Implantation d'algorithmes spécifiés en virgule flottante dans les DSP virgule fixe

Daniel Menard — Taofik Saïdi — Daniel Chillet — Olivier Sentieys

Équipe de recherche R2D2 - IRISA/INRIA
ENSSAT - Université de Rennes I
6, rue de Kérampont
F-22300 Lannion
{nom}@enssat.fr

RÉSUMÉ. *Les contraintes au niveau du coût, de la consommation et du temps de mise sur le marché des applications de traitement numérique du signal exigent la mise en œuvre de méthodologies d'implantation automatique d'algorithmes spécifiés en virgule flottante au sein d'architectures en virgule fixe. Dans cet article, une nouvelle méthodologie d'implantation au sein des processeurs de traitement du signal (DSP) sous contrainte de précision est définie. Par rapport aux méthodologies existantes, l'architecture du processeur DSP est entièrement prise en compte pour l'optimisation du codage des données. Les justifications de la structure de cette méthodologie et les différentes étapes la composant sont détaillées.*

ABSTRACT. *The development of methodologies for the automatic implementation of floating-point algorithms in fixed-point architectures is required for the minimization of cost, power consumption and time-to-market of digital signal processing applications. In this paper, a new methodology of implementation in Digital Signal Processors (DSP) under accuracy constraint is presented. In comparison with the existing methodologies, the DSP architecture is completely taken into account for optimizing the execution time under accuracy constraint. The justification and the different stages of our methodology are presented.*

MOTS-CLÉS : *arithmétique à virgule fixe, précision finie, processeurs de traitement du signal, DSP, génération de code.*

KEYWORDS: *fixed-point arithmetic, finite precision, digital signal processors, DSP, code generation.*

1. Introduction

La majorité des applications de traitement numérique du signal est implantée au sein d'architecture en virgule fixe afin de satisfaire les contraintes de coût, de consommation et d'encombrement inhérentes aux systèmes embarqués [EYR 00, GOO 97]. En effet, la largeur des bus et des mémoires au sein des architectures virgule fixe étant plus faible, le prix et la consommation d'énergie de ces architectures sont moins importants. L'arithmétique à virgule flottante basée sur la norme IEEE-754 nécessite d'utiliser des données codées sur au moins 32 bits. A titre d'exemple, la majorité des processeurs de traitement numérique du signal programmables (DSP) virgule fixe possède une largeur naturelle nettement plus faible (16 bits). De plus, les opérateurs utilisant l'arithmétique à 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 à virgule flottante au sein d'architecture virgule fixe étant prohibitif (facteur 10 à 500 [WIL 97b]), il est nécessaire de coder l'ensemble des données en virgule fixe. Le codage manuel des données 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, des méthodologies de codage automatique des données en virgule fixe sont nécessaires car le codage manuel se révèle être un frein important à la diminution du temps de conception [GRÖ 96]. Pour illustrer ce processus de conversion en virgule fixe, un exemple est proposé à la figure 8.

L'utilisation de l'arithmétique à virgule fixe conduit à une dégradation de la précision des calculs réalisés. Pour satisfaire les performances associées à l'application, il est nécessaire que la spécification en virgule fixe implantée garantisse une précision minimale. Dans le cadre d'une implantation logicielle au sein de DSP, l'objectif de la méthodologie est d'optimiser le codage des données afin de minimiser le temps d'exécution et la taille du code sous contrainte de précision. Les méthodologies existantes [KUM 00, WIL 97b] réalisent la conversion en virgule fixe au niveau du code source sans prendre en considération l'architecture du processeur. L'analyse de l'influence de l'architecture et des différentes phases de la génération de code sur la précision des calculs [MEN 02b] 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. Ainsi, dans cet article, une nouvelle méthodologie d'implantation d'algorithmes de traitement signal spécifiés en virgule flottante au sein des architectures DSP virgule fixe sous contrainte de précision est présentée. Dans la section 2, un état de l'art des méthodologies existantes est réalisé. Ensuite, une nouvelle méthodologie est définie dans la section 3 et les différentes phases de celle-ci sont détaillées dans la section 4.

2. État de l'art des méthodologies

Dans cette partie, les deux principales méthodes existantes pour implanter automatiquement un algorithme spécifié en virgule flottante au sein d'un processeur program-

mable virgule fixe sont analysées. Ces méthodes réalisent la conversion en virgule fixe au niveau du code source.

La méthodologie FRIDGE [WIL 97a], à l'origine de l'outil *CoCentric Fixed-point Designer* (Synopsys Inc.), réalise une transformation du code C virgule flottante en un code C intégrant uniquement des types entiers. Dans une première étape appelée *annotations*, l'utilisateur définit à l'aide des types en virgule fixe disponibles au sein de *SystemC* [OPE 01], le format de quelques données. De plus des annotations globales peuvent être ajoutées afin de spécifier des caractéristiques pour l'ensemble de l'algorithme (largeur maximale des données, règles de *casting*). La seconde phase appelée *interpolation* [WIL 97a] correspond à la détermination du nombre de bits pour la partie entière et fractionnaire. Le format des données est obtenu à partir de règles prédéfinies et de l'analyse du flot de contrôle du programme. En sortie de cette phase, une spécification virgule fixe complète de l'application est obtenue. Cette description est simulée afin de vérifier que les contraintes de précision sont satisfaites. Dans le cas contraire, l'utilisateur modifie le format de certaines opérandes et le processus d'*interpolation* est réitéré. Un code C est généré à partir de la spécification en virgule fixe obtenue précédemment [WIL 97b]. Chaque donnée dont la largeur est quelconque est codée à l'aide d'un type supporté par le processeur. Afin que la spécification en virgule fixe associée au code C généré soit identique au bit près à celle d'origine, des opérations de quantification sont insérées au sein du code. Cette méthode permet de transformer un code C virgule flottante en une spécification en virgule fixe et de transcrire celle-ci en un code C n'utilisant que des types entiers. Cependant, l'insertion d'opérations de quantification augmente inutilement le temps d'exécution du code. De plus, cette méthodologie ne réalise pas de manière automatique l'optimisation du format des données en virgule fixe en vue de minimiser le temps d'exécution du code. Ce processus d'optimisation est à la charge de l'utilisateur qui modifie manuellement le format de certaines données au cours de la phase d'*annotations*.

L'outil présenté dans [KUM 00] a pour objectif de transformer un code C virgule flottante en un code C virgule fixe afin d'obtenir un code indépendant de la cible. Cette transformation intègre une optimisation du codage des données afin de minimiser le nombre de recadrages[1]. L'outil se décompose en deux modules. Le premier module correspond à la conversion de la description en virgule flottante en un code C virgule fixe. Elle est réalisée à l'aide de l'outil SUIF [WIL 94] qui permet de transformer le code C en une représentation intermédiaire. Tout d'abord, l'outil substitue le type entier au type flottant et ajoute ensuite les instructions de recadrage permettant de modifier le format des données. Les différents cadrages sont déterminés et optimisés par le second module. Finalement le code C virgule fixe est généré à partir de la représentation intermédiaire modifiée. Le second module définit le codage des données et les cadrages à réaliser. Il détermine tout d'abord la dynamique des différentes données à l'aide d'une méthode statistique [KIM 98]. La connaissance de la dynamique permet de définir le format de codage de l'ensemble des données et d'en déduire les différents recadrages nécessaires pour respecter les règles de l'arithmétique à virgule fixe.

1. Opération de décalage réalisée sur la donnée pour modifier son format (position de la virgule).

La réduction du nombre de recadrages est basée sur l'égalisation du format de données pertinentes permettant de minimiser le coût global des recadrages. Cette fonction de coût intègre le nombre d'exécutions de chaque expression et est fonction des capacités de décalage du processeur cible. Si un registre à décalage en barillet est présent, chaque décalage nécessite un cycle, sinon le nombre de cycles est égal au nombre de bits à décaler.

Cette méthodologie s'intègre bien dans notre problématique de recherche mais différents points faibles peuvent être soulignés. La méthode ne minimise pas le temps d'exécution du code avec une contrainte de précision globale en sortie de l'algorithme. La contrainte de précision est uniquement présente à travers la définition d'une dégradation maximale autorisée de la précision de chaque donnée et il est très difficile à l'utilisateur de déterminer la valeur de cette dégradation maximale en fonction des performances souhaitées en sortie de l'application. Le modèle utilisé pour déterminer le temps d'exécution des opérations de recadrage, ne conduit pas à des estimations réalistes. Celui-ci ne prend pas en considération les registres à décalage spécialisés, qui réalisent certains décalages sans cycle supplémentaire. La nécessité de prendre en compte l'architecture du processeur et le processus de génération de code pour estimer correctement le temps d'exécution des opérations de recadrage réalisées par un registre à décalage en barillet est montrée dans la partie suivante.

