N° d'ordre : 2791

THÈSE

présentée

DEVANT L'UNIVERSITÉ DE RENNES 1

pour obtenir

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

Mention : TRAITEMENT DU SIGNAL ET TÉLÉCOMMUNICATIONS

PAR

Raphaël DAVID

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

Architecture reconfigurable dynamiquement pour applications mobiles

Soutenue le 9 juillet 2003 devant la commission d'examen

COMPOSITION DU JURY :

MM.	G. Cambon	Professeur à l'Université de Montpellier II	Rapporteurs
	D. Demigny	Professeur à l'ENSEA	
	T. Collette	Ingénieur au CEA	Examinateurs
	D. Lavenier	Directeur de recherche à l'IRISA	
	S. Pillement	Maître de conférence à l'IUT de Lannion	
	O. Sentieys	Professeur à l'ENSSAT	

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 pour m'avoir accueilli dans son laboratoire.

Messieurs S. Pillement, maître de conférence à l'IUT de Lannion, et O. Sentieys, professeur à l'ENSSAT, m'ont fait l'honneur de diriger cette thèse. Je tiens à leur exprimer ma profonde et sincère reconnaissance pour leur aide et leurs conseils dans la préparation de cette thèse et de l'"après-thèse". En m'accordant leur confiance et en pariant sur un concept obscur, ils m'ont offert la chance de m'épanouir professionnellement durant ces trois années. Qu'ils soient assurés de ma gratitude et de ma sympathie.

Je tiens à remercier monsieur D. Lavenier, directeur de recherche au CNRS, pour m'avoir fait l'honneur de présider le jury de cette thèse.

J'adresse mes remerciements à messieurs G. Cambon, professeur à l'université de Montpellier II, et D. Demigny, professeur à l'ENSEA, pour avoir accepter de juger mon travail en tant que rapporteurs.

Je remercie également monsieur T. Collette, ingénieur au CEA, pour sa participation au jury de thèse.

Ce travail de recherche est le fruit de nombreuses contributions. Que messieurs B. Pottier, maître de conférence à l'université de Brest, T. Ben-Ismaël et D. Moulin, ingénieurs à ST microelectronics, soient remerciés pour leurs collaborations. Que les membres de l'équipe R2D2 à l'IRISA, et en particulier F. Charrot, soient remerciés pour m'avoir fait partager leurs compétences dans le domaine de la compilation. Que monsieur T. Saïdi, ingénieur de recherche à l'ENSSAT, soit certain de ma gratitude. Sa contribution à ce projet fût une aide précieuse. Que monsieur D. Menard soit finalement remercié pour m'avoir fait profiter de sa culture en traitement du signal et pour ses conseils lors de la rédaction de cette thèse.

Je remercie tous les membres du LASTI ainsi que le personnel technique et administratif de l'ENSSAT pour avoir contribué, de près ou de loin, à ce travail. Que l'équipe enseignante de l'ENSSAT, et en particulier mon tuteur D. Chillet, soit également remerciée pour m'avoir accompagné dans la découverte de l'enseignement supérieur.

Que mes parents, frères et amis soient finalement remerciés pour leur soutient et leurs encouragements tout au long de la préparation de cette thèse.

À Laëtitia, que cette thèse a parfois privé de l'attention qu'elle mérite, je dédie cette thèse avec amour.

Résumé

Conjointement à l'évolution rapide et constante des technologies de la microélectronique, l'essor des applications embarquées grand public amène des contraintes de conception sans précédents. Aux traditionnelles exigences de performance et de moindre coût, s'ajoutent désormais des contraintes de faible consommation et de flexibilité qui imposent la définition de nouveaux paradigmes architecturaux.

À cette fin, nous étudions ici une architecture enfouie reconfigurable dynamiquement, DART. La définition de cette architecture fait suite à un état de l'art en matière d'architecture reconfigurable et à une analyse du domaine applicatif. Le premier aspect de ce travail porte donc sur l'extraction de mécanismes architecturaux autorisant l'association des différentes contraintes inhérentes aux applications mobiles de prochaines générations.

Suite à la définition des fondements de l'architecture, la mise en œuvre de ces concepts constitue le second aspect de ce travail. Dans un premier temps, l'architecture système de DART est présentée. Par raffinements successifs, nous atteignons progressivement les plus bas niveaux de la hiérarchie constituant l'architecture et détaillons les primitives de calcul, de contrôle et de mémorisation sur lesquelles est bâti le circuit. Les modèles de programmation sous-jacents à cette architecture sont par ailleurs explicités.

Une architecture n'étant rien sans outils permettant de l'exploiter, la chaîne de développement associée à DART fait l'objet du troisième point de cette étude. Nous démontrons en particulier que par l'association d'un *front-end* permettant la transformation et l'optimisation de codes C, d'un compilateur reciblable et d'un outil de synthèse de haut niveau, une chaîne de développement efficace peut être construite. Les différentes optimisations logicielles mises en œuvre dans ces outils sont par ailleurs détaillées, de même que le simulateur permettant leur validation et l'évaluation de la pertinence de la solution proposée.

Le dernier aspect de ce travail concerne la validation des fondements de l'architecture et de leur mise en œuvre. Nous présentons pour cela les résultats issus de l'implémentation de fonctions clés des télécommunications de troisième génération et analysons le comportement de DART sur ces applications. Une évaluation poussée de la consommation d'énergie et des performances nous conduit par ailleurs à comparer DART à des paradigmes architecturaux plus communs que sont les processeurs programmables et les FPGAs.

Pour assurer le suivi de nos recherches, nous effectuons finalement une analyse critique de cette architecture et des outils qui lui sont associés. Nous évoquons également les nombreuses perspectives de ce travail.

Mots clés : traitement du signal, télécommunications, applications mobiles, architecture reconfigurable, FPGA, processeur programmable, arithmétique des ordinateurs, compilation, synthèse de haut niveau, hiérarchie mémoire, estimation de la consommation.

Abstract

Together with rapid evolution of microelectronics technology, consumer embedded applications expansion brings unprecedented conception constraints. To traditional performance and low-cost requirements, are nowadays added low-consumption and flexibility constraints which imply the definition of new architectural paradigms.

Consequently we are here studying a new embedded reconfigurable architecture, DART. The definition of this architecture comes from the state of the art in reconfigurable architectures, and to an application domain analysis. The first part of our work consists in extracting architectural mechanisms which allow association of the different constraints inherent to next generation embedded applications.

Following these architectural principles definitions, implementation of these concepts constitutes the second part of this work. In a first time the system level architecture of DART is presented. By successive refinements, we gradually reach the lowest levels of the hierarchy constituting architecture, and detail the processing, control and storage primitives. The programming models underlying to this architecture are moreover extracted.

An architecture without programming environnement is unusable. Hence the third part of our work consists in developing an efficient compilation framework. In particular, we demonstrate that by the association of a front-end, allowing transformation and optimisation of C codes, of a retargetable compiler and a high-level synthesis tool, an efficient development flow can be built. The different software optimisations implemented in these tools are moreover depicted, as well as the simulator allowing their validation and the evaluation of the proposed solution quality.

The last part of our work concerns the architecture validation. For that, we are presenting results coming from implementations of third generation telecommunication key applications, and analyse the behavior of DART on these applications. Thanks to a fine evaluation of the energy consumption and of the performance, we are also comparing DART to more traditional architectural paradigms which are the programmable processors and FPGAs.

To follow up our research, we have finally carried out a critical analysis on this architecture and its associated developpement tools. We are also discussing about the numerous perspectives of this research work.

key words : signal processing, telecommunications, embedded applications, reconfigurable architecture, FPGA, programmable processor, computer arithmetic, compilation, high-level synthesis, memory hierarchy, consumption estimation.

