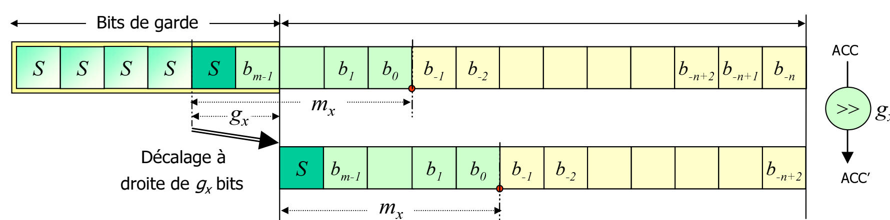


FIG. 4.14: Représentation du processus d'alignement des bits de garde après une série d'accumulations

Dans le cadre du déplacement des opérations de décalage, des règles de propagation de ces opérations au sein des opérateurs sont définies. Pour l'addition, la règle est définie afin que la position de la virgule des entrées reste alignée après la propagation de l'opération de décalage. Dans le cadre du déplacement d'une opération de décalage de la sortie d'une opération de multiplication vers ses entrées, il est nécessaire de définir le type de l'entrée sur laquelle l'opération de décalage est transférée.

Dans le cadre des systèmes linéaires, une technique analysant les opportunités de recombinaisons d'opérations de décalage et prenant en compte les directives spécifiées par l'utilisateur a été mise en œuvre. Pour les systèmes linéaires, les entrées des opérations de multiplication sont composées d'un signal et d'une constante. Ainsi, l'utilisateur de la méthodologie doit définir si les opérations doivent être transférées vers la partie signal ou vers les coefficients. Dans ce dernier cas, si le signal considéré est multiplié par plusieurs constantes, la constante la plus proche en termes de nombre d'opérations est sélectionnée.

Dans un premier temps, la méthode analyse pour chaque entrée du multiplieur, l'opportunité de recombiner cette opération de décalage avec une autre opération de décalage présente au niveau des entrées du multiplieur. Si une opportunité de recombinaison permettant d'éliminer une opération de décalage r_{e_1} est présente sur l'entrée e_1 et si l'opération doit être transmise sur l'autre entrée e_2, alors l'opération de décalage r_0 est scindée en deux afin de pouvoir éliminer r_{e_1}. L'autre partie de l'opération de décalage est transférée sur l'entrée e_2 afin de suivre les directives de l'utilisateur. Pour illustrer ce processus un exemple est fourni à la figure 4.15. Le décalage à droite r_0 de d_d bits est propagé vers les constantes de l'application. La présence d'un décalage à gauche r_{e_1} de d_g bits va aboutir après avoir scindé en deux l'opération de décalage

r_0 et propagé celles-ci, à un décalage à droite r_0' de $d_d - d_g$ bits en entrée. Dans le cas, où la valeur du décalage à gauche d_g est supérieure à la valeur du décalage à droite d_d alors, nous aboutissons à la recombinaison des deux opérations de décalage et à la présence d'un décalage à gauche de $d_g - d_d$ bits.

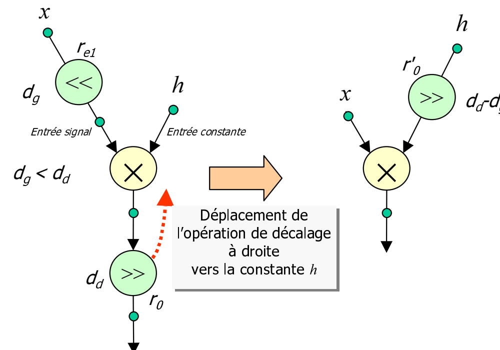


FIG. 4.15: Exemple de déplacement d'une opération de décalage

Les opérations de décalage sont déplacées au sein des blocs de base mais aussi, à travers les blocs de contrôle. Ceci nécessite de connaître le cheminement des données entre les blocs de contrôle. Ce cheminement des données est spécifié à travers les liens présents entre les sorties et les entrées des blocs de contrôle.

Évaluation de la contrainte de précision

L'évaluation de la précision est réalisée à l'aide de l'expression du RSBQ obtenue à l'étape précédente. L'obtention de cette expression est présentée dans la paragraphe 4.4.1 situé à la page 139. Le graphe flot de données et de contrôle utilisé pour déterminer l'expression du RSBQ est le GFDC de départ n'incluant pas d'opérations de décalage. En effet, le RSBQ doit être déterminé pour une configuration quelconque des opérations de décalage. Les opérations de décalage étant amenées à être déplacées au cours de la phase d'optimisation, elles ne peuvent pas être présentes au sein du GFDC utilisé pour déterminer l'expression du RSBQ. La prise en compte de ces opérations de décalage est réalisée à travers les modifications qu'elles apportent au niveau de la position de la virgule des entrées et de la sortie des opérateurs.

La largeur des données de l'application étant fixée pour cette phase, l'expression du RSBQ est fonction uniquement de la position de la virgule des entrées et de la sortie des opérateurs. Ces positions vont être modifiées lors du déplacement des opérations de décalage.

4.5.2 Architecture sans parallélisme d'instruction

Dans cette partie, les architectures pour lesquelles le parallélisme éventuel est encodé au sein des instructions sont considérées. En conséquence, le coût d'une opération de recadrage dépend des instructions utilisées pour implanter celui-ci.

Évaluation du temps d'exécution global des opérations de recadrage

Pour cette classe d'architectures, si un code possède un temps d'exécution t_1 et si les éventuels aléas de pipeline ne sont pas considérés, alors l'ajout au sein de ce code d'une série d'instructions dont le temps d'exécution est t_2, conduit à un temps d'exécution global de $t_1 + t_2$. En effet, pour les architectures considérées, les opérations réalisées en parallèle sont spécifiées au sein de la même instruction et une seule instruction est exécutée par cycle. Soient t_{r_i} le temps d'exécution de l'opération de recadrage r_i et n_{r_i} le nombre d'exécutions de cette opération. L'expression

du temps d'exécution T_{or} de l'ensemble des N_r opérations de recadrage présentes au sein de l'application est présentée à l'équation 4.36. Le nombre d'exécutions de chaque opération est calculé selon la méthode présentée dans la partie 4.4.1 à la page 141. La technique utilisée pour déterminer le temps d'exécution t_{r_i} de chaque opération de recadrage est présentée ci-dessous.

$$T_{or} = \sum_{i=0}^{N_r-1} n_{r_i}.t_{r_i} \tag{4.36}$$

Évaluation des temps d'exécution t_{r_i}

Pour les architectures considérées, différents types de registres à décalage doivent être considérés. Les registres à décalage spécialisés permettent de réaliser quelques décalages particuliers sans cycle supplémentaire. Ces décalages sont situés en entrée ou en sortie d'un opérateur et sont réalisés en parallèle à l'exécution de l'instruction commandant l'opérateur. Ces registres sont commandés de deux manières différentes. La valeur du décalage est soit spécifiée au sein de l'instruction ou au sein d'un registre de mode. Dans les deux cas, la méthodologie doit détecter si une opération de recadrage peut être réalisée avec l'un des registres à décalage spécialisés présents au sein de l'architecture. Pour les opérations de décalage commandées par des instructions, cette détection peut être réalisée lors de la phase de sélection d'instructions. En revanche, pour les opérations de décalage commandées par des registres de mode, la phase de sélection d'instructions ne permet pas toujours de détecter ce type d'opération. Ainsi, un module spécifique doit être mis en œuvre pour les opérations de décalage spécialisées commandées par un registre de mode.

Les registres à décalage en barillet fournissent une flexibilité plus importante. Ce type de registre permet de réaliser un décalage à droite ou à gauche quelconque en 1 cycle. Cependant, comme nous l'avons montré dans la paragraphe 2.1.2 situé à la page 46 le temps d'exécution réel de l'opération de recadrage dépend de la localisation au sein de l'unité de traitement de la donnée à recadrer. De plus, cette opération de recadrage modifie le motif de calcul et l'insertion de cette opération peut ralentir le traitement en ne permettant plus de sélectionner une instruction complexe pour le motif considéré. Le temps d'exécution va dépendre de la suite d'instructions utilisées pour pouvoir réaliser l'opération de recadrage et de la suite d'instructions utilisées si l'opération de recadrage n'est pas présente.

Opérations de décalage spécifiées par les instructions : Dans ce paragraphe, la technique utilisée pour mesurer le temps d'exécution t_{r_i} d'une opération de recadrage r_i implantée à l'aide d'une opération de décalage spécifiée par une instruction, est présentée. Le concept retenu pour estimer ce temps d'exécution consiste à déterminer le temps d'exécution du code t_{ar} en présence de l'opération de recadrage et le temps d'exécution du code t_{sr} sans l'opération de recadrage. Le temps d'exécution de l'opération de recadrage correspond à la différence entre les deux temps t_{ar} et t_{sr}. Le synoptique de la méthode mise en œuvre pour estimer le temps d'exécution de l'opération de recadrage est présenté à la figure 4.16. La première étape consiste à extraire le motif contenant l'opération de recadrage à traiter. Les opérations de traitement étant modélisées sous SUIF par des arbres d'expression, l'arbre contenant l'opération de recadrage est sélectionné. Ensuite, une sélection de code est réalisée sur l'arbre d'expression avec et sans l'opération de recadrage. Ce traitement est réalisé avec l'outil de sélection d'instructions OLIVE présent au sein de l'infrastructure CALIFE et conduit à la génération de deux listes d'instructions I_{ar} et I_{sr}. Le temps d'exécution associé à chaque liste est mesuré à l'aide de la technique présentée ci-dessous. L'utilisation de l'outil de sélection de code OLIVE permet de

prendre en compte les différents problèmes exposés ci-dessus. Les opérations de décalage spécialisées et spécifiées par une instruction sont détectées à l'aide de ce type d'outil, si les capacités de décalage sont spécifiées au sein du motif de l'instruction. Pour les opérations de décalage réalisées à partie d'un registre à décalage en barillet, l'outil de sélection de code intègre les éventuels transferts de données entre les registres nécessaires dans le cas où la donnée à recadrer n'est pas située dans le registre adéquat. De plus, l'outil OLIVE permet de sélectionner les instructions complexes. Ainsi, la non sélection d'une instruction complexe liée à la présence de l'opération de recadrage est détectée.

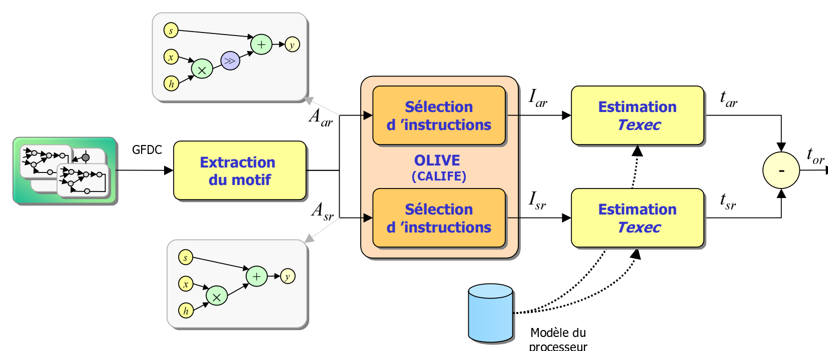


FIG. 4.16: Synoptique de la technique de détermination du temps d'exécution des opérations de recadrage dans le cas d'architectures sans parallélisme au niveau instruction

Considérons un arbre d'expression pour lequel la sélection d'instructions a conduit à une liste d'instructions I. Le temps d'exécution de ce code est obtenu à partir de la somme du temps d'exécution de chaque instruction de la liste I. Pour les architectures considérées dans cette partie, l'ensemble du parallélisme étant spécifié au sein des instructions complexes, la méthode d'estimation proposée permet de prendre en compte ce parallélisme. Cependant, les éventuels aléas de pipeline ne sont pas pris en compte. Une méthode permettant d'affiner cette estimation est proposée dans [82]. Elle permet de mieux prendre en compte les effets du pipeline en analysant pour chaque instruction les instructions adjacentes.

Opérations de décalage commandées par un registre de mode : Les fonctionnalités spécifiées par des registres de mode ne sont pas toujours bien prises en compte par les modules de sélection d'instructions. Ainsi, un modèle permettant de prendre en compte ces capacités de décalage commandées par des registres de mode a été mis en œuvre. Les capacités de décalage commandées par un registre de mode et associées à un opérateur donné sont spécifiées au niveau des instructions utilisant cet opérateur. Ainsi, le processeur est modélisé à travers son jeu d'instructions arithmétiques. Chaque instruction spécifie les capacités de décalage commandées par des registres de mode disponibles en entrée et en sortie de l'opération associée à l'instruction.

Une sélection de code est réalisée au niveau de l'arbre d'expression associé à l'opération de décalage mais sans cette opération de décalage. Pour l'opération de recadrage considérée, les instructions associées au nœud prédécesseur et au nœud successeur sont traitées. La méthode analyse si l'instruction associée au nœud prédécesseur possède les capacités de décalage en sortie pour réaliser le recadrage souhaité. De même, les capacités de décalage en entrée de l'instruction associée au nœud successeur de l'opération de décalage sont analysées.

Ces opérations de décalage sont réalisées en parallèle aux autres opérations et ainsi ne nécessitent pas de cycle supplémentaire. Cependant, il est nécessaire d'initialiser le registre de mode avant l'utilisation du registre à décalage. Ainsi, le temps d'exécution de ce type d'opération de décalage est fixé au temps nécessaire à l'initialisation du registre de mode.

Pour tirer profit de ce type de mécanisme, il est nécessaire d'analyser pour les autres instructions utilisant cet opérateur, les valeurs du décalage utilisées afin de minimiser le coût de l'initialisation du registre de mode. Cependant, ce type de registres étant peu flexible, ils sont de moins en moins présents au sein des processeurs DSP actuels. Ainsi, nous n'avons pas mis en œuvre de méthodologie spécifique permettant d'analyser la valeur des décalages des autres opérations et d'optimiser l'initialisation du registre de mode.

Stratégie d'optimisation

L'insertion d'une opération de décalage entraîne un surcoût en termes de temps d'exécution. Dans le cadre des architectures considérées, le surcoût de chaque opération de recadrage correspond au temps d'exécution de celle-ci. Le surcoût global des opérations de recadrage correspond au temps T_{or} défini à l'équation 4.36 et est égal à la somme du surcoût de chaque opération de recadrage r_i. Chaque opération de recadrage est caractérisée par sa position p_{r_i} au sein du GFDC. Cette position courante p_{r_i} appartient à l'ensemble P_{r_i} des positions possibles pour cette opération. Soient p, l'ensemble des positions courantes des opérations de recadrage de l'application et P, l'ensemble des positions possibles pour les opérations de recadrage du GFDC. Ainsi, l'objectif de cette phase d'optimisation est de minimiser le temps d'exécution global des opérations de recadrage T_{or} tant que la précision de l'implantation ($RSBQ(m)$) est supérieure à la précision minimale $RSBQ_{min}$:

$$\min_{p \in P} \Big(T_{or}(p) \Big) \quad \text{tel que} \quad RSBQ(m) \geq RSBQ_{min} \tag{4.37}$$

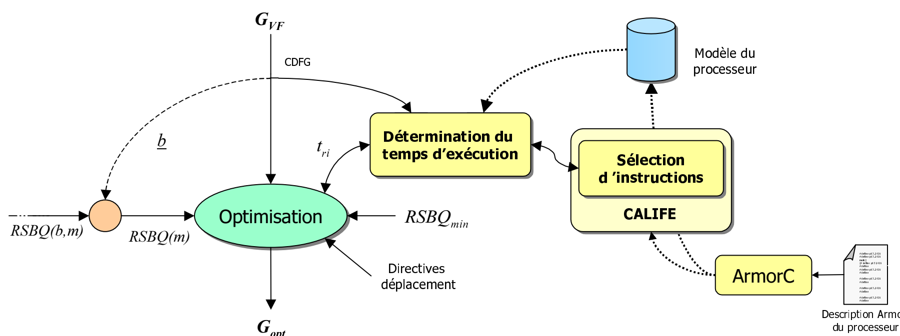


FIG. 4.17: Synoptique du processus d'optimisation de la position des opérations de recadrage

Le synoptique du processus d'optimisation est proposé à la figure 4.17. L'algorithme mis en œuvre pour résoudre ce problème d'optimisation se base sur un traitement itératif des opérations de recadrage. Chaque itération est composée d'un déplacement d'une opération de recadrage et d'une validation de celui-ci après évaluation de la précision de la spécification en virgule fixe obtenue. L'objectif étant de diminuer le temps d'exécution lié aux opérations de recadrage, les opérations les plus coûteuses sont déplacées en priorité. Ainsi, les opérations de décalage sont traitées par ordre décroissant du surcoût qu'elles entraînent. L'algorithme conçu pour minimiser le surcoût des opérations de recadrage est présenté à la figure 4.18.

Après chaque déplacement[1] d'une opération de recadrage, la précision de la solution obtenue est évaluée. Si celle-ci reste supérieure à la contrainte minimale, le déplacement est validé. Ensuite, le temps d'exécution des opérations ayant été modifiées est évalué et la liste des opérations de recadrage à traiter est mise à jour. Au cours de l'itération suivante, la nouvelle opération de recadrage dont le surcoût est maximal est déplacée. Chaque opération de recadrage r_i est caractérisée par un chemin C_i spécifiant les nœuds parcourus lors des déplacements de l'opération. Si

[1] La notion d'un déplacement d'une opération de recadrage r_i, correspond au transfert de r_i des entrées vers la sortie d'une opération ou inversement.

```
OptimisationRecadrage()
{
    L_OpsDec = ConstitutionListeOpsDec()  -- Constitution de la liste L_OpsDec des opérations de recadrage à déplacer

    Faire tant que (L_OpsDec ≠ ∅)
    {
        L_OpsDecNouv = DeplacementOpsDec(L_OpsDec(1));  -- Déplacement de l'opération de recadrage dont le sur-coût est maximal
                                                          L_OpsDecNouv regroupe les opérations de décalages issues du déplacement
        Si (L_OpsDecNouv = ∅)      -- L'opération de recadrage courante ne peut être déplacée, ainsi, elles est enlevée de la liste L_OpsDec
        {
            L_OpsDec = L_OpsDec - L_OpsDec(1) ;
        }
        Sinon
        {
            RSBQ = CalculRSBQ() ;    -- Calcul de la précision en sortie de l'application
:
            Si (RSBQ < RSBQ_min)  -- Le déplacement n'est pas valide car la précision obtenue est inférieure à la précision minimale
            {
                DéplacementVersPositionOptimale(L_OpsDec(1)) ; -- L 'opération de recadrage courante est replacée
                                                                  dans la position conduisant à un sur-coût minimal
                L_OpsDec = L_OpsDec - L_OpsDec(1) ;  -- L 'opération de recadrage courante ne sera plus déplacée ainsi, elles est enlevée
                                                      de la liste L_OpsDec
            }
            Sinon – Validation du déplacement de l'opération de recadrage courante
            {
                CalculTexec(L_OpsDecNouv) ; -- Évaluation du temps d 'exécution des nouvelles opérations de recadrage
                L_OpsDec = L_OpsDec - L_OpsDec(1) ;  -- Suppression de l'opération de recadrage traitée
                L_OpsDec = L_OpsDec + L_OpsDecNouv ; -- Insertion des nouvelles opérations
            }
        }
    }
}
```

FIG. 4.18: Algorithme d'optimisation du placement des opérations de recadrage

le déplacement d'une opération de recadrage conduit à l'extrémité de son chemin C_i ou à une précision ne respectant plus la contrainte fixée, cette opération de recadrage est repositionnée à l'emplacement du chemin C_i conduisant au surcoût minimal et cette opération n'est plus déplacée par la suite. Ce processus est réitéré tant que la liste des opérations de recadrage pouvant être déplacées n'est pas vide. Les opérations de décalage ne pouvant plus être déplacées sont retirées de la liste des opérations de décalage. Ces opérations de recadrage correspondent aux :

- opérations de décalage à droite situées en entrée du système ;
- opérations de décalage à gauche situées en sortie du système ;
- opérations de décalage dont le déplacement a conduit à l'obtention d'une précision en sortie du système insuffisante ;
- opérations de décalage nécessaires pour recadrer les bits de garde, en sortie d'une opération d'accumulation.

4.5.3 Architecture avec parallélisme d'instruction

Dans cette partie, les architectures possédant du parallélisme au niveau instruction sont considérées. Pour ces architectures permettant d'exécuter en parallèle plusieurs instructions partielles, le coût réel d'une opération de recadrage dépend de la manière dont les instructions sont ordonnancées comme nous l'avons montré dans la partie 2.2.4 à la page 71. En conséquence, une méthode couplant l'ordonnancement des instructions et le déplacement des opérations de recadrage a été définie afin de minimiser le coût de ces opérations.

Évaluation du coût des opérations de recadrage

Dans le cadre des processeurs possédant du parallélisme au niveau instruction, le coût réel d'une opération dépend de l'opportunité d'exécuter cette opération en parallèle avec les autres instructions. Ainsi, pour évaluer ce coût, une métrique η définissant la capacité d'accueil en opérations de recadrage est introduite. L'objectif de cette métrique η_{ij}, associée à l'arc (o_i, o_j), est d'évaluer le nombre maximal d'opérations de décalage pouvant être insérées entre les deux opérations consécutives o_i et o_j sans augmenter le temps d'exécution global du bloc considéré. Cette capacité d'accueil η_{ij} dépend de la mobilité des opérations o_i et o_j et du degré d'utilisation des unités fonctionnelles pouvant réaliser l'opération de décalage.

Considérons une opération de décalage r_k dont le temps d'exécution est t_k. Pour pouvoir insérer cette opération r_k entre les opérations o_i et o_j, il faut que la différence entre la date $d_{d,j}$ de début d'exécution de l'opération o_j et la date $d_{f,i}$ de fin d'exécution de l'opération o_i soit supérieure ou égale au temps d'exécution t_k et qu'une unité fonctionnelle permettant d'exécuter r_k soit disponible dans l'intervalle de temps $[d_{f,i}, d_{d,j}]$. Pour obtenir la valeur maximale de la capacité d'accueil η_{ij}, il faut se placer dans le cas le plus favorable. Pour cela, l'opération o_i doit être exécutée le plus tôt possible et l'opération o_j doit être exécutée le plus tard possible afin de maximiser la différence entre les dates $d_{d,j}$ et $d_{f,i}$. De plus, le maximum d'unités fonctionnelles permettant d'exécuter le recadrage doit être libéré pendant l'intervalle de temps où l'opération de décalage peut potentiellement être exécutée.

L'ordonnancement par liste permet d'exécuter les opérations au plus tôt en tenant compte des contraintes de dépendances entre les opérations et des contraintes de ressources au niveau des unités fonctionnelles. Cet algorithme conduit à un temps d'exécution du bloc ordonnancé b_l égal à t_{b_l}. Pour chaque opération, une date d^{min} d'exécution au plus tôt est obtenue à l'aide de cet ordonnancement. L'ordonnancement par liste en sens inverse permet d'exécuter les opérations au plus tard en fonction des contraintes de dépendances et de ressources. Cet ordonnancement procède par décroissance du pas de contrôle en débutant par les racines du graphe pour aboutir aux sources de celui-ci. Dans ce cas, une opération est prête à être ordonnancée si tous ses successeurs ont été ordonnancés. Cet ordonnancement aboutit au même temps d'exécution t_{b_l}, mais dans ce cas les instructions sont exécutées au plus tard. Ainsi, une date d^{max} d'exécution au plus tard est obtenue à l'aide de cet ordonnancement en sens inverse.

Soit $\varepsilon_{ops,m}$ l'ensemble des ressources permettant d'exécuter les opérations de type m. Soit $z_u(d_1, d_2)$ le paramètre de type booléen indiquant si l'unité fonctionnelle u est libre dans l'intervalle de temps $[d_1, d_2]$. Si cette unité fonctionnelle est utilisée pendant une partie de l'intervalle $[d_1, d_2]$, la valeur de ce booléen est nulle. Soit $Ord(o_i, [d_1, d_2])$, la fonction booléenne indiquant si l'opération o_i réalisant la fonctionnalité γ_{o_i} peut être ordonnancée pendant l'intervalle de temps $[d_1, d_2]$. Une opération o_i dont le temps d'exécution est t_i peut être ordonnancée dans l'intervalle de temps $[d_1, d_2]$, s'il existe un intervalle de temps $[t, t + t_i]$ appartenant à $[d_1, d_2]$ pour lequel une unité fonctionnelle pouvant réaliser la fonctionnalité γ_{o_i} de l'opération o_i est disponible :

$$Ord(o_i, [d_1, d_2]) = 1 \quad \text{si} \quad \sum_{d=d_1}^{d_2 - t_i} \sum_{l} z_l(d, d + t_i) > 0 \qquad \forall\, l \in \varepsilon_{ops,\gamma_{o_i}} \tag{4.38}$$

Pour déterminer la métrique η_{ij}, il est nécessaire de se placer dans le cas le plus favorable afin de libérer le maximum d'unités fonctionnelles dans l'intervalle de temps $[d_{f,i}^{min}, d_{d,j}^{max}]$. Considérons, les résultats de l'ordonnancement dans le sens direct, pour libérer le maximum d'unités fonctionnelles dans l'intervalle de temps $[d_{f,i}^{min}, d_{d,j}^{max}]$, il faut ordonnancer au plus tôt les opérations dont la date de fin d'exécution est inférieure ou égale à $d_{f,i}^{min}$ et ordonnancer au plus tard les opérations dont la date de début d'exécution est supérieure ou égale à $d_{f,i}^{min}$. Ainsi, pour le calcul de la métrique les dates d_p associées à l'opération p sont fixées soit à celles obtenues dans le cas de l'ordonnancement en sens direct (d_p^{min}) ou soit à celles obtenues dans le cas de l'ordonnancement en sens inverse (d_p^{max}), de la manière suivante :

$$\begin{cases} \text{si } d_{f,p}^{min} \leq d_{f,i}^{min} & \text{alors} \quad d_p = d_p^{min} \\ \\ \text{si } d_{d,p}^{min} \geq d_{f,i}^{min} & \text{alors} \quad d_p = d_p^{max} \end{cases} \tag{4.39}$$

La métrique η_{ij} dénombre le nombre d'opérations de décalage r_k pouvant être ordonnancées dans l'intervalle de temps $[d_{f,i}^{min}, d_{d,j}^{max}]$, dans le cas le plus favorable. Son expression est présentée

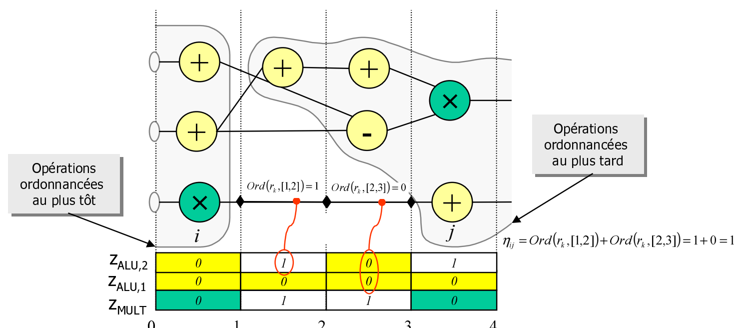


FIG. 4.19: Exemple de calcul de la métrique η_{ij}

à l'équation 4.40 et un exemple de calcul de celle-ci est proposé à la figure 4.19.

$$\eta_{ij} = \sum_{t=d_{f,i}^{min}}^{d_{d,j}^{max}-t_k} Ord(r_k,[t,t+t_k]) \tag{4.40}$$

Si la mobilité de l'opération o_j est nulle alors, les dates $d_{f,i}^{min}$ et $d_{d,j}^{max}$ sont identiques et la valeur de η_{ij} est nulle. Dans ce cas, l'insertion d'une opération de décalage entre ces deux opérations se traduit par l'augmentation du temps d'exécution de l'opération de décalage. Ainsi, pour un GFD quelconque, il existe au moins un chemin *critique* dont le nœud terminal est la dernière opération ordonnancée et pour lequel l'ensemble des arcs composants ce chemin possède un paramètre η égal 0.