Pour ces deux méthodologies réalisant la conversion en virgule fixe avant la génération de code, l'architecture du processeur n'est que très partiellement prise en compte lors du processus de codage en virgule fixe. La nécessité de considérer le processus de compilation et l'architecture pour obtenir une implantation optimisée en termes de précision et de temps d'exécution est soulignée dans la partie suivante.

3. Définition d'une nouvelle méthodologie de conversion en virgule fixe

Dans cette partie, une nouvelle approche permettant d'implanter un algorithme spécifié en virgule flottante au sein d'un DSP virgule fixe sous contrainte de précision globale est proposée. La structure de cette méthodologie, présentée dans la section 3.3, a été définie après avoir étudié l'influence de l'architecture sur la précision des calculs et l'interaction entre les différentes phases de génération de code et de codage des données. Les éléments de l'architecture des DSP influençant la précision des calculs sont détaillés dans la section 3.1 et l'interaction entre les processus de codage des données et de compilation est résumée dans la section 3.2.

3.1. *Architecture des DSP*

L'architecture des DSP est conçue pour traiter efficacement les différentes opérations arithmétiques présentes au sein des applications de traitement du signal. Deux catégories de DSP peuvent être distinguées. Les DSP conventionnels sont basés sur un chemin de données spécialisé caractérisé par des registres dédiés aux unités fonction-

nelles. Pour ce type d'architecture, le parallélisme est encodé au sein de l'instruction. La seconde catégorie correspond aux processeurs possédant du parallélisme au niveau instruction, tels que les processeurs VLIW (*Very Long Instruction Word*). L'architecture de l'unité de traitement est homogène et est composée d'une file de registres et de différentes unités fonctionnelles pouvant travailler en parallèle. Le processeur exécute à chaque cycle une instruction globale correspondant à la mise en parallèle de plusieurs instructions partielles commandant chacune une unité fonctionnelle.

Différents éléments de l'unité de traitement des DSP, interviennent au niveau de la précision des calculs réalisés. Une analyse quantitative de l'influence de l'architecture sur la précision des calculs est présentée dans [MEN 02b]. L'élément principal déterminant la précision des calculs correspond à la largeur des données traitées. Celle-ci résulte d'un compromis entre la précision et le temps d'exécution de l'opération. Trois catégories d'instructions, correspondant aux instructions arithmétiques classiques, double-précision ou SWP (*Sub-Word Parallelism*), peuvent être distinguées. Les processeurs DSP possèdent des instructions arithmétiques classiques permettant de réaliser une multiplication suivie d'une addition sans perte d'information. Dans ce cas, la largeur des opérandes de l'additionneur et de sortie du multiplieur, est égale au double de la largeur des opérandes d'entrée du multiplieur. Cependant, l'accroissement de la dynamique des données lors d'accumulations successives peut engendrer des débordements. Ainsi, certains processeurs tels que le C54x [TEX 99] possèdent des bits de garde au sein de l'additionneur pour stocker les bits supplémentaires issus d'accumulations successives. La majorité des DSP propose des instructions double-précision permettant de manipuler des données stockées en mémoire avec une précision plus importante. Chaque instruction double-précision étant composée d'une suite d'instructions arithmétiques classiques, son temps d'exécution est plus important. Afin de diminuer le temps d'exécution du code, certains DSP récents tels que le C64x [TEX 00] et le TigerSharc [WOL 98], permettent l'exploitation du parallélisme présent au niveau des données. Ils intègrent des instructions SWP réalisant le traitement en parallèle de données dont la largeur est plus faible par rapport aux instructions classiques. Cette technique divise les opérateurs de largeur N afin de pouvoir exécuter en parallèle k opérations sur des fractions de mot de largeur N/k [FRI 00]. Ainsi, ces processeurs permettent de manipuler divers types de données.

La largeur des données codées au sein d'un DSP étant limitée, il est nécessaire de modifier le format de certaines d'entre elles afin de maintenir une précision maximale. Pour réaliser ces opérations de recadrage, différents types de registre à décalage sont disponibles. Au sein des premiers DSP conventionnels tels que le C50 [TEX 98], des registres à décalage dédiés pouvant réaliser quelques décalages prédéfinis sans cycle supplémentaire sont présents en entrée ou en sortie des opérateurs arithmétiques. Pour offrir plus de flexibilité, les générations de DSP suivantes intègrent un registre à décalage en barillet pouvant effectuer un décalage quelconque en un cycle d'instruction. Cependant, le coût réel de l'opération de recadrage dépend de l'architecture des DSP. Pour les DSP conventionnels tels que le C54x, le coût du recadrage dépend de la localisation de l'opérande à décaler au sein de l'unité de traitement. Ainsi, le transfert de l'opérande entre les registres peut augmenter fortement le temps d'exécution du reca-

drage. Pour les processeurs possédant du parallélisme au niveau instruction tels que le C64x ou le TigerSharc, le coût réel d'un recadrage dépend de l'opportunité d'exécuter cette opération en parallèle avec les autres instructions [MEN 02b].

Lors d'un changement de format, le DSP réalise par défaut une quantification par troncature. Pour éliminer le biais engendré par cette loi de quantification, certains DSP (C54x, TigerSharc) proposent l'utilisation d'une loi de quantification par arrondi.

Les caractéristiques des DSP C50, C54x, C64x utilisés dans les expérimentations présentées dans les parties suivantes sont résumées dans le tableau 1.

Processeurs	C50	C54x	C64x
Fréquence d'horloge (MHz)	50	160	600
Performances maximales (MIPS)	50	160	4800
Type d'architecture	conventionnel	conventionnel	VLIW
Nombre de bits de garde	0	8	0 / 8
Capacités de décalage	3 registres dédiés	1 registre en barillet	2 registres en barillet
Capacités SWP	non	addition	addition, décalage, multiplication
Types des données	16, 32	16, 32, 40	8, 16, 32, 40, 64
Loi de quantification par arrondi	non	oui	non

Tableau 1. *Caractéristiques des DSP C50, C54x, 64x*

3.2. *Interaction entre le codage des données et la génération de code*

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'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. La sélection d'instructions nécessite que le type (largeur des données) des opérandes d'entrée et de sortie de chaque opération arithmétique soit défini. Le report du choix des types des opérandes après la phase de sélection d'instructions rend nettement plus complexe les phases de génération de code. Ainsi, les types des données de l'application seront définis avant cette phase au niveau de la représentation intermédiaire.

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. Les résultats des calculs intermédiaires étant stockés avec une précision plus importante au sein de l'unité de traitement, leur renvoi en mémoire sera un compromis entre la précision et le temps de transfert de la donnée. La méthodologie doit déterminer le format des données renvoyées en mémoire en fonction de la précision souhaitée en sortie de l'algorithme.

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 ce code. Pour les processeurs possédant du parallélisme au niveau instruction (ILP), un compactage du code est réalisé afin de regrouper les instructions partielles pouvant s'exécuter en parallèle. Comme indiqué dans le paragraphe précédent, pour les processeurs possédant de l'ILP, le coût réel d'une opération de recadrage dépend de la manière dont les instructions sont ordonnancées et celui-ci ne peut être déterminé que lors de la phase d'ordonnancement. En conséquence, pour ce type de processeurs la phase d'optimisation des recadrages en vue de minimiser le temps d'exécution est réalisée en parallèle avec l'ordonnancement des instructions.