Table des matières

Table des figures

Liste des tableaux

Liste des listings

Introduction

Sommaire

L'apparition des systèmes numériques, au début des années 50, a permis à des utilisateurs hautement qualifiés de traiter automatiquement certaines applications de faible complexité. Bien que la puissance de calcul, la capacité de mémorisation et le coût de ces machines en limitaient alors l'intérêt, les évolutions technologiques survenues depuis lors ont renversé cette tendance.

La réduction de la taille des lithographies et l'augmentation de celle des *wafers*, ont notamment autorisé la diminution de la surface des systèmes numériques et par voie de conséquence leur coût, leur ouvrant ainsi la porte du marché grand public. Ces mêmes effets ont par ailleurs eu pour conséquence d'autoriser l'intégration d'un nombre toujours croissant de transistors dans des systèmes de plus en plus performants. Les applications développées en vue d'être intégrées dans ces systèmes se sont dès lors traduites par de nouvelles contraintes dont certaines n'ont pu être supportées que suite à l'apparition de nouvelles technologies. L'exemple le plus probant de cet aspect de l'évolution des machines numériques est sans aucun doute l'apparition des technologies CMOS (Complementary Metal-Oxyde Semiconductor) qui, en éliminant la consommation des transistors au repos, a permis d'embarquer les circuits numériques dans des systèmes mobiles.

Depuis lors, les applications embarquées ont connu un essor important et les quelques dizaines d'opérations que devaient traiter les premières machines numériques à chaque seconde sont devenues des centaines de millions. Afin de supporter ces très fortes puissances de calcul, les concepteurs de systèmes ont aujourd'hui la possibilité de concevoir des circuits pouvant contenir quelques centaines de millions de transistors et pourront, à l'horizon 2010, en intégrer plusieurs milliards. Ces contraintes de puissance de calcul peuvent ainsi être supportées à la condition que les méthodes de conception soient suffisamment performantes pour autoriser le développement de ces systèmes en respectant des contraintes de temps de mise sur le marché (*time-to-market* - TTM) sévères.

Les nouvelles applications embarquées associent cependant à ces très gros besoins en puissance de calcul d'autres caractéristiques tout aussi contraignantes. Parmi celles-ci, le faible coût est sans aucun doute la plus commune, le grand public étant le principal consommateur de ce marché. Conjointement à cette nécessité de concevoir des circuits au moindre coût, leur

durée de vie se réduit aussi inexorablement que le temps séparant leur spécification de leur mise sur le marché. La flexibilité de ces nouveaux systèmes numériques, et la disponibilité d'outils de conception performants, deviennent donc les garants de leur succès. Concevoir un système numérique n'a plus pour objectif de définir un circuit fonctionnel mais un circuit efficace.

Les assistants personnels (PDA : Personal Data Assistant), l'intelligence ambiante, les réseaux sans fils, etc., sont autant d'exemples d'applications mobiles grand public, chacun de ces domaines applicatifs ayant ses propres contraintes et caractéristiques. Dans le cadre de cette étude, les télécommunications mobiles de prochaine génération (UMTS : Universal Mobile Telecommunication System) ont été considérées comme le domaine applicatif de référence et exploitées dans le cadre de nos expérimentations. Elles sont brièvement décrites dans la section suivante et constituent le fil conducteur de ce mémoire.

Contexte de l'étude

L'idée fondatrice de l'UMTS est d'intégrer tous les réseaux de génération actuelle (GSM : Global System for Mobile Communications, DECT : Digital Enhanced Cordless Telecommunications, IS-95 : Interim Standard-95) en un seul et de lui adjoindre des capacités multimédia. Émettre un signal par le biais d'un tel système nécessite de traverser une chaîne de traitement comparable à celle modélisée sur la figure 1.

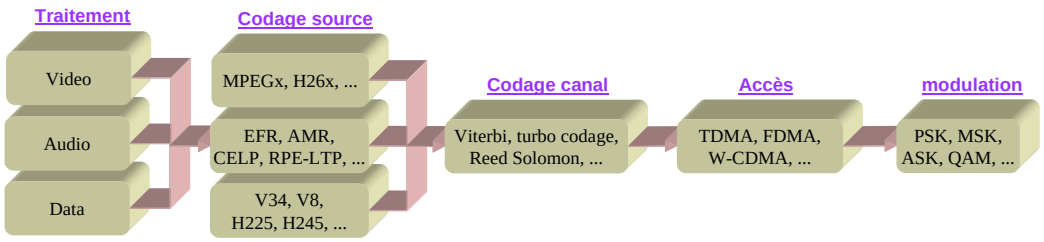

FIG. 1 – Synoptique d'un système d'émission de troisième génération

Concrètement, pour émettre un signal de parole, un terminal mobile doit successivement lui appliquer les transformations décrites ci-dessous.

Traitements haut-niveau Les capacités multimédia intégrées dans la norme UMTS se traduisent par la possibilité d'appliquer des transformations de très haut niveau sur les signaux. Le signal de parole peut ainsi être manipulé afin d'améliorer sa qualité (e.g. annulation d'écho, de bruit, ...) ou de rajouter certains effets (e.g. mixage, fadding) ;

Codage de source Afin de réduire la quantité de données à transmettre, un codage de source doit être appliqué. Compte tenu de la diversité des standards intégrés dans l'UMTS, celui-ci varie en fonction de l'environnement extérieur. Le codage de source, pour ce signal de parole, peut par exemple suivre les recommandations de la norme et être du type AMR (Adaptative Multi-Rate), ou autoriser une communication compatible avec le réseau GSM via un codage EFR (Enhanced Full-Rate) ;

Codage de canal Afin d'améliorer la robustesse du signal et de faciliter le retour sur erreur, un codage canal doit également être appliqué. Là encore, la diversité des standards intégrés implique la nécessité de pouvoir supporter plusieurs types de codage, tels que les codages de Viterbi ou par turbo-code ;

Techniques d'accès Afin de multiplexer les différents utilisateurs accédant simultanément au support de transmission, diverses techniques d'accès doivent également être mises en œuvre. Aux traditionnelles techniques de multiplexage fréquentiel ou temporel, s'ajoutent désormais des techniques de multiplexage par code (CDMA : Code Division Multiple Access), beaucoup plus efficaces mais également beaucoup plus complexes ;

Modulation Avant d'accéder au support de transmission, le signal doit finalement être modulé. Cette dernière étape constitue la partie Radio-Fréquence des systèmes de troisième génération et ne sera pas abordée dans ce mémoire.

Les contraintes associées à ce domaine applicatif sont aussi diverses que sévères. En particulier, la complexité de ces télécommunications dites de troisième génération (T3G) est considérable, et estimée à 12 milliards d'opérations par secondes (GOPS) [1]. Compte tenu de l'aspect portatif des terminaux multimédia mobiles, et de l'évolution lente des performances des batteries qui les alimentent [2], des contraintes toutes aussi sévères s'appliquent à la consommation puisque pour assurer un degré d'autonomie raisonnable, ces terminaux doivent consommer moins de 500 milliWatt (mW). Une excellente efficacité énergétique est donc indispensable pour les terminaux mobiles puisque les contraintes de performance et de consommation de puissance se traduisent par une efficacité énergétique minimale de 24 MOPS/mW, i.e. pour chaque mW consommé, le terminal doit être en mesure d'exécuter 24 millions d'opérations (MOPS).