Pour chaque opération de décalage, la métrique $c_{k,ij}$ correspondant au coût probable de l'opération de décalage r_k, si celle-ci est placée entre les opérations o_i et o_j, est déterminée. L'expression de cette métrique est proposée à l'équation 4.41. Si cette métrique est égale à 1, alors l'opération de recadrage ne peut pas être insérée sans augmenter le temps d'exécution du bloc.

$$c_{k,ij} = \frac{1}{1+\eta_{ij}} \tag{4.41}$$

Stratégie d'optimisation

L'objectif de la première étape du processus d'optimisation est de définir l'ordre d'évaluation des blocs de base. Ainsi, un surcoût $C_{b_i,r}$ lié à la présence des opérations de décalage est déterminé pour chaque bloc de base b_i. Soient $t_{b_i,ar}$ et $t_{b_i,sr}$, les temps d'exécution du bloc de base b_i respectivement avec et sans la présence des opérations de décalage. Soit n_i, le nombre d'exécutions du bloc de base n_i. Le surcoût $C_{b_i,r}$, lié aux opérations de décalage au sein du bloc b_i est égal à :

$$C_{b_i,r} = n_i(t_{b_i,ar} - t_{b_i,sr}) \tag{4.42}$$

Les blocs de base sont traités indépendamment en donnant la priorité à ceux dont le surcoût est le plus élevé afin de diminuer rapidement et de manière conséquente le temps d'exécution des opérations de recadrage. Ainsi, les blocs de base sont traités de manière itérative par ordre décroissant de leur surcoût.

Optimisation au niveau du bloc de base b_i : L'objectif du traitement de chaque bloc b_i est de rechercher la position des opérations de recadrage permettant d'obtenir un ordonnancement conduisant à un temps d'exécution équivalent au temps d'exécution $t_{b_i,sr}$ obtenu sans

opération de recadrage. Pour aboutir à cet objectif de temps d'exécution, pour chaque opération de recadrage, deux alternatives de placement de l'opération sont possibles. L'opération est soit placée au sein du bloc de base ou soit transférée à l'extérieur si aucune position permettant de respecter la contrainte de temps ne peut être trouvée. Cependant, ce transfert sera validé uniquement si le déplacement de l'opération de recadrage vers un autre bloc permet de réduire le surcoût global des instructions de recadrage.

Le processus d'optimisation du placement des opérations de recadrage au sein du GFD représentant le bloc de base débute avec la spécification virgule fixe d'entrée pour laquelle les opérations de décalage ont été positionnées afin d'obtenir une précision maximale. Dans un premier temps, le temps d'exécution de l'ensemble des opérations de décalage est fixé à une valeur nulle. Ainsi, dans ce cas, aucune opération de recadrage n'est prise en compte lors de l'ordonnancement du bloc. Ensuite, lors du traitement d'une opération de décalage son temps d'exécution est fixé à sa valeur réelle et l'opération est prise en compte lors de l'ordonnancement. Cette technique permet d'insérer progressivement les opérations de recadrage et d'obtenir une spécification toujours correcte d'un point de vue virgule fixe. Les opérations de décalage sont prises en compte progressivement au sein du processus d'ordonnancement par ordre décroissant de leur surcoût afin de traiter et de positionner en priorité les opérations dont le surcoût probable est élevé.

La minimisation du temps d'exécution étant réalisée sous contrainte de précision, après chaque déplacement d'une opération de recadrage la précision de la nouvelle spécification en virgule fixe est évaluée afin de vérifier si celle-ci respecte la contrainte fixée.

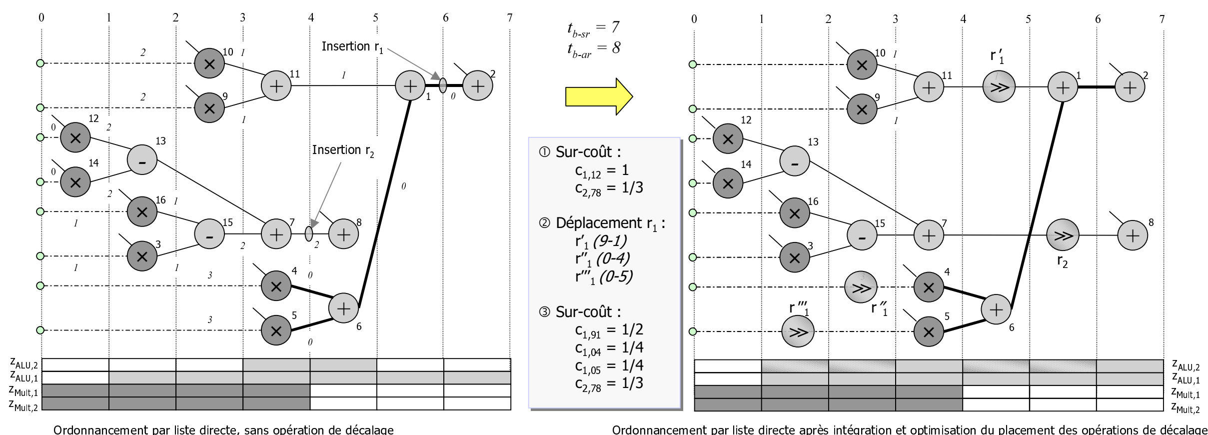


FIG. 4.20: Exemple d'ordonnancement intégrant les opérations de recadrage pour un filtre FIR complexe

L'optimisation du placement des opérations de recadrage est un processus itératif dont chaque itération est composée d'une phase de calcul du surcoût des opérations de recadrage, de déplacement de certaines opérations puis d'une phase d'ordonnancement du bloc de base. Pour illustrer la technique mise en œuvre, un exemple correspondant à un filtre FIR complexe est fourni à la figure 4.20. Le GFD spécifié représente le cœur de la boucle et permet de traiter en parallèle deux cellules du filtre. L'architecture considérée est composée de 4 unités fonctionnelles dédiées aux traitements des données. Ces unités correspondent à deux multiplieurs et deux unités arithmétiques et logiques permettant de réaliser les additions et soustractions et les opérations de décalage.

Dans un premier temps, les opérations de décalage dont le surcoût est égal à 1 sont déplacées tant que le surcoût de l'opération après déplacement reste égal à 1 et que la précision obtenue est suffisante. Ainsi, les opérations de recadrage devant être insérées au sein du chemin critique sont

positionnées avant le nœud initial de ces différents chemins. Si le déplacement d'une opération de recadrage ne permet plus de respecter la précision souhaitée, cette opération de recadrage est replacée à sa position initiale. Dans l'exemple traité à la figure 4.20, l'opération de décalage r_1 possède un surcoût unitaire. Ainsi, celle-ci est déplacée en suivant les règles de propagation propres à chaque type d'opération. Le GFD est composé de 2 chemins critiques $(4, 6, 1, 2)$ et $(5, 6, 1, 2)$. Le déplacement de r_1 vers les extrémités de ces deux chemins se traduit par la présence de trois opérations de recadrage r'_1, r''_1 et r'''_1, dont les surcoûts sont égaux respectivement à 1/2, 1/4 et 1/4.

Lorsque l'ensemble des déplacements possibles des opérations de recadrage ayant un surcoût unitaire est réalisé, l'intégration des opérations de recadrage au sein du processus d'ordonnancement est mise en œuvre. Après cette phase de déplacement, les différentes opérations de recadrage peuvent être potentiellement ordonnancées sans augmenter le temps $t_{b_i,sr}$. Les opérations de recadrage sont classées par ordre décroissant de leur surcoût. L'opération de recadrage r_{k_0} dont le surcoût est maximal est traitée. Pour cela, son temps d'exécution est fixé à sa valeur réelle et le bloc de base est ordonnancé. Cependant, il n'est pas nécessaire de réaliser cet ordonnancement sur l'ensemble du bloc de base. En effet, ce nouvel ordonnancement ne va pas modifier les dates d'exécution des opérations qui débutaient avant la fin de l'exécution de l'opération o_i correspondant au prédécesseur de r_{k_0}. Ainsi, le nouvel ordonnancement peut débuter à la date $d_{d,i}$. Une augmentation du temps d'exécution du bloc après ordonnancement par rapport au temps $t_{b_i,sr}$ signifie que les surcoûts ne reflètent plus la réalité. Dans ce cas, le processus de calcul des surcoûts, de déplacement d'opérations de décalage et d'ordonnancement est réitéré.

Dans le cadre de l'exemple considéré, l'opération r'_1 est traitée en premier et cette opération est ordonnancée au $5^{ème}$ cycle. En conséquence, l'opération o_8 est retardée d'un cycle. Ensuite, l'opération de recadrage r_2 est traitée. Celle-ci est ordonnancée au $6^{ème}$ cycle et retarde l'opération o_8 d'un cycle. Les deux autres opérations r''_1 et r'''_1 sont insérées au sein du GFD sans nécessiter la modification de la date d'exécution des autres opérations.

4.5.4 Expérimentations

Processeurs sans parallélisme au niveau instruction

Implantation d'un *Rake Receiver* au sein du TMS320C54x : L'application testée au sein de cette expérimentation, permet d'implanter une technique de réception utilisant la diversité des trajets dans le cadre des transmissions à étalement de spectre par séquence directe. Cette application est présentée en détail dans la partie 5.2.2 et correspond à celle utilisée au niveau du mobile. Le synoptique de celle-ci est proposé aux figures 5.4 et 5.7. Les différents traitements réalisés sont résumés ci-dessous afin d'analyser les résultats obtenus au niveau de l'optimisation des opérations de recadrage. Les différents blocs de base présents au sein de l'application sont représentés à la figure 4.21.b. De plus, le nombre d'exécutions de chaque bloc et le nombre d'opérations de décalage présentes au sein de chaque bloc sont détaillés. L'application est composée de plusieurs *fingers* (b_{0k}, b_{1k}) et les résultats des différents *fingers* sont combinés afin d'obtenir le signal de sortie. Chaque *finger* réalise la corrélation complexe entre le code bipolaire généré en interne et le signal d'entrée. Pour l'application considérée, la longueur du code est de 4 éléments. Ensuite, le bloc de combinaison (b_2) modifie l'argument des sorties complexes des *fingers* avant de les sommer.

Les différentes solutions possibles de déplacement des opérations de décalage ont été testées afin d'analyser l'évolution de la précision en fonction du temps d'exécution des opérations de recadrage. Le modèle d'architecture utilisé au sein de cette expérimentation correspond à celui du DSP TMS320C54x. Les courbes représentant l'évolution au cours du processus d'optimisation

du RSBQ présent au niveau de la partie réelle de la sortie, en fonction du temps d'exécution des opérations de décalage T_{or} sont présentées à la figure 4.21.a.

La solution utilisant les bits de garde conduit à un temps d'exécution des opérations de recadrage de 9 cycles et un RSBQ de 56.5 dB. Ce temps est lié aux opérations de recadrage présentes en sortie des accumulations et mises en œuvre afin de recadrer les bits de garde. Ainsi, ces opérations de décalage ne peuvent pas être déplacées. Dans le cadre de cette application, elles sont situées en sortie de chaque *finger* et en sortie du bloc de combinaison.

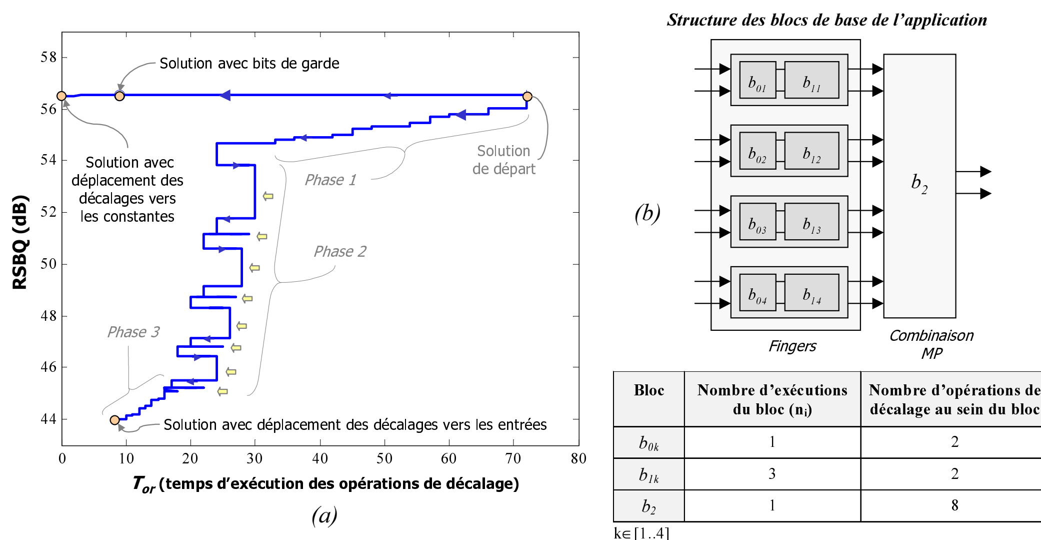
Si les bits de garde ne sont pas utilisés, la solution obtenue en entrée de cette phase d'optimisation se traduit par un RSBQ de 56.5 dB et un temps d'exécution T_{or} important et égal à 72 cycles. La solution basée sur le déplacement des opérations de décalage vers les constantes de l'application permet d'éliminer toutes les opérations de décalage. Ceci se traduit par une dégradation de la précision en sortie du système négligeable. Les opérations de décalage sont transférées vers les coefficients $\widehat{\alpha}_i^*$ et vers les codes. La modification du format des coefficients $\widehat{\alpha}_i^*$ est peu importante ainsi, celle-ci entraîne une dégradation de la précision en sortie très faible. Étant donné que les codes utilisés pour la corrélation sont composés des valeurs ± 1, une modification du format des constantes C représentant le code ne modifie pas la valeur de celles-ci ($\Delta C = 0$). En conséquence, la précision en sortie de l'application n'est pas altérée par la modification du format des codes.

La solution basée sur le déplacement des opérations de décalage vers les entrées se traduit par un RSBQ de 43.9 dB et un temps d'exécution T_{or} de 8 cycles. Ce temps est lié aux opérations de recadrage présentes sur chaque entrée. L'évolution de la précision en fonction du temps d'exécution T_{or} permet de suivre le parcours des différentes opérations de décalage. Trois phases distinctes peuvent être identifiées sur le graphique présenté à la figure 4.21.a.

La première phase correspond au traitement des nœuds présents au sein des blocs de base b_{1i}. En effet, ces opérations de recadrage possèdent le surcoût maximal. Le déplacement des opérations de recadrage se traduit par une diminution importante du temps T_{or} (39 cycles) et une dégradation de la précision de l'implantation relativement faible (1.7 dB). Ensuite, la seconde phase correspond au déplacement des opérations de décalage présentes au sein du bloc b_2. La dégradation du RSBQ est élevée, car ces opérations de décalage situées initialement en sortie du système sont déplacées jusqu'aux entrées. Le déplacement de ces 8 opérations de décalage situées dans le bloc b_2 nécessite de les transférer par les blocs b_{1i}. Ainsi, une augmentation du temps T_{or} lors du passage des opérations de décalage au sein des blocs b_{1i} peur être observée sur la courbe d'évolution du RSBQ en fonction de T_{or}. Ces augmentations sont repérées sur la courbe par le sigle $\Leftarrow$. La troisième phase correspond au déplacement des opérations de recadrage présentes dans les blocs b_{0i} situés en entrée de l'application.

Les opérations de recadrage sont ordonnées en fonction du surcoût qu'elles entraînent. Les résultats présentés à la figure 4.21 montrent qu'il serait judicieux d'inverser les phases 2 et 3, car la dégradation du RSBQ est plus importante au niveau de la phase 2. En effet, ces deux types de recadrage possèdent le même surcoût, mais les recadrages présents au sein du bloc b_2 ont une influence plus importante sur le RSBQ. Par exemple, si la contrainte de précision est fixée à 50 dB, le temps d'exécution T_{or} obtenu est égal à 22 cycles. Si les phases 2 et 3 sont inversées, alors le temps d'exécution T_{or} pour la même contrainte sera égal à 16 cycles. En conséquence, pour les opérations ayant le même surcoût, la dégradation potentielle liée à leur déplacement doit être prise en compte. Pour ordonner les opérations de recadrage en fonction de leur influence sur la précision, la technique présentée dans la partie 4.4.1 à la page 145 peut être mise en œuvre.

Pour cette application, la solution optimale est obtenue en déplaçant les opérations de décalage vers les constantes. Cette solution permet d'éliminer toutes les opérations de décalage



Bloc	Nombre d'exécutions du bloc (n_i)	Nombre d'opérations de décalage au sein du bloc
b_{0k}	1	2
b_{1k}	3	2
b_2	1	8

k∈[1..4]

FIG. 4.21: Évolution de la précision en fonction du coût des recadrages

et d'obtenir une dégradation de la précision en sortie quasiment nulle. Ceci est lié au fait que l'application utilise des constantes particulières pour lesquelles une modification du format ne modifie pas leur valeur. Cependant, pour d'autres types d'application, la solution basée sur le déplacement des opérations de décalage vers les constantes entraînera une dégradation de la précision en sortie du système. Les autres solutions permettent d'obtenir un temps d'exécution global des opérations de recadrage faible, mais pour la solution basée sur le déplacement vers les entrées, la dégradation de la précision est importante.

Implantation d'applications au sein des DSP TMS320C54x et TMS320C50 : Dans cette partie, les différentes solutions de déplacement des opérations de décalage ont été testées dans le cadre de différentes applications et de deux modèles d'architectures différents. Les processeurs TMS320C54x et TMS320C50 sont basés sur une architecture traditionnelle de type MAC, mais les architectures de ces processeurs et leurs capacités de recadrage sont différentes. Le DSP C54x possède 8 bits de garde au niveau de l'accumulateur et un registre en barillet. Le DSP C50 ne possède pas de bit de garde et ses capacités de décalage sont présentées dans la partie 2.1.2 à la page 46. Ce processeur possède un registre à décalage spécialisé en entrée et en sortie de l'accumulateur. Les résultats du processus d'optimisation sont présentés dans le tableau 4.2. La solution dénommée *initiale* correspond à la spécification virgule fixe obtenue avant la phase de déplacement des opérations de recadrage. Pour chaque solution, le temps d'exécution global des opérations de décalage T_{or} est reporté. Pour les solutions de déplacement vers les constantes et vers les entrées, la dégradation de la précision Δ_p a été évaluée. Cette dégradation correspond à la différence des RSBQ exprimés en dB, obtenus avant et après la phase d'optimisation.

Dans le cadre du DSP C54x, la présence de bits de garde au sein de l'accumulateur permet d'obtenir une spécification en virgule fixe contenant un nombre d'opérations de recadrage limité. Les opérations de décalage nécessaires pour recadrer les bits de garde ne pouvant être déplacées, la phase d'optimisation ne permet pas de diminuer le coût des opérations de recadrage. En conséquence, la précision obtenue correspond à la précision optimale et initiale dans le cas des structures non-récursives. Pour les structures récursives telles que le filtre IIR, la présence de bits de garde n'est pas suffisante ainsi, une opération de recadrage est nécessaire pour adapter le format des données entre les sorties des parties récursives et non-récursives.

	TMS320C54x					TMS320C50				
Applications	*Initiale*	*Constantes*		*Entrées*		*Initiale*	*Constantes*		*Entrées*	
	T_{or}	Δ_p	T_{or}	Δ_p	T_{or}	T_{or}	Δ_p	T_{or}	Δ_p	T_{or}
FIR 32	1	0	1	0	1	128	-1	0	-4,5	0
IIR 2	2	-0.5	1	-8.6	2	7	-0.5	0	-8.6	0
CC 32	1	0	1	0	1	160	0	0	-18,26	0
CC 128	1	0	1	0	1	896	0	0	-29,9	0
DW 32 - MS	1	0	1	0	1	128	-2,3	0	-12,4	0
RR 4 - MS	9	0	9	0	9	80	-0.04	0	-12,6	0
DW 32 - BS	1	0	1	0	1	128	0	0	-9,5	0
RR 8 - BS	5	0	5	0	5	50	-0.02	0	-2,5	0

CC α : Corrélateur complexe avec un code binaire de longueur α
DW β : Décodage des données (WCDMA), facteur d'étalement de β, MS : terminal mobile (fig : 5.7), BS : station de base (fig : 5.6)
RR β : Récepteur *Rake Receiver* composé de 5 *finger* avec un facteur d'étalement β (fig : 5.4), MS : terminal mobile, BS : station de base

TAB. 4.2: Optimisation du placement des opérations de recadrage dans le cadre de l'implantation de différentes applications au sein des DSP TMS320C54x et TMS320C50

Dans le cadre du DSP C50, l'absence de bit de garde se traduit par une spécification initiale pour laquelle le temps d'exécution des opérations de décalage est très élevé. En effet, ces différentes opérations de décalage sont insérées afin de coder les données au plus précis. De plus, les capacités de recadrage de ce processeur sont limitées. La majorité des opérations de recadrage est implantée en utilisant une instruction réalisant un décalage par cycle. Ainsi, le temps d'exécution de l'opération de décalage est proportionnel à la valeur du décalage. Les résultats obtenus après la phase d'optimisation montrent la réduction importante du temps d'exécution des opérations de décalage et l'importance de cette phase pour obtenir un code efficace.

Le déplacement des opérations de décalage vers les constantes permet d'éliminer les différentes opérations de recadrage. La dégradation de la précision peut être nulle dans certaines applications telles que les applications pour la corrélation complexe ou le décodage des données au sein du WCDMA. Ce type d'application manipule des constantes particulières égales à ± 1 dont la valeur est insensible à un changement de format. Cependant, pour l'application de décodage des données dans le cadre de la réception WCDMA au sein des terminaux mobiles, le déplacement des opérations de décalage vers les constantes entraîne une dégradation de la précision liée à la recombinaison de certaines opérations de recadrage. Pour les applications telles que les filtres linéaires, la dégradation liée au déplacement des recadrages vers les constantes n'est pas nulle et dépend de l'application. Le déplacement des opérations de recadrage vers les entrées permet d'obtenir pour le C50, un temps d'exécution T_{or} nul mais ce type de recadrage entraîne une dégradation de la précision relativement importante.

Processeurs avec parallélisme au niveau instruction

Dans cette section, quelques résultats obtenus avec la méthode d'optimisation du placement des opérations de recadrage lors de la phase d'ordonnancement, sont présentés. L'outil permettant d'implanter cette méthodologie n'étant pas développé, ces résultats ont été obtenus manuellement. L'application considérée est un *Rake Receiver* implanté au sein des stations de base. Cette application est détaillée dans la partie 5.2.2 à la page 172 et les synoptiques sont présentés aux figures 5.6 et 5.4. Les différents blocs de base présents au sein de l'application

sont identiques à ceux présentés à la figure 4.21.

Pour ces expérimentations, deux architectures VLIW ont été testées. Elles sont composées respectivement de trois et de quatre unités fonctionnelles dédiées exclusivement aux opérations de traitement du signal. La première architecture possède deux multiplieurs et une UAL et la seconde est composée de deux multiplieurs et de deux UAL. Les taux d'occupation des UAL Γ_{UAL} et des multiplieurs Γ_{MULT} ont été mesurés avant et après la phase d'optimisation. La valeur du taux d'occupation fournie avant la phase d'optimisation correspond à celle obtenue sans les opérations de recadrage. De plus, le coût des opérations de recadrage $C_{b_i,r}$ a été déterminé pour chaque bloc de base avant et après la phase d'optimisation. Le paramètre P_r associé à chaque bloc de base, définit le taux d'opérations de recadrage restées au sein de ce bloc après la phase d'optimisation.

Pour la première architecture, la présence d'une seule UAL ne permet pas de positionner les opérations de recadrage au sein des blocs de base sans augmenter les temps d'exécution de ceux-ci. En conséquence, au cours de la phase d'optimisation, les différentes opérations sont transférées vers les entrées du système et conduisent à un surcoût global en termes de temps d'exécution de 12 cycles.