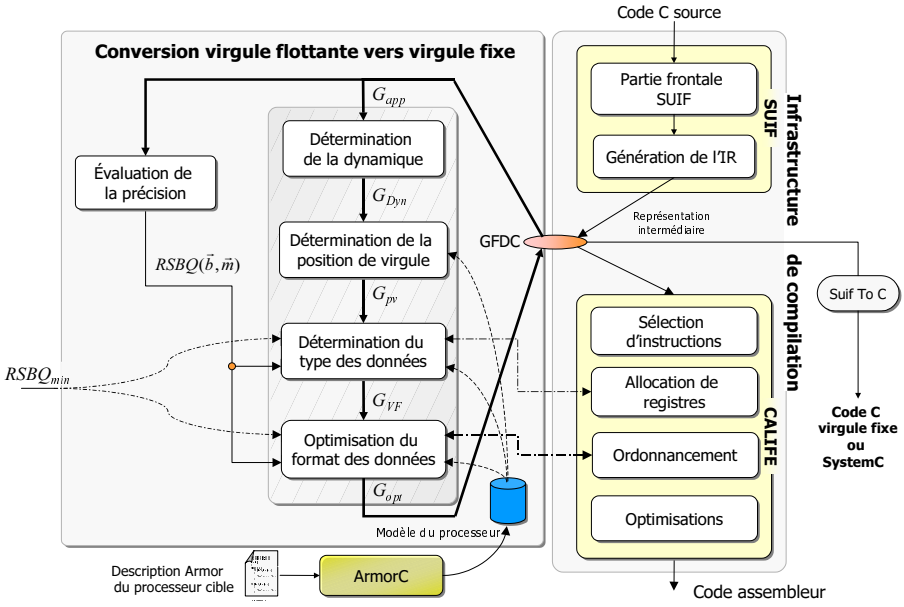


Figure 1. *Synoptique de la méthodologie d'implantation d'algorithmes spécifiés en virgule flottante au sein des architectures DSP virgule fixe sous contrainte de précision*

3.3. *Présentation de la méthodologie*

L'étude des méthodologies existantes et l'analyse de l'influence de l'architecture et de la génération de code permettent de définir une nouvelle méthodologie. Par rapport aux méthodologies existantes [KUM 00, WIL 97b], la détermination et l'optimisation du format des données en virgule fixe sont réalisées sous contrainte de précision en sortie de l'algorithme. De plus, l'architecture du processeur cible est entièrement prise en compte lors de ces deux phases. Le synoptique de cette méthode est présenté à la figure 1. Les différentes phases de conversion en virgule fixe sont réalisées sur une représentation intermédiaire correspondant à un graphe flot de données et de contrôle (GFDC). La première étape de cette méthodologie consiste à déterminer la dynamique (domaine de définition) des données. Les résultats obtenus sont utilisés pour déterminer la position de la virgule de chaque donnée. Ensuite, pour obtenir un format des données en virgule fixe complet, la largeur de chaque donnée est déterminée. La dernière étape de ce processus de conversion correspond à l'optimisation du format des données en vue de minimiser le temps d'exécution du code à travers le déplacement des opérations de recadrage. La détermination et l'optimisation du codage des données étant réalisées sous contrainte de précision, il est nécessaire d'évaluer celle-ci.

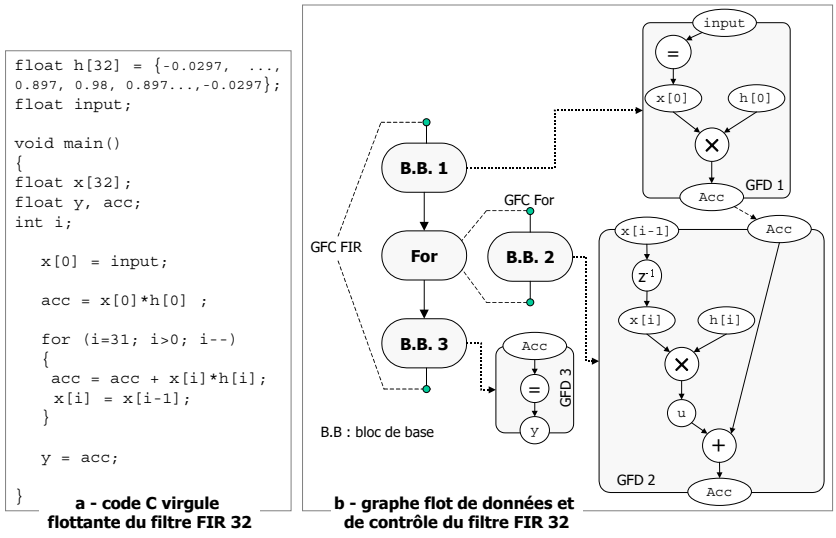


Figure 2. *Spécification du filtre FIR 32 cellules. Code C virgule flottante et graphe flot de données et de contrôle associé*

4. Présentation des différents éléments de la méthodologie

Dans cette partie, les différents éléments de la méthodologie de conversion en virgule fixe sous contrainte de précision sont détaillés. Après avoir présenté la struc-

ture de compilation utilisée et la méthode d'évaluation de la précision, les différentes phases de conversion sont exposées. Elles correspondent à l'évaluation de la dynamique des données, à la détermination de la position de la virgule et de la largeur des données et à l'optimisation du placement des opérations de recadrage. Pour illustrer ces différentes phases de conversion en virgule fixe, un exemple simple est détaillé dans les sections suivantes. Ce dernier correspond à un filtre à réponse impulsionnelle finie (FIR) composé de 32 cellules dont le code C virgule flottante est présenté à la figure 2.a.

4.1. *Structure de compilation*

La structure de compilation utilisée pour mettre en œuvre notre méthodologie est composée d'une partie frontale SUIF [WIL 94] et d'un générateur de code CALIFE [CHA 99]. L'environnement SUIF, développé à l'Université de Stanford, inclut la partie frontale d'un compilateur permettant la transformation d'une application spécifiée en langage C vers une représentation intermédiaire (IR). Celle-ci est transformée en un graphe flot de données et de contrôle (GFDC) sur lequel les différentes phases de conversion en virgule fixe sont réalisées. Au sein de ce GFDC, les différentes structures de contrôle de l'application sont représentées par un graphe flot de contrôle (GFC) dont chaque noeud correspond soit à un bloc de base soit à une structure répétitive ou conditionnelle. Les traitements réalisés au sein d'un bloc de base sont modélisés par un graphe flot de données (GFD) composé de nœuds représentant soit une donnée soit une opération. Le GFDC associé au filtre FIR spécifié à la figure 2.a est présenté à la figure 2.b.

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. Ce dernier permet de décrire de manière compacte le jeu d'instructions du processeur. Cette description est ensuite compilée afin de générer l'ensemble des informations nécessaires aux différentes passes de la génération de code.

4.2. *Évaluation de la précision*

Dans le domaine du traitement du signal, la métrique la plus utilisée pour évaluer la précision d'une implantation en virgule fixe est le Rapport Signal à Bruit de Quantification (RSBQ). Elle correspond au rapport entre la puissance du signal et la puissance du bruit de quantification. Différentes méthodes peuvent être utilisées pour évaluer la précision d'un système en virgule fixe.

Les méthodes basées sur la simulation estiment la puissance du bruit de quantification de manière statistique à partir des signaux obtenus après simulation de l'algo-

rithme en virgule fixe et en virgule flottante. Dans ce cas, l'implantation en virgule flottante est utilisée comme la référence car pour les applications de traitement du signal considérées, l'erreur liée à l'utilisation de l'arithmétique à virgule flottante est négligeable par rapport à celle due à l'arithmétique à virgule fixe. La simulation de l'algorithme en virgule fixe nécessite d'émuler l'ensemble des mécanismes associés à cette arithmétique. Cela augmente nettement le temps de simulation par rapport à une simulation traditionnelle en virgule flottante. De plus, pour obtenir une estimation précise, il est nécessaire d'effectuer un nombre de réalisations de l'algorithme important. La combinaison de ces deux phénomènes conduit à des temps de simulation élevés. Au sein de la méthodologie proposée, 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. Dans ce cas, l'évaluation de la précision basée sur la simulation aboutit à un processus itératif nécessitant de nombreuses simulations. Ainsi pour les systèmes complexes, le temps d'optimisation devient prohibitif et des heuristiques doivent être utilisées afin de limiter fortement l'espace de recherche.

L'approche stochastique proposée au sein de la méthode CESTAC [VIG 93] représente une alternative à ces méthodes basées sur la simulation et nécessitant de nombreuses réalisations. Cette technique a été définie dans le cas de l'arithmétique à virgule flottante pour estimer l'erreur d'arrondi à partir de quelques réalisations. Le résultat en sortie du système est considéré suivre une loi Gaussienne et le test de Student est utilisé pour déterminer l'intervalle de confiance de la moyenne de cette Gaussienne. Ensuite, le nombre de bits significatifs du résultat est déterminé à partir de cette intervalle de confiance. Cette méthode est en cours d'adaptation pour l'arithmétique à virgule fixe.

Une nouvelle méthode analytique d'évaluation de la précision a été définie afin d'obtenir des temps d'optimisation du codage des données raisonnables. Cette méthode permet de déterminer automatiquement l'expression analytique de la puissance du bruit de quantification et de la puissance du signal dans le cas des systèmes linéaires et des systèmes non linéaires et non récursifs. Dans le cas des systèmes linéaires, le bruit de quantification en sortie correspond à la somme des bruits issus de la quantification des coefficients et de différentes sources de bruit filtrées. Ces différentes sources correspondent au bruit associé à chaque entrée et aux bruits générés lors d'un changement de format au sein de l'application. L'expression de la puissance du bruit est détaillée dans [MEN 02c]. Elle est fonction des paramètres statistiques des sources de bruit et de la fonction de transfert entre la sortie et chaque source de bruit. La méthode utilisée pour déterminer automatiquement les différentes fonctions de transfert du système et l'outil développé pour implanter cette méthode sont présentés dans [MEN 02d]. Dans un premier temps, l'application est transformée en une représentation intermédiaire correspondant à un graphe flot de signal (GFD intégrant les opérations de retard entre les données), puis elle est modélisée au niveau bruit de quantification. Ensuite les différentes fonctions de transfert régissant le système sont déterminées en vue d'obtenir l'expression analytique du RSBQ.