Outre ces deux aspects, un troisième, né de la définition même du standard, vient compliquer le problème. En effet, l'intégration de multiples réseaux de communication impose la mise en œuvre de nombreux algorithmes, et ce à tous les stades de la chaîne de communication, depuis les traitements de haut-niveau jusqu'aux techniques d'accès. Les différents algorithmes employés dans la norme sont par ailleurs susceptibles d'évoluer au fil du temps et l'un des aspects clé du succès de l'UMTS tient sans aucun doute dans la flexibilité des terminaux multimédia mobiles qui la supportent. Ceux-ci n'auront en effet de cesse d'évoluer en intégrant de nouvelles applications, certaines d'entre-elles restant aujourd'hui encore à imaginer.

À cette flexibilité des algorithmes, s'ajoute par ailleurs celle de l'architecture. En effet, les tâches qui cohabitent au sein d'un système mobile de prochaine génération manipulent des données de tailles très différentes, avec des motifs de calcul et d'accès aux données eux-aussi très variés. À titre d'exemple, un codeur source vidéo travaillant par bloc sur des données arithmétiques peut être suivi d'un codage canal travaillant sur un flot de données binaires. Les contraintes architecturales associées à ces deux calculs sont alors très différentes et excluent la possibilité de les traiter efficacement sur une architecture dans laquelle le modèle d'exécution est invariant.

Finalement, ce domaine applicatif ciblant clairement le marché grand public, les contraintes de coût sont primordiales. Les architectures définies doivent ainsi être aussi peu onéreuses que possible. Elles doivent donc d'une part occuper une surface de silicium minimale, et d'autre part autoriser l'amortissement des coûts non périodiques (NRE : Non-Recurring Engineering) associés à la conception et au test des premiers circuits. Cet amortissement n'est possible que dans la mesure où les volumes de production sont très importants. Pour cela, la durée de vie de ces architectures doit être accrue en les rendant plus évolutives, i.e. plus flexibles.

Si supporter chacune de ces contraintes est un exercice à la portée de tous, les associer rend le problème beaucoup plus délicat, d'autant que les contraintes de temps de mise sur le marché imposent la définition d'outils de développement aussi portables qu'efficaces. Lorsque par ailleurs, le résultat doit consommer très peu d'énergie, il devient insoluble si l'on se limite

aux moyens actuels (ASIC : Application Specific Integrated Circuit, FPGA : Field Programmable Gate Arrays, processeurs programmables, ASIP : Application Specific Instruction set Processor).

Problématique de l'étude

L'augmentation des densités d'intégration autorise désormais la conception de systèmes sur silicium (SoC : System-on-Chip). De tels systèmes associent sur un même substrat différents blocs de natures hétérogènes. Un système sur silicium orienté GSM, représenté sur la figure 2, est par exemple centré sur un processeur généraliste de type microcontrôleur qui se charge principalement de l'interface utilisateur et de la gestion des protocoles de communication. Les calculs de faible complexité sont assurés par des processeurs spécialisés dans la mise en œuvre d'applications de traitement du signal (DSP : Digital Signal Processor). Les traitements critiques et réguliers sont quant à eux exécutés sur des accélérateurs matériels dédiés (ASIC). L'aspect radio fréquence est finalement traité par le sous-ensemble analogique du système.

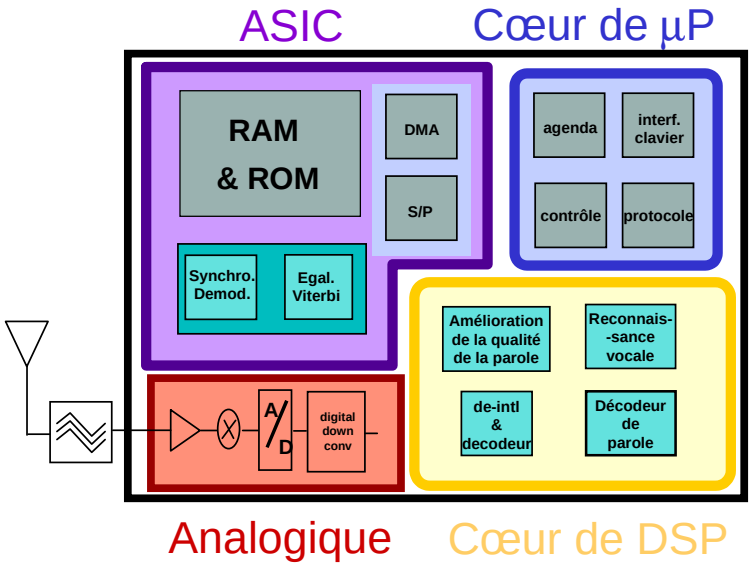

FIG. 2 – *Schéma bloc d'un système sur silicium. Pour le GSM, il se compose d'un microprocesseur se chargeant des traitements de faible complexité et de la gestion du système. Il fait appel à des processeurs de traitement du signal pour les traitements de complexité moyenne et à des blocs dédiés pour les traitements critiques. Un sous-ensemble analogique vient compléter le système pour assurer l'interface avec les traitements Radio-Fréquence.*

Si ce type de système est parfaitement adapté aux réseaux de génération actuelle, il en est tout autrement pour les télécommunications de troisième génération. En effet, la diversité des traitements devant être implantés dans des accélérateurs matériels impose l'intégration d'un très grand nombre de blocs dédiés, et conduit à un taux d'utilisation extrêmement réduit de ces ressources. Le coût de ces systèmes devient dès lors prohibitif, d'autant que leur faible flexibilité restreint leur durée de vie lorsque dans le même temps, des méthodologies de conception immatures accroissent leur temps de mise sur le marché.

La viabilité des terminaux multimédia mobiles dépendant de leur capacité à supporter concurremment des contraintes de faible coût, de haute performance, de flexibilité et de faible

consommation, de nouvelles solutions architecturales doivent être envisagées. À ce titre, bien que les circuits de type FPGA aient longtemps ciblé les seuls marchés du prototypage et de l'émulation, le monde de la recherche a commencé, il y a peu, à les considérer comme des solutions potentielles aux problèmes tels que ceux évoqués précédemment.

Dans ce mémoire, nous montrons que si ces architectures ne peuvent répondre totalement à nos attentes, elles peuvent néanmoins contribuer à définir une nouvelle classe d'architectures : les architectures reconfigurables. Par l'association des concepts ayant fait le succès de ces architectures et ceux ayant permis l'essor des processeurs programmables, ainsi que par l'introduction de nouveaux mécanismes architecturaux, nous proposons une architecture reconfigurable dynamiquement, DART. Celle-ci vise à supporter l'ensemble des contraintes inhérentes aux télécommunications mobiles de prochaines générations.

Plan du mémoire

Le premier chapitre défini la notion d'architecture reconfigurable. En explorant l'espace de conception de ces architectures, nous montrons qu'elles méritent d'être considérées comme de nouvelles alternatives au traditionnel compromis performance/flexibilité qui borne l'espace de conception des systèmes numériques aux seules architectures dédiées et programmables. La caractérisation de cet espace de conception nous conduit par la suite à classer les différents projets menés jusqu'à lors, et à évoquer de leur potentiel et leurs limitations.

Dans le second chapitre de ce mémoire, l'architecture que nous proposons est définie. Cette définition se base sur l'analyse qualitative des architectures reconfigurables examinées dans l'état de l'art, et vise à répondre aux contraintes associées aux applications mobiles de prochaines générations. La description détaillée de DART nous amènera à exposer les concepts clés utilisés pour mettre cette architecture en adéquation avec notre domaine applicatif et à en extraire ses modèles de programmation.