Pour la seconde architecture, la présence de deux UAL, permet d'aboutir à un surcoût lié aux opérations de recadrage nul. Pour les blocs b_0 et b_1, l'ensemble des opérations de recadrage a été placé au sein du bloc de base d'origine sans augmenter le temps d'exécution de celui-ci. Pour le bloc b_2, uniquement la moitié des opérations de recadrage a pu être placée au sein du bloc. Les autres opérations sont transférées au sein du bloc b_2 où elles sont combinées aux autres opérations de recadrage présentes au sein de ce bloc.

Architecture	**Blocs**	**Avant Optimisation**			**Après Optimisation**			
		Γ_{UAL}	Γ_{MULT}	$C_{b_i,r}$	Γ_{UAL}	Γ_{MULT}	$C_{b_i,r}$	P_r
Archi 1	b_0	0.75	0.714	**8**	0.90	0.714	**12**	1
	b_1	0.75	0.714	**56**	0.75	0.714	**0**	0
	b_2	1	0	**4**	1	0	**0**	0
Archi 2	b_0	0.428	0.714	**4**	0.714	0.714	**0**	1
	b_1	0.428	0.714	**28**	0.714	0.714	**0**	1
	b_2	0.5	0	**1**	0.833	0	**0**	0.5

TAB. 4.3: Optimisation du placement des opérations de recadrage pour deux architectures VLIW

4.5.5 Conclusions

Dans cette partie, la méthode d'optimisation du format des données en vue de minimiser le temps d'exécution du code a été présentée. Deux types de méthode ont été proposés en fonction de l'architecture du processeur cible. Pour les processeurs ne possédant pas de parallélisme au niveau instruction, le coût d'une opération de recadrage peut être estimé avant la phase de génération de code en utilisant la technique basée sur l'évaluation du temps d'exécution de la suite d'opérations nécessaires pour implanter l'opération de recadrage. Ainsi, pour cette classe d'architectures, la phase d'optimisation du format des données est réalisée avant de débuter le processus de génération de code. Pour les processeurs possédant du parallélisme au niveau instruction, cette phase d'optimisation est réalisée au cours de la phase d'ordonnancement.

4.6 Conclusions

Dans ce chapitre, les différentes étapes du processus de conversion en virgule fixe ont été exposées. Le synoptique détaillé de cette méthodologie est présenté à la figure 4.22. Pour chaque étape, les objectifs et la méthode utilisée sont résumés. Dans la première partie, après une présentation de l'infrastructure de compilation, la représentation intermédiaire utilisée au sein du processus de conversion en virgule fixe, a été définie.

Dans la seconde partie du chapitre, la méthode utilisée pour estimer la dynamique des données et basée sur une approche analytique a été présentée. Elle permet de traiter les systèmes linéaires invariants dans le temps et les systèmes non-linéaires et non-récursifs. Pour compléter ce module d'estimation de la dynamique, les techniques basées sur la simulation peuvent être incorporées au sein de l'outil afin d'enrichir les estimations. Ceci permettrait de traiter certaines structures telles que les filtres adaptatifs. L'utilisateur pourrait alors choisir la méthode d'estimation en fonction de l'application cible et des caractéristiques des signaux d'entrée.

La méthode pour déterminer le format des différentes données de l'application a été détaillée dans la troisième et la quatrième partie. Tout d'abord, la position de la virgule est déterminée à partir de la dynamique des données et des règles de propagation de la virgule au sein des opérateurs. Ensuite, la largeur des données est définie en vue d'obtenir le format de chaque donnée. Cette étape d'optimisation de la largeur des données consiste à sélectionner pour chaque opération les instructions permettant de respecter la précision souhaitée et de minimiser le temps d'exécution. Au niveau de cette phase d'optimisation, il est possible d'intégrer le choix de la loi de quantification utilisée lors d'un changement de format. Actuellement, nous considérons que la loi de quantification utilisée correspond à la troncature. Cependant, certains DSP permettent d'utiliser une loi de quantification par arrondi. Pour intégrer ce choix de la loi de quantification, la modélisation du processeur doit être enrichie à travers la définition de la loi de quantification utilisée pour chaque instruction. Au niveau de l'outil d'évaluation de la précision, la possibilité de spécifier la loi de quantification utilisée doit être ajoutée. Ainsi, la prise en compte de la loi de quantification peut être réalisée aisément, cependant le processus d'optimisation sera ralenti. En effet, pour chaque opération le nombre d'instructions disponibles pour réaliser celle-ci sera plus important.

La dernière partie de ce chapitre est consacrée à l'optimisation du format des données en vue de minimiser le temps d'exécution du code. Deux types de méthodologie ont été mis en œuvre en fonction des capacités de parallélisme au niveau instruction offertes par le processeur. Pour les processeurs ne possédant pas de parallélisme au niveau instruction, une technique itérative de déplacement des opérations de recadrage a été mise en œuvre. Cette méthode permet d'obtenir des résultats intéressants, mais la méthode ne fournit pas toujours la solution optimale. Ainsi, des techniques d'optimisation basées par exemple sur les algorithmes génétiques peuvent être envisagées afin de trouver la position optimale des opérations de décalage. En contrepartie, le temps d'optimisation sera plus élevé. Pour les processeurs possédant du parallélisme au niveau instruction, une méthode de déplacement des opérations de décalage intégrée à la phase d'ordonnancement des instructions a été proposée. L'implantation de cette méthode au sein de l'infrastructure CALIFE nécessite le développement d'une interface entre notre représentation intermédiaire et la représentation utilisée au sein de CALIFE pour réaliser la phase d'ordonnancement. Les différents résultats obtenus sur quelques exemples montrent la nécessité d'optimiser le placement des opérations de recadrage afin d'obtenir un code efficace en termes de temps d'exécution et plus particulièrement si les capacités de décalage du processeur sont limitées.

La méthodologie développée au sein de ce travail de recherche s'adresse aux processeurs pro-

grammables. Celle-ci peut être utilisée dans le cadre des DSP, des cœurs de DSP ou des ASIP mais aussi pour les processeurs à usage général possédant des capacités de calcul pour le traitement du signal. Plus généralement, les architectures pour lesquelles la largeur des opérateurs est fixe peuvent bénéficier de cette méthodologie. En particulier, cette méthode peut être utilisée pour les architectures reconfigurables au niveau fonctionnel telles que l'architecture DART [29, 30] développée au sein du laboratoire. Pour ce type d'architecture, il est nécessaire d'adapter la phase d'optimisation du placement des données. En effet, l'interconnexion entre les unités fonctionnelles est définie afin d'optimiser le chemin de données par rapport aux motifs de calcul présents au niveau de l'application. Ceci conduit à une implantation spatiale de l'application au lieu d'une implantation temporelle dans le cas des processeurs programmables.

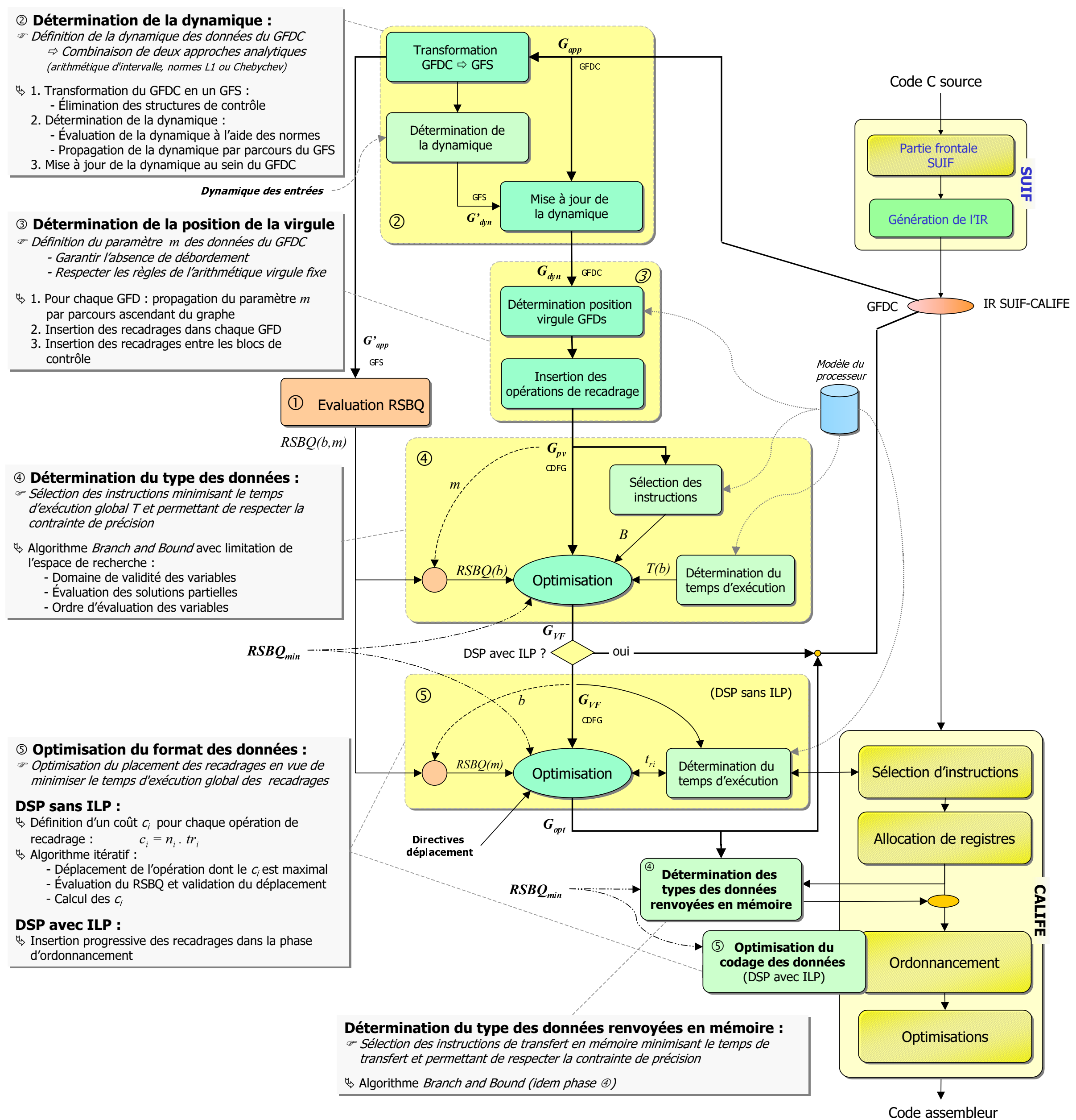


FIG. 4.22: Synoptique détaillé de la méthodologie de codage en virgule fixe

Chapitre 5

Applications

5.1 Introduction

Les applications retenues pour tester notre méthodologie appartiennent au domaine des télécommunications et plus particulièrement aux systèmes de radiocommunications de troisième génération (systèmes 3G). L'objectif de cette nouvelle génération est d'obtenir un système universel (UMTS : Universal Mobile Telecommunications System) permettant de transmettre des données à un débit élevé dans le cadre de services multimédia et liés à l'internet. De plus, les techniques de transmission utilisées doivent permettre d'augmenter la capacité des cellules en nombre d'utilisateurs. Ainsi, des techniques de réception avancées telles que les antennes adaptatives ou la détection multi-utilisateurs pourront être utilisées. Ces différents objectifs se traduisent par la mise en œuvre de nombreux algorithmes de TNS issus de différents domaines tels que la compression audio et vidéo, le traitement de la parole, les communications numériques.

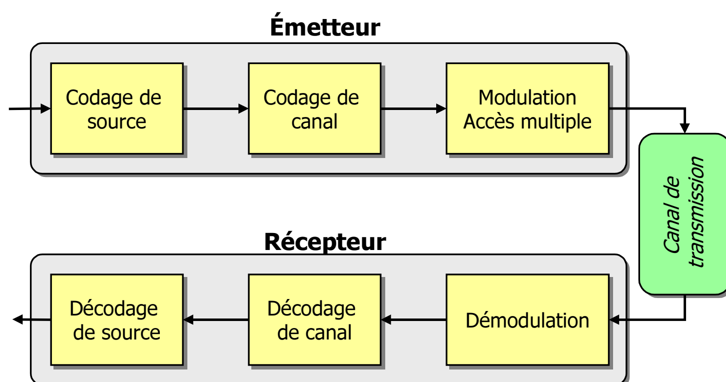


Fig. 5.1: Synoptique d'un chaîne de transmission numérique

Le synoptique d'une chaîne de transmission en communications numériques est représenté à la figure 5.1. L'objectif des systèmes de radiocommunications de troisième génération est de transmettre différents types d'information tels que la parole, l'audio, la vidéo ou les données. Ainsi, différents algorithmes de codage de source sont mis en œuvre en fonction du type de source d'information. Le but de ce codage de source est de diminuer le volume d'information à transmettre à travers l'élimination des informations redondantes.

L'objectif du codage de canal est de rajouter de la redondance au sein des données à transmettre afin de se protéger contre les erreurs de transmission. Cette redondance est utilisée au niveau de l'algorithme de décodage de canal afin de détecter et de corriger les erreurs. Dans le

cadre de l'UMTS, les techniques basées sur les codes convolutifs sont utilisées. En fonction du type de codage utilisé, l'algorithme de Viterbi ou les turbo-décodeurs sont mis en œuvre.

Le dernier traitement réalisé permet d'adapter l'information numérique au support de transmission. Pour cela, les données à transmettre sont modulées et des techniques d'accès multiples sont mises en œuvre afin de partager le support de transmission avec les autres utilisateurs. Dans le cadre des normes pour l'UMTS en vigueur en Europe et en Asie, l'accès multiple est basé sur les techniques d'étalement de spectre à large bande par séquence directe (WCDMA pour Wideband Code Division Multiple Access). Les applications utilisées pour tester notre méthodologie correspondent à la partie décodage des données dans le cadre de l'utilisation du WCDMA.

Au niveau des communications mobiles cellulaires, le canal de transmission entre l'émetteur et le récepteur possède des propriétés particulières. Le signal reçu au niveau du récepteur est composé de différentes répliques du signal émis, liées à la présence de multiples chemins de propagation de l'onde radioélectrique entre l'émetteur et le récepteur. Chaque trajet est caractérisé par un retard et une amplitude complexe. De plus, le déplacement du mobile entraîne une variation de la puissance reçue. Ce phénomène dénommé *fading* produit des évanouissements du signal lorsque la combinaison des multi-trajets est destructrice.

Contraintes architecturales

Les systèmes de radiocommunications cellulaires sont composés d'un émetteur-récepteur mobile et d'un émetteur-récepteur fixe dénommé station de base. Cette dernière est affectée à la cellule et est reliée au réseau de télécommunications. Les contraintes architecturales de ces deux types d'éléments sont différentes.

La définition de l'architecture dans le cadre d'une station de base est soumise à deux contraintes majeures correspondant à la capacité de calcul et à la flexibilité. L'architecture doit être flexible afin de supporter les différents standards de radiocommunications cellulaires et leurs évolutions au cours du temps. En effet, la durée de vie des infrastructures de télécommunications doit être relativement élevée. Le traitement des différents utilisateurs présents au sein de la cellule nécessite des capacités de calcul importantes. De plus, la mise en œuvre de techniques de réception avancées au niveau de la station de base requière des capacités de calcul supplémentaires.

Le mobile étant un produit grand public, son coût doit être le plus faible possible. De plus, la contrainte de mobilité nécessite une consommation d'énergie réduite. La durée de vie de ces produits étant relativement faible, les concepteurs s'orientent vers des plate-formes utilisées pour plusieurs générations de produits. De plus, la durée de vie limitée des produits nécessite des temps de mise sur le marché très réduits. Ainsi, cette plate-forme doit être assez flexible pour supporter les évolutions des normes, l'ajout de nouveaux services et le développement rapide de nouvelles applications.

5.2 Structure d'un émetteur et d'un récepteur WCDMA

5.2.1 Émetteur WCDMA

Le synoptique d'un émetteur WCDMA est présenté à la figure 5.2. Les émetteurs pour la voie montante (terminal mobile vers station de base) et pour la voie descendante (station de base vers terminal mobile) sont légèrement différents.

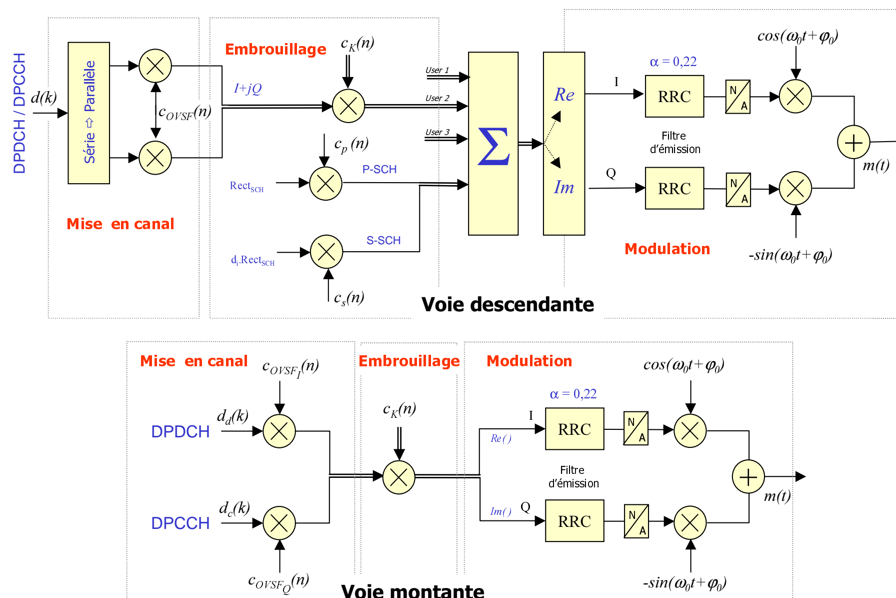


FIG. 5.2: Synoptique des émetteurs WCDMA

Les données à transmettre sont regroupées au sein de différents canaux physiques en fonction du type d'information à transmettre [33]. Ces canaux physiques sont soit communs (SCH, . . .) à toute la cellule ou soit dédiés à la communication entre le mobile et la station de base. Les canaux dédiés permettent de transporter des informations correspondant aux données (DPDCH) et celles destinées au contrôle (DPCCH). Ces dernières spécifient les informations pour le contrôle de puissance, le facteur d'étalement utilisé pour les données transmises et une séquence de bits de référence servant à l'estimation du canal au niveau du récepteur. Ces canaux sont organisés en trame de 10 ms composées chacune de 15 *slots*.

Les symboles à transmettre $d(k)$ possèdent un débit variable D_s variant de 15 $Ksymbol/s$ à 960 $Ksymbol/s$ pour la voie montante et de 7.5 $Ksymbol/s$ à 960 $Ksymbol/s$ pour la voie descendante. La première étape dénommée *mise en canal* correspond à la multiplication des données par un code c_{OVSF} orthogonal de longueur variable (*OVSF : Orthogonal Variable Spreading Factor*) [31]. L'élément correspondant à un symbole du code est dénommé *chip* et sa durée T_c est inférieure à la durée d'un symbole à transmettre T_s. Le débit du code D_c est fixe et égal à 3.84 Mbits/s. Le facteur d'étalement (SF) est défini comme le rapport entre la durée d'un symbole T_s et la durée d'un *chip* T_c. La période du code correspond à la durée d'un symbole T_s. Ainsi, le nombre de bits composant une période de ce code est égal au facteur d'étalement. Ces codes OVSF permettent d'identifier et de séparer les différents canaux. Ces codes étant orthogonaux entre eux, l'intercorrélation entre deux codes OVSF est nulle si ces codes sont synchronisés[1].

La seconde étape dénommée *embrouillage* (*scrambling*) correspond à la multiplication complexe du signal issu de la phase de mise en canal par un code pseudo-aléatoire (codes PN) [31].

[1]Deux codes sont synchronisés si le retard entre ces deux codes est nul.

Ces codes PN permettent d'identifier et de séparer les utilisateurs. Les codes PN utilisés dans le cadre du WCDMA sont les codes de Kasami. L'intercorrélation entre deux séquences PN n'est pas nulle, mais ces codes possèdent de très bonnes propriétés au niveau de leur autocorrélation. En effet, la fonction d'autocorrélation d'une séquence PN est proche d'une impulsion de Dirac. Ainsi, ces codes sont utilisés pour synchroniser les codes générés en interne (codes OVSF et code de Kasami) par rapport au signal reçu. Le couplage de ces deux types de code permet de séparer deux canaux dont les facteurs d'étalement peuvent être différents.

La dernière étape correspond à la modulation du signal support de l'information. Une modulation de phase à quatre états (QPSK) est réalisée. Pour éliminer l'interférence entre symboles au niveau de la réception, un filtre de Nyquist réalisé à l'aide d'un filtre en cosinus surélevé possédant un facteur de retombée de 0.22 est inséré au niveau des voies réelles et imaginaires. Ensuite, le signal est converti en analogique avant d'être mélangé avec les porteuses en phase et en quadrature.

5.2.2 Récepteur WCDMA

Le synoptique d'un récepteur WCDMA utilisant la diversité des trajets pour améliorer les performances de réception est présenté à la figure 5.3. Le signal reçu au niveau de l'antenne est mélangé avec les porteuses en phase et en quadrature générées en interne afin d'obtenir un signal en bande de base. Le signal $s_r(n)$, en entrée de la partie numérique, est un signal complexe issu de la numérisation des voies en phase et en quadrature. Ce signal est numérisé à une fréquence multiple de la fréquence d'un *chip* ($F_{chip} = 3.84\ MHz$). Le facteur de sur-échantillonnage N_{se} ne doit pas être trop élevé afin de limiter la complexité des calculs réalisés par la suite. Dans le cadre de notre implantation, la valeur du sur-échantillonnage a été fixée à 4.

Un contrôle automatique de gain est mis en œuvre afin de maintenir le niveau du signal en entrée des convertisseurs analogique-numérique (CAN) pour bénéficier de la dynamique complète offerte par les CAN. En effet, le phénomène de *fading* modifie significativement la puissance du signal reçu. Ainsi, la puissance du signal complexe numérisé est mesurée afin de commander les amplificateurs présents en entrée des CAN. Différentes techniques peuvent être mises en œuvre pour obtenir un signal fonction de la puissance du signal numérisé.

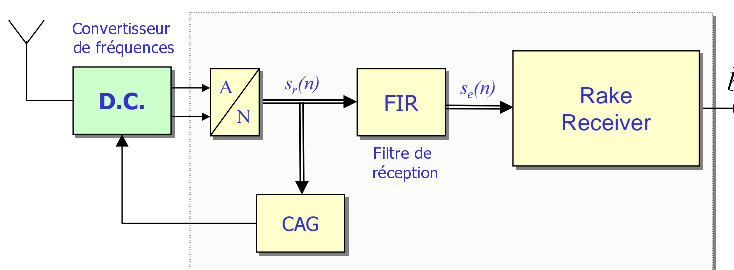


FIG. 5.3: Synoptique d'un récepteur WCDMA

Filtre de réception

Le premier traitement réalisé sur le signal $s_r(n)$ correspond au filtrage de Nyquist mis en œuvre pour éliminer l'interférence entre symboles. Ce type de filtre est implanté à l'aide d'un filtre à réponse impulsionnelle finie (FIR) afin de ne pas introduire de distorsion de phase. Les parties réelles et imaginaires du signal $s_r(n)$ sont traitées indépendamment par un filtre FIR à coefficients réels et composé de N_{FIR} cellules. La complexité des deux filtres FIR de réception est égale à :

$$\mathcal{C}_{FIR} = 2.N_{FIR}.N_{se}.F_{chip}\ (MAC/s) \tag{5.1}$$

Pour limiter la complexité du filtre FIR, il est nécessaire de limiter la réponse impulsionnelle du filtre de Nyquist. Dans [118], la dégradation du taux d'erreur binaire (Γ_r), liée à l'utilisation d'un filtre de réception de longueur limitée, est étudiée. Si un filtre de longueur limitée est utilisé, il est nécessaire d'avoir un rapport signal à bruit (RSB) supérieur en entrée du récepteur pour obtenir le même taux d'erreur binaire que celui obtenu avec un filtre idéal. Ce RSB correspond au rapport entre la puissance du signal reçu et la puissance du bruit présent au niveau du récepteur. Ce bruit est lié au bruit thermique présent au niveau de l'antenne et au bruit généré au sein de la partie analogique du récepteur. Soit $RSB_{ideal}(\Gamma_{r,0})$ et $RSB_{limité}(\Gamma_{r,0})$, les rapports signal à bruit nécessaires pour obtenir un taux d'erreur binaire égal à $\Gamma_{r,0}$, avec respectivement un filtre idéal et un filtre de longueur limitée. Soit $\Delta RSB(\Gamma_{r,0})$, le RSB supplémentaire lié à l'utilisation d'un filtre de longueur limitée, défini à l'équation 5.2. Pour un filtre composé de 64 cellules, ΔRSB est égal à $0,5\ dB$ pour un taux d'erreur binaire de 10^{-3} et $1\ dB$ pour un taux d'erreur de 10^{-6}. Pour un filtre composé de 32 cellules, ΔRSB est égal à $1\ dB$ pour un taux d'erreur de 10^{-3} et $1,6\ dB$ pour un taux d'erreur de 10^{-6}.

$$\Delta RSB(\Gamma_{r,0}) = RSB_{ideal}(\Gamma_{r,0}) - RSB_{limité}(\Gamma_{r,0}) \tag{5.2}$$

Pour limiter la complexité des calculs, nous avons retenu un filtre dont le nombre de cellules est égal à 32. En conséquence, la longueur du filtre correspond à 8 symboles. Ceci conduit pour les deux filtres de réception à une complexité égale à 983.10^6 opérations MAC par seconde. Face à la complexité élevée de ce filtre, il est nécessaire de mettre en œuvre des solutions efficaces fournissant des capacités de calcul très élevés.