4.3. *Détermination du domaine de définition de chaque donnée*

Deux types de méthode peuvent être utilisés pour déterminer le domaine de définition (dynamique) des différentes données présentes au sein d'une application. Cette dynamique peut être estimée à partir des paramètres statistiques de chaque donnée obtenus par simulation de l'algorithme en virgule flottante [KIM 98]. Ces méthodes statistiques permettent d'estimer relativement précisément la dynamique des données en utilisant les informations liées aux caractéristiques du signal. Elles garantissent une probabilité de débordement très faible pour les signaux dont les paramètres statistiques sont proches de ceux des signaux d'entrée utilisés pour l'estimation. Cependant l'absence de débordement n'est pas garantie pour des signaux ayant des caractéristiques nettement différentes.

Le second type de méthode correspond aux approches analytiques. Ces méthodes permettent de déterminer l'expression du domaine de définition des données à partir de celui des entrées du système en utilisant les résultats de l'arithmétique d'intervalles [KEA 96]. Ceux-ci définissent le domaine de définition de la sortie d'un opérateur à partir du domaine de définition de ses entrées dans le cas le plus défavorable. Ces méthodes garantissent l'absence de débordement mais conduisent à des estimations parfois conservatrices car celles-ci sont basées sur une analyse dans le pire cas. De plus, cette approche basée sur la propagation de la dynamique au sein du graphe flot de l'application ne peut s'appliquer aux structures récursives de manière simple. Ce type de méthode a été implanté au sein de l'outil FRIDGE [WIL 97a]. Dans le cadre des systèmes linéaires à coefficients constants, la dynamique des données peut être déterminée à l'aide des normes L1 [PAR 87] ou de Chebychev en fonction de la nature du signal en entrée de l'application. Ces normes sont basées sur la connaissance de la fonction de transfert entre la donnée cible et les entrées. Ainsi, l'utilisation de la notion de fonction de transfert permet de traiter aussi les structures récursives.

Les deux types d'approche analytique présentés ci-dessus ont été combinés afin de pouvoir déterminer la dynamique des données au sein des systèmes linéaires et des systèmes non linéaires et non récursifs. Ces deux techniques sont appliquées à un graphe flot de signal obtenu à partir du GFDC de l'application. Les concepts présentés dans le paragraphe 4.2 pour déterminer automatiquement les fonctions de transfert associées à une application ont été utilisés dans le cadre des systèmes linéaires récursifs.

4.4. *Détermination de la position de la virgule des données*

4.4.1. *Détermination de la position de la virgule des données de l'application*

L'objectif de ce module est de déterminer la position de la virgule pour chaque donnée du GFDC de l'application afin d'aboutir à une spécification correcte d'un point de vue de l'arithmétique à virgule fixe. Cette transformation doit garantir l'absence de débordement et respecter les règles de l'arithmétique à 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 à virgule fixe sont insérées. La détermination de la position de la virgule de chaque donnée au sein du GFDC G_{Dyn} est réalisée en deux temps. Tout d'abord, chaque graphe flot de données (GFD) de G_{Dyn} est traité indépendamment selon la technique présentée dans le paragraphe suivant. Ensuite un traitement global est réalisé afin d'assurer la cohérence de la position des virgules des données partagées entre plusieurs structures de contrôle. Si la position de la virgule d'une donnée présente dans deux structures est différente, alors une opération de recadrage est insérée afin d'assurer le déplacement de la virgule.

4.4.2. *Détermination de la position de la virgule des données au sein d'un GFD*

La position de la virgule associée aux données et aux 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.

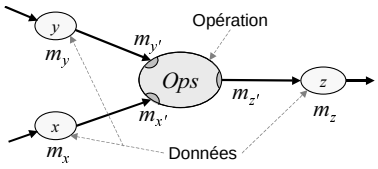


Figure 3. *Modélisation des positions de la virgule d'une opération et de ses opérandes*

Le modèle d'une opération est présenté à la figure 3. Une position de la virgule est associée à chaque donnée du GFD et à chaque entrée et sortie d'une opération. Pour une donnée x la position de sa virgule m_x par rapport au bit le plus significatif est directement obtenue à partir de son domaine de définition $[x_{min}, x_{max}]$ selon la relation suivante :

$$m_x = \lceil \log_2 (\max (|x_{min}|, |x_{max}|)) \rceil \qquad [1]$$

Une règle de propagation de la position de la virgule est définie pour chaque type d'opération. 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 $m_{z'}$ de la sortie d'une multiplication est directement obtenue à partir de celle des entrées selon la relation [2]. Le bit de signe redondant est intégré à la partie entière du résultat.

$$m_{z'} = m_x + m_y + 1 \quad [2]$$

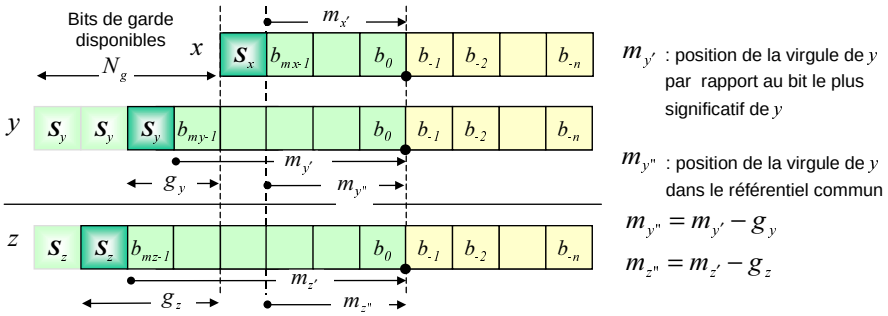


Figure 4. *Spécification de la position de la virgule des opérandes d'une addition*

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. Pour les additionneurs possédant des bits de garde, les largeurs des opérandes d'entrée et de sortie étant différentes, 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. Un paramètre g_x est associé à chaque donnée x pour représenter le nombre de bits de garde utilisés au sein de la donnée. Soit $m_{x''}, m_{y''}, m_{z''}$, les positions des virgules des données dans le référentiel commun telles que représentées à la figure 4. Ces positions sont référencées par rapport au bit le plus significatif de l'entrée dont la largeur est la plus faible. Une position de la virgule m_c commune aux entrées et à la sortie de l'additionneur dans le réfentiel commun est définie. Cette position doit garantir l'absence de débordement et 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 par cette donnée. Cette valeur n'étant pas connue avant de déterminer le format commun, celle-ci est fixée à sa valeur maximale N_g, correspondant au nombre de bits de garde disponibles au niveau de l'additionneur. La position de la virgule commune m_c est définie à partir de la relation suivante :

$$m_c = \max\left(m_x - g_x, m_y - g_y, m_z - N_g\right) \quad [3]$$

En conséquence, afin d'aligner les virgules avant de réaliser l'addition, les positions des virgules associées aux entrées de l'opération doivent être égales à :

$$\begin{cases} m_{x'} = m_c + g_x \\ m_{y'} = m_c + g_y \end{cases} \quad [4]$$

Le nombre de bits de garde réellement utilisés par la sortie de l'additionneur est défini par la relation [5] et la position de la virgule $m_{z'}$ de la sortie de l'additionneur est égale à $m_c + g_z$.

$$\begin{cases} g_z = m_z - m_c & \text{si } m_z > m_c \\ g_z = 0 & \text{si } m_z \leq m_c \end{cases} \quad [5]$$

Lorsque la position de la virgule de l'ensemble des données des 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.