Au sein du troisième chapitre, la méthodologie de développement associée à cette architecture est définie. Cette méthodologie découle des modèles de programmation extraits précédemment et se base sur l'utilisation conjointe de techniques de compilation et de synthèse de haut niveau. Nous démontrons dans ce chapitre que par la définition de plusieurs modèles de programmation, nous sommes en mesure de partitionner le flot de développement en modules de complexité raisonnable. Les différents modules constituant notre chaîne de développement sont ainsi décrits, et par leur assemblage nous présentons la chaîne de développement associée à notre architecture.

La validation de DART fait l'objet du quatrième chapitre. Nous présentons ici les résultats d'implémentation des principales applications constituant les télécommunications de troisième génération. L'efficacité de DART, en terme de performance et d'efficacité énergétique, sur ces applications est analysée et les atouts de cette architecture sont mis en évidence. Par la comparaison des performances de DART et de celles d'architectures représentatives de l'offre commerciale en matière d'architectures reconfigurables, nous discutons par ailleurs des avantages et des limitations de la reconfiguration au niveau fonctionnel, au regard d'autres paradigmes de reconfiguration.

Finalement, dans le chapitre de conclusion, nous résumons les contributions de ce travail de recherche. Les perspectives de cette étude seront ensuite exposées. À court terme, celles-ci concernent essentiellement la poursuite de l'exploitation de DART en tant que support d'étude autour de la reconfiguration. À plus long terme il s'agira de poursuivre l'exploration de l'espace

de conception des architectures reconfigurables et de formaliser le travail réalisé jusqu'à lors, dans les nombreuses équipes de recherche intéressées par le "*reconfigurable computing*", autour des modèles de programmation et des outils de développement.

Chapitre 1

État de l'art

Sommaire

Dans la littérature [3], le terme reconfigurable qualifie une très large gamme de systèmes. Ceci s'explique par la variété de formes que revêt cette propriété. En effet, une architecture est qualifiée de reconfigurable dès lors qu'elle dispose d'un support lui permettant de s'adapter aux traitements qui lui sont assignés. L'ambiguïté des termes "support" et "traitement" dans cette définition fait de son interprétation un exercice délicat et explique la relative obscurité du concept de *reconfigurable* ainsi que la quantité de systèmes qui peuvent justifier cette dénomination. C'est dans le but de clarifier cette définition, et par la caractérisation de ce "support" et de ces "traitements", que ce chapitre tente de raffiner le concept de reconfigurable et d'examiner l'espace de conception de ces architectures.

1.1 Espace de conception des architectures reconfigurables

Par définition, une architecture programmable[1] est reconfigurable : elle est capable de supporter une modification de ses ressources pour répondre à une sollicitation extérieure. Le

[1] Tout processeur, qu'il soit micro-programmé, SIMD, vectoriel ou VLIW est inclus dans cette catégorie.

support de la reconfiguration est alors une instruction. Ces architectures sont dites reconfigurables au sens logiciel du terme. Autrement dit, l'utilisateur doit spécifier une tâche sous la forme d'un programme écrit dans un langage de plus ou moins haut niveau. Le langage est interprété par le système comme une succession de configurations et de données à manipuler. La reconfiguration de l'architecture est alors dynamique et concurrente à l'exécution du traitement.

Si la reconfiguration des architectures programmables est qualifiée de logicielle, il semble naturel qu'à l'autre extrémité du spectre, elle soit qualifiée de matérielle. Par opposition aux processeurs programmables, nous entendons ici les architectures de type FPGA dont les ressources peuvent également être reconfigurées après fabrication. L'approvisionnement en données est fait via un chemin différent de celui des configurations et à des instants disjoints. Cette dernière caractéristique justifie la qualification de statique de la reconfiguration. L'enjeu de la reconfiguration est, cette fois, l'initialisation d'un ensemble de cellules de mémorisation.

1.1.1 Terminologie

La distinction matérielle/logicielle est par ailleurs comparable à la distinction spatiale/temporelle qui décrit la façon dont une application est implantée sur une architecture [4] (Fig. 1.1).

Implémentation spatiale - implémentation temporelle

Les implémentations spatiales sont réalisées sur des architectures dites *Hardware* (HW), dans lesquelles chaque opérateur existe en différents points de l'espace et traite des opérandes directement câblées sur ses entrées. La multiplicité des ressources de calcul autorise dès lors leur spécialisation. De ce fait, chacun des opérateurs de l'architecture peut être assigné à une opération unique ce qui se traduit par des solutions dites "full-custom", dans lesquelles les ressources de calcul ne sont ni plus grosses, ni plus complexes que nécessaire.

Par ailleurs, puisqu'il n'existe aucune ambiguïté quant aux traitements qui devront être réalisés à un instant donné, aucun mécanisme n'a à être introduit pour assurer un certain degré de flexibilité au sein de l'architecture. L'exécution d'un algorithme est donc totalement déterministe et contrôlée par une simple machine d'état. Ces implémentations étant optimisées pour une application unique, elles permettent de maîtriser la consommation, la surface et la vitesse d'exécution.

À l'inverse, les architectures *Software* (SW) implémentent leurs traitements de manière temporelle. Ces architectures n'utilisent qu'un faible nombre de ressources de calcul et de stockage qui sont réutilisées dans le temps. Ces architectures représentent donc la solution optimale en terme de flexibilité, mais celle-ci est obtenue au prix d'une baisse sensible des performances et de l'efficacité énergétique. En effet, dans ce type d'architecture, chaque pas de calcul (e.g. addition de deux nombres) nécessite de rechercher des instructions, de les décoder, d'accéder à des données stockées dans de larges bancs mémoires via de longs bus fortement chargés, puis finalement d'exécuter l'opération sur une unité fonctionnelle générique sur-dimensionnée. Ces architectures ont donc l'avantage d'être compactes et de supporter un très grand nombre de traitements arithmétiques. En contrepartie, elles sont peu performantes.

Reconfiguration statique - Reconfiguration dynamique

Le concept sur lequel reposent les architectures reconfigurables consiste à mixer les implémentations spatiales et temporelles. L'exécution d'un algorithme peut être vue comme une

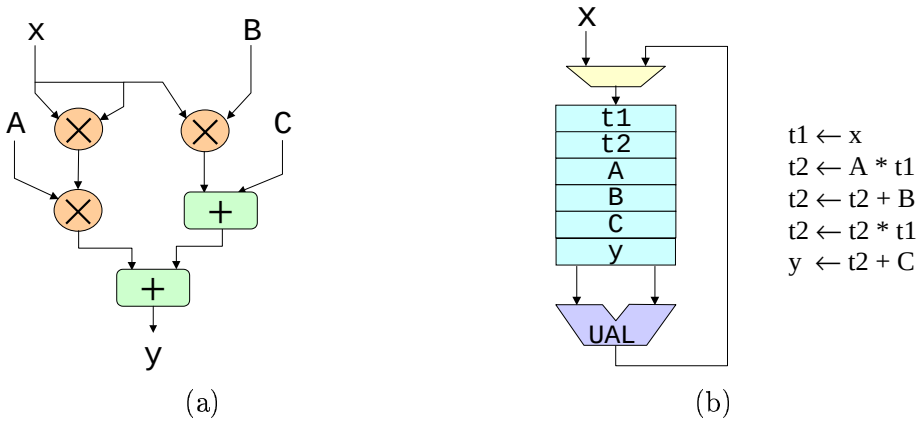

FIG. 1.1 – *Implémentation spatiale (a) et temporelle (b) d'une équation second degré : $y=A.x^2+B.x+C$. Dans le cas d'une implémentation spatiale, les opérations sont assignées à des opérateurs dupliqués en différents points de l'espace. À l'inverse, pour une implémentation temporelle, une ressource de calcul unique est partagée dans le temps entre les différentes opérations.*

succession de phases, chacune d'elle ayant une partie configuration et exécution. Schématiquement, à une extrémité de l'espace de conception des architectures reconfigurables se trouvent les architectures programmables qui à chaque phase traitent une donnée, et qui requièrent un nombre d'étapes de calcul proportionnel au volume de données à traiter. À l'autre extrémité, les circuits FPGA agissent en une seule phase constituée d'une configuration suivie du traitement des données. Entre ces deux extrêmes, il existe, bien sûr, une myriade de possibilités qui offrent donc de nouvelles alternatives aux traditionnels compromis entre performance et flexibilité [5, 4]. En fonction de la complexité des traitements et de la fréquence des reconfigurations, le concepteur peut ainsi définir le degré de performance et de flexibilité qui convient à son système, en parcourant l'espace de conception borné par les processeurs programmables et les circuits FPGA (Fig. 1.2).