Récepteur avec diversité de trajets

Dans cette partie, les traitements réalisés pour décoder les données sont présentés [38]. Dans un canal de transmission à L trajets multiples, le signal transmis par l'émetteur se réfléchit sur des obstacles tels que les bâtiments ou le relief. Ainsi, le récepteur reçoit L répliques du même signal avec des délais différents. Soit M le nombre d'utilisateurs reçus et w le bruit introduit par le canal et présent en sortie du filtre de Nyquist. L'expression du signal $s_e(n)$ en sortie du filtre de réception est la suivante :

$$s_e(n) = \sum_{m=1}^{M} \sum_{l=1}^{L} s_m(n - \tau_l) + w(n) \tag{5.3}$$

La composante $s_m(n - \tau_l)$ correspond au signal reçu de l'utilisateur m affecté d'un retard τ_l. Si les signaux arrivent avec plus d'un *chip* de retard les uns par rapport aux autres alors le système peut les séparer. Pour un signal issu d'un multi-trajet, les autres signaux sont perçus comme des interférences et sont éliminés par le gain de traitement. Un bénéfice supplémentaire est obtenu en combinant les signaux issus des différents multi-trajets après traitement de ceux-ci au sein d'un récepteur en râteau (*rake receiver*).

Supposons une diversité d'ordre L au niveau du récepteur, c'est à dire que L répliques du signal transmis sont disponibles en entrée du récepteur. Si le filtrage global émetteur - récepteur satisfait le critère de Nyquist et si les évanouissements liés au phénomène de *fading* sont non sélectifs en temps et en fréquence, alors l'expression de la sortie du filtre adapté de la $l^{ème}$ branche r_l est égale à :

$$r_l(k) = \alpha_l(k).c(k) + n_l(k) \qquad l = 1, 2, ..., L \tag{5.4}$$

Le terme $c(k)$ représente le $k^{ème}$ symbole transmis sur K et $\alpha_l(k)$ l'amplitude complexe du canal pour le trajet l. Le terme $n_l(k)$ correspond au bruit additif gaussien présent en sortie de cette branche. Le signal $r_l(k)$ évolue au rythme symbole.

Selon le critère du maximum de vraisemblance, le symbole est estimé à partir du signal $y(k)$ défini de la manière suivante [110] :

$$y(k) = \sum_{l=1}^{L} y_l(k) = \sum_{l=1}^{L} \alpha_l^*(k) r_l(k) \qquad k = 1, 2, \cdots, K \tag{5.5}$$

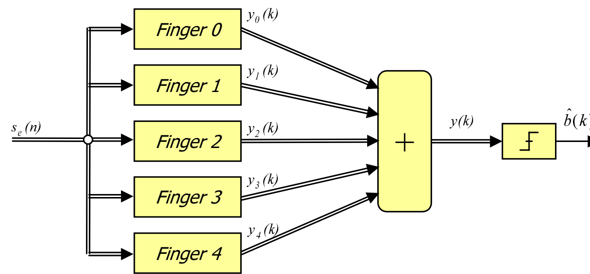


Fig. 5.4: Synoptique d'un *rake receiver*

Cette détection combine linéairement toutes les sorties des filtres adaptés à la période symbole T_s, de façon à réaliser une décision optimale sur $b(k)$. La multiplication par $\alpha_l^*(k)$ élimine la distorsion de phase présente au niveau du signal reçu pour le trajet l. Le synoptique de ce type de récepteur en râteau (*rake receiver*) est présenté à la figure 5.4. Chaque trajet l est traité par un module appelé *finger* dont le synoptique est présenté à la figure 5.5. Chaque *finger* réalise le décodage des données, l'estimation de l'amplitude complexe du canal et la synchronisation des codes générés en interne.

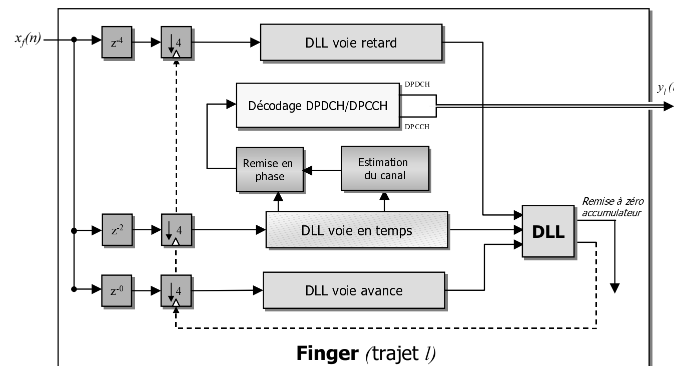


Fig. 5.5: Synoptique d'un *finger* du *rake receiver*

Décodage des données au niveau de la station de base

Dans cette partie, la technique utilisée pour décoder les données au sein d'une station de base est présentée. Le signal reçu correspond à celui de la voie montante. Les symboles correspondant aux données de l'utilisateur sont transmis sur la voie réelle et les symboles de contrôle sur la voie imaginaire. Ces deux voies sont caractérisées par des facteurs d'étalement différents.

Pour présenter le décodage des données nous supposons que le système est synchronisé et que l'amplitude complexe du canal a été estimée. Ces deux parties sont présentées dans l'annexe C.

La sortie complexe du *finger* associé à un trajet l, est obtenue à partir des traitements présentés à la figure 5.6.

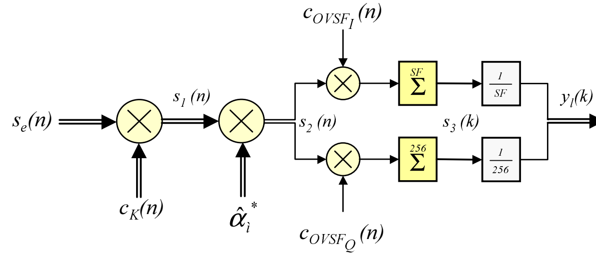


FIG. 5.6: Synoptique de la partie décodage des données au niveau de la station de base

Dans un premier temps, le signal est désembrouillé en réalisant la multiplication complexe entre le signal d'entrée $s_e(n)$ et le conjugué du code de Kasami c_K^*, utilisé à l'émission. Sachant que les codes $c_K^*(n)$ et $c_K(n)$ sont synchronisés, nous obtenons $|c_K(n)|^2 = 2$. Ainsi, l'expression du signal de sortie s_1 est la suivante :

$$s_1(n) = s_e(n).c_K^*(n) = a_l.e^{j\theta_l}x(n).c_K(n).c_K^*(n) = A_l\ e^{j\theta_l}x(n) \tag{5.6}$$

avec :

$$Re(x(n)) = d_d.c_{OVSF_I} \tag{5.7}$$
$$Im(x(n)) = d_c.c_{OVSF_Q} \tag{5.8}$$

Pour remettre en phase le signal complexe $s_1(n)$, celui-ci est multiplié par le conjugué de la phase complexe estimée $\widehat{\alpha_l}$:

$$s_2(n) = s_1(n).\widehat{\alpha_l}^* = A_l\ e^{j\theta_l}.A_l\ e^{-j\theta_l}.x(n) = A_l^2.x(n) \tag{5.9}$$

La phase introduite par le canal de transmission étant éliminée, les bits de données sont présents uniquement sur la voie réelle et les bits de contrôle ne sont présents que sur la voie imaginaire. Ensuite, l'opération de désétalement (*despreading*) est réalisée. La partie réelle de s_2 est multipliée par la partie réelle du code OVSF et le résultat de la multiplication est accumulé sur un nombre d'échantillons égal au facteur d'étalement. Le signal en sortie de l'accumulateur évolue au rythme symbole. La durée d'un symbole est égale à $SF.T_c$. L'expression de la partie réelle du signal s_3 est égale à :

$$Re(s_3(k')) = \sum_{n=SF.k'}^{SF(k'+1)-1} Re\left(A_l^2 x(n)\right).c_{OVSF_I}(n) \tag{5.10}$$
$$= A_l^2 \sum_{n=SF.k'}^{SF(k'+1)-1} d_d(n).c_{OVSF_I}(n).c_{OVSF_I}(n) = SF.A_l^2.d_d(k') \tag{5.11}$$

La partie imaginaire de s_2 est multipliée par la partie imaginaire du code OVSF et le résultat est accumulé sur 256 échantillons. La durée d'un symbole au niveau de la voie imaginaire est égale à $256.T_c$. En suivant une démarche identique à celle de la partie réelle, nous obtenons l'expression de la partie imaginaire du signal s_3 suivante :

$$Im(s_3(k)) = 256.A_l^2.d_c(k) \tag{5.12}$$

Afin d'obtenir une dynamique identique sur les voies réelles et imaginaires, les parties réelles et imaginaires sont normalisées par leur facteur d'étalement. Ainsi, l'expression de la sortie du *finger* est la suivante :

$$\begin{cases} Re(y(k')) = A_l^2.d_d(k') \\ Im(y(k)) = A_l^2.d_c(k) \end{cases} \tag{5.13}$$

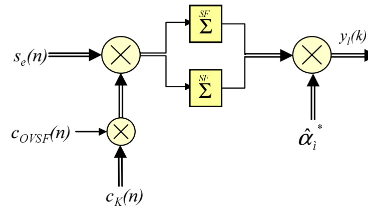


FIG. 5.7: Synoptique de la partie décodage des données au niveau d'un terminal mobile

Décodage des données au niveau de la station mobile

La technique présentée dans le paragraphe précédent peut être utilisée pour décoder les données au sein de la station mobile. Cependant, la structure des données reçues étant différente, il est possible de modifier et de simplifier le récepteur en termes de complexité, au niveau du terminal mobile. Dans le cadre de la réception des données au niveau de la station de base, il est nécessaire de remettre en phase le signal $s_1(n)$ issu de l'opération de désembrouillage avant de réaliser l'opération de désétalement. En effet, les codes OVSF utilisés pour les voies réelles et imaginaires sont différents et possèdent des facteurs d'étalement différents. Dans le cas de la voie descendante, les symboles représentant les données et le contrôle sont multiplexés afin de former un signal complexe et le code OVSF appliqué aux voies réelles et imaginaires est identique. Ainsi, il n'est pas nécessaire de remettre en phase le signal $s_1(n)$ avant de réaliser l'opération de désétalement. Cette opération de remise en phase peut être réalisée après l'accumulation réalisée au niveau de l'opération de désétalement car l'amplitude complexe du trajet est constante pendant la durée d'un *slot*. Dans ce cas, la fréquence de traitement de l'opération de remise en phase est égale à F_c/SF au lieu de F_c. Le signal complexe $s_1(n)$ est multiplié par le code de Kasami puis le code OVSF. Afin de limiter le nombre de multiplications entre le signal et le code nous proposons de réaliser d'abord la multiplication des codes puis la multiplication du code résultant par le signal $s_1(n)$. La multiplication des codes peut être réalisée simplement par des opérations logiques lors de la génération des codes. Le synoptique de la partie décodage est présenté à la figure 5.7.

Complexité des traitements

La complexité des différentes parties d'un récepteur de type *rake receiver* a été déterminée à partir des algorithmes utilisés pour le décodage des données, l'estimation du canal et la synchronisation fine. Pour une station de base, la complexité d'un *finger* est de 92.10^6 multiplications par seconde et 61.10^6 additions par seconde. En conséquence, la complexité d'un *rake receiver* composé de 5 *fingers* est de 462.10^6 multiplications par seconde et 307.10^6 additions par seconde. Pour la station de base, un *rake receiver* est affecté à chaque utilisateur.

5.3 Implantation d'un récepteur WCDMA

5.3.1 Définition de la contrainte de précision

Le but d'un récepteur de communications numériques est d'estimer à partir du signal reçu les symboles transmis d_k. A partir de la valeur estimée du symbole reçu et présente en sortie du *rake receiver*, une décision est prise afin d'obtenir la valeur binaire de ce symbole. Au sein d'un récepteur complet, un module de correction d'erreurs basé par exemple sur l'algorithme de Viterbi ou les turbo-décodeurs est utilisé. Pour améliorer les performances du système, une décision douce est réalisée au niveau de l'entrée de ce module de correction d'erreurs. Dans ce cas, le signal en sortie du *rake receiver* est quantifié sur quelques bits. Cependant, le module de correction d'erreurs n'étant pas développé actuellement, nous réalisons une décision ferme en sortie du *rake receiver*. Pour la modulation utilisée, la décision est basée sur l'analyse du signe du signal présent sur les voies réelles et imaginaires. L'objectif du récepteur est de minimiser la probabilité d'erreur Γ_r entre les symboles estimés $\widehat{d_k}$ et les symboles réellement transmis d_k. Ce taux d'erreur binaire est fonction du rapport entre la puissance du signal reçu et la puissance du bruit présent au niveau du récepteur (RSB). Pour ce type de récepteur, l'évolution du taux d'erreur binaire en fonction du RSB présent en entrée du *rake receiver* est présentée à la figure 5.8 [118].

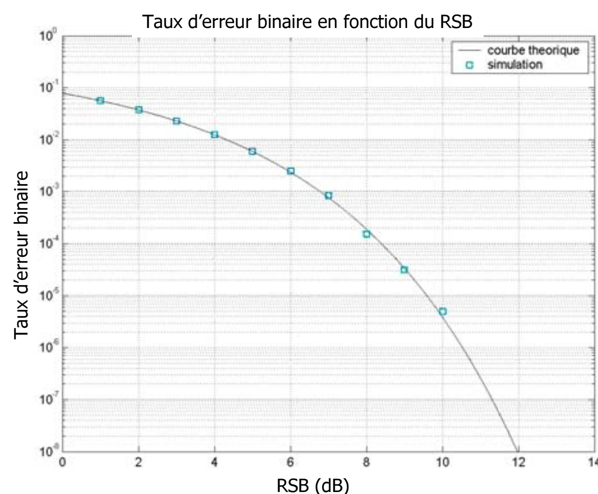


FIG. 5.8: Évolution du taux d'erreur binaire en sortie du *rake receiver* en fonction du RSB en entrée du récepteur

L'utilisation de l'arithmétique virgule fixe altère la valeur estimée des données et peut conduire à une modification du résultat de la décision. En conséquence, l'implantation de l'algorithme en virgule fixe est viable, si le taux d'erreur binaire Γ_{FP} lié à l'utilisation de l'arithmétique virgule fixe est limité. Cette métrique Γ_{FP}, détermine le taux d'échantillons pour lesquels la décision effectuée sur la donnée en précision infinie $\widehat{d_k}$ et en précision finie $\widetilde{d_k}$ est différente. Pour évaluer ce taux d'erreur binaire en fonction de la puissance du bruit de quantification présent au niveau de la donnée estimée, l'expérimentation dont le synoptique est présenté à la figure 5.9 a été réalisée.

Le taux d'erreur binaire Γ_{FP} lié à l'arithmétique virgule fixe a été mesuré en fonction du rapport signal à bruit de quantification (RSBQ) présent en sortie du *rake receiver*. Ce RSBQ est ajusté en agissant sur la puissance de la source de bruit b_y, modélisant le bruit de quantification lié à l'arithmétique virgule fixe selon la technique présentée dans la partie 3.6 à la page 120. Ce taux d'erreur binaire a été déterminé pour deux valeurs du RSB égales à 1 dB et 2.5 dB. Cette expérimentation a été réalisée pour des valeurs faibles du RSB afin de limiter le nombre

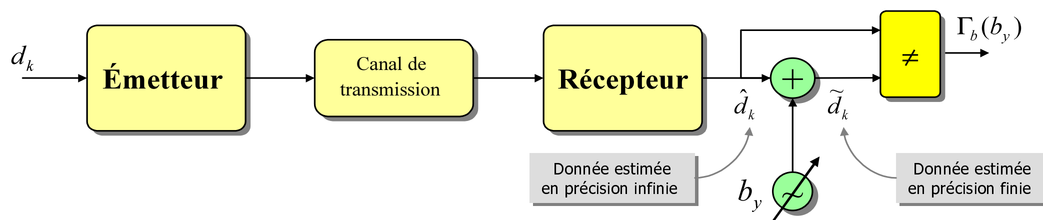


FIG. 5.9: Synoptique de l'expérimentation réalisée pour déterminer $\Gamma_{FP} = f\,(RSBQ)$

d'échantillons utilisés au sein de la simulation en virgule flottante du système. Celui-ci a été fixé à 2.10^6. Les résultats obtenus sont présentés à la figure 5.10.

Afin que l'influence du codage des données en virgule fixe soit négligeable sur les performances du récepteur, nous imposons que, pour une valeur donnée du RSB, le taux d'erreur binaire Γ_{FP} lié à l'arithmétique virgule fixe soit égal à 1/10 du taux d'erreur binaire Γ_r obtenu en précision infinie. Cette contrainte permet de fixer la valeur minimale du RSBQ à respecter lors du codage des données. D'après les courbes présentées à la figure 5.10, la valeur minimale du RSBQ est de 12.5 dB pour un RSB de 1 dB et de 12 pour une valeur du RSB de 2.5 dB. Ainsi, la valeur minimale du RSBQ devant être respectée est fixée à 12.5 dB

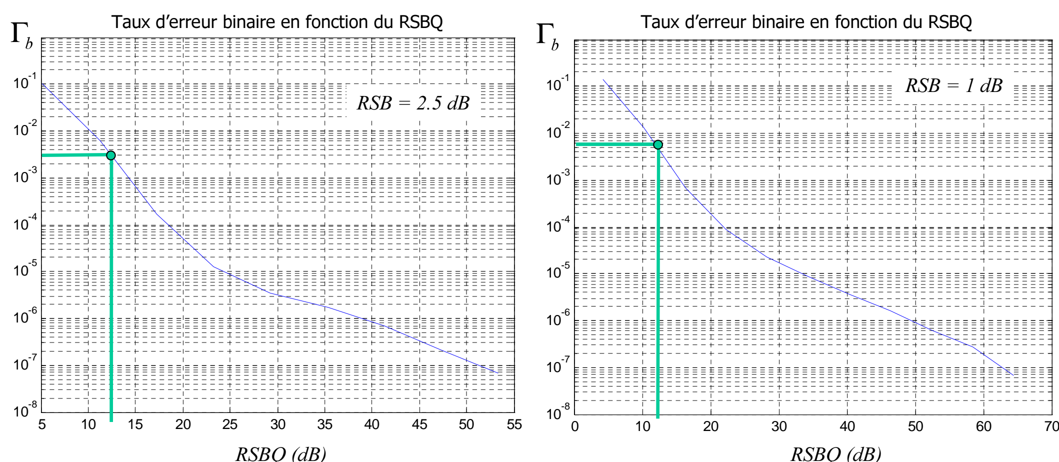


FIG. 5.10: Évolution du taux d'erreur binaire Γ_b en fonction du RSBQ pour deux valeurs du RSB en entrée du récepteur

5.3.2 Filtre de réception

Le filtre FIR de réception possède une complexité proche de 1 $GMAC$ par seconde. Deux solutions architecturales fournissant les performances nécessaires en termes de capacités de calcul ont été testées. La première solution correspond à un ASIC classique et la seconde solution se base sur l'architecture reconfigurable au niveau chemin de données DART [29, 30] et développée au sein du laboratoire.

Dans [114], les études menées sur l'influence de la largeur des convertisseurs analogique-numérique montrent que la dégradation au niveau de la réception est négligeable pour des largeurs supérieures à 5 bits. Ainsi, nous avons fixé la largeur de l'entrée du filtre à 6 bits. La sortie du filtre est utilisée en entrée du *rake receiver* et celui-ci est implanté au sein d'un processeur programmable. En conséquence, la largeur de l'entrée du *rake receiver* est fixée en

fonction des types des données manipulées par le processeur programmable. Nous considérons que celui-ci peut manipuler des données sur 8 ou 16 bits.

Nous considérons que le signal présent au niveau du convertisseur analogique-numérique possède un domaine de définition égal à $[-1, 1]$. Dans le cadre d'un service téléphonique pour une transmission de parole, le rapport entre le signal utile et le bruit présent au niveau du récepteur est supérieur à 5 dB. Dans ce cas, le niveau[2] du signal utile est d'environ -10 dB. Le niveau du bruit lié à la quantification du signal en sortie du CAN est de -34 dB. Ces différents niveaux sont représentés à la figure 5.11. Le niveau du signal utile en sortie du filtre FIR est inchangé car le gain du filtre au sein de sa bande passante est unitaire. La propagation du bruit de quantification d'entrée au sein du filtre conduit à un niveau en sortie égal à -29 dB. Ceci conduit à une dégradation du RSBQ entre l'entrée et la sortie de 5 dB. Si la sortie du filtre est codée sur 16 bits, le niveau du bruit est de -29 dB. Si la sortie du filtre est codée sur 8 bits, le niveau du bruit est de -26 dB, si les calculs sont réalisés avec le maximum de précision. Afin d'obtenir la précision maximale dans le cas où la sortie est codée sur 8 bits, nous fixons la contrainte minimale $RSBQ_{min}$ à 16 dB pour optimiser la spécification en virgule fixe du filtre.

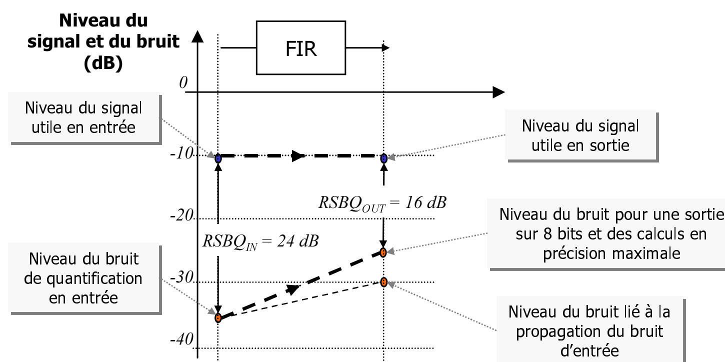


FIG. 5.11: Évolution du niveau du bruit et du signal au sein du filtre FIR

Solution basée sur un ASIC

La méthodologie mise en œuvre au sein de ce travail de recherche n'est pas destinée à l'implantation d'algorithmes au sein des ASIC. Cependant, certaines phases de la méthodologie peuvent être utilisées pour déterminer les caractéristiques de l'architecture de l'ASIC. Ainsi, les phases d'évaluation de la précision, de détermination de la dynamique et de la position de la virgule des données peuvent être utilisées dans l'état, dans le cadre des ASIC. Ensuite, notre méthode de détermination du type des données a été adaptée afin de pouvoir optimiser la largeur des opérateurs au sein de l'ASIC. Pour cela, un modèle d'architecture permettant de traiter des données de largeur quelconque a été défini. Le nombre de variables à optimiser au sein de l'application étant très faible, la technique d'optimisation mise en œuvre au sein de notre méthodologie peut être utilisée pour résoudre ce type de problème. De plus, la largeur des entrées et de la sortie du filtre étant fixée, uniquement la largeur de l'additionneur est à déterminer. En effet, la largeur des entrées du multiplieur est fixée par celle des entrées du filtre et des coefficients. La largeur des coefficients a été fixée à 8 bits, à partir des résultats présents dans [118]. Cependant, la largeur des coefficients peut être déterminée automatiquement en intégrant dans le processus d'optimisation une contrainte sur la déviation maximale de la réponse fréquentielle du filtre. Ainsi, le processus d'optimisation va uniquement déterminer la largeur de l'additionneur.

[2]Le niveau d'un signal x de puissance P_x est égal à $10.log_{10}(P_x)$.

Les résultats du processus d'optimisation sont présentés à la figure 5.12 pour différentes valeurs de la largeur de la sortie. Pour chaque courbe de la figure 5.12, deux zones peuvent être distinguées. Lorsque la largeur de l'additionneur est proche de la largeur des entrées du multiplieur, les bruits générés en sortie de chaque multiplieur ont une influence très importante sur le bruit global. Dans ce cas, l'évolution du RSBQ exprimé en dB, est quasiment linéaire par rapport au nombre de bits. Lorsque la largeur de l'additionneur est proche de celle du multiplieur les bits éliminés ont une influence moindre sur le bruit global. Pour satisfaire les spécifications sur la largeur de l'entrée et de la sortie du filtre et la contrainte de RSBQ minimale fixée à 16 dB, la largeur de l'additionneur doit être égale à 14 bits. Les formats des différentes données présentes au sein du filtre FIR sont détaillés à la figure 5.13.

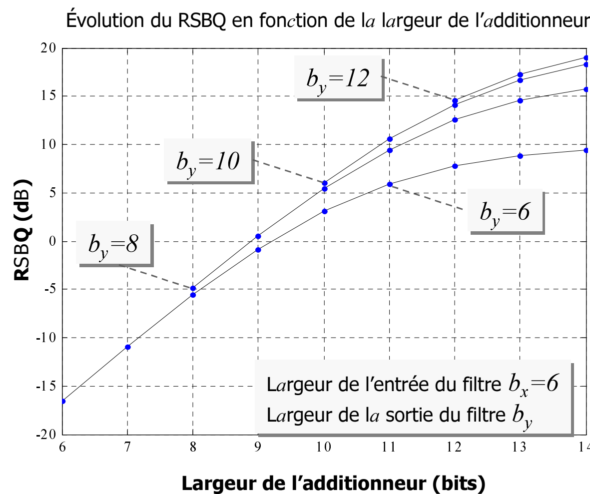


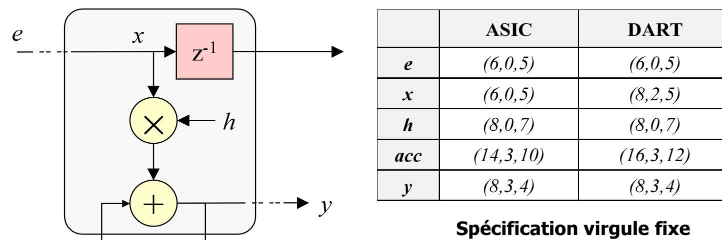
FIG. 5.12: Évolution du RSBQ en fonction de la largeur de l'additionneur utilisé au sein du filtre de réception

Solution basée sur l'architecture reconfigurable DART

Présentation de l'architecture : L'architecture DART, présentée dans [29, 30], a pour objectif de combiner au sein d'une même architecture les capacités de calcul des ASIC et la flexibilité des processeurs programmables. Cette architecture reconfigurable dynamiquement au niveau de son chemin de données est découpée en *clusters* possédant chacun leurs ressources de contrôle et de mémorisation. Chaque *cluster* est composé de six chemins de données reconfigurables (DPR). Chaque DPR est composé de deux multiplieurs, de deux UAL et de 4 bancs mémoires possédant chacun son propre générateur d'adresses. Deux approches d'utilisation de cette architecture peuvent être envisagées. Pour la première approche, l'architecture DART est autonome et un contrôleur de tâche est présent afin d'assigner les tâches aux *clusters*. La seconde approche consiste à considérer un *cluster* de l'architecture DART comme un bloc IP et à intégrer celui-ci dans un SoC (System on Chip).