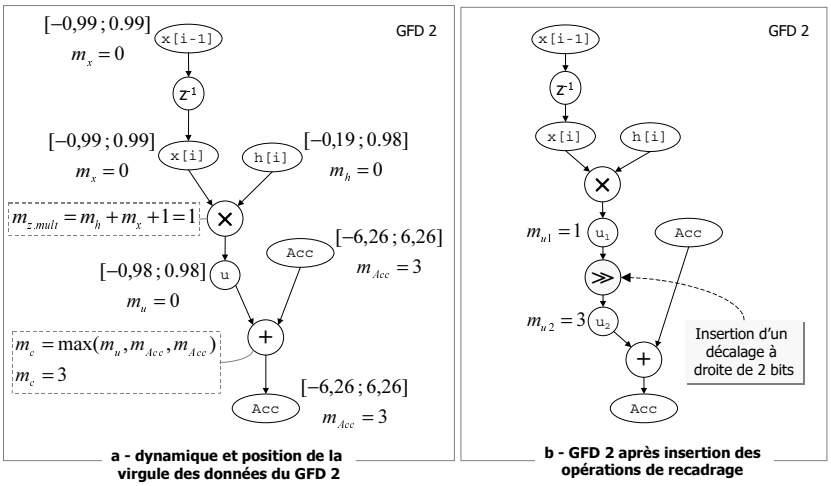


Figure 5. *GFD associé à la structure répétitive du filtre FIR. Les données sont annotées avec la valeur de leur dynamique et la position de leur virgule. Pour les opérations, la position de la virgule de la sortie a été déterminée. Les opérations de recadrage nécessaires ont été insérées*

La dynamique et les positions de la virgule des différentes données du GFD associé à la structure répétitive du filtre FIR spécifié à la figure 2.a, sont représentées à la figure 5.a. La position de la virgule en sortie de la multiplication $m_{z.mult}$ et la position de la virgule commune à l'addition m_c ont été déterminées à l'aide des relations [2] et [3] et en considérant une architecture sans bit de garde. Les positions de la virgule de la sortie de la multiplication, de la donnée u et de l'entrée de l'addition étant différentes, il est nécessaire d'insérer une opération de recadrage entre la multiplication et la donnée u et entre la donnée u et l'addition. Après avoir fusionné ces deux opérations, le graphe flot de données obtenu est représenté à la figure 5.b.

4.5. *Détermination de la largeur des données*

4.5.1. *Présentation de la méthode de détermination de la largeur des données*

L'objectif de cette partie est de définir la largeur de chaque donnée afin d'obtenir une spécification de l'application en virgule fixe complète. Le choix de cette largeur doit prendre en compte la diversité des types des données manipulées au sein du DSP. Comme présenté dans la partie 3.1, les DSP peuvent proposer en plus de leurs instructions arithmétiques classiques, des instructions double-précision pour accroître la qualité des calculs ou des instructions SWP pour accélérer les traitements. 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.

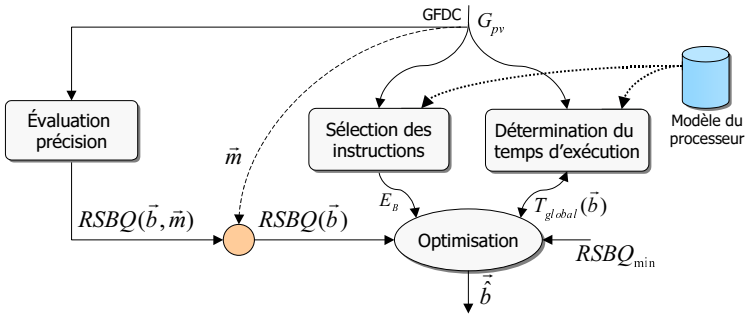


Figure 6. *Synoptique de la méthode de détermination du type des données*

Le synoptique de la méthode mise en œuvre pour déterminer la largeur des données est présenté à la figure 6. L'entrée de ce module est le GFDC G_{PV} au sein duquel la position de la virgule de chaque donnée est spécifiée. Soit $\overrightarrow{b} = [b_0, \ldots, b_i, \ldots]$ et $\overrightarrow{m} = [m_0, \ldots, m_i, \ldots]$, les vecteurs regroupant respectivement la largeur et la position de la virgule des opérandes des différentes opérations de l'application. Soit E_B, l'ensemble regroupant les différentes valeurs pouvant être prises par $\overrightarrow{b}$. Cet ensemble E_B est obtenu à partir de la sélection, pour chaque opération o_i, de l'ensemble E_{I_i} des instructions du processeur permettant de réaliser o_i. La précision du système en virgule fixe est évaluée à l'aide de la méthode présentée dans la partie 4.2. La position de la virgule de chaque donnée, obtenue à l'étape précédente, n'étant pas modifiée au cours de cette phase, l'expression analytique du RSBQ est fonction uniquement de la largeur des opérandes des différentes opérations du système ($\overrightarrow{b}$).

Un module permettant d'estimer le temps d'exécution global T_{global} du code en fonction des instructions sélectionnées est mis en œuvre. Un modèle d'évaluation de T_{global} relativement simple à été retenu afin de limiter le temps nécessaire pour l'évaluation de T_{global} car de nombreuses évaluations de celui-ci sont réalisées au cours du processus d'optimisation. L'objectif de cet estimateur n'est pas de déterminer pré-

cisément le temps d'exécution de l'application, mais de permettre la comparaison de deux séquences d'instruction différentes. Le temps d'exécution global T_{global} correspond à la somme du temps d'exécution t_i de chaque opération o_i de l'application. Ce temps t_i est fonction du nombre d'exécutions de l'opération et du temps d'exécution de l'instruction sélectionnée pour exécuter cette opération. Le temps d'exécution de l'instruction dépend du type d'instruction et ainsi de la largeur des opérandes d'entrée et de sortie de l'opération.

Cette approche permet d'estimer correctement le temps d'exécution d'un code vertical (code sans parallélisme) 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'instruction dans le cas des processeurs possédant du parallélisme au niveau instruction. 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'opération. En effet, les différentes séquences d'opérations classiques ou SWP utilisent pendant les mêmes cycles d'horloge les mêmes unités fonctionnelles. 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. Les instructions classiques et SWP étant identiques d'un point de vue de l'exécution du pipeline, les éventuels aléas de pipeline auront les mêmes conséquences sur les deux séquences d'instruction. Les instructions double-précision ne sont sélectionnées que si les instructions classiques ne peuvent pas fournir la précision requise. Ainsi, ces instructions n'étant jamais sélectionnées sur le critère du temps d'exécution, la surestimation de leur temps d'exécution est sans conséquence sur le processus d'optimisation. Ce modèle peut être amélioré à l'aide des techniques présentées dans [PEG 99] mais en contrepartie, le processus d'optimisation sera ralenti.

Le processus d'optimisation doit déterminer pour chaque opération o_i, la largeur optimale $\widehat{b_i}$ appartenant à l'ensemble E_{B_i}, permettant de minimiser le temps d'exécution global $T_{global}(\overrightarrow{b})$ et de respecter la contrainte de précision définie à travers $RSBQ_{min}$:

$$\min_{\overrightarrow{b} \in E_B} \left(T_{global}(\overrightarrow{b}) \right) \quad \text{tel que} \quad RSBQ(\overrightarrow{b}) \geq RSBQ_{min} \qquad [6]$$

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 double-précision. Ainsi, ces variables b_i ne peuvent prendre que quelques valeurs entières et prédéfinies. Pour les architectures DSP actuelles, entre une à quatre valeurs par variable sont disponibles. Le domaine de définition très restreint des variables b_i permet de générer de manière exhaustive, sous forme d'arbre, l'ensemble des solutions possibles à ce problème d'optimisation. Le niveau l de l'arbre, correspond au traitement de la variable b_l associée à l'opération o_l. La modélisation obtenue dans le cadre du filtre FIR spécifié à la figure 2.a est présentée à la figure 7. Dans cet exemple, unique-

ment les opérations de multiplication et d'accumulation du GFD 2 sont considérées. Un algorithme de type *branch and bound* est utilisé pour obtenir la solution à ce problème d'optimisation. Le succès de ce type de méthode réside dans la possibilité de limiter fortement l'espace de recherche des solutions. Les techniques utilisées pour limiter celui-ci sont résumées dans les paragraphes suivants.

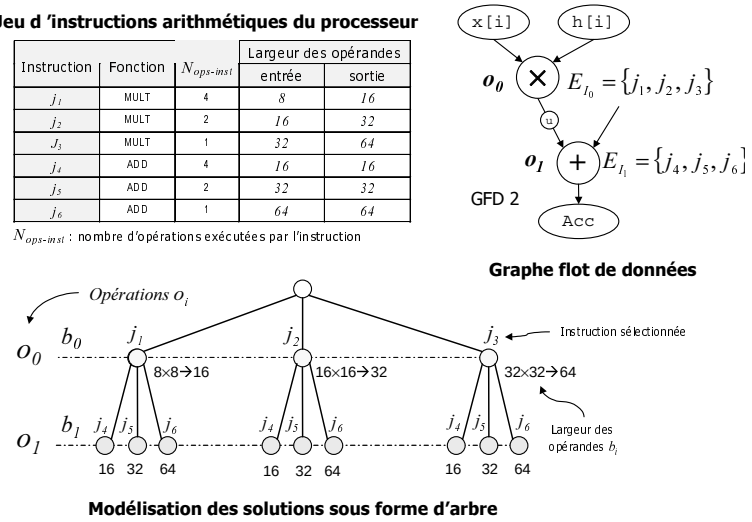


Figure 7. *Modélisation du problème d'optimisation de la largeur des données dans le cadre du filtre FIR*

La modélisation du problème sous forme d'arbre permet de représenter de manière exhaustive l'ensemble des solutions possibles. Cependant, toutes les combinaisons d'instructions ne sont pas valides. En effet, si la donnée correspondant à l'entrée de l'opération o_i est le résultat de l'opération o_k, alors le nombre de bits réservés pour la partie fractionnaire de l'entrée de o_i ne peut être strictement supérieur au nombre de bits pour la partie fractionnaire du résultat de o_k. Ainsi, lors de l'initialisation de chaque largeur b_i, si la condition présentée ci-dessus n'est pas vérifiée, alors cette solution est rejetée et une nouvelle valeur de b_i est testée. Cette condition sur le domaine de validité des variables b_i permet de limiter fortement l'espace de recherche.