La distribution du contrôle à travers le système est également un point important pour évaluer différentes solutions architecturales en raison du surcoût, éventuellement important, qu'elle implique. Comme il a été dit dans le paragraphe précédent, ce surcoût est très limité pour les solutions de type ASIC puisque la distribution des données est spécifiée au moment de la fabrication et qu'il n'existe aucune ambiguïté quant au traitement que devra réaliser une unité fonctionnelle. Le problème est en revanche beaucoup plus délicat pour les processeurs programmables puisqu'à chaque cycle, il est nécessaire de contrôler toutes les ressources de calcul et d'orienter toutes les données utilisées.

Les architectures reconfigurables se présentent, de nouveau, comme une solution intermédiaire. En effet, pour de telles architectures, il s'agit de configurer localement, le temps d'un traitement, des unités fonctionnelles et un réseau d'interconnexions pour associer haute performance et flexibilité. Une fois encore, la définition de la complexité associée à un traitement a un impact décisif sur l'efficacité des architectures : celle-ci peut aller de la simple opération, exécutée en un cycle, à des traitements très lourds, tels qu'une DCT (Discrete Cosine Transform) ou un codage de Viterbi, qui seront exécutés pendant de très longues périodes, éventuellement sur un flux continu de données. Du fait de l'influence de la complexité des traitements sur le modèle de programmation, deux types de reconfiguration sont définis :

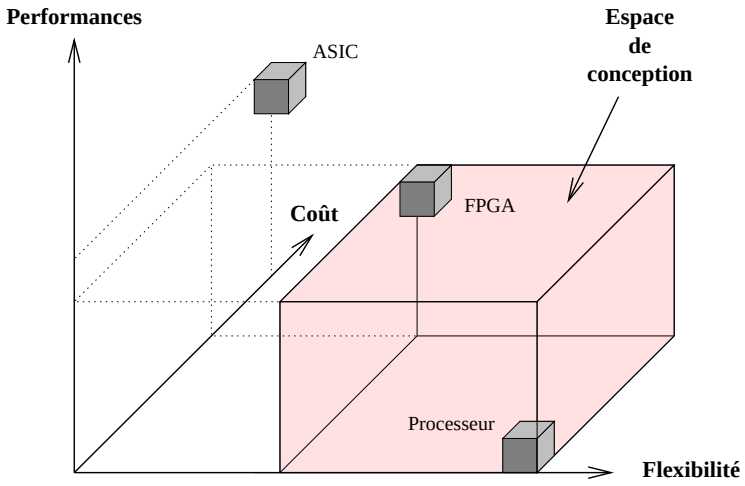

FIG. 1.2 – *Espace de conception des architectures reconfigurables : cet espace est délimité par trois axes symbolisant les performances, la flexibilité et le coût. Dans cet espace de conception, les processeurs et les circuits FPGA occupent deux coins opposés. Les architectures reconfigurables se situent entre ces deux extrêmes*

- la **reconfiguration dynamique** s'applique lorsque les données et les configurations sont distribuées simultanément, i.e. lorsque les phases de reconfiguration et d'exécution sont concurrentes (e.g. processeurs programmables) ;
- la **reconfiguration statique** caractérise les systèmes dans lesquels la reconfiguration et les données sont distribuées via des chemins différents et à des instants disjoints (e.g. circuits FPGA).

1.1.2 Granularité de la reconfiguration

Historiquement, le terme programmable a été appliqué aux systèmes utilisant un programme. Ce dernier est typiquement un groupe d'instructions qui modifie dynamiquement le comportement de modules statiquement connectés (mémoires, registres, UALs, ...). Au cours des dernières décennies, ce concept a progressivement été étendu pour inclure les réseaux d'interconnexions reconfigurables et les fonctionnalités logiques [5, 6]. Aujourd'hui, la programmation peut intervenir à différents niveaux de granularité, chacun ayant un domaine d'application privilégié pour que l'architecture soit en adéquation avec l'algorithme. De ce fait, les applications orientées opérations logiques, (encryptage, codage canal, ...) sont implémentées sur des architectures dont les opérateurs travaillent au niveau du bit. À l'inverse, les applications de traitement du signal ou de l'image, par exemple, qui opèrent au niveau arithmétique, nécessitent des opérateurs travaillant sur des mots de largeurs variables.

Ainsi, du point de vue de la reconfiguration, trois niveaux de granularité [7] (Fig. 1.3) peuvent être distingués : le niveau système, le niveau fonctionnel, et le niveau logique.

La reconfiguration au niveau système

Typiquement, ce niveau de reconfiguration est celui des processeurs programmables. La reconfiguration concerne essentiellement la fonctionnalité des unités de calcul (une UAL exécute une addition ou une soustraction) et l'orientation du chemin de données (de/vers la mémoire et/ou les registres). À ce niveau de granularité, la reconfiguration est toujours dynamique

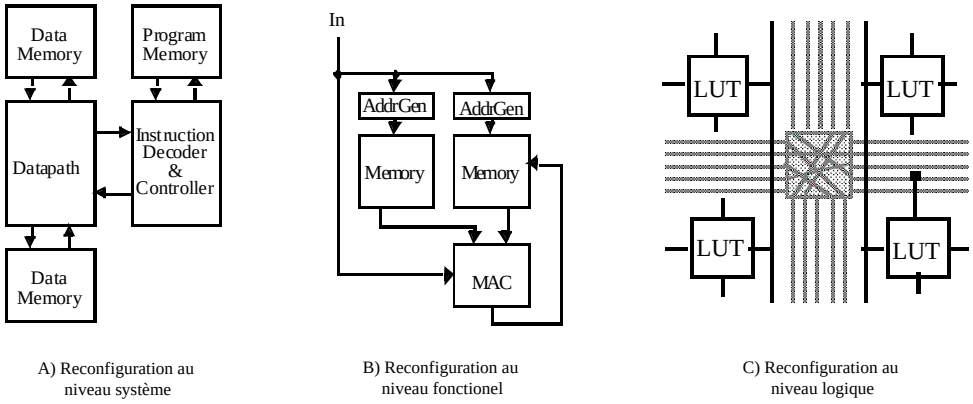

FIG. 1.3 – *Schématisation des différents grains de reconfiguration [5]. La reconfiguration système (A) correspond à la programmation des ressources de calcul et de stockage des processeurs. La reconfiguration fonctionnelle (B) cible les interconnexions entre des ressources de calcul arithmétiques. La reconfiguration logique (C) opère quant à elle sur des LUTs et sur un réseau d'interconnexions transportant des données de très petite taille (typiquement 1 bit).*

puisqu'effective à chaque cycle. Pour la systématiser, la quantité de données à lire en mémoire doit être relativement faible, ce qui explique que les possibilités d'optimisation offertes par la reconfiguration soient limitées. Les architectures exploitant une reconfiguration au niveau système sont donc extrêmement flexibles mais ne disposent que d'un faible potentiel d'optimisation.