Cette architecture possède des capacités de traitement SWP. Chaque multiplieur permet de réaliser soit une multiplication de deux données sur 16 bits avec un résultat sur 32 bits ou soit deux multiplications de données sur 8 bits avec un résultat sur 16 bits. Les UAL peuvent réaliser 4 opérations sur 8 bits, 2 opérations sur 16 bits ou 1 opération sur 32 bits. Les opérations de recadrage sont réalisées à l'aide des UAL ou de registres à décalage spécialisés situés en sortie des multiplieurs et des UAL ou en entrée des UAL. La valeur du décalage est spécifiée au sein des instructions de configuration de l'architecture.

Implantation du filtre FIR : La méthode d'optimisation de la largeur des données a été utilisée afin de déterminer la spécification virgule fixe du filtre FIR en vue de son implantation



	ASIC	DART
e	(6,0,5)	(6,0,5)
x	(6,0,5)	(8,2,5)
h	(8,0,7)	(8,0,7)
acc	(14,3,10)	(16,3,12)
y	(8,3,4)	(8,3,4)

Spécification virgule fixe

FIG. 5.13: Spécification virgule fixe des deux solutions testées

dans l'architecture DART. Ce processus d'optimisation conduit à une solution fournissant un RSBQ en sortie du filtre de 16 dB. Les formats des différentes données présentes au sein du filtre FIR sont détaillés à la figure 5.13. Le recadrage des données est réalisé au niveau de l'entrée du filtre. En effet, la donnée issue du convertisseur analogique-numérique étant codée sur 6 bits, son insertion au sein d'une donnée sur 8 bits se traduit par deux bits supplémentaires non utilisés. La présence de ces deux bits permet de réaliser un recadrage de l'entrée sans dégrader la précision des calculs par rapport à l'implantation utilisant des recadrages en sortie des multiplications. L'exécution en temps réel de ce filtre FIR au sein de l'architecture DART se traduit par l'utilisation d'un *cluster* complet pendant 31.6% du temps et une consommation de 58 mW [30].

5.3.3 Décodage des données au sein d'un terminal mobile

Évolution de la précision en fonction du facteur d'étalement

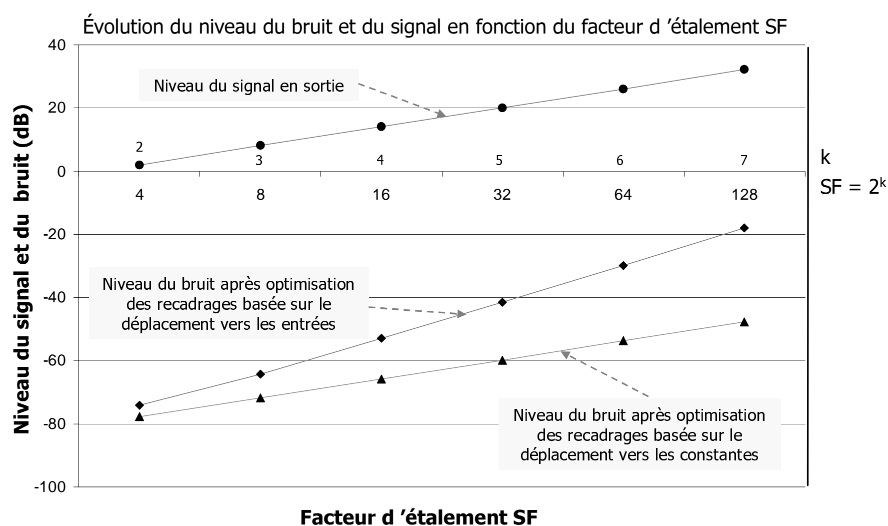


FIG. 5.14: Évolution des niveaux du bruit et du signal en sortie du *rake receiver* en fonction du facteur d'étalement

La corrélation entre le signal et les codes est paramétrée par le facteur d'étalement (SF) utilisé au niveau de la transmission. Ainsi, le codage des données dépend de ce facteur d'étalement, car celui-ci fixe la longueur des codes et le nombre d'accumulations réalisées au sein du processus de corrélation. Pour étudier la précision de la sortie du corrélateur en fonction du facteur d'étalement, les niveaux du bruit et du signal en sortie d'un corrélateur ont été mesurés pour différentes valeurs de SF et en considérant une même architecture. Les résultats dans le

cadre d'un DSP de largeur naturelle égale à 16 bits, sont présentés à la figure 5.14.

Pour optimiser le placement des opérations de recadrage, les solutions basées sur le déplacement de celles-ci vers les constantes et vers les entrées ont été testées. Les résultats montrent que les performances obtenues avec un déplacement vers les constantes sont supérieures en raison des valeurs particulières utilisées au niveau de ces constantes. L'analyse des courbes montre que les niveaux du bruit et du signal exprimés en dB sont proportionnels au paramètre k_{SF} défini tel que $SF = 2^{k_{SF}}$. De plus, pour le bruit associé au déplacement vers les constantes, le facteur de proportionnalité est identique à celui obtenu au niveau du signal. En conséquence, pour ce type de codage, le RSBQ, correspondant à la différence entre les deux niveaux, est constant et indépendant du facteur d'étalement. Ainsi, par la suite, un facteur d'étalement SF égal à 4 est utilisé.

Spécification virgule fixe du *rake receiver*

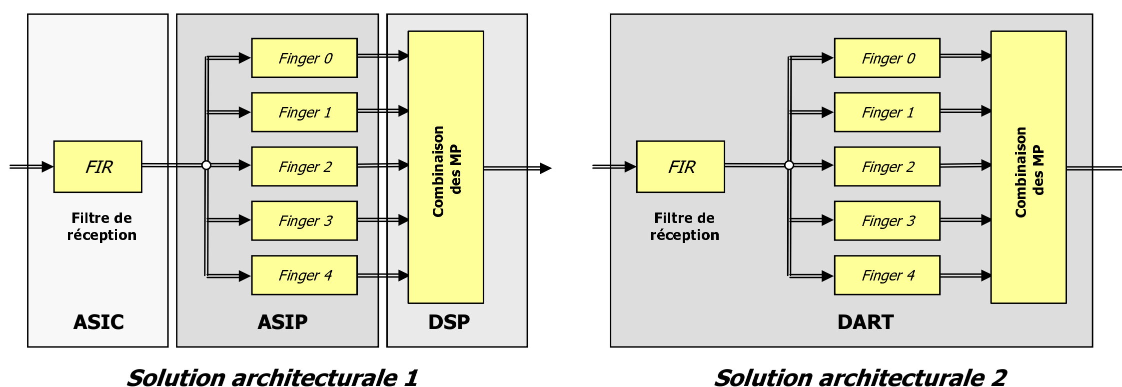


FIG. 5.15: Synoptique des solutions architecturales testées

Pour l'implantation du *rake receiver* au sein d'un mobile, deux architectures sont considérées. Les contraintes architecturales liées à la station mobile ont été présentées dans le paragraphe 5.1. Pour ce type d'architecture, le coût et la consommation d'énergie doivent être minimisés. La première solution associe trois architectures différentes. L'application est partitionnée en fonction de la fréquence de traitement des différentes parties. Le filtre de réception traitant des échantillons à une fréquence quadruple de la fréquence *chip*, est implanté au sein d'un ASIC, afin d'obtenir des capacités de calcul importantes. Le *rake receiver* est implanté au sein d'un DSP utilisant un accélérateur pour réaliser la fonction de corrélation traitant les échantillons à la fréquence *chip*. Cet accélérateur est réalisé à l'aide d'un ASIP spécialisé pour réaliser des traitements de type multiplication-accumulation sur des données complexes codées sur 8 bits. Les traitements à la fréquence symbole, correspondant à la combinaison des résultats des *fingers* sont implantés au sein d'un DSP faible consommation tel que le DSP TMS320C55. Ce DSP correspond à la nouvelle génération de DSP basse consommation de la société Texas Instruments et est basé sur l'architecture du TMS320C54x. La seconde solution est basée sur l'architecture reconfigurable DART présentée dans la section précédente. Pour les deux solutions, l'affectation des différents modules aux architectures est représentée à la figure 5.15.

Le processus de corrélation permet d'accroître la puissance du signal utile. Pour déterminer la puissance du signal utile en sortie du *rake receiver*, nous avons considéré un canal de propagation présent au sein de la norme UMTS. Celui-ci modélise le canal entre la station de base et un mobile se déplaçant à une vitesse élevée. Ce canal de propagation est relativement défavorable en termes de puissance du signal en sortie du *rake receiver*. En effet, les amplitudes des trajets

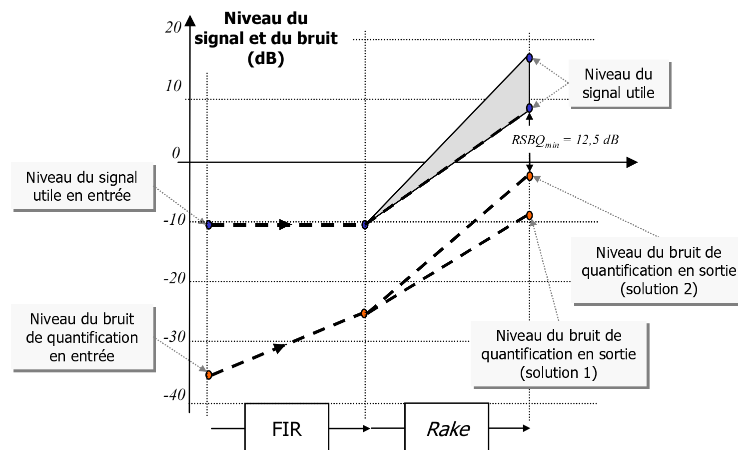


FIG. 5.16: Niveau du bruit et du signal au sein du récepteur WCDMA

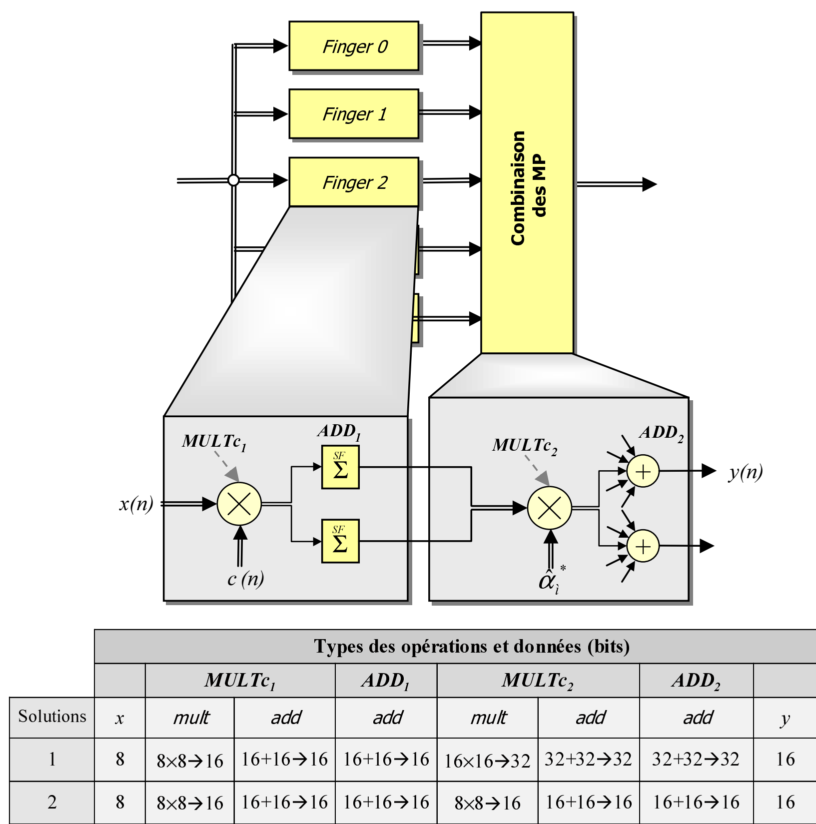
autres que le trajet principal sont très faibles. Ainsi, la puissance du signal[3] utile en sortie du *rake receiver* est égale à 9 dB. Les différents niveaux obtenus pour le signal et le bruit de quantification sont représentés à la figure 5.16.

Notre méthodologie a été utilisée pour obtenir la spécification virgule fixe du *rake receiver* dont le code source est présenté dans l'annexe D. La localisation des opérations de recadrage a été optimisée en déplaçant celles-ci vers les constantes. La contrainte de RSBQ minimale a été fixée à $12,5\ dB$. La largeur des entrées du *rake receiver* a été fixée à 8 bits. Les différentes largeurs de données obtenues à l'aide de la méthodologie sont représentées à la figure 5.17. Pour ces expérimentations, aucune contrainte n'a été fixée au niveau de la sélection des instructions SWP. Le RSBQ obtenu pour la première solution architecturale est égal à $18,86\ dB$ et $12,8\ dB$ pour la seconde solution. Les données au sein des *fingers* sont codées de manière identique pour les deux solutions. La différence de codage entre les deux solutions est présente sur le module de recombinaison des multi-trajets. Pour la seconde solution, la largeur des données en entrée du module de recombinaison est égale à 8 bits. Pour la première solution, le DSP ne permettant pas de traiter des données sur 8 bits, les traitements sont réalisés sur des données sur 16 bits.

5.3.4 Décodage des données au sein d'une station de base

Dans le cadre de l'implantation du *rake receiver* au sein de la station de base, la solution architecturale retenue doit fournir les capacités de calcul requises pour le traitement de plusieurs utilisateurs en temps réel et la flexibilité nécessaire pour implanter aisément les évolutions futures du standard. La solution architecturale retenue est basée sur un processeur DSP permettant de fournir des capacités de calcul élevées, couplé à un accélérateur pour réaliser les traitements intensifs et réguliers. Cette solution est schématisée à la figure 5.18. De nombreux DSP récents basés sur une architecture proposant du parallélisme au niveau instruction sont conçus en vue de leur intégration dans les infrastructures de télécommunications de troisième génération. Le modèle d'architecture retenu pour nos expérimentations correspond à celui du TMS320C64x de la société Texas Instruments. Ce processeur est basé sur une architecture VLIW composée de deux *clusters* permettant de réaliser chacun 4 instructions en parallèle. L'accélérateur est basé sur un *cluster* de l'architecture DART présentée dans la section 5.3.2 à la page 180. Cet accélérateur est utilisé pour implanter le filtre FIR qui nécessite des capacités de calcul très élevées. La partie du *rake receiver* réalisant la recombinaison des multi-trajets est implantée au sein du DSP VLIW. Les différents *fingers* sont implantés soit au sein du DSP VLIW soit au sein de l'accélérateur DART. Les deux architectures possédant des capacités SWP relativement

[3]Résultats obtenus sans normalisation en sortie de la phase de désétalement



	Types des opérations et données (bits)							
		MULTc₁		*ADD₁*	*MULTc₂*		*ADD₂*	
Solutions	x	*mult*	*add*	*add*	*mult*	*add*	*add*	y
1	8	8×8→16	16+16→16	16+16→16	16×16→32	32+32→32	32+32→32	16
2	8	8×8→16	16+16→16	16+16→16	8×8→16	16+16→16	16+16→16	16

FIG. 5.17: Largeur des données au sein du *rake receiver* (terminal mobile)

proches, les spécifications virgule fixe de l'application, obtenues à l'aide de notre méthodologie pour les deux types d'architecture sont identiques. Le choix du DSP ou de l'architecture DART pour l'implantation de cette partie sera fonction des coûts de chaque implantation.

Le code source de cette application est présenté dans l'annexe D et les résultats du codage des données en virgule fixe sont exposés à la figure 5.19. Les contraintes de précision imposées à la méthodologie sont identiques à celles utilisées dans le cas de la station mobile. Le RSBQ obtenu est égal à $21,4$ dB alors que la contrainte de précision est de $12,5$ dB. La spécification obtenue regroupe des données de largeur 8, 16 et 32 bits. La première multiplication complexe traite des données sur 8 bits et conserve un résultat en double précision (16 bits). Ensuite, la

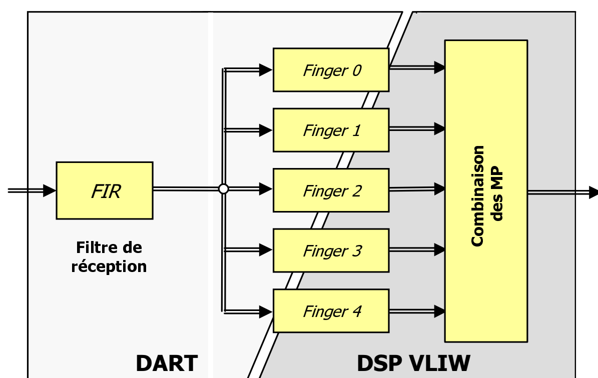
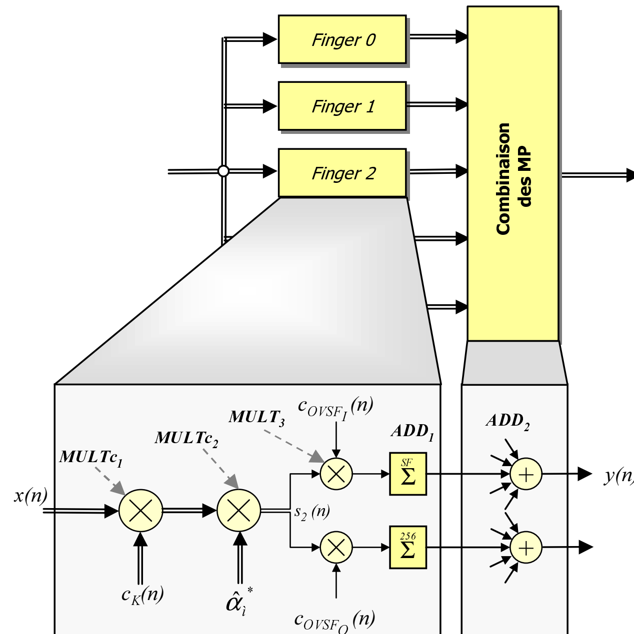


FIG. 5.18: Synoptique de la solution architecturale testée

seconde multiplication complexe traite des données sur 16 bits et conserve un résultat en simple précision (16 bits). De même, la troisième opération de multiplication puis d'accumulation traite des données sur 16 bits et conserve un résultat en simple précision. Ainsi, cette spécification est une combinaison d'opérations de type multiplication-addition sur des données 8 bits puis sur des données 16 bits. En effet, le traitement de données uniquement sur 8 bits en entrée des multiplications ne permettait pas de respecter la contrainte de précision souhaitée.



Types des opérations et données (bits)								
	MULTc₁		*MULTc₂*		*MULT₃*	*ADD₁*	*ADD₂*	
x	*mult*	*add*	*mult*	*add*	*mult*	*add*	*add*	*y*
8	8×8→16	16+16→16	16×16→32	16+16→16	16×16→32	16+16→16	16+16→16	16

Fig. 5.19: Largeur des données au sein du *rake receiver* (station de base)

5.4 Conclusions

Dans ce chapitre, les résultats de l'expérimentation de notre méthodologie sur des applications issues des systèmes de radio-communications de troisième génération ont été présentés. Ces applications réalisent la réception des données dans le cadre de transmissions à étalement de spectre par séquence directe. Après une présentation de la structure d'un émetteur et d'un récepteur WCDMA et des différents algorithmes utilisés, les solutions architecturales envisagées et les résultats obtenus avec l'outil ont été exposés.

A travers ces expérimentations, nous avons montré la capacité de notre méthodologie à traiter des applications de traitement du signal de complexité moyenne. Ces applications représentent entre 50 et 100 lignes de code C. L'obtention d'une spécification en virgule fixe optimisée nécessite quelques minutes. La majorité du temps est consacrée à la détermination de l'expression du RSBQ et aux deux phases d'optimisation correspondant à la détermination de la largeur des données et à l'optimisation du placement des opérations de recadrage. Ces temps d'optimisation sont nettement plus faibles que ceux obtenus avec une méthode d'évaluation de la précision basée sur la simulation et nécessitant un nombre de simulations très élevé.

Ces expérimentations montrent la réduction importante du temps de conversion en virgule fixe par rapport à une transformation manuelle. Dans le cadre de l'application considérée, le temps nécessaire à la définition de la contrainte de précision est relativement important. En effet, cette phase nécessite de simuler l'application avec un nombre d'échantillons important en vue de déterminer les taux d'erreur binaire. Lorsque cette contrainte de précision est définie le temps nécessaire à la transformation de l'algorithme de traitement du signal en virgule flottante en une spécification en virgule fixe, est relativement faible. En conséquence, la méthodologie proposée au sein de ce travail de recherche contribue à la réduction du temps de développement des applications de traitement du signal.

Conclusions et perspectives

Conclusions

L'implantation efficace des algorithmes de traitement numérique du signal (TNS) dans les systèmes embarqués requiert l'utilisation de l'arithmétique virgule fixe afin de satisfaire les contraintes de coût, de consommation et d'encombrement exigées par ces applications. Le codage manuel des données en virgule fixe est une tâche fastidieuse et source d'erreurs. De plus, l'augmentation de la complexité des algorithmes de TNS et la réduction du temps de mise sur le marché des applications exigent l'utilisation d'outils de développement de haut niveau, permettant d'automatiser certaines tâches. Ainsi, la mise en œuvre de méthodologies de codage automatique des données en virgule fixe est nécessaire car le codage manuel des données se révèle être un frein important à la diminution du temps de conception.

Dans le cadre de ce travail de recherche, une nouvelle méthodologie de compilation d'algorithmes de TNS pour les processeurs en virgule fixe a été définie et mise en œuvre. Cette méthodologie prend en compte l'architecture cible et est couplée avec certaines phases de la génération de code afin d'optimiser l'implantation en virgule fixe de l'algorithme au sein du processeur cible. De plus, certains paramètres de l'implantation tels que le temps d'exécution sont optimisés sous contrainte de respect des critères de qualité associés à l'application. Ces contraintes sur les performances du système sont traduites en une contrainte de précision minimale devant être respectée par le système en virgule fixe.

Trois aspects ont été abordés au sein de ce travail de recherche. Le premier aspect correspond à la définition de la méthodologie. Tout d'abord, l'analyse de l'influence de l'architecture sur la précision des calculs a permis de montrer la nécessité de prendre en compte l'architecture pour obtenir une implantation optimisée en termes de précision et de temps d'exécution [103]. Ensuite, l'étude de l'interaction entre les processus de génération de code et de codage des données a permis de définir les couplages nécessaires entre ces deux processus [100].

Le second aspect concerne la mise en œuvre d'une méthodologie d'évaluation de la précision d'une spécification en virgule fixe. L'objectif de cette méthodologie est de déterminer automatiquement l'expression du rapport signal à bruit de quantification (RSBQ) en sortie d'un système en virgule fixe. Tout d'abord, un nouveau modèle de bruit a été proposé. Celui-ci correspond au bruit généré lors de la quantification du résultat de la multiplication d'un signal par une constante. Ensuite, les expressions de la puissance du bruit de quantification en sortie des systèmes linéaires et des systèmes non-linéaires et non-récursifs ont été détaillées [104]. La précision des estimations obtenues pour différentes applications montre la qualité de cette approche pour estimer la puissance du bruit de quantification. Une méthode de détermination des différents éléments nécessaires à la constitution de l'expression analytique du RSBQ a été proposée dans le cadre des systèmes linéaires [105]. Cette méthode permet de déterminer automatiquement les fonctions de transfert régissant un système représenté sous la forme d'un graphe flot de signal. Les résultats obtenus avec l'outil développé pour implanter cette méthode soulignent l'efficacité de notre approche en termes de temps d'exécution. En particulier, l'utilisation de

cette approche au sein du processus de détermination et d'optimisation du format des données permet d'obtenir des temps d'optimisation nettement plus faibles que ceux obtenus avec des méthodes basées sur la simulation.

Le troisième aspect abordé au sein de ce travail de recherche correspond à la définition et à la mise en œuvre de la méthode de codage des données en virgule fixe. La première étape permettant d'évaluer la dynamique des données de l'application, est réalisée en combinant deux techniques présentes au sein de la littérature et basées sur une approche analytique.

La seconde et la troisième étape permettent de déterminer le format des données en virgule fixe. Dans un premier temps, la position de la virgule des données est définie. L'originalité de notre méthode réside dans la définition d'une modélisation permettant de prendre en compte la présence éventuelle de bits de garde au sein du processeur cible [101]. Ensuite, la largeur des différentes données est déterminée. Une approche nouvelle est proposée pour prendre en compte les différents types de données manipulées au sein du processeur cible. Cette méthode permet de sélectionner la séquence d'instructions dont la largeur des opérandes permet de respecter la contrainte de précision imposée et de minimiser le temps d'exécution global du code [99].

La dernière étape a pour objectif d'optimiser le format des données en virgule fixe en vue de réduire le temps d'exécution du code. Pour cela, les opérations de recadrage présentes au sein de l'application sont déplacées sous contrainte de précision. Par rapport à la méthode proposée dans [70], notre méthode prend en compte le coût réel de l'opération de recadrage. En conséquence, deux types de méthode ont été proposés en fonction du processeur cible [102]. Pour les processeurs dont le parallélisme est spécifié au sein d'instructions complexes, le temps d'exécution de l'opération de recadrage est déterminé à partir de la suite d'instructions utilisées pour implanter cette opération. Cette suite d'instructions est obtenue à l'aide d'une phase de sélection d'instructions. La stratégie d'optimisation définie est basée sur un processus itératif. Chaque itération consiste à déplacer l'opération de recadrage la plus coûteuse et à valider ce déplacement après évaluation de la précision de la nouvelle spécification en virgule fixe issue de ce déplacement. Pour les processeurs dont le parallélisme est obtenu par le regroupement d'instructions partielles, le coût d'un recadrage dépend de la manière dont les instructions sont ordonnancées. Ainsi, une méthode de placement des opérations de recadrage pendant la phase d'ordonnancement est proposée.