Une technique classique utilisée 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. Cela nécessite de pouvoir évaluer une solution partielle à un niveau quelconque de l'arbre, d'un point de vue de la précision 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. Ce temps minimal est déterminé en fixant les temps d'exécution des opérations appartenant au sous-arbre à leur valeur minimale. 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. C'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.

Le temps d'exécution de ce processus de minimisation basée sur l'exploration d'arbre est sensible à l'ordre d'évaluation des variables. Afin de trouver plus rapidement la solution optimale, il est nécessaire de traiter en premier les variables ayant le plus d'influence sur le processus d'optimisation en termes de précision et de temps d'exécution global du code [MEN 02a].

4.5.2. *Expérimentations*

Pour illustrer cette phase de détermination de la largeur des données un exemple relativement simple permettant de détailler les largeurs des différentes données est proposé. L'application testée correspond à un corrélateur complexe dont la sortie y à l'instant n correspond à la corrélation entre un code complexe bipolaire C de 32 éléments et les 32 derniers échantillons du signal complexe x. Le code C virgule flottante de cette application est décrit à la figure 8.a. Le processeur ciblé est le DSP VLIW TMS320C64x [TEX 00] dont les capacités SWP sont similaires à celles du processeur utilisé dans l'exemple présenté à la figure 7.

```
float CI[32] = {1, 1,-1,...}; // Partie réelle du code
float CQ[32] = {1,-1, 1,...}; // Partie imaginaire du code
float InputI, InputQ; // Partie réelle et imaginaire de l'entrée

int main(){
int i;
float xI[32], xQ[32], yI, yQ;
float   zI, zQ, AccI,  AccQ;

    xI[0] = InputI;
    xQ[0] = InputQ;

    AccI = xI[0]*CI[0]-xQ[0]*CQ[0];
    AccQ = xI[0]*CQ[0]+xQ[0]*CI[0];

    for (i = 31; i > 0; i--)
    {   zI = xI[i]*CI[i]-xQ[i]*CQ[i];
        zQ = xI[i]*CQ[i]+xQ[i]*CI[i];
        AccI = AccI + zI;
        AccQ = AccQ + zQ;
        xI[i] = xI[i-1];
        xQ[i] = xQ[i-1];
    }
        yI = AccI;
        yQ = AccQ;
}
```

a - code C virgule flottante

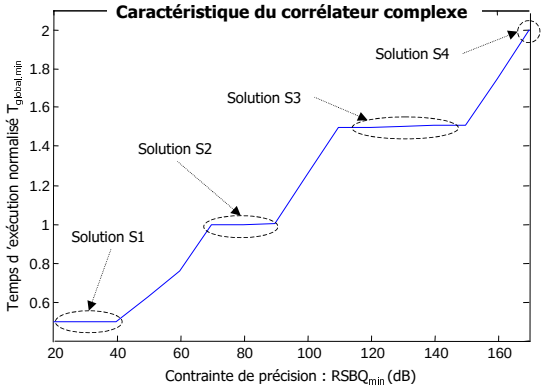
```
short CI[32] = {16384, 16384,-16384,...};
short CQ[32] = {16384, 16384,-16384,...};
short InputI, InputQ;

int main(){
int i;
short xI[32], xQ[32], yI, yQ;
int   zI, zQ, AccI,  AccQ;

    xI[0] = InputI;
    xQ[0] = InputQ;

    AccI = (xI[0]*CI[0]-xQ[0]*CQ[0])>> 4;
    AccQ = (xI[0]*CQ[0]+xQ[0]*CI[0])>> 4;

    for (i = 31; i > 0; i--)
    {   zI = xI[i]*CI[i]-xQ[i]*CQ[i];
        zQ = xI[i]*CQ[i]+xQ[i]*CI[i];
        AccI = AccI + zI >> 4;
        AccQ = AccQ + zQ >> 4;
        xI[i] = xI[i-1];
        xQ[i] = xQ[i-1];
    }
        yI = AccI;
        yQ = AccQ;
}
```

b - code C virgule fixe (S2)

Figure 8. *Code C virgule flottante et virgule fixe d'un corrélateur complexe*

Cette méthode d'optimisation de la largeur des données permet de déterminer l'évolution du temps d'exécution minimal estimé ($T_{global.min}$) en fonction de la contrainte de RSBQ souhaitée ($RSBQ_{min}$) pour l'implantation d'une application donnée dans un DSP donné. La caractéristique obtenue pour l'application considérée est présentée à la figure 9. Pour comparer les différentes implantations, le temps d'exécution

a été normalisé par rapport à la solution 2 dont le code C virgule fixe est présenté à la figure 8.b. La caractéristique $T_{global.min} = f(RSBQ_{min})$ évolue par paliers. Les quatre paliers présents sur la courbe correspondent à des implantations particulières pour lesquelles les largeurs des différentes données sont détaillées au sein du tableau présenté à la figure 9. Les solutions 1, 2 et 4 sont basées sur la même structure de codage des données. Les données x et y sont stockées en mémoire sur L bits et les résultats intermédiaires sont codées sur $2L$ 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 des données de 32 bits et permettent ainsi d'obtenir un temps d'exécution plus faible.



Données	Largeur des données (bits)			
	S1	S2	S3	S4
`Input`	*8*	*16*	*32*	*32*
`xI, xQ`	*8*	*16*	*32*	*32*
`zI, zQ`	*16*	*32*	*32*	*64*
`AccI, AccQ`	*16*	*32*	*32*	*64*
`yI, yQ`	*8*	*16*	*32*	*32*

Largeur des données pour les différentes solution Si (aucune contrainte sur la largeur des entrées et sortie de l'application n 'a été fixée)

Figure 9. *Caractéristique $T_{global.min} = f(RSBQ_{min})$ du corrélateur complexe. Chaque point 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 méthodologie a été utilisée pour implanter un récepteur WCDMA (Wideband Code Division Multiple Access) [OJA 98] au sein de ce même DSP. La contrainte de précision a été définie en fonction des performances souhaitées pour le récepteur et la spécification en virgule fixe permettant de respecter cette contrainte a été déterminée. L'implantation à l'aide du compilateur C associé au DSP, de cette spécification utilisant les capacités SWP du DSP permet d'accélérer d'un facteur 2,6 le temps d'exécution de l'application par rapport à un code n'utilisant que des instructions classiques. Pour cette application, le problème d'optimisation de la largeur des données est composé de 30 variables et la solution a été obtenue en moins de 150 s. Ce temps d'optimisation est raisonnable mais celui-ci peut devenir important pour des problèmes comportant un nombre de variables plus élevé. Dans ce cas, il peut être envisagé de scinder ce problème d'optimisation en deux phases. La première phase réalise l'optimisation en considérant les variables comme des nombres entiers positifs. La solution obtenue permet de réduire le domaine de définition des variables en ne retenant pour chacune d'elle que les deux valeurs de leur domaine de définition encadrant la valeur obtenue au cours de la première phase. Ensuite, la méthode basée

sur l'algorithme *branch and bound* est utilisée pour trouver la solution à ce problème d'optimisation.

Ces expérimentations montrent l'intérêt de cette méthodologie pour explorer les différentes instructions proposées par les DSP en vue d'obtenir une spécification en virgule fixe optimisée et respectant les critères de précision spécifiés par l'utilisateur. Cependant, cette spécification optimisée ne conduit à une implantation optimisée que si le processus de génération de code est capable de prendre en compte les capacités SWP du processeur. Dans [LEU 00], l'auteur propose une technique de sélection d'instructions permettant de prendre en compte ces instructions SWP.