Les études menées autour du calcul reconfigurable excluent bien souvent ce niveau de reconfiguration. Ceci s'explique notamment par le fait que les architectures supportant cette dénomination, les processeurs programmables, ont été particulièrement bien étudiées dans les années passées [8]. Dès lors, une remise en cause de leur condition de "modèle architectural" est problématique, la communauté scientifique étant plus à même de supporter la définition d'une nouvelle classe d'architecture que d'accepter la définition d'un modèle plus général dans lequel le processeur ne serait plus la solution de référence, mais un cas particulier.

L'analyse sémantique[2] des mots "programmable" et "reconfigurable" tend également à confirmer cette distinction. En effet, **Programme** signifie[3] (1677) *ce qui est écrit avant*. En informatique (1954), il désigne l'ensemble des dispositions déterminant l'ordre de fonctionnement d'une machine. **Configuration**, quant à lui, signifie à l'origine (XIII^{ème} s.) *façonner à la ressemblance de*, et a pris le sens de *figure, aspect, disposition relative d'éléments*.

Une architecture est constituée d'une disposition relative d'éléments organisés selon un certain schéma. La reconfiguration, en permettant un choix des éléments d'une part, et de leur disposition relative d'autre part, autorise une variabilité de schémas et donc d'architectures. Cette nouvelle définition différencie alors les architectures reconfigurables, permettant un changement des dispositions relatives des éléments, des processeurs qui n'ont pas cette propriété et qui sont de fait programmables.

Les architectures programmables satisfaisant la définition donnée dans l'introduction de ce chapitre, il apparaît cependant problématique de les exclure de cette étude. Ainsi, compte tenu de l'impact considérable de ces architectures sur les modèles de programmation et les

[2]Les auteurs remercient D. Demigny pour cette étude.
[3]Le Robert, dictionnaire historique de la langue française sous la direction d'Alain Rey.

outils de conception, nous considérons, dans le cadre de cette étude, ces architectures comme parties intégrantes de l'espace de conception des architectures reconfigurables.

La reconfiguration au niveau fonctionnel

Pour augmenter le potentiel d'optimisation des processeurs standards, la reconfiguration des interconnexions des ressources (opérateurs arithmétiques ou logiques, ressources de mémorisation, générateur d'adresses, etc.) peut être envisagée pour définir un nouveau grain de reconfiguration dit fonctionnel ou flot de données. Dans ce cas, les distributions du contrôle et des données sont réalisées via des chemins différents. Dans la mesure où ces distributions sont réalisées concurremment, les reconfigurations peuvent être dynamiques dès lors que la quantité de données à transmettre est réduite. Schématiquement, les architectures intégrant un grand nombre d'unités fonctionnelles seront reconfigurables statiquement et celles ne disposant que d'un nombre restreint d'unités pourront être reconfigurées dynamiquement.

La reconfiguration des interconnexions offre, par rapport à la granularité précédente, des possibilités d'optimisation beaucoup plus radicales. En effet, les architectures reconfigurables au niveau système maintiennent un même modèle de programmation, quelle que soit l'application à exécuter : par exemple, un processeur RISC (Reduce Instruction Set Computer) traite exclusivement des opérations registre-registre, un processeur de traitement du signal (DSP : Digital Signal Processor) conventionnel des instructions mémoire-mémoire, etc. Dans la mesure où le modèle de calcul est invariant, cette limitation est transparente et le passage d'une application à une autre n'entraîne aucune perte d'efficacité.

La situation est cependant beaucoup plus délicate lorsque le spectre des traitements s'élargit. En effet, si les applications qui manipulent un grand nombre de données sont exécutées efficacement avec des opérations mémoire-mémoire, leur implémentation sur des architectures centrées sur de larges bancs de registres est très sérieusement perturbée par les incessants transferts de données entre la mémoire et les registres. Dans le même ordre d'idée, il est pénalisant de systématiser de lents accès aux mémoires de données, à la manière des DSPs, lorsque ce sont toujours les mêmes données qui sont manipulées. Ces deux exemples mettent en avant la complexité de la tâche qui consiste à définir un modèle de programmation commun à un ensemble d'algorithmes. La reconfiguration des interconnexions permet de s'affranchir de cette difficulté en autorisant le passage d'un modèle de programmation de type RISC à un modèle de type DSP, par simple transformation du pipeline d'exécution.

La reconfiguration au niveau logique

La reconfiguration au niveau logique est celle qui est réalisée sur les composants FPGA. Les architectures supportant ce type de reconfiguration sont qualifiées de grain fin en raison de la faible largeur de leur chemin de données. La reconfiguration opère au niveau logique sur des LUTs (Look-Up Table) et sur leurs interconnexions. Étant donnée la quantité de LUTs nécessaire à la définition de fonctions évoluées[4], ces réseaux sont très complexes et nécessitent un très grand nombre de données de configuration. En conséquence, les configurations qui sont appliquées à ce type d'architecture sont généralement maintenues pendant un nombre de cycles très important afin de ne pas perturber l'exécution de l'application par des phases de *gel* trop fréquentes.

En reconfigurant une architecture au niveau logique, il est possible de définir n'importe quel type de chemin de données, en synthétisant les opérateurs requis par l'application. Cette

[4]La complexité des équations booléennes implantées dans ces LUTs est limitée par le nombre d'entrées et non par un nombre de portes logiques : typiquement quatre entrées.

caractéristique est intéressante pour les traitements logiques pour lesquels chaque bit est susceptible d'être traité indépendamment du reste d'un flot de données. Si cette flexibilité facilite la spécialisation des chemins de données disposant d'un fort parallélisme au niveau bit, elle est en revanche défavorable aux traitements arithmétiques. En effet, l'emploi de cellules standards limite fortement les possibilités d'optimisation, et donc l'efficacité des opérateurs synthétisés sur l'architecture, par rapport à des solutions *full-custom* où ces opérateurs sont dédiés au traitement d'une opération unique (e.g. addition, multiplication). La perte en performance diffère suivant la complexité de la logique à implémenter d'un facteur variant entre 4 et 10. Cette tendance est par exemple vérifiée dans [9] par l'implémentation de multiplieurs dans un FPGA Xilinx Xc4010, un Altera Flex8k et un National Semiconductor CLAy31.

Il est à noter que sont récemment apparues des architectures dont les structures sont directement issues des FPGAs mais dont les ressources de calcul sont légèrement plus "grosses" [3]. Celles-ci ne manipulent plus des bits mais des données dont la largeur varie typiquement de 2 à 4 bits. Dans [4], nous avons considéré ces architectures comme partie intégrante d'une quatrième classe d'architecture : les architectures reconfigurables au niveau opérateur. Cependant, dans le cadre de cette étude, ces architectures n'ont qu'un intérêt limité puisqu'elles disposent du même modèle d'exécution que les FPGAs. Les considérations sur les architectures reconfigurables de grain fin restant valides pour ces architectures, elles seront présentées dans ce mémoire comme des cas particuliers d'architectures reconfigurables au niveau logique.