L'expérimentation de notre méthodologie sur des applications de traitement du signal a montré la capacité de celle-ci à traiter des systèmes de complexité moyenne et sa contribution à la réduction du temps de développement des applications de traitement du signal.

Perspectives

Évaluation de la précision

Dans le cadre de l'évaluation de la précision, les perspectives se situent à différents niveaux. D'un point de vue de la méthode d'évaluation de la précision, nous souhaitons étendre la classe des applications supportées par notre approche. En particulier, nous désirons définir la puissance du bruit de quantification en sortie des systèmes adaptatifs et permettre l'automatisation de la détermination de l'expression de cette puissance. Le second aspect correspond à la poursuite du développement de l'outil permettant d'implanter cette méthodologie d'évaluation de la précision. Ce développement concerne à la fois l'amélioration de l'efficacité de l'outil et l'intégration des nouvelles techniques telles que celles présentées pour les systèmes non-linéaires et non-récursifs. De plus, les fonctionnalités réalisées par Matlab doivent être intégrées dans l'outil.

La présence au sein d'une même application, de parties récursives et de parties non récursives composées de nombreux nœuds, conduit à des temps d'évaluation de la précision pouvant

devenir élevés si l'algorithme d'énumération des circuits est appliqué à l'ensemble du graphe. Différentes voies sont possibles pour résoudre ce problème. Dans un premier temps, le temps d'exécution de la méthodologie proposée peut être optimisé. Des techniques de réduction du nombre de nœuds au sein du graphe peuvent être mises en œuvre pour réaliser certains traitements. Ensuite, l'utilisation d'une approche hiérarchique pour traiter ce type de problème nous semble être une voie prometteuse pour obtenir une méthodologie efficace dans le cadre des applications complexes. En effet, l'analyse de haut niveau de l'application permet de décomposer l'application en blocs distincts. Ensuite, les graphes représentant chaque bloc sont traités indépendamment afin de déterminer les fonctions de transfert associées à ces blocs. Les fonctions de transfert globales associées à l'application complète sont déterminées à partir des fonctions de transfert des blocs. Le traitement de graphes de taille moyenne permet d'obtenir un temps d'exécution global raisonnable. Par ailleurs, la mise en œuvre de techniques permettant de déterminer la fonction de transfert associée à une structure répétitive sans réaliser le déroulage de la boucle permettrait de diminuer le temps d'exécution global.

Dans le cadre des filtres linéaires, le codage des données doit permettre de respecter la contrainte de précision minimale ($RSBQ_{min}$). De plus, le codage des coefficients ne doit pas modifier la réponse fréquentielle de manière trop importante. Ainsi, une déviation maximale entre la fonction de transfert en précision infinie et en précision finie est définie.

Dans le cadre des méthodes basées sur la simulation, le respect de cette contrainte nécessite de discrétiser l'espace fréquentiel en N_f points et de vérifier pour chaque fréquence f_k de cet espace que la réponse fréquentielle obtenue en ce point $\widehat{H}(e^{j2\pi f_k})$ vérifie la contrainte spécifiée au niveau de la déviation maximale de la réponse fréquentielle. Pour chaque fréquence f_k, une simulation en virgule fixe du filtre est effectuée en utilisant en entrée un signal sinusoïdal de fréquence f_k. Ce processus nécessite de réaliser N_f simulations du filtre en virgule fixe pour chaque modification du format des coefficients. En conséquence, le temps d'exécution du processus d'optimisation est très fortement augmenté.

Dans le cadre des systèmes linéaires, la méthodologie mise en œuvre permet de déterminer l'ensemble des fonctions de transfert présentes au sein d'un système. Ainsi, la fonction de transfert obtenue après la quantification des coefficients est connue et la contrainte définie au niveau de la réponse fréquentielle peut être vérifiée aisément sans efforts supplémentaires.

Cette méthodologie d'évaluation de la précision est exploitée dans le cadre de notre méthode d'implantation d'algorithmes de TNS au sein des processeurs en virgule fixe, présentée dans le chapitre 4. De plus, elle peut être avantageusement utilisée dans le cadre de l'implantation matérielle au sein d'ASIC ou de FPGA. En effet, cette méthodologie peut être utilisée au sein du processus d'optimisation de la largeur des opérateurs. Cette méthode a été utilisée dans le cadre d'une méthodologie de définition de blocs IP (Intellectual Properties) génériques [126, 122]. Pour certaines applications typiques, l'utilisateur dispose de blocs IP génériques pouvant être adaptés à l'application ciblée. En particulier, la largeur des données est optimisée en fonction d'une contrainte minimale de précision définie par l'utilisateur. La détermination de la largeur des données est réalisée à l'aide de règles définies par le concepteur de l'IP. Ces règles ont été établies à partir de l'expression du RSBQ obtenue à l'aide de notre méthodologie.

Au travers de ce travail de recherche, nous avons été amenés à mettre en œuvre une méthodologie permettant de déterminer la fonction de transfert d'un système spécifié sous la forme d'un graphe flot de signal. Cette technique peut être utilisée dans d'autres contextes que celui du codage des données en virgule fixe. Par exemple, elle peut être mise en œuvre pour vérifier la conformité entre un code source décrivant un système linéaire et la spécification de l'application. La fonction de transfert associée au code source est déterminée et comparée avec la fonction de transfert définie au sein des spécifications.

Codage des données en virgule fixe

Pour resituer notre travail de recherche, le cycle de développement d'une application de traitement du signal intégrant notre méthodologie de conversion en virgule fixe est présenté à la figure 5.20. La phase de définition de l'algorithme de TNS permet d'obtenir une description de l'algorithme à l'aide d'un code C en virgule flottante. Une méthodologie de compilation de cet algorithme pour un processeur en virgule fixe a été proposée. Au niveau de l'outil permettant d'implanter cette méthodologie, certains modules restent à développer. Les structures conditionnelles doivent être intégrées au sein de la représentation de type GFDC et les différentes phases de conversion en virgule fixe doivent être étendues afin de supporter ce type de structure de contrôle. L'interface entre les phases de codage en virgule fixe et l'outil de génération de code CALIFE doit être complétée. Ce développement permettra de mettre en œuvre la phase d'optimisation de la largeur des données issues du renvoi de variables intermédiaires en mémoire et la phase d'optimisation du placement des opérations de recadrage pour les processeurs possédant du parallélisme au niveau instruction. Ensuite, le module de conversion de la représentation intermédiaire en une description *SystemC* doit être développé afin de pouvoir simuler la spécification en virgule fixe obtenue à l'aide de notre méthodologie. Cette simulation permet de vérifier si les performances requises au niveau de l'application sont respectées. Dans le cas contraire, la contrainte de précision minimale $RSBQ_{min}$ peut être ajustée et une nouvelle itération du processus d'optimisation est réalisée.

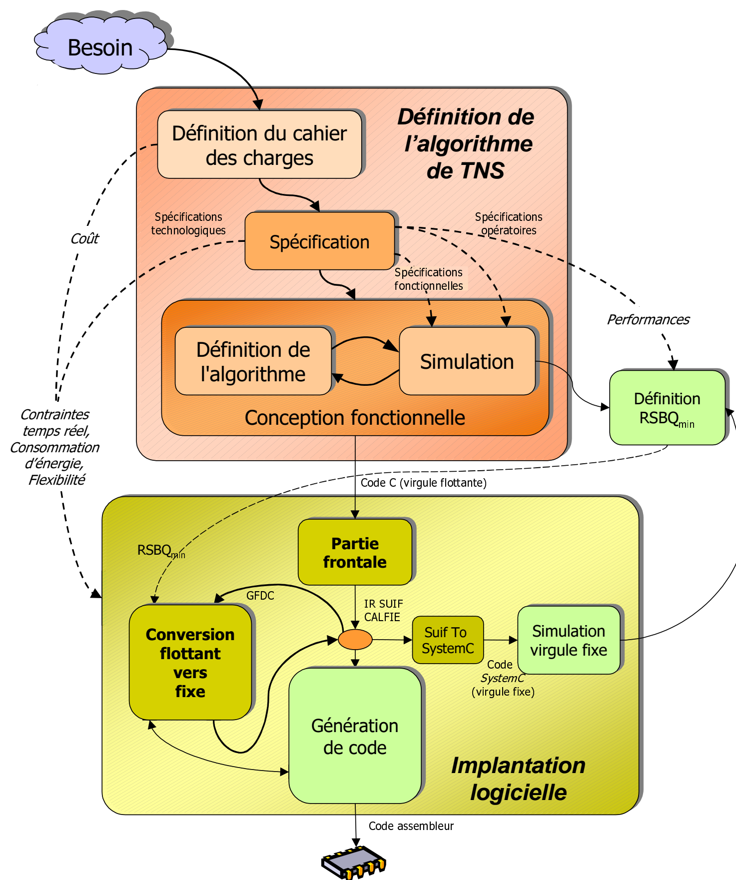


FIG. 5.20: Cycle de développement d'une application de TNS intégrant la méthodologie de conversion en virgule fixe

La méthode mise en œuvre pour évaluer la dynamique des données est basée sur une estimation de celle-ci dans le pire cas. Ce type de technique conduit à des estimations pessimistes et

peut se traduire par la présence de bits non utilisés au niveau de la partie entière des données. En conséquence, pour les architectures figées, la précision des calculs au sein du processeur est diminuée. Ainsi, il est nécessaire de définir de nouvelles techniques d'estimation de la dynamique des données plus précises et aussi robustes. Une méthode fusionnant les résultats des estimations obtenues par les approches analytiques et statistiques peut être envisagée.

Au sein de notre méthodologie, les phases de détermination et d'optimisation du format des données minimisent le temps d'exécution du code tant que la contrainte de précision est respectée. Des métriques différentes du temps d'exécution, peuvent être utilisées au sein de ces deux processus d'optimisation. En particulier, dans le cadre des systèmes embarqués, la taille du code et la consommation d'énergie sont deux facteurs à minimiser. L'intégration de ces métriques au sein du processus d'optimisation nécessite de définir un modèle d'estimation de la taille globale du code ou de la consommation d'énergie totale. Dans le cadre de la minimisation de la taille du code, un modèle d'estimation basé sur la somme de la taille de chaque instruction peut être utilisé. Pour l'optimisation de la consommation d'énergie au sein des processeurs DSP, des modèles d'estimation à différents niveau d'abstraction ont été proposés [53].

L'utilisation de l'infrastructure de compilation reciblable CALIFE et d'une modélisation du processeur relativement simple permet d'envisager l'usage de notre méthodologie dans un contexte d'exploration architecturale. L'objectif de cette approche est de paramétrer un cœur de DSP ou d'optimiser l'architecture d'un ASIP à la classe des applications ciblées. L'architecture de l'unité de traitement du processeur va être optimisée en fonction des contraintes associées à l'application. Le concepteur peut intervenir sur les différents éléments de l'architecture influençant le temps d'exécution ou la précision des traitements. Ainsi, la largeur des bus et des opérateurs, le nombre de bits de garde présents au niveau des additionneurs et le nombre de registres peuvent être modifiés. De plus, certains opérateurs tels que les registres à décalage ou les opérateurs d'arrondi ou SWP peuvent insérés.

Une des perspectives de ce travail de recherche est de définir et de mettre en œuvre une méthodologie dans le cadre de l'implantation d'algorithmes de traitement du signal dans les architectures matérielles telles que les ASIC ou les FPGA. Pour ce type d'implantation, l'objectif est d'optimiser la surface et la consommation d'énergie du circuit sous contrainte de précision. Cette optimisation est réalisée à travers la détermination de la largeur des opérateurs de l'architecture. Pour les méthodologies d'implantation logicielle et matérielle, différentes phases de conversion en virgule fixe sont identiques. Ces phases correspondent à la détermination de la dynamique, la définition de la position de la virgule des données et à l'évaluation de la précision. L'approche proposée pour évaluer la précision permet d'envisager des temps d'exécution du processus d'optimisation de la largeur des opérateurs, nettement plus faibles que ceux obtenus avec les méthodes basées sur la simulation [62, 132, 57]. Lorsque la position de la virgule des données et l'expression du RSBQ sont déterminées, la surface du circuit est minimisée tant que la contrainte de précision est satisfaite. Une technique différente de celle mise en œuvre pour l'optimisation du type des données au sein de notre méthodologie doit être utilisée. En effet, dans le contexte d'une implantation matérielle, le nombre de valeurs possibles pour chaque variable du processus d'optimisation est nettement plus élevé.

Annexe A

Exemple de résultats obtenus avec l'outil de détermination du RSBQ

Dans cette partie, les résultats obtenus avec l'outil de détermination de l'expression du RSBQ sont exposés. Nous présentons, plus particulièrement, les différentes fonctions de transfert déterminées par l'outil dans le cadre d'un filtre récursif.

A.1 Filtre IIR d'ordre 4 cascadé

A.1.1 Spécifications

Dans cette partie, nous considérons un filtre à réponse impulsionnelle infinie d'ordre 4. Ce filtre est implanté avec deux cellules d'ordre 2 cascadées. La fonction de transfert $H(z)$ de ce filtre est la suivante :

$$H(z) = H_1(z).H_2(z) = \frac{b_{10} + b_{11}z^{-1} + b_{12}z^{-2}}{1 - a_{11}z^{-1} - a_{12}z^{-2}} \cdot \frac{b_{20} + b_{21}z^{-1} + b_{22}z^{-2}}{1 - a_{21}z^{-1} - a_{22}z^{-2}} \quad \text{(A.1)}$$

Les deux cellules sont implantées à l'aide d'une forme directe I. Le synoptique du filtre est présenté à la figure A.1.

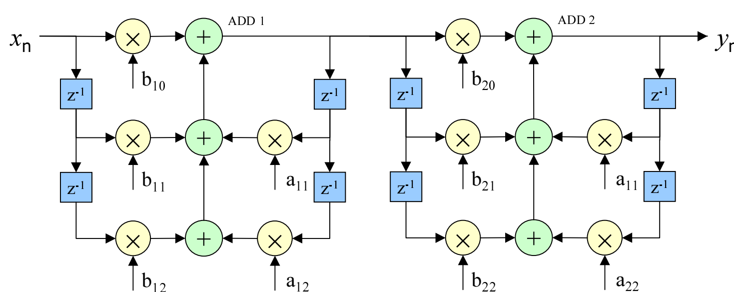


FIG. A.1: Synoptique du filtre IIR 4

A.1.2 Transformation $T_{21} - T_{22}$: graphe G_{eq}

Le graphe G_{eq} de ce filtre est présenté à la figure A.2. Soit y', la sortie du filtre représentant la partie *bruit*. Soient x et x', respectivement la partie *signal* et la partie *bruit* de l'entrée du filtre. Deux sources de bruit b_{g_1} et b_{g_2} sont présentes au sein du filtre. Elles correspondent au changement de format en sortie des additionneurs ADD1 et ADD2.

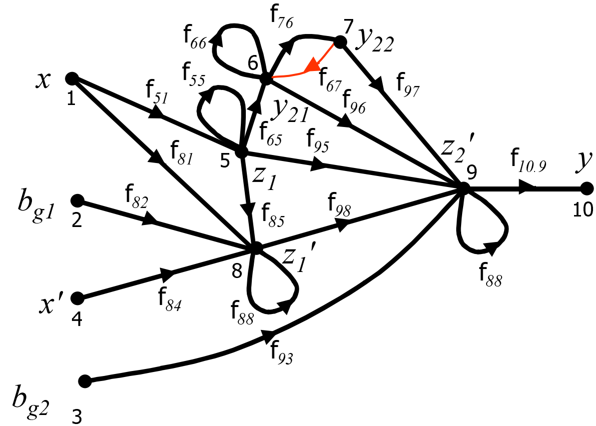


FIG. A.2: Graphe G_{eq} du filtre IIR

A.1.3 Transformation $T_{23} - T_{24}$: graph A_H

L'arbre G_{Hi} associé à ce filtre est présenté à la figure A.3. Les différentes fonctions de transfert globales sont présentées ci-dessous :

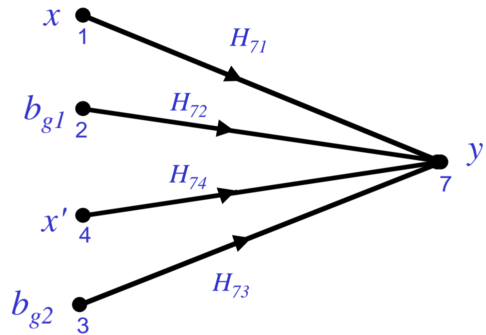


FIG. A.3: Arbre A_H du filtre IIR

Fonction de transfert entre la sortie du filtre $Y'(z)$ et la source de bruit $B_{g1}(z)$:

$$H_{10.2}(z) = \frac{Y'(z)}{B_{g1}(z)} = H_{10.9}(z)H_{98}(z)H_{82}(z) \tag{A.2}$$

$$H_{10.2}(z) = \frac{1}{1 - a_{11}z^{-1} - a_{12}z^{-2}}.H_2(z) \tag{A.3}$$

Fonction de transfert entre la sortie du filtre $Y'(z)$ et la source de bruit $B_{g2}(z)$:

$$H_{10.3}(z) = \frac{Y'(z)}{B_{g2}(z)} = H_{10.9}(z)H_{93}(z) \tag{A.4}$$

$$H_{10.3}(z) = \frac{1}{1 - a_{21}z^{-1} - a_{22}z^{-2}} \tag{A.5}$$

Fonction de transfert entre la sortie du filtre $Y'(z)$ et l'entrée du filtre $X(z)$:

$$\begin{aligned} H_{10.1}(z) = \frac{Y'(z)}{X(z)} &= H_{10.9}(z)H_{95}(z)H_{51}(z) + \\ & H_{10.9}(z)H_{96}(z)H_{65}(z)H_{51}(z) + \\ & H_{10.9}(z)H_{97}(z)H_{76}(z)H_{65}(z)H_{51}(z) + \\ & H_{10.9}(z)H_{98}(z)H_{81}(z) + \\ & H_{10.9}(z)H_{98}(z)H_{85}(z)H_{51}(z) \end{aligned} \tag{A.6}$$

$$\begin{aligned} H_{10.1}(z) &= H_1(z)\Big(\frac{b'_{20} + b'_{21}z^{-1} + b'_{22}z^{-2}}{1 - a_{21}z^{-1} - a_{22}z^{-2}} + H_2(z)\frac{a'_{21}z^{-1} + a'_{22}z^{-2}}{1 - a_{21}z^{-1} - a_{22}z^{-2}}\Big) + \\ & H_2(z)\Big(\frac{b'_{10} + b'_{11}z^{-1} + b'_{12}z^{-2}}{1 - a_{11}z^{-1} - a_{12}z^{-2}} + H_{1(z)}.\frac{a'_{11}z^{-1} + a'_{12}z^{-2}}{1 - a_{11}z^{-1} - a_{12}z^{-2}}\Big) \end{aligned} \tag{A.7}$$

Fonction de transfert entre la sortie du filtre $Y'(z)$ et le bruit associé à l'entrée du filtre $X'(z)$:

$$H_{10.4}(z) = \frac{Y'(z)}{X'(z)} = H_{10.9}(z)H_{98}(z)H_{84}(z) \tag{A.8}$$

$$H_{10.4}(z) = H_1(z)H_2(z) \tag{A.9}$$

Annexe B

Exemple de génération d'un code C virgule fixe

B.1 Code virgule flottante

Le code C du filtre FIR composé de 128 cellules est présenté ci-dessous.

```
float h[128] = { -0.01129701633153,   ... 0.10071514234281,   0.10467901971720,
                  0.10467901971720,  0.10071514234281,   ... -0.01129701633153 };

void main()
{
 float x[128];
 float in[1];
 float acc;
 float y;

 int i;

    x[0] = in[0];

    acc = x[0]*h[0] ;

    for(i=127; i>0; i--)
    {
       acc = acc + x[i]*h[i];
        x[i] = x[i-1];
    }

    y = acc;
}
```

B.2 Code virgule fixe

Le code C obtenu après la phase de conversion en virgule fixe est présenté ci-dessous. La spécification en virgule fixe associée à ce code correspond à celle obtenue en sortie de la phase de détermination du type des données et avant l'optimisation du placement des opérations de recadrage. Pour cette expérimentation le processeur cible est un DSP 16 bits ne possédant pas de bit de garde.

```
/*
 * This file was created automatically from SUIF
 *   on Mon Oct 21 18:14:59 2002.
 *
 * Created by:
 * s2c 1.3.0.1 compiled Thu Jan 3 11:39:46 MET 2002
 */


extern short h[128];
```

```
extern void main();

short h[128] =  {(short)-2961, ...  (short)26401, (short)27440,
                 (short)27440, (short)26401, ...  (short)-2961};

extern void main()
  {
    short x[128];
    short in[1];
    int acc;
    short y;
    int i;

    *x = *in;
    acc = *x * *h >> 4;
    for(i=127; i>0; i--)
    {
        acc = acc + (x[i] * h[i] >> 4);
        x[i] = x[i - 1];
    }
    y = (short)acc;
    return;
  }
```

Annexe C

Estimation du canal et synchronisation

C.1 Estimation de l'amplitude complexe du canal

Le signal $s_1(n)$ est traité de manière à obtenir les bits de référence (bits pilotes) pour réaliser l'estimation du canal. Ces bits de référence, présents au sein des bits de contrôle, se trouvent non seulement sur la voie imaginaire, mais aussi sur la voie réelle car la réponse impulsionnelle du canal est complexe. Ainsi, les deux voies sont traitées de la même manière pour extraire ces bits. Le synoptique du traitement pour l'estimation de canal est présenté à la figure C.1.

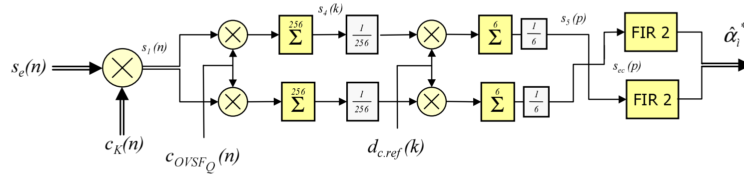


FIG. C.1: Synoptique du traitement pour l'estimation de canal

La partie réelle et la partie imaginaire de $s_1(n)$ sont multipliées par le code c_{OVSF_Q} puis le résultat est accumulé sur la durée d'un symbole de contrôle (256 *chips*). Les échantillons en sortie de ces accumulateurs évoluent au rythme symbole ($T_s = 256.T_c$). L'expression du signal complexe $s_4(k)$ représentant la sortie des accumulateurs aux instants $k.T_s$ est la suivante :

$$Re\left(s_4(k.T_s)\right) = \sum_{m=256k}^{256(k+1)-1} Re\left(A_l.e^{j\theta_l}x(m.T_c)\right).c_{OVSF_Q}(m.T_c) \tag{C.1}$$

$$Im\left(s_4(k.T_s)\right) = \sum_{m=256k}^{256(k+1)-1} Im\left(A_l.e^{j\theta_l}x(m.T_c)\right).c_{OVSF_Q}(m.T_c) \tag{C.2}$$

Sachant que les codes c_{OVSF_Q} et c_{OVSF_I} sont orthogonaux, la sommation sur 256 symboles du produit $c_{OVSF_Q}.c_{OVSF_I}$ est nulle tandis que la sommation des produits $c_{OVSF_Q}.c_{OVSF_Q}$ est égale 256. Ainsi, l'expression de $s_4(k)$ est égale à :

$$Re(s_4(k)) = -A_l.256.\sin\theta.d_c(k) \tag{C.3}$$

$$Im(s_4(k)) = A_l.256.\cos\theta.d_c(k) \tag{C.4}$$

La suite du traitement est effectuée dans le but d'éliminer le terme correspondant aux bits de contrôle. Pour cela les bits de référence présents au sein de ces bits de contrôle sont utilisés. Pour la partie du *slot* correspondant aux bits de référence, les voies réelles et imaginaires de

$s_4(k)$ sont corrélées avec la séquence de bits de référence (bits pilotes) déterministe et connue au niveau du récepteur. Dans le cadre de l'application considérée, la séquence de référence de chaque *slot* est composée de 6 bits. L'expression du signal s_5 en sortie des corrélateurs est la suivante :

$$Re(s_5) = \frac{1}{6}\sum_{m=1}^{6} Re(s_4(mT_p)).d_{c.ref}(mT_p) = -A_l.\sin\theta \tag{C.5}$$

$$Im(s_5) = \frac{1}{6}\sum_{m=1}^{6} Im(s_4(mT_p)).d_{c.ref}(mT_p) = A_l.\cos\theta \tag{C.6}$$

Les voies réelles et imaginaires sont échangées afin d'obtenir le signal s_{ec} correspondant à l'estimation du conjugué de l'amplitude complexe du canal associé au trajet l pour un *slot* donné :

$$s_{ec} = A'_l.(\cos\theta - j\,\sin\theta) = A'_l e^{-j\,\theta} \tag{C.7}$$

Cette valeur estimée est moyennée avec la valeur estimée du *slot* précédent à l'aide d'un filtre FIR (bloc FIR2) composé de deux cellules dont les coefficients sont égaux à 0.5.

C.2 Synchronisation des codes

Les opérations de corrélation utilisées dans les parties précédentes requièrent une synchronisation précise entre le code généré en interne au sein du récepteur et le code contenu dans le signal transmis. Pour cela le récepteur effectue une synchronisation en deux temps. Dans un premier temps, une estimation du retard de chaque trajet avec une précision de $\pm T_c/2$ est réalisée. Chaque trajet est caractérisé par un maximum au niveau de la fonction d'autocorrélation $\varphi_{CS}(\tau)$ entre le signal reçu et le code généré en interne au sein du récepteur. Ces maximums sont recherchés de manière séquentielle sur des fenêtres temporelles de largeur T_c. Ensuite, les L multi-trajets dont les puissances sont les plus élevées, sont affectés à un *finger* afin de réaliser le décodage des données. Cette opération nécessite une synchronisation fine dont la précision est inférieure au temps *chip* T_c. Ainsi, une boucle d'asservissement de retard (DLL : *Delay Locked Loop*) est mise en œuvre pour synchroniser le code généré en interne. La structure de la boucle de poursuite pour les systèmes à étalement de spectre à séquence directe utilisant une modulation de phase, est présentée à la figure C.2.