4.6. *Optimisation du format des données*

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 optimisée 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. Ces dernières 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 peut être diminuée car certaines données ne sont plus codées aussi précisément. La solution optimisée 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 optimisée en fonction de la contrainte de précision minimale. 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 la position de la virgule et le type de chaque donnée sont spécifiés. 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 de registre à décalage utilisé 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. Dans ce cas, la phase de sélection d'instructions permet de détecter ces différentes instructions complexes. La technique d'optimisation du format des données pour ce type d'architecture est présentée dans le paragraphe 4.6.2. Le second type d'architecture correspond aux processeurs pour lesquels le parallélisme au niveau instruction est spécifié par le regroupement au sein d'une instruction globale de

plusieurs instructions partielles commandant chacune une seule unité fonctionnelle. La mise en parallèle des instructions partielles étant réalisée lors de l'ordonnancement des instructions, l'optimisation du format des données est effectuée au cours de cette phase selon la technique présentée dans [MEN 02a].

4.6.1. *Stratégie de 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. Dans ce cas, le nombre de bits m_x utilisés pour coder la partie entière est diminué car celui-ci est trop important par rapport à la dynamique réelle de la donnée. 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 cette opération 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. 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 cette opération soit située 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 afin d'aligner le bit le plus significatif de la donnée, présent au sein des bits de garde, sur le bit le plus significatif du registre d'accumulation sans bit de garde. Si ce décalage n'est pas réalisé immédiatement après la série d'accumulations, les bits de garde sont perdus lors du renvoi de la donnée en mémoire 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.

Pour le déplacement des opérations de décalage, des règles de propagation au sein de chaque type d'opérateur sont définies. 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. Ainsi, dans le cadre des systèmes linéaires, l'utilisateur doit définir si les opérations doivent être transférées vers les entrées représentant la partie signal ou celles représentant les coefficients (constantes). Dans le cadre du filtre FIR spécifié à la figure 2.a, les opérations de recadrage ont été déplacées vers l'entrée du filtre. Le code C virgule fixe obtenu est présenté à la figure 10.

4.6.2. *Architecture sans parallélisme d'instruction*

Dans cette partie, les architectures pour lesquelles le parallélisme éventuel est encodé au sein des instructions sont considérées. Soit t_{r_i} le temps d'exécution de l'opération de recadrage r_i et n_{r_i} le nombre d'exécutions de r_i. Pour la classe d'architectures considérée, l'expression du temps d'exécution T_{dec} de l'ensemble des N_r opérations de recadrage présentes au sein de l'application est la suivante :

```
short h[32]={-973,..., 29418, 32112, 29418,...-973};
short input;

void main()
{
short x[32];
short y;
int acc;
int i;

    *x = input >> 2;
    acc = *x * *h;

    for (i=31; i>0; i--)
    {
        acc = acc + x[i] * h[i];
        x[i] = x[i - 1];
    }
    y = (int)(acc);
}
```

Figure 10. *Code C virgule fixe du filtre FIR après déplacement des opérations de recadrage vers l'entrée*

$$T_{dec} = \sum_{i=0}^{N_r-1} n_{r_i}.t_{r_i} \qquad [7]$$

Le temps d'exécution t_{r_i} de l'opération de recadrage r_i est égal à la différence entre les temps t_{ar} et t_{sr} correspondant respectivement aux temps d'exécution du code avec et sans l'opération de recadrage. Le processus de détermination du temps t_{r_i} d'exécution des opérations de recadrage est schématisé à la figure 11. La première étape consiste à extraire l'arbre d'expression contenant l'opération de recadrage à traiter. Ensuite, une sélection de code réalisée sur l'arbre d'expression avec et sans l'opération de recadrage permet de générer deux listes d'instructions I_{ar} et I_{sr}. Le temps d'exécution de chaque liste d'instructions est obtenu à partir de la somme du temps d'exécution des différentes instructions. 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. Cette technique détermine le temps d'exécution réel des opérations de recadrage réalisées à l'aide d'un registre à décalage en barillet ou spécialisé. Dans ce dernier cas, la valeur du décalage doit être spécifiée au sein de l'instruction. Pour affiner cette estimation, les éventuels aléas de pipeline lors de l'exécution de l'opération de recadrage peuvent être détectés à l'aide de la méthode proposée dans [LI 95] en analysant pour chaque instruction les instructions adjacentes.

Le synoptique du processus d'optimisation est proposé à la figure 11. 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 du déplacement d'une

opération de recadrage et d'une validation de celui-ci après évaluation de la précision de l'implantation. L'objectif étant de diminuer le temps d'exécution lié aux opérations de recadrage, ces opérations sont traitées par ordre décroissant du surcoût qu'elles entraînent. Après chaque déplacement 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. Si le déplacement d'une opération de recadrage conduit à une précision ne respectant plus la contrainte fixée, cette opération de recadrage est repositionnée à l'emplacement fournissant un 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. L'heuristique proposée permet de réduire le temps d'exécution des opérations de recadrage, mais la méthode ne garantit pas d'obtenir la solution optimale. Des techniques d'optimisation basées par exemple sur les algorithmes génétiques peuvent être envisagées afin de trouver la position optimale des recadrages mais en contrepartie, le temps d'optimisation sera nettement plus élevé.

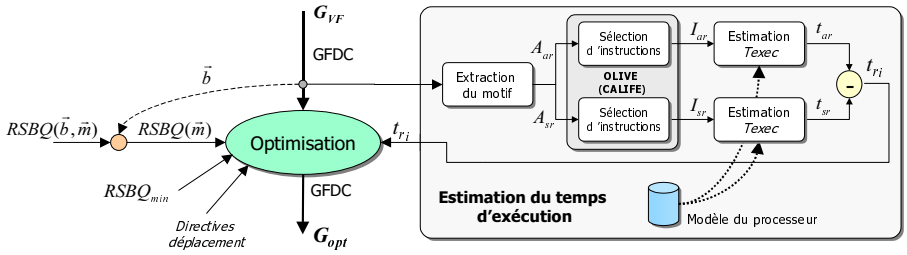


Figure 11. *Synoptique du processus d'optimisation du format des données en vue de réduire le temps d'exécution des opérations de recadrage*

4.6.3. *Expérimentations*

Pour illustrer cette phase d'optimisation du format des données, différentes applications ont été testées sur les processeurs C54x et C50. Ces DSP sont basés sur une architecture conventionnelle de type MAC, mais leurs opérateurs et leurs capacités de recadrage sont différents. Le DSP C54x possède 8 bits de garde au niveau de l'accumulateur et un registre à décalage en barillet. Le DSP C50 n'intègre pas de bit de garde et ses capacités de décalage sont limitées. Il possède des registres à décalage dédiés aux opérateurs et un registre à décalage réalisant le décalage d'un bit par cycle. Les applications considérées sont un filtre à réponse impulsionnelle finie composé de 32 cellules (FIR 32), un filtre à réponse impulsionnelle infinie d'ordre 2 (IIR 2), un corrélateur complexe sur 128 éléments (CC 128) et une application correspondant à un récepteur en râteau (*rake receiver*) pour les systèmes de transmission WCDMA [OJA 98]. Ce dernier est composé de 4 entrées complexes.

Les résultats du processus d'optimisation du format des données sont présentés dans le tableau 2. Les temps d'exécution des opérations de recadrage avant et après la phase d'optimisation sont dénommés respectivement T_{dec-AV} et T_{dec-AP}. La dégradation de la précision Δ_p liée aux déplacements des opérations de recadrage a été évaluée. Elle 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 pour les structures non récursives, 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 et la précision n'est pas modifiée. 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 du filtre. Dans le cadre du DSP C50, l'absence de bit de garde se traduit par une spécification en virgule fixe avant optimisation pour laquelle le temps d'exécution des opérations de recadrage est très élevé. En effet, ces opérations sont insérées afin de coder les données au plus précis. De plus, ces opérations de recadrage sont implantées à l'aide d'une instruction réalisant le décalage d'un bit par cycle. Ainsi, le temps d'exécution de l'opération de recadrage est proportionnel à la valeur du décalage. Après la phase d'optimisation, le temps d'exécution des opérations de recadrage est nul car ces opérations sont réalisées à l'aide des registres à décalage dédiés aux opérateurs qui ne nécessitent pas de cycle supplémentaire pour réaliser le décalage. En contrepartie, la dégradation de la précision est relativement importante.

Applications	**TMS320C54x**			**TMS320C50**		
	T_{dec-AV} (cycles)	Δ_p (dB)	T_{dec-AP} (cycles)	T_{dec-AV} (cycles)	Δ_p (dB)	T_{dec-AP} (cycles)
FIR 32	1	0	1	128	4,5	0
IIR 2	4	8.6	2	7	8.6	0
CC 128	1	0	1	896	29,9	0
Récepteur WCDMA	9	0	9	80	12,6	0

Tableau 2. *Optimisation du placement des opérations de recadrage dans le cadre de l'implantation de différentes applications au sein des DSP TMS320C54x et TMS320C50*

Ces résultats montrent l'intérêt de prendre en compte la présence de bit de garde au sein du DSP afin de limiter le coût des opérations de recadrage dans le cas des systèmes non récursifs. De plus, cette phase d'optimisation du format des données permet d'obtenir une implantation plus efficace en réduisant le temps d'exécution des opérations de recadrage si la contrainte de précision le permet.