1.1.3 Les réseaux d'interconnexions

Le réseau d'interconnexions assure la distribution des données entre les différentes unités fonctionnelles. Sa topologie a un impact de tout premier ordre sur les performances, la consommation et la flexibilité du système. Trois grandes catégories de réseaux [6] peuvent être distinguées, chacune étant caractérisée par un temps de traversée, une complexité et une consommation.

Les réseaux d'interconnexions globaux

Les réseaux d'interconnexions globaux garantissent une totale connectivité des ressources, au sens où chaque élément (calcul, mémorisation, contrôle, ...) peut potentiellement communiquer avec tous les autres. Les deux principaux motifs de ce type sont les réseaux de type *crossbar* et *multi-bus* (Fig. 1.4).

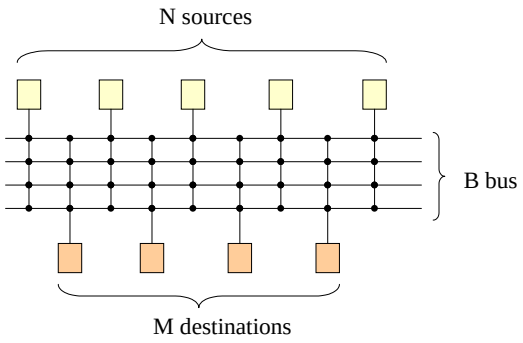

FIG. 1.4 – *Exemple de réseau multi-bus : les N sources approvisionnent en données les M destinations, sans aucune contrainte quant au nombre de communications simultanées.*

La flexibilité de ces deux types de réseau est identique. Ils sont désignés comme étant des réseaux totalement flexibles. Leur nature extrêmement orthogonale simplifie le développement de l'architecture. Ces motifs souffrent cependant d'une complexité (nombre de points d'interconnexion) très importante. De plus, leur consommation et leur temps de traversée sont proportionnels à leur taille et au nombre d'éléments connectés. Aussi, sachant que la surface croît comme le carré du nombre de modules connectés et que le temps de traversée décroît linéairement dans le même temps [6], ce type de réseau est généralement réservé aux systèmes de faible complexité.

Les réseaux d'interconnexions point-à-point

Dans le monde des systèmes multiprocesseurs, il existe pléthore de topologies relatives aux réseaux point-à-point [11] dont le seul but est de fournir des connexions locales efficaces. Parmi ces motifs les réseaux en anneaux et les réseaux *mesh* 2-D généralisés (Fig. 1.5) peuvent être cités.

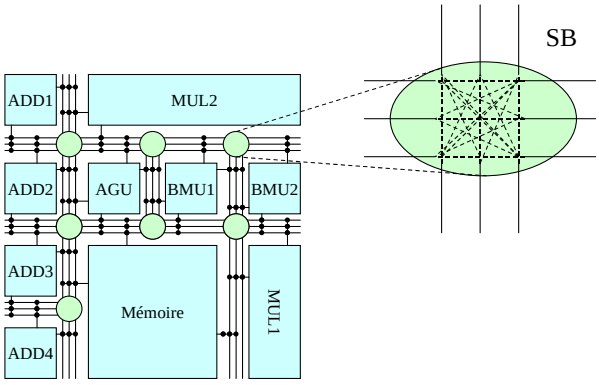

FIG. 1.5 – *Exemple de réseau mesh généralisé : les différents éléments de l'architecture communiquent par le biais de lignes de communication partagées. Les données sont orientées dans le circuit par des boites de commutation (SB).*

Ce type de motif optimise les communications locales en temps et en surface tout en limitant le nombre de connexions. Ceci se fait cependant au prix de communications longue distance très nettement détériorées par le fait que les communications entre deux modules nécessitent la traversée d'un nombre de boîtes de commutations (SB : Switch Box) proportionnel à la distance entre la source et la destination. Dans les technologies actuelles, les délais des commutations restent bien supérieurs à ceux de la propagation des signaux le long des fils. Les connexions longues distances sont de ce fait lentes et consommatrices d'énergie. Par ailleurs, contrairement aux réseaux totalement connectés, les réseaux point-à-point sont très peu orthogonaux et donc plus complexes à exploiter. Les étapes de placement et de routage ont donc une importance toute particulière dans le flot de conception des architectures centrées sur de tels motifs.

Les réseaux d'interconnexions hiérarchiques

Une notion de hiérarchie peut être adjointe aux topologies précédentes pour pallier leurs insuffisances. Ceci conduit alors à la définition de motifs tels que les réseaux hiérarchiques segmentés, les réseaux en arbre ou encore les réseaux *mesh* hiérarchiques (Fig. 1.6). Pour ces derniers, l'architecture est décomposée en *clusters* à l'intérieur desquels les ressources sont interconnectées en favorisant les communications locales. Pour les connexions longues

distances, un second niveau d'interconnexions, global, est défini. Celui-ci permet de proposer des communications *inter-cluster* efficaces en terme de performance et de consommation.

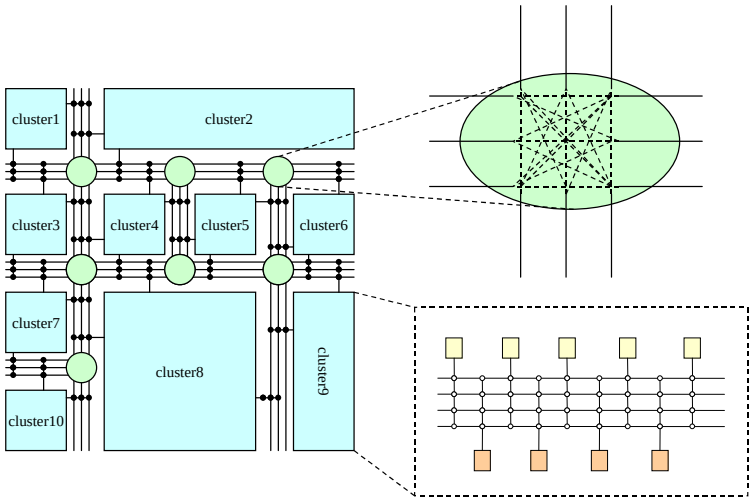

FIG. 1.6 – *Exemple de réseau mesh hiérarchique : les différents clusters de l'architecture communiquent par le biais d'un réseau mesh généralisé. Les communications locales sont assurées par un réseau de type multi-bus.*

L'inconvénient majeur de ces topologies est la possibilité d'être confronté à des conflits lors des communications *inter-cluster*. Ceci impose une décomposition en *clusters* consciencieuse. Plus encore que pour les topologies précédentes, le placement est donc ici une étape cruciale puisqu'il détermine la forme et la surface des *clusters*. De même, le placement des ports d'interface des *clusters* influe de manière significative sur l'efficacité de la connectique. En terme de performance, l'efficacité s'obtient finalement au prix d'un développement fastidieux.

1.1.4 Modes d'utilisation des architectures reconfigurables

Trouver une classification à l'usage des architectures reconfigurables relève de la quadrature du cercle tant leurs natures extrêmement flexibles les prédestinent à un large éventail de traitements. Trois grandes catégories peuvent néanmoins être distinguées : les systèmes de prototypage et/ou d'émulation, les co-processeurs ou accélérateurs matériels et les systèmes autonomes.

Prototypage / émulation

Lors de la conception de systèmes numériques complexes, une validation poussée conduit inexorablement à des temps de simulation extrêmement longs (plusieurs milliers d'heures pour la validation d'un micro-processeur contemporain, par exemple) [12]. L'usage des architectures reconfigurables offre deux alternatives : le prototypage rapide ou l'émulation matérielle. Dans les deux cas, l'idée est d'accélérer le processus de validation en câblant directement l'architecture à tester sur une structure reconfigurable.