Le début du traitement appliqué aux trois lignes est identique au traitement utilisé pour l'estimation du canal. Ce traitement réalise la multiplication complexe avec le code de Kasami, puis la multiplication avec le code $OVSF_Q$ et l'accumulation du résultat sur 256 échantillons. Les données de contrôle présentes au sein du signal s_4 sont éliminées en multipliant celui-ci par les bits de référence. Le module au carré du signal complexe résultant s_5, est calculé puis la composante basse fréquence de celui-ci est isolée à l'aide d'un filtre passe-bas. Ce dernier est implanté à l'aide d'un filtre à réponse impulsionnelle infinie du premier ordre dont la fonction de transfert est la suivante :

$$H(z) = \frac{b_0}{1 - a_1 z^{-1}} \qquad \text{avec } b_0 = 0.01 \text{ et } a_1 = 0.99 \tag{C.8}$$

Le signal d'erreur $s_{er}(k)$ est calculé de la manière suivante :

$$s_{er}(k) = \frac{y_{early} - y_{late}}{y_{on-time}} \tag{C.9}$$

Le signal de commande $s_{cse}(k)$ des sous-échantillonneurs permet de choisir parmi les 4 échantillons composant un *chip* celui qui sera traité. Sachant que ce signal de commande spécifie la

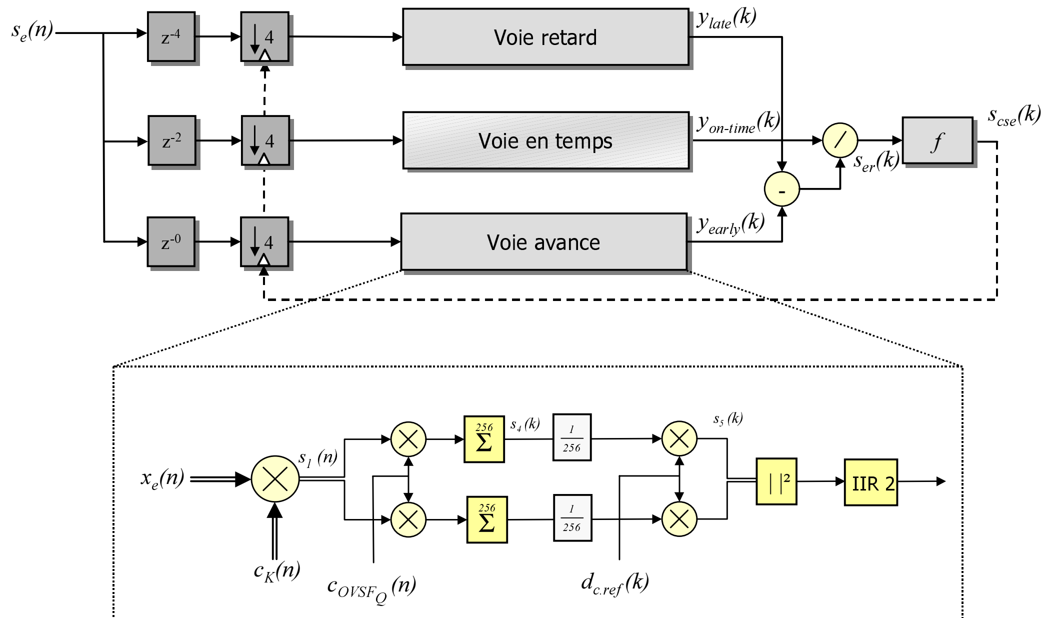


FIG. C.2: Synoptique de la boucle d'asservissement de retard

position de l'échantillon à traiter, ce signal doit être représenté par des entiers compris entre -2 et 2. Ce signal est proportionnel au signal d'erreur $s_{er}(k)$.

Annexe D

Code C des applications

Nous présentons dans cette annexe le code C d'un *rake receiver* pour un terminal mobile et pour une station de base. Ces récepteurs sont implantés sous la forme de filtres adaptés.

D.1 *Rake receiver* terminal mobile

Le code C d'un *rake receiver* destiné à un terminal mobile est présenté ci-dessous. Celui-ci est composé de 4 *fingers* et le facteur d'étalement est fixé à 4.

```
 /* Defintion des codes */

 float CksI[4] = {1,-1, 1,-1 };
 float CksQ[4] = {1,-1, 1,-1 };

 /* Definition des amplitudes complexes des trajets, elles       */
 /* sont considerees constantes pendant le traitement d'un slot */

 float a0I[1]= {0.99};
 float a0Q[1]= {0.99};
 float a1I[1]= {0.99};
 float a1Q[1]= {0.99};
 float a2I[1]= {0.99};
 float a2Q[1]= {0.99};
 float a3I[1]= {0.99};
 float a3Q[1]= {0.99};
 float a4I[1]= {0.99};
 float a4Q[1]= {0.99};

int main() {
int i;

 float x0I[4], x0Q[4];
 float x1I[4], x1Q[4];
 float x2I[4]; x2Q[4];
 float x3I[4], x3Q[4];
 float x4I[4], x4Q[4];

 float In0I[1], In0Q[1];
 float In1I[1], In1Q[1];
 float In2I[1], In2Q[1];
 float In3I[1], In3Q[1];
 float In4I[1], In4Q[1];

 float AccI, AccQ;
 float Acc0I, Acc0Q, Acc1I, Acc1Q, Acc2I, Acc2Q, Acc3I, Acc3Q, Acc4I, Acc4Q;
 float y0I, y0Q, y1I, y1Q, y2I, y2Q, y3I, y3Q, y4I, y4Q;
 float yn;

        /* Initialisation des entrees des filtres adaptes */

            x0I[0] = In0I[0];
```

```
        x0Q[0] = In0Q[0];

        x1I[0] = In1I[0];
        x1Q[0] = In1Q[0];

        x2I[0] = In2I[0];
        x2Q[0] = In2Q[0];

        x3I[0] = In3I[0];
        x3Q[0] = In3Q[0];

        x4I[0] = In4I[0];
        x4Q[0] = In4Q[0];


    /* Phase de desembrouillage et desetalement */

        Acc0I =   (x0I[0] * CksI[0] - x0Q[0] * CksQ[0]);
        Acc0Q =   (x0I[0] * CksQ[0] + x0Q[0] * CksI[0]);

        Acc1I =   (x1I[0] * CksI[0] - x1Q[0] * CksQ[0]);
        Acc1Q =   (x1I[0] * CksQ[0] + x1Q[0] * CksI[0]);

        Acc2I =   (x2I[0] * CksI[0] - x2Q[0] * CksQ[0]);
        Acc2Q =   (x2I[0] * CksQ[0] + x2Q[0] * CksI[0]);

        Acc3I =   (x3I[0] * CksI[0] - x3Q[0] * CksQ[0]);
        Acc3Q =   (x3I[0] * CksQ[0] + x3Q[0] * CksI[0]);

        Acc4I =   (x4I[0] * CksI[0] - x4Q[0] * CksQ[0]);
        Acc4Q =   (x4I[0] * CksQ[0] + x4Q[0] * CksI[0]);

    for (i = 3; i > 0; i--)
    {
        Acc0I = Acc0I + (x0I[i] * CksI[i] - x0Q[i] * CksQ[i]);
        Acc0Q = Acc0Q + (x0I[i] * CksQ[i] + x0Q[i] * CksI[i]);

        x0I[i] = x0I[i-1];
        x0Q[i] = x0Q[i-1];

        Acc1I = Acc1I + (x1I[i] * CksI[i] - x1Q[i] * CksQ[i]);
        Acc1Q = Acc1Q + (x1I[i] * CksQ[i] + x1Q[i] * CksI[i]);

        x1I[i] = x1I[i-1];
        x1Q[i] = x1Q[i-1];

        Acc2I = Acc2I + (x2I[i] * CksI[i] - x2Q[i] * CksQ[i]);
        Acc2Q = Acc2Q + (x2I[i] * CksQ[i] + x2Q[i] * CksI[i]);

        x2I[i] = x2I[i-1];
        x2Q[i] = x2Q[i-1];

        Acc3I = Acc3I + (x3I[i] * CksI[i] - x3Q[i] * CksQ[i]);
        Acc3Q = Acc3Q + (x3I[i] * CksQ[i] + x3Q[i] * CksI[i]);

        x3I[i] = x3I[i-1];
        x3Q[i] = x3Q[i-1];

        Acc4I = Acc4I + (x4I[i] * CksI[i] - x4Q[i] * CksQ[i]);
        Acc4Q = Acc4Q + (x4I[i] * CksQ[i] + x4Q[i] * CksI[i]);

        x4I[i] = x4I[i-1];
        x4Q[i] = x4Q[i-1];
    }

y0I = Acc0I;
y0Q = Acc0Q;
y1I = Acc1I;
y1Q = Acc1Q;
y2I = Acc2I;
y2Q = Acc2Q;
y3I = Acc3I;
y3Q = Acc3Q;
```

```
    y4I = Acc4I;
    y4Q = Acc4Q;

    /* Remise en phase et combinaison des trajets */

    AccI = a0I[0]*y0I - a0Q[0]*y0Q + a1I[0]*y1I - a1Q[0]*y1Q + a2I[0]*y2I - a2Q[0]*y2Q
         + a3I[0] *y3I - a3Q[0] *y3Q + a4I[0] *y4I - a4Q[0] *y4Q;

    AccQ = a0Q[0]*y0I + a0I[0]*y0Q + a1Q[0]*y1I + a1I[0]*y1Q + a2Q[0]*y2I + a2I[0]*y2Q
         + a3Q[0] *y3I + a3I[0] *y3Q + a4Q[0] *y4I + a4I[0] *y4Q;

    yn = Acc;
}
```

D.2 *Rake receiver* station de base

Le code C d'un *rake receiver* destiné à une station de base est présenté ci-dessous. Celui-ci est composé de 4 *fingers* et le facteur d'étalement est fixé à 8.

```
 /* Defintion des codes de Kasami */

 float CksI[8] = {1,-1, 1,-1, 1,-1, 1,-1 };
 float CksQ[8] = {1,-1, 1,-1, 1,-1, 1,-1 };

  /* Defintion du code OVSF */

 float Co[8]   = {1,-1, 1,-1, 1,-1, 1,-1 };

 /* Definition des amplitudes complexes des trajets, elles       */
 /* sont considerees constantes pendant le traitement d'un slot */

 float a0I[1]= {0.99};
 float a0Q[1]= {0.99};
 float a1I[1]= {0.99};
 float a1Q[1]= {0.99};
 float a2I[1]= {0.99};
 float a2Q[1]= {0.99};
 float a3I[1]= {0.99};
 float a3Q[1]= {0.99};

int main() { int i;

 float x0I[8], x0Q[8];
 float x1I[8], x1Q[8];
 float x2I[8]; x2Q[8];
 float x3I[8], x3Q[8];
 float x4I[8], x4Q[8];

 float In0I[1], In0Q[1];
 float In1I[1], In1Q[1];
 float In2I[1], In2Q[1];
 float In3I[1], In3Q[1];
 float In4I[1], In4Q[1];

 float Acc0I, Acc0Q, Acc1I, Acc1Q, Acc2I, Acc2Q, Acc3I, Acc3Q;
 float AccI, AccQ;

 float   y0I[8], y0Q[8];
 float   y1I[8], y1Q[8];
 float   y2I[8], y2Q[8];
 float   y3I[8], y3Q[8];

 float z0I, z1I, z2I, z3I, z0Q, z1Q, z2Q, z3Q;
 float yn;

        /* Initialisation des entrees des filtres adaptes */

           x0I[0] = In0I[0];
           x0Q[0] = In0Q[0];
```

```
    x1I[0] = In1I[0];
    x1Q[0] = In1Q[0];

    x2I[0] = In2I[0];
    x2Q[0] = In2Q[0];

    x3I[0] = In3I[0];
    x3Q[0] = In3Q[0];

   /* Phase de desembrouillage, de remise en phase et desetalement */

    y0I[0] = x0I[0] * CksI[0] - x0Q[0] * CksQ[0];
    y0Q[0] = x0I[0] * CksQ[0] + x0Q[0] * CksI[0];
    Acc0I = (y0I[0] * a0I[0]  - y0Q[0] * a0Q[0])* Co[0];
    Acc0Q = (y0I[0] * a0Q[0]  + y0Q[0] * a0I[0])* Co[0];

    y1I[0] = x1I[0] * CksI[0] - x1Q[0] * CksQ[0];
    y1Q[0] = x1I[0] * CksQ[0] + x1Q[0] * CksI[0];
    Acc1I = (y1I[0] * a1I[0]  - y1Q[0] * a1Q[0])* Co[0];
    Acc1Q = (y1I[0] * a1Q[0]  + y1Q[0] * a1I[0])* Co[0];

    y2I[0] = x2I[0] * CksI[0] - x2Q[0] * CksQ[0];
    y2Q[0] = x2I[0] * CksQ[0] + x2Q[0] * CksI[0];
    Acc2I = (y2I[0] * a2I[0]  - y2Q[0] * a2Q[0])* Co[0];
    Acc2Q = (y2I[0] * a2Q[0]  + y2Q[0] * a2I[0])* Co[0];

    y3I[0] = x3I[0] * CksI[0] - x3Q[0] * CksQ[0];
    y3Q[0] = x3I[0] * CksQ[0] + x3Q[0] * CksI[0];
    Acc3I = (y3I[0] * a3I[0]  - y3Q[0] * a3Q[0])* Co[0];
    Acc3Q = (y3I[0] * a3Q[0]  + y3Q[0] * a3I[0])* Co[0];

for (i = 7; i > 0; i--)
{
    y0I[i] = x0I[i] * CksI[i] - x0Q[i] * CksQ[i];
    y0Q[i] = x0I[i] * CksQ[i] + x0Q[i] * CksI[i];
    Acc0I = Acc0I + ( y0I[i]*a0I[0] -   y0Q[i]*a0Q[0] )* Co[i];
    Acc0Q = Acc0Q + ( y0I[i]*a0Q[0] +   y0Q[i]*a0I[0] )* Co[i];

    x0I[i] = x0I[i-1];
    x0Q[i] = x0Q[i-1];

    y1I[i] = x1I[i] * CksI[i] - x1Q[i] * CksQ[i];
    y1Q[i] = x1I[i] * CksQ[i] + x1Q[i] * CksI[i];
    Acc1I = Acc1I + ( y1I[i]*a1I[0] -   y1Q[i]*a1Q[0] )* Co[i];
    Acc1Q = Acc1Q + ( y1I[i]*a1Q[0] +   y1Q[i]*a1I[0] )* Co[i];

    x1I[i] = x1I[i-1];
    x1Q[i] = x1Q[i-1];

    y2I[i] = x2I[i] * CksI[i] - x2Q[i] * CksQ[i];
    y2Q[i] = x2I[i] * CksQ[i] + x2Q[i] * CksI[i];
    Acc2I = Acc2I + ( y2I[i]*a2I[0] -   y2Q[i]*a2Q[0] )* Co[i];
    Acc2Q = Acc2Q + ( y2I[i]*a2Q[0] +   y2Q[i]*a2I[0] )* Co[i];

    x2I[i] = x2I[i-1];
    x2Q[i] = x2Q[i-1];

    y3I[i] = x3I[i] * CksI[i] - x3Q[i] * CksQ[i];
    y3Q[i] = x3I[i] * CksQ[i] + x3Q[i] * CksI[i];
    Acc3I = Acc3I + ( y3I[i]*a3I[0] -   y3Q[i]*a3Q[0] )* Co[i];
    Acc3Q = Acc3Q + ( y3I[i]*a3Q[0] +   y3Q[i]*a3I[0] )* Co[i];

    x3I[i] = x3I[i-1];
    x3Q[i] = x3Q[i-1];
}

z0I = Acc0I;
z1I = Acc1I;
z2I = Acc2I;
z3I = Acc3I;
z0Q = Acc0Q;
z1Q = Acc1Q;
z2Q = Acc2Q;
```

```
        z3Q = Acc3Q;

        /* Combinaison des trajets */

        AccI = z0I + z1I + z2I + z3I;
        AccQ = z0Q + z1Q + z2Q + z3Q;

        yn = AccI;
}
```

Bibliographie

[1] 3DSP. *SP-5 Fixed-point Signal Processor Core.* 3DSP Corporation, July 1999.

[2] T. Aamodt. Floating-point To Fixed-point Compilation and Embedded Architectural Support. Master's thesis, University of Toronto, January 2001.

[3] T. Aamodt and P.Chow. Embedded ISA Support for Enhanced Floating-point To Fixed-point ANSI-C Compilation. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES-2000)*, San Jose, US, November 2000.

[4] ACE. DSP-C An extension to ISO/IEC IS 9899 :1990. Technical Report CoSy-8025P-dsp-c, ACE Associated Compiler Experts, October 1998.

[5] A.V. Aho and S.C. Johnson. Optimal code generation for expression trees. *Journal of ACM*, 23(3), July 1976.

[6] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers – principles, techniques, and tools.* AddisonWesley, 1986.

[7] Analog Device. *ADSP-2100 Family User's Manual*, 3 edition, September 1995.

[8] Analog Device. *TigerSHARC Hardware Specification.* Analog Device, December 1999.

[9] G. Araujo. *Code Generation Algorithms for Digital Signal Processors.* PhD thesis, Princeton University Department of EE, Princeton, May 1997.

[10] C. Barnes, B. N. Tran, and S. Leung. On the Statistics of Fixed-Point Roundoff Error. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 33(3), 1985.

[11] BDTi. Selecting and Designing with DSP Cores. In *6th International Conference on Signal Processing Applications and Technology (ICSPAT 95).* Miller Freeman, October 1995.

[12] BDTi. The BDTImark2000 : A Measure of DSP Execution Speed. Technical report, Berkeley Design Technology Inc, 2001.

[13] A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations : the SIGNAL language and its semantics. *Science of Computer Programming*, 16(2) :103–149, September 1991.

[14] J. Bier. DSP16xxx Targets Communications Apps. *Microprocessor Report*, 11(12), September 1997.

[15] A. Buisson. *Implémentation effiace d'un codeur vidéo hiérarchique granulaire sur une architecture multi-pentium.* PhD thesis, Université de Rennes I, Décembre 2002.

[16] D. Cachera and T. Risset. Advances in bit width selection methodology. In *The IEEE International Conference on Application-Specific Systems, Architectures and Processors, 2002* , pages 381 –390, San Jose, California, July 2002.

[17] J.P. Calvez. *Spécification et conception des systèmes.* Masson, 3 ème edition, 1992.

[18] C. Caraiscos and B. Liu. A Roundoff Erro Analysis of the LMS Adaptive Algorithm. *IEEE Transactions on Acoustics, Speech and Signal Processing*, ASSP-32(1) :34–41, February 1984.

[19] G.J. Chaitin. Register allocation and spilling via graph coloring. In *SIGPLAN Conference Programming Language Design and Implementation*, 1982.

[20] F. Charot and F. Djieya. Approche d'estimation flexible de performances pour DSP. In *Septième Symposium en Architectures nouvelles de machines (SYMPA 2001)*, Paris, Avril 2001.

[21] F. Charot, F. Djieya, and C. Wagner. Retargetable Compilation In The Service Of Interactive ASIP Design. Technical Report 1173, IRISA, Rennes, November 2000.

[22] F. Charot, G. Le Fol, and V. Messé. Programmable Processor Modeling for Retargetable Compiler Design and Architecture Exploration. Technical Report 1167, IRISA, Rennes, January 1998.

[23] R. Cmar, L. Rijnders, P. Schaumont, and I. Bolsens. A Methodology and Design Environment for DSP ASIC Fixed Point Refinement. In *Proceedings of the Design Automation and Test in Europe Conference (DATE 99)*, pages 271–276, Munich, 1999.

[24] G. Constantinides, P. Cheung, and W. Luk. Truncation Noise in Fixed-Point SFGs. *IEE Electronics Letters*, 35(23) :2012–2014, November 1999.

[25] G. Constantinides, P. Cheung, and W. Luk. Heuristic Datapath Allocation for Multiple Wordlength Systems. In *Design Automation and Test in Europe (DATE 01)*, pages 791–796, March 2001.

[26] M. Coors, H. Keding, O. Luthje, and H. Meyr. Integer Code Generation For the TI TMS320C62x. In *International Conference on Acoustics, Speech and Signal Processing 2001 (ICASSP 01)*, Sate Lake City, US, May 2001.

[27] L. De Coster, M. Ade, R. Lauwereins, and J.A. Peperstraete. Code Generation for Compiled Bit-True Simulation of DSP Applications. In *Proceedings of the 11th International Symposium on System Synthesis (ISSS 98)*, Taiwan, December 1998.

[28] F. De Coulon. *Théorie et traitement des signaux*, volume 6 of *Traité d'électricité*. Presses polytechniques et universitaires romandes, CH-1015 Lausanne, 3 edition, 1996.

[29] R. David, D. Chillet, S. Pillement, and O. Sentieys. DART A Dynamically Reconfigurable Architecture dealing with Next Generation Telecommunications Constraints. In *Reconfigurable Architecture Workshop (RAW 02)*, April 2002.

[30] R. David, D. Chillet, S. Pillement, and O. Sentieys. *SOC Design Methodologies*, chapter A Dynamically Reconfigurable Architecture for Low-Power Multimedia Terminals, pages 51–62. Kluwer Academic Publishers, 2002.

[31] E. Dinan, B. Jabbari, and G. Mason. Spreading Codes for Direct sequence CDMA and Wideband CDMA Cellular Networks. *IEEE Communications Magazine*, pages 48–54, September 1998.

[32] D. Esftathiou, J. Fridman, and Z. Zvonar. Recent developpements in enabling technologies for the software-defined radio. *IEEE Communication Magazine*, pages 112–117, August 1999.

[33] ETSI. Universal Mobile Telecommunications System (UMTS) ; Physical channels and mapping of transport channels onto physical channls (FDD)(3G TS 25.211 version 3.3.0). Technical specification, June 2000.

[34] J. Eyre and J. Bier. Carmel Enables Customizable DSP. *Microprocessor Report*, 12(17), December 1998.

[35] J. Eyre and J. Bier. VLIW Architectures for DSP. In *10th International Conference on Signal Processing Applications and Technology (ICSPAT 99) - DSP World*, Orlando, November 1999. Miller freeman.

[36] J. Eyre and J. Bier. The Evolution of DSP Processors. *IEEE Signal Processing Magazine*, 17(2) :44–51, March 2000.

[37] J. Fridman. Sub-Word Parallelism in Digital Signal Processing. *IEEE Signal Processing Magazine*, 17(2) :27–35, March 2000.

[38] E. Gaudry. Estimation de la complexité algorithmique d'une chaine de traitement WCDMA en télécommunication mobile : application au processeur Lx. Master's thesis, DEA STIR, Enssat - Université de Rennes 1, Lannion, Septembre 2001.

[39] R. Gonzales. Xtensa : A Configurable and Extensible Processor. *IEEE Micro*, 20(2), March/April 2000.

[40] G. Goossens and al. Programmable Chips in Consumer Electronics and Telecommunications. In G. De Micheli and M SAmi, editors, *Hardware Software Co-Design*, pages 135–164. Kluwer Academic Publishers, 1996.

[41] G. Goossens and al. Embedded Software in Real-Time Signal Processing Systems : Design Technologies. *Proceedings of the IEEE*, 85(3) :436–453, March 1997.

[42] T. Grötker, E. Multhaup, and O.Mauss. Evaluation of HW/SW Tradeoffs Using Behavioral Synthesis. In *7th International Conference on Signal Processing Applications and Technology (ICSPAT 96)*, Boston, October 1996.

[43] P. Le Guernic, B. Chéron, T. Gautier, and C. Le Maire. Développer en langage Signal. *Annales des Télécommunications*, 46(1-2) :3–24, Janvier 1991.

[44] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming Real-Time Applications with Signal. *Proceedings of the IEEE*, 79(9) :1321–1336, September 1991.

[45] S. Hacker. Static Superscalar Design : A new architecture for the Tiger SHARC DSP Processor. http ://www.analog.com/publications/whitepapers/products/sharc.html.

[46] J. L. Hennessy and D. A. Patterson. *Achitecture des Ordinnateurs - Une approche quantitative.* Morgan Kaufmann Publishers, San Francisco, 2 edition, 1996.

[47] P. Hilfinger and J Rabaey. DSP Specification Using the Silage Language. In R.W. Brodersen, editor, *Anatomy of a Silicon Compiler*. Kluwer Academic Publishers, 1992.

[48] E. Horowitz and Sahni S. *Fundamentals Of Computer Algorithms*. W. H. Freeman Company, November 1990.

[49] E. Van Der Horst, W. Kloosterhuis, and J. van der Heyden. A C Compiler for the Embedded R.E.A.L. DSP Architecture. In *9th International Conference on Signal Processing Applications and Technology (ICSPAT 98)*, 1998.

[50] ITRS. International Technology Roadmap for Semiconductors 1999 Edition : Design. Technical report, International Technology Roadmap for Semiconductors, 1999.

[51] ITRS. Overall Roadmap Technology Characteristics. Technical report, International Technology Roadmap for Semiconductors, 2000.

[52] D. B. Johnson. Finding All the Elementary Circuits of a Directed Graph. *SIAM Journal on Computing*, 4(1) :77–84, March 1975.

[53] N. Julien, E. Senn, J. Laurent, and E. Martin. Power Estimation of a C algorithm on a VLIW Processor. In *Proceedings of the Workshop IEEE WCED*, May 2002.

[54] K. Kalliojärvi and J. Astola. Roundoff Errors in Block-Floating Point Systems. *IEEE Transactions on Signal Processing*, 44(4) :783–790, April 1996.

[55] J. Kang and W. Sung. Fixed-Point C Compiler for TMS320C50 Digital Signal Processor. In *International Conference on Acoustics, Speech and Signal Processing 1997 (ICASSP 97)*, 1997.