5. Conclusion

Dans cet article, une nouvelle méthodologie d'implantation d'algorithmes spécifiés en virgule flottante au sein de processeurs programmables virgule fixe sous contrainte de précision a été proposée. Les justifications de la structure de cette méthodologie et les différentes étapes de celle-ci ont été présentées. Au sein de cette méthodologie, la détermination et l'optimisation du format des données sont réalisées sous une contrainte de précision globale en sortie de l'application. De plus, les différentes phases de conversion en virgule fixe prennent en compte l'architecture du processeur à travers les bits de garde, les types de données supportés et les capacités de recadrage. Les résultats des différentes expérimentations réalisées montrent l'intérêt de ce type d'approche pour optimiser la conversion en virgule fixe.

Une nouvelle méthodologie de conversion automatique en virgule fixe dans le cadre de l'implantation d'algorithmes au sein d'architectures matérielles, telles que les ASIC ou les FPGA, est en cours de développement. Dans ce cas, différents aspects de la méthodologie tels que la détermination de la dynamique et de la position de la virgule des données ou l'évaluation de la précision de la spécification en virgule fixe peuvent être avantageusement réutilisés. Pour une implantation matérielle, 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. La largeur de ces opérateurs étant un entier positif, l'espace de recherche du processus d'optimisation du type des données est nettement plus important que dans le cas d'une implantation logicielle. En conséquence, une technique d'optimisation différente de celle présentée dans cet article devra être mise en œuvre.

6. Bibliographie

[CHA 99] CHAROT F., MESSE V., « A Flexible Code Generation Framework for the Design of Application Specific Programmable Processors », *7th international workshop on Hardware/Software Codesign, CODES'99*, Rome, Italy, May 1999.

[EYR 00] EYRE J., BIER J., « The Evolution of DSP Processors », *IEEE Signal Processing Magazine*, vol. 17, n° 2, 2000, p. 44-51.

[FRI 00] FRIDMAN J., « Sub-Word Parallelism in Digital Signal Processing », *IEEE Signal Processing Magazine*, vol. 17, n° 2, 2000, p. 27-35.

[GOO 97] GOOSENS G. *et al.*, « Embedded Software in Real-Time Signal Processing Systems : Design Technologies », *Proceedings of the IEEE*, vol. 85, n° 3, 1997, p. 436-453.

[GRÖ 96] GRÖTKER T., MULTHAUP E., MAUSS O., « Evaluation of HW/SW Tradeoffs Using Behavioral Synthesis », *ICSPAT'96*, Boston, October 1996.

[KEA 96] KEARFOTT R., « Interval Computations : Introduction, Uses, and Resources », *Euromath Bulletin*, vol. 2, n° 1, 1996, p. 95-112.

[KIM 98] KIM S., KUM K., WONYONG S., « Fixed-Point Optimization Utility for C and C++ Based Digital Signal Processing Programs », *IEEE Transactions on Circuits and Systems II*, vol. 45, n° 11, 1998.

[KUM 00] KUM K., KANG J., SUNG W., « 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*, vol. 47, 2000, p. 840-848.

[LEU 00] LEUPERS R., *Code Optimization techniques for Embedded Processors*, Kluwer academic publishers, 2000.

[LI 95] LI Y., MALIK S., « Performance Analysis of Embedded Software Using Implicit Path Enumeration », *Design Automation Conference 1995*, ACM Press, 1995, p. 456–461.

[MEN 02a] MENARD D., « Méthodologie de compilation d'algorithmes de traitement du signal pour les processeurs en virgule fixe, sous contrainte de précision », PhD thesis, Université de Rennes 1, Lannion, Décembre 2002.

[MEN 02b] MENARD D., QUEMERAIS P., SENTIEYS O., « Influence of fixed-point DSP architecture on computation accuracy », *XI European Signal Processing Conference*, Toulouse, September 2002.

[MEN 02c] MENARD D., SENTIEYS O., « A methodology for evaluating the precision of fixed-point systems », *International Conference on Acoustics, Speech and Signal Processing 2002* , Orlando, May 2002.

[MEN 02d] MENARD D., SENTIEYS O., « Automatic Evaluation of the Accuracy of Fixed-point Algorithms », *Design, Automation and Test in Europe 2002*, Paris, March 2002.

[OJA 98] OJANPERÄ T., PRASAD R., *Wideband CDMA for Third Generation Mobile Communications*, Artech House Universal Personal Communications Series, 1998.

[OPE 01] OPEN SYSTEMC INITIATIVE, « SystemC User's Guide (ver 2.0) », (www.systemc.org), 2001.

[PAR 87] PARKS T., BURRUS C., *Digital Filter Design*, Jhon Willey and Sons Inc, 1987.

[PEG 99] PEGATOQUET A., « 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.

[TEX 98] TEXAS INSTRUMENTS, « TMS320C5X User's Guide », Texas Instruments, 1998.

[TEX 99] TEXAS INSTRUMENTS, « TMS320C54X DSP CPU And Peripherals Reference Set », Texas Instruments, 1999.

[TEX 00] TEXAS INSTRUMENTS, « TMS320C64x Technical Overview », Texas Instruments, 2000.

[VIG 93] VIGNES J., « A stochastic arithmetic for reliable scientific computation », *Mathematics and Computers in Simulation*, vol. 35, 1993, p. 233-261.

[WIL 94] WILSON R. *et al.*, « SUIF : An Infrastructure for Research on Parallelizing and Optimizing Compilers », rapport n° CA 94305-4055, May 1994, CSL, Stanford University.

[WIL 97a] WILLEMS M., BURSGENS V., KEDING H., MEYR H., « System Level Fixed-Point Design Based On An Interpolative Approach », *Design Automation Conference*, 1997.

[WIL 97b] WILLEMS M., BURSGENS V., MEYR H., « FRIDGE : Floating-Point Programming of Fixed-Point Digital Signal Processors », *ICSPAT'97*, 1997.

[WOL 98] WOLF O., BIER J., « TigerSHARC Sinks Teeth into VLIW », *Microprocessor Report*, vol. 12, n° 16, 1998.

Daniel Menard, *docteur de l'Université de Rennes 1 en 2002, est attaché temporaire d'enseignement et de recherche à l'ENSSAT (Lannion), école d'ingénieurs de l'Université de Rennes 1. Il est membre de l'équipe de recherche R2D2 de l'IRISA. Ses activités de recherche s'articulent autour de la définition de méthodologies d'aide à la conception de systèmes intégrés dans le cadre d'applications de traitement du signal. Ses travaux concernent plus particulièrement la définition et la mise en œuvre d'une méthodologie de conversion automatique en virgule fixe dans le cas d'une implantation matérielle ou logicielle.*

Taofik Saïdi, *titulaire du diplôme d'ingénieur de l'ENSSAT (Lannion), est actuellement ingénieur de recherche au sein de l'équipe de recherche R2D2 de l'IRISA. Ses travaux portent sur les circuits reconfigurables, la compilation flexible et la logique ternaire.*

Daniel Chillet, *docteur de l'Université de Rennes 1 en 1997, est actuellement maître de conférences à l'ENSSAT (Lannion), école d'ingénieurs de l'Université de Rennes 1. Il est membre de l'équipe de recherche R2D2 de l'IRISA. Ses activités de recherche s'articulent autour de la définition de méthodologies de conception de circuits. Plus particulièrement, ses travaux concernent la conception de la partie mémorisation avec notamment la prise en compte de la hiérarchisation de cette unité.*

Olivier Sentieys, *docteur de l'Université de Rennes 1 en 1993, est actuellement professeur des Universités à l'ENSSAT (Lannion), école d'ingénieurs de l'Université de Rennes 1. Il est coresponsable de l'équipe de recherche R2D2 (Reconfigurable Retargetable Digital Devices) de l'IRISA. Ses travaux de recherche portent sur les architectures et les méthodes de conception de systèmes intégrés pour les domaines du traitement du signal et des télécommunications. En particulier, il aborde les thèmes du calcul reconfigurable (architectures et outils de compilation), les systèmes à faible consommation et l'arithmétique en virgule fixe.*

Article reçu le 9 octobre 2002
Version révisée le 31 mars 2003
Rédacteur responsable : Abdelaziz M'Zoughi