[56] R. Kearfott. Interval Computations : Introduction, Uses, and Resources. *Euromath Bulletin*, 2(1) :95–112, 1996.

[57] H. Keding, F. Hurtgen, M. Willems, and M. Coors. Transformation of Floating-Point into Fixed-Point Algorithms by Interpolation Applying a Statistical Approach. In *9th International Conference on Signal Processing Applications and Technology (ICSPAT 98)*, 1998.

[58] H. Keding, M. Willems, M. Coors, and H. Meyr. FRIDGE : A Fixed-Point Design And Simulation Environment. In *Design, Automation and Test in Europe 1998 (DATE-98)*, 1998.

[59] P. Kievits, E Lambers, C. Moerman, and R. Woudsma. R.E.A.L. DSP Technology for Telecom Baseband Processing. In *9th International Conference on Signal Processing Applications and Technology (ICSPAT 99)*, Toronto, September 1998. I, Miller Freeman Inc.

[60] P. Kievits and F. Vermeire. Next Generation R.E.A.L. DSP for Consumer Applications. In *10th International Conference on Signal Processing Applications and Technology (ICSPAT 99)*, Orlando, November 1999. Miller Freeman Inc.

[61] S. Kim, K. Kum, and W. Sung. Fixed-Point Optimization Utility for C and C++ Based Digital Signal Processing Programs. In *Workshop on VLSI and Signal Processing '95*, Osaka, Nov. 1995.

[62] S. Kim, K. Kum, and S. Wonyong. Fixed-Point Optimization Utility for C and C++ Based Digital Signal Processing Programs. *IEEE Transactions on Circuits and Systems II*, 45(11), November 1998.

[63] S. Kim and W. Sung. An Autoscaling Assembler for the TMS320C25. In *4th International Conference on Signal Processing Applications and Technology (ICSPAT 93)*, pages 543–552, Oct. 1993.

[64] S. Kim and W. Sung. A Floating-point to Fixed-point Assembly program Translator for the TMS 320C25. *IEEE Transactions on Circuits and Systems*, 41(11) :730–739, Nov. 1994.

[65] S. Kim and W. Sung. Fixed-Point-Simulation Utility for C and C++ Based Digital Signal Processing Programs. In *Twenty-eighth Annual Asilomar Conference on Signals, Systems, and Computer*, Oct. 1994.

[66] S. Kim and W. Sung. Fixed-Point Error Analysis and Word Length Optimization of 8x8 IDCT Architectures. *IEEE Transactions on Circuits and Systems for Video Technology*, 8(8) :935–940, December 1998.

[67] I. Kollár. The Noise Model of Quantization. In *Proc. 1st IMEKO TC4 Symposium*, pages 125–129, Como (Italy), June 19-21 1986.

[68] K. Kum, J. Kang, and W. Sung. A Floating-point to Fixed-point C Converter for Fixed-point Digital Signal Processors. In *Proc. of the Second SUIF Compiler Workshop*, Aug. 1997.

[69] K. Kum, J. Kang, and W. Sung. A Floating-Point to Integer C Converter with Shift Reduction for Fixed-Point Digital Signal Processors. In *Proceedings of the International Conference on Acoustics, Speech and Signal Processing (ICASSP 99)*, pages 2163–2166, 1999.

[70] K. Kum, J.Y. Kang, and W.Y. Sung. AUTOSCALER for C : An optimizing floating-point to integer C program converter for fixed-point digital signal processors. *IEEE Transactions on Circuits and Systems II - Analog and Digital Signal Processing*, 47 :840–848, September 2000.

[71] T. Kumura, D. Ishii, M. Ikekawa, and M. Yoshida. A Low-Power Programmable DSP Core Architecture For 3G Mobile Terminals. In *International Conference on Acoustics, Speech and Signal Processing 2001 (ICASSP 2001)*, Sate Lake City, May 2001.

[72] M. Kunt, M. Bellanger, F De Coulon, and C Guegen. *Techniques modernes de traitement numérique des signaux*, volume 1 of *Traitement de l'information*. Presses polytechniques et universitaires romandes et CNET-ENST, 1 edition, 1991.

[73] P. Landman and J. Rabaey. Architectural power analysis : the dual bit type method. *IEEE Transaction on VLSI systems*, 3(36) :173–187, June 1995.

[74] P. Landman and J. Rabaey. Activity-Sensitive Architectural Power Analysis. *IEEE Transaction on Computer Aided Design of Integrated Circuits and Systems*, 15(6) :571–587, June 1996.

[75] P. Lapsley. Fixed-Point DSP Processors : History and Trends. In *U.C. Berkeley (VLSI Signal Processing)*, April 1996.

[76] P. Lapsley, J. Bier, A. Shoham, and E. A. Lee. *DSP Processor Fundamentals : Architectures and Features*. Berkeley Design Technology, Inc, Fremont, CA, 1996.

[77] E. Lee. Programmable DSP Architectures : Part I. *IEEE ASSP Magazine*, pages 4–14, Oct. 1988.

[78] E. Lee. Programmable DSP Architectures : Part II. *IEEE ASSP Magazine*, pages 4–14, Janv. 1989.

[79] R. Leupers. *Code Optimization techniques for Embedded Processors*. Kluwer academic publishers, 2000.

[80] M. Levy. C compilers for DSP flex their muscles. *EDN*, June, 1997.

[81] M. Levy. 1999 DSP Architetcure directory. *EDN*, April 1999.

[82] Y. Li and S. Malik. Performance Analysis of Embedded Software Using Implicit Path Enumeration. In *Proceedings of the Design Automation Conference (DAC 95)*, pages 456–461. ACM Press, 1995.

[83] S. Liao. *Code Generation an Optimization for Embedded Digital Signal Processors*. PhD thesis, MIT, 1996.

[84] S. Liao and Al. Code Generation and Optimization Techniques for Embedded Digital Signal Processors. In G. De Micheli, editor, *Hardware-Software Co-Design : Application Domains and Design Technologies*. Kluwer Academic Publishers, 1996.

[85] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang. Code optimization techniques for embedded dsp microprocessors. In *Proceedings of the 32nd ACM/IEEE conference on Design automation conference*, pages 599–604. ACM Press, 1995.

[86] S. Lipshitz, R. Wannamaker, and J. Vanderkooy. Quantization and Dither : A Theorical Survey. *Journal of the Audio Engineering Society*, 40(5) :355–375, may 1992.

[87] B. Liu. Effect of Finite Word Length on the Accuracy of Digital Filters - A Review. *IEEE Transaction on Circuit Theory*, 18(6), November 1971.

[88] Lucent Technologies. *DSP16xx*. Lucent Technologies.

[89] Lucent Technologies, Motorola. *SC140 DSP Core Reference Manual*. Lucent Technologies, December 1999.

[90] V. Madisetti. *VLSI Digital Signal Processors, An introduction to Rapid Prototyping and Design Synthesis*. IEEE Press, Butterworth Heinemann, 1995.

[91] E. Martin, O.Sentieys, and J.L.Philippe. Synthèse Architecturale de Cœur de processeurs de Traitement du Signal. *Technique et Science Informatiques*, 13, 1994.

[92] E. Martin, J. Toureilles, and C. Nouet. Conception optimisée d'architectures en précision finie pour les applications de traitement du signal. *Traitement du Signal 2001*, 18(1) :47–56, 2001.

[93] P. Marwedel. Code Generation for Embedded Processors : an Introduction. In P. Marwedel and G. Goossens, editors, *Code Generation for Embedded Processors*. KluwerAcademic Publishers, 1995.

[94] S.J. Mason and H.J.Zimmermann. *Electronic Circuits,Signals and Systems*. J.Wiley and Sons Inc., 1960.

[95] P. Mateti and N. Deo. On algorithms for enumerating all circuits of a graph. *SIAM Journal on Computing*, 5(1) :90–99, March 1976.

[96] Mathworks. *System-Level Design Products for DSP and communications*, 2001.

[97] D. Menard. Méthodologies d'implantation d'algorithmes spécifiés en virgule flottante dans les architectures en virgule fixe. Technical report, LASTI-ENSSAT-Université de Rennes 1, Lannion, Décembre 2000.

[98] D. Menard. Precision Evaluation of Fixed-point Systems. Technical report, LASTI-ENSSAT-Université de Rennes 1, Lannion, Sepember 2001.

[99] D. Menard, D. Chillet, F.Charot, and O. Sentieys. Automatic Floating-point to Fixed-point Conversion for DSP Code Generation. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems 2002 (CASES 2002)*, Grenoble, October 2002.

[100] D. Menard, P. Quemerais, and O. Sentieys. Influence of fixed-point DSP architecture on computation accuracy. In *XI European Signal Processing Conference (EUSIPCO 2002)*, Toulouse, September 2002.

[101] D. Menard, T. Saidi, D. Chillet, and O. Sentieys. Implantation d'algorithmes spécifiés en virgule flottante dans les DSP virgule fixe. In *Huitième Symposium en Architectures nouvelles de machines (SYMPA 2002)*, Hammamet, Tunisie, Avril 2002.

[102] D. Menard, T. Saidi, D. Chillet, and O. Sentieys. Implantation d'algorithmes spécifiés en virgule flottante dans les DSP virgule fixe. *Sélectionné pour un numéro spécial de Technique et Science Informatiques*, publication fin 2003.

[103] D. Menard and O. Sentieys. Influence du modèle de l'architecture des DSPs virgule fixe sur la précision des calculs. In *Dix huitième colloque GRETSI sur le traitement du signal et des images*, Toulouse, Septembre 2001.

[104] D. Menard and O. Sentieys. A methodology for evaluating the precision of fixed-point systems. In *International Conference on Acoustics, Speech and Signal Processing 2002 (ICASSP 2002)*, Orlando, May 2002.

[105] D. Menard and O. Sentieys. Automatic Evaluation of the Accuracy of Fixed-point Algorithms. In *Design, Automation and Test in Europe 2002 (DATE-02)*, Paris, March 2002.

[106] V. Messé. *Production de compilateurs flexibles pour la conception de processeurs programmables spécialisés*. PhD thesis, Université de Rennes I, mars 1999.

[107] C. Moerman. Instruction Sets in DSP Architectures. In *10th International Conference on Signal Processing Applications and Technology (ICSPAT 99)*, Orlando, November 1999. Miller Freeman.

[108] C. Moerman, P. Kievits, E. Lambers, and R. Woudsma. R.E.A.L. DSP : Reconfigurable Embedded DSP Architecture for Low-Power/ Low-cost Applications. In *8th International Conference on Signal Processing Applications and Technology (ICSPAT 99)*, San Diego, 1997. Miller Freeman Inc.

[109] C. Moerman, R. Woudsma, and P. Kievits. Embedded DSP Technologies in Consumer applications. In *9th International Conference on Signal Processing Applications and Technology (ICSPAT 98)*, Toronto, September 1998. Miller Freeman.

[110] A. Molisch. *Wideband Wireless Digital Communication.* Prentice Hall, November 2000.

[111] Motorola. *DSP56000 24-Bit Digital Signal Processor User'S Manual.* Motorola, 1994.

[112] Y. Le Moullec, J.P. Diguet, and J.L. Philippe. Design-Trotter : a Multimedia Embedded Systems Design Space Exploration Tool. In *Proceedings of the IEEE Workshop on Multimedia Signal Processing (MMSP02)*, December 2002.

[113] J.M. Muller. *Arithmetique des Ordinateurs.* Masson, Paris, 1989.

[114] T. Ojanperä and R. Prasad. *WCDMA : Towards IP mobility and mobile internet.* Artech House Universal Personal Communications Series, 2002.

[115] Open SystemC Initiative. SystemC User's Guide (ver 2.0). Technical report, www.systemc.org, 2001.

[116] B. Ovadia and G. Wertheizer. PalmDSPCore - Dual MAC and Parallel Modular Architecture. In *10th International Conference on Signal Processing Applications and Technology (ICSPAT 99)*, Orlando, November 1999. Miller Freeman Inc.

[117] P. Papamichalis and J. So. Implementation of Fast Fourrier Transform Algorithms with the TMS32020. Technical Report SPRA122, Texas Instruments, 1989.

[118] J. Le Pape. Etude d'un filtrage de Nyquist optimum dans les communications mobiles WCDMA. Technical report, LASTI-ENSSAT, Lannion, Septembre 2002.

[119] T.W. Parks and C.S. Burrus. *Digital Filter Design.* Jhon Willey and Sons Inc, 1987.

[120] P. Paulin and al. Embedded Software in Real-Time Signal Processing Systems : Application and Architecture Trends. *Proceedings of the IEEE*, 85(3) :419–435, March 1997.

[121] A. Pegatoquet. *Méthode d'estimation de performance logicielle : application au développement rapide de code optimisé pour une classe de DSP.* PhD thesis, Université de Nice Sophia Antipolis, 1999.

[122] S. Pillement and O. Sentieys. Sous projet 2 : Méthode de spécification des applications. Rapport final du projet milpat, LASTI, Université Rennes 1, November 2001.

[123] D. Preece. Distributions of the Final Digits in Data. *The Statistician*, 30(1) :31–60, March 1981.

[124] J.M. Rabaey. Managing Power Dissipation in the Generation after Next Wireless Systems. In *2ème journées francophones d'études Faible Tension Faible Consommation FTFC'99*, Paris, France, 26-28 mai 1999.

[125] G. Saporata. *Probabilités, Analyse des données et Statistiques.* Editions Technip, 1990.

[126] O. Sentieys, S. Pillement, and D. Chillet. Behavioral ip specification and integration framework for high-level design reuse. In *ISQED 2002 : 3rd International Symposium on Quality Electronic Design*, March 2002.

[127] A. Sripad and D. L. Snyder. A Necessary and Sufficient Condition for Quantization Error to be Uniform and White. *IEEE Trans. ASSP.*, 25(5) :442–448, Oct. 1977.

[128] W. Strauss. Internet Audio-Beyond Mp3. Technical Report 1030, Foward Concepts, Tempe, Arizona, USA, May 2000.

[129] W. Strauss. Forward Concepts' Dsp Market Bulletin. Technical report, Foward Concepts, Tempe, Arizona, USA, March 2002.

[130] W. Strauss. VoIP and Packet Voice DSP Markets. Technical Report 2201, Foward Concepts, Tempe, Arizona, USA, April 2002.

[131] W. Sung and J. Kang. Fixed-Point C Language for Digital Signal Processing. In *Proceeding of the ASILOMAR Conference*, 1995.

[132] W. Sung and K. Kum. Simulation-Based Word-Length Optimization Method for Fixed-Point Digital Signal Processing Systems. *IEEE Transactions on Signal Processing*, 43(12), Dec. 1995.

[133] Synopsys. *Converting ANSI-C into Fixed-Point using CoCentric Fixe-Point Designer.* Synopsys Inc., April 2000.

[134] Tensilica. *Xtensa Product Brief*, 2002.

[135] Texas Instruments. *TMS320C5X User's Guide.* Texas Instruments, June 1998.

[136] Texas Instruments. *TMS320C54X Dsp Cpu And Peripherals Reference Set Volume I.* Texas Instruments, Dallas, January 1999.

[137] Texas Instruments. *TMS320C6000 Cpu And Instruction Set Reference Guide.* Texas Instruments, Dallas, Sept 1999.

[138] Texas Instruments. *TMS320C64x Technical Overview.* Texas Instruments, February 2000.

[139] J. C. Tiernan. An Efficient Search Algorithm to Find the Elementary Circuits of a Graph. *Communicarions of the ACM*, 13(12) :722–726, December 1970.

[140] I. T. Tokaji and C. Barnes. Roundoff Error Statistics for a Continuous Range of Multiplier Coefficients. *IEEE Transactions on Transactions on Circuits and Systems*, 34(1), 1987.

[141] J.M. Toureilles. *Conception d'architectures pour le traitement du signal en précision finie.* PhD thesis, Université de Rennes I, Janvier 1999.

[142] J. Turley and H. Hakkarainen. TI's New 'C6x DSP Screams at 1,600 MIPS. *Microprocessor Report*, 11(2), Frebruary 1997.

[143] L. Turner, D. Graham, and P. Denyer. The analysis and Implementation of Digital Filters Using a special Purpose CAD Tool. *IEEE Transaction On Education*, 32(3) :287–297, August 1989.

[144] VLSI Technology. *VVF 3500 DSPCore Rev. 1.2.* VLSI Technology, June 1998.

[145] B. Widrow. A Study of Rough Amplitude Quantization by Means of Nyquist Sampling Theory. *RE Trans. on Circuit Theory*, CT-3(4) :266–276, Dec. 1956.

[146] B. Widrow. Statistical Analysis of Amplitude Quantized Sampled-Data Systems. *Trans. AIEE, Part. II :Applications and Industry*, 79 :555–568, 1960.

[147] B. Widrow, I. Kollár, and M.-C. Liu. Statistical Theory of Quantization. *IEEE Trans. on Instrumentation and Measurement*, 45(2) :353–61, Apr. 1996.

[148] M. Willems, V. Bursgens, H. Keding, and H. Meyr. System Level Fixed-Point Design Based On An Interpolative Approach. In *Design Automation Conference 1997 (DAC 97)*, June 1997.

[149] M. Willems, V. Bursgens, and H. Meyr. FRIDGE : Floating-Point Programming of Fixed-Point Digital Signal Processors. In *8th International Conference on Signal Processing Applications and Technology (ICSPAT 97)*, 1997.

[150] M. Willems and H. Keding. FRIDGE : Fixed-point pRogrammIng DesiGn Environment. http ://www.ert.rwth-aachen.de/Projekte/Tools/FRIDGE/fridge.html, Sept. 1997.

[151] M. Willems and V. Zivojnovic. DSP-Compiler : Product Quality for Control Oriented Applications ? In *7th International Conference on Signal Processing Applications and Technology (ICSPAT 96)*, pages pp.752–756, Boston, October 1996. Miller Freeman.

[152] R. Wilson and al. SUIF : An Infrastructure for Research on Parallelizing and Optimizing Compilers. Technical Report CA 94305-4055, Computer Systems Laboratory, Stanford University, May 1994.

[153] O. Wolf and J. Bier. StarCore Launches First Architecture. *Microprocessor Report*, 12(14), October 1998.

[154] O. Wolf and J. Bier. TigerSHARC Sinks Teeth into VLIW. *Microprocessor Report*, 12(16), December 1998.

[155] P. Wong. Quantization Noise, Fixed-Point Multiplicative Roundoff Noise, and Dithering. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 38(2), 1990.

[156] P. Wong. Quantization and Roundoff Noises in Fixed-Point FIR Digital Filters. *IEEE Transactions on Signal Processing*, 39(7), 1991.

[157] K. Yarlagadda. Programmable DSP Architectures. In *DSP World*, Orlando, Florida, Nov. 1999. Miller Freeman.

[158] V. Zivojnovic and al. DSP Processor/Compiler Co-Design : A Quantitative Approach. In *Proceedings of the 9th International Symposium on System Synthesis (ISSS 96)*, La Jolla, November 1996.

[159] V. Zivojnovic, J. Martinez, C. Schläger, and H. Meyr. DSPStone : A DSP-Oriented Benchmarking Methodology. In *5th International Conference on Signal Processing Applications and Technology (ICSPAT 94)*, Dallas, October 1994. Miller Freeman.

[160] V. Zivojnovic, H. Schraut, M. Willems, and R. Schoenen. DSPs, GPPs, and multimedia applications : An evaluation using DSPStone. In *6th International Conference on Signal Processing Applications and Technology (ICSPAT 95)*, Boston, October 1995. Miller Freeman.

[161] U. Zölzer. *Digital Audio Signal Processing.* John Wiley and Sons, Aug 1997.

Publications et Rapports

Revue nationale

[1] D. Menard, T. Saidi, D. Chillet, and O. Sentieys. Implantation d'algorithmes spécifiés en virgule flottante dans les DSP virgule fixe. *Sélectionné pour un numéro spécial Architecture des Systèmes Embarqués de Techniques et Sciences Informatiques*, publication fin 2003.

Conférences internationales

[2] D. Menard and O. Sentieys. Automatic Evaluation of the Accuracy of Fixed-point Algorithms. In *Design, Automation and Test in Europe 2002 (DATE-02)*, Paris, March 2002.

[3] D. Menard and O. Sentieys. A methodology for evaluating the precision of fixed-point systems. In *International Conference on Acoustics, Speech and Signal Processing 2002 (ICASSP 2002)*, Orlando, May 2002.

[4] D. Menard, P. Quemerais, and O. Sentieys. Influence of fixed-point DSP architecture on computation accuracy. In *XI European Signal Processing Conference (EUSIPCO 2002)*, Toulouse, September 2002.

[5] D. Menard, D. Chillet, F.Charot, and O. Sentieys. Automatic Floating-point to Fixed-point Conversion for DSP Code Generation. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems 2002 (CASES 2002)*, Grenoble, October 2002.

[6] D. Menard, M. Guitton, S. Pillement and O. Sentieys, Design and Implementation of WCDMA Platforms : Challenges and Trade-offs In *International Signal Processing Conference*, April 03, Dallas.

Conférences nationales

[7] D. Menard and O. Sentieys. Influence du modèle de l'architecture des DSPs virgule fixe sur la précision des calculs. In 18ème *colloque GRETSI sur le traitement du signal et des images*, Toulouse, Septembre 2001.

[8] D. Menard, T. Saidi, D. Chillet, and O. Sentieys. Implantation d'algorithmes spécifiés en virgule flottante dans les DSP virgule fixe. In *Huitième Symposium en Architectures nouvelles de machines (SYMPA 2002)*, Hammamet, Tunisie, Avril 2002.

[9] D. Menard, M. Guitton, R. David, S. Pillement, O. Sentieys. Évaluation comparative de plates-formes reconfigurables et programmables pour les télécommunications de $3^{ème}$ génération. **Soumission à** $19^{ème}$ *colloque GRETSI sur le traitement du signal et des images*, Paris, Septembre 2003.

Rapports de recherche interne

[10] D. Menard. Méthodologies d'implantation d'algorithmes spécifiés en virgule flottante dans les architectures en virgule fixe. Technical report, LASTI-ENSSAT-Université de Rennes 1, Lannion, Décembre 2000.

[11] D. Menard. Precision Evaluation of Fixed-point Systems. Technical report, LASTI-ENSSAT-Université de Rennes 1, Lannion, Sepember 2001.

Résumé

L'implantation efficace des algorithmes de traitement numérique du signal (TNS) dans les systèmes embarqués requiert l'utilisation de l'arithmétique virgule fixe afin de satisfaire les contraintes de coût, de consommation et d'encombrement exigées par ces applications. Le codage manuel des données en virgule fixe est une tâche fastidieuse et source d'erreurs. De plus, la réduction du temps de mise sur le marché des applications exige l'utilisation d'outils de développement de haut niveau, permettant d'automatiser certaines tâches. Ainsi, le développement de méthodologies de codage automatique des données en virgule fixe est nécessaire. Dans le cadre des processeurs programmables de traitement du signal, la méthodologie doit déterminer le codage optimal, permettant de maximiser la précision et de minimiser le temps d'exécution et la taille du code. L'objectif de ce travail de recherche est de définir une nouvelle méthodologie de compilation d'algorithmes spécifiés en virgule flottante au sein d'architectures programmables en virgule fixe sous contrainte de respect des critères de qualité associés à l'application. Ce travail de recherche s'articule autour de trois points principaux.

Le premier aspect de notre travail a consisté à définir la structure de la méthodologie. L'analyse de l'influence de l'architecture sur la précision des calculs montre la nécessité de tenir compte de l'architecture cible pour obtenir une implantation optimisée d'un point de vue du temps d'exécution et de la précision. De plus, l'étude de l'interaction entre les phases de compilation et de codage des données permet de définir le couplage nécessaire entre les phases de conversion en virgule fixe et le processus de génération de code.

Le second aspect de ce travail de recherche concerne l'évaluation de la précision au sein d'un système en virgule fixe à travers la détermination du Rapport Signal à Bruit de Quantification (RSBQ). Une méthodologie permettant de déterminer automatiquement l'expression analytique du RSBQ en fonction du format des données en virgule fixe est proposée. Dans un premier temps, un nouveau modèle de bruit est présenté. Ensuite, les concepts théoriques pour déterminer la puissance du bruit de quantification en sortie des systèmes linéaires et des systèmes non-linéaires et non-récursifs sont détaillés. Finalement, la méthodologie mise en œuvre pour obtenir automatiquement l'expression du RSBQ dans le cadre des systèmes linéaires est exposée.

Le troisième aspect de ce travail de recherche correspond à la mise en œuvre de la méthodologie de codage des données en virgule fixe. Dans un premier temps, la dynamique des données est déterminée à l'aide d'une approche analytique combinant deux techniques différentes. Ces informations sur la dynamique permettent de déterminer la position de la virgule de chaque donnée en tenant compte de la présence éventuelle de bits de garde au sein de l'architecture. Pour obtenir un format des données en virgule fixe complet, la largeur de chaque donnée est déterminée en prenant en compte l'ensemble des types des données manipulées au sein du DSP. La méthode sélectionne la séquence d'instructions permettant de fournir une précision suffisante en sortie de l'algorithme et de minimiser le temps d'exécution du code. La dernière phase du processus de codage correspond à l'optimisation du format des données en vue d'obtenir une implantation plus efficace. Les différentes opérations de recadrage sont déplacées afin de minimiser le temps d'exécution global tant que la précision en sortie de l'algorithme est supérieure à la contrainte. Deux types de méthode ont été mis en œuvre en fonction des capacités de parallélisme au niveau instruction de l'architecture ciblée.

Cette méthodologie a été testée sur différents algorithmes de traitement numérique du signal présents au sein des systèmes de radio-communications de troisième génération. Les résultats obtenus montrent l'intérêt de notre méthodologie pour réduire le temps de développement des systèmes en virgule fixe.

Mots-clés : traitement du signal, adéquation algorithme architecture, arithmétique virgule fixe, précision finie, bruit de quantification, processeurs de traitement du signal, processeurs spécialisés, DSP, ASIP, compilation, génération de code, télécommunications, WCDMA.