Avant la conception définitive d'un système complet, tout ou partie de l'architecture est câblée dans le ou les composants reconfigurables. Le but est généralement la validation *in situ* ou le test grandeur nature de diverses options d'implémentation. Il existe en effet une

multitude de situations où la simulation seule n'est absolument pas satisfaisante, soit à cause de conditions environnementales très complexes à modéliser, ou soit parce que les résultats des traitements ne peuvent être jugés que sur la base d'une évaluation subjective humaine. À titre d'exemple peut être citée la qualité d'un système de compression/décompression d'un flux vidéo que seul l'œil humain peut apprécier. Ainsi, seul le développement d'un prototype permet d'évaluer la pertinence de la solution proposée. Il existe aujourd'hui de nombreuses plate-formes qui évitent la mise au point de plusieurs systèmes physiques différents [12] : la partie en test est implantée dans la structure reconfigurable et est modifiée par reprogrammation des composants.

Accélérateur matériel de traitements

Les architectures reconfigurables offrent également de belles opportunités en tant que co-processeur pour l'accélération de traitements arithmétiques ou logiques spécifiques. Dans cette optique, la ressource reconfigurable est exploitée pour décharger le processeur hôte d'un traitement trop complexe ou pour lequel il n'est pas adapté. Plusieurs modes de couplage sont possibles entre l'hôte (CPU : Central Processing Unit) et le co-processeur [13], comme l'illustre la figure 1.7.

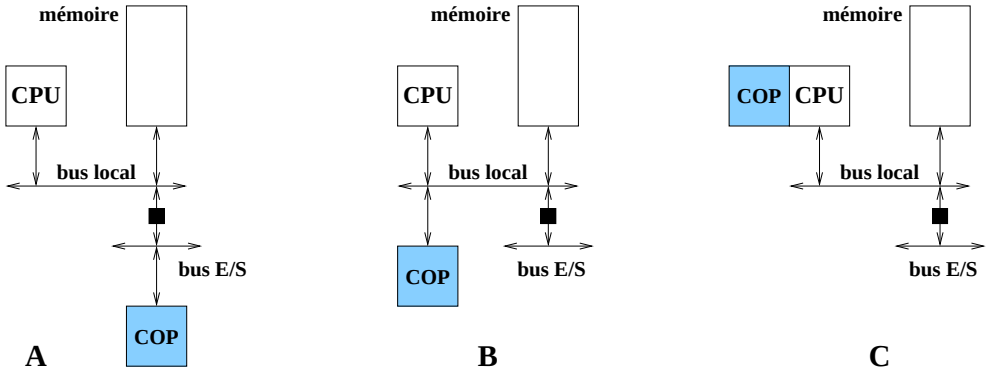

FIG. 1.7 – *Modes de couplage d'une ressource reconfigurable avec un processeur : (A) couplage en mode périphérique ; (B) couplage de type co-processeur ; (C) couplage direct sur le chemin de données.*

Le premier mode (A) s'apparente à une connexion de type périphérique. Les échanges s'effectuent par l'intermédiaire du bus d'entrées/sorties à l'aide, par exemple, d'accès DMA (Direct Memory Access). Un tampon de style FIFO (First In First Out) règle le flux d'informations entre les deux entités. Le temps de latence est élevé, mais la bande passante disponible assure une alimentation efficace de la ressource reconfigurable. Les calculs peuvent s'effectuer en recouvrement avec ceux de l'hôte et la configuration reste généralement stable pendant toute la durée d'un traitement.

Le second mode (B) connecte le co-processeur au bus local du CPU. Ils partagent alors un même espace mémoire. Le co-processeur est "réveillé" par une instruction particulière ou par une interruption. Après reconfiguration, il manipule des données stockées dans la mémoire de l'hôte, sans nécessiter l'intervention du CPU. Puisque c'est le co-processeur qui est chargé de se reconfigurer, de manière à pouvoir traiter l'opération demandée, le processeur hôte peut continuer l'exécution de son programme dans la mesure où il n'entre pas en conflit au niveau des accès mémoire.

Le troisième mode (C) couple directement le co-processeur au CPU. La ressource reconfi-

gurable est située sur les chemins de données internes du processeur, au même titre qu'une unité entière ou flottante [14]. Son alimentation en données est alors réalisée efficacement par des transferts avec les registres généraux du processeur. Le temps de propagation des signaux dans les circuits reconfigurables étant plus important que dans les unités fixes, le contrôle du processeur doit considérer des opérations configurables multicycles et gérer les aléas éventuels. Ces ressources doivent avoir la propriété de pouvoir se reconfigurer très rapidement.

Système autonome

Cette dernière catégorie est directement issue des progrès technologiques considérables en matière d'intégration. Une puce possède aujourd'hui l'équivalent d'une dizaine de millions de portes reconfigurables. Les estimations laissent présager que la centaine de millions de portes est à portée de main, et plus encore en considérant un avenir à plus long terme [2]. L'intégration de systèmes numérique entiers est donc d'ores et déjà envisageable sur des structures entièrement reconfigurables.

Contrairement à l'approche co-processeur, la ressource reconfigurable est le support unique de l'application. Elle inclut toutes les entités d'un système, la plupart basée sur des cœurs IP (Intellectual Property), tels que des microprocesseurs RISC par exemple. Les performances de ces derniers, même si elles n'atteignent pas celles des microprocesseurs *full custom* actuels, sont suffisantes pour une majorité d'applications. À titre d'exemple peuvent être cités les microprocesseurs *MicroBlaze* développé par Xilinx [15] et *NIOS* développé par Altera [16], tous deux capables de délivrer une puissance de calcul de 125 MIPS.

1.2 Classification et caractérisation des architectures reconfigurables

Compte tenu de la dimension de l'espace de conception des architectures reconfigurables, il apparaît nécessaire de définir un critère principal de classification autorisant une rapide évaluation des architectures proposées dans la littérature. À ce titre, nous avons considéré la granularité de la reconfiguration comme étant le plus adéquat. Ce critère de classification permet en effet de mettre en avant certains paramètres clés des architectures examinées, parmi lesquels les modèles de programmation, l'aspect statique ou dynamique de la reconfiguration et son mode d'utilisation.

1.2.1 Les architectures reconfigurables au niveau logique

Dans cette section nous décrivons les architectures reconfigurables au niveau logique. Nous ne retiendrons ici que les FPGAs au détriment des circuits de type PAL (Programmable Array Logic) ou CPLD (Complex Programmable Logic Device) qui ne disposent pas d'une puissance de calcul suffisante pour implémenter des systèmes complets. Par ailleurs, la recherche sur les circuits FPGA ayant débutée dès le milieu des années 80, il existe aujourd'hui une très grande variété de produits et les passer exhaustivement en revue nécessiterait un chapitre complet. Aussi, ne seront mentionnés ici que deux grandes familles de composants reconfigurables au niveau logique, à savoir les produits des sociétés Xilinx et Altera, leaders mondiaux du marché. En dehors de celles-ci, le lecteur pourra vérifier dans la littérature [17, 18] que c'est dans la réalisation des cellules et de leurs interconnexions que chaque constructeur apporte ses spécificités.

L'architecture générique des circuits reconfigurables au niveau logique est représentée sur la figure 1.8. Elle consiste en une matrice d'éléments configurables, reliés par un réseau d'interconnexions, qui communiquent vers l'extérieur au moyen de blocs d'entrées/sorties configurables. La taille, le type et le nombre de cellules, de même que leurs interconnexions varient suivant les familles et les constructeurs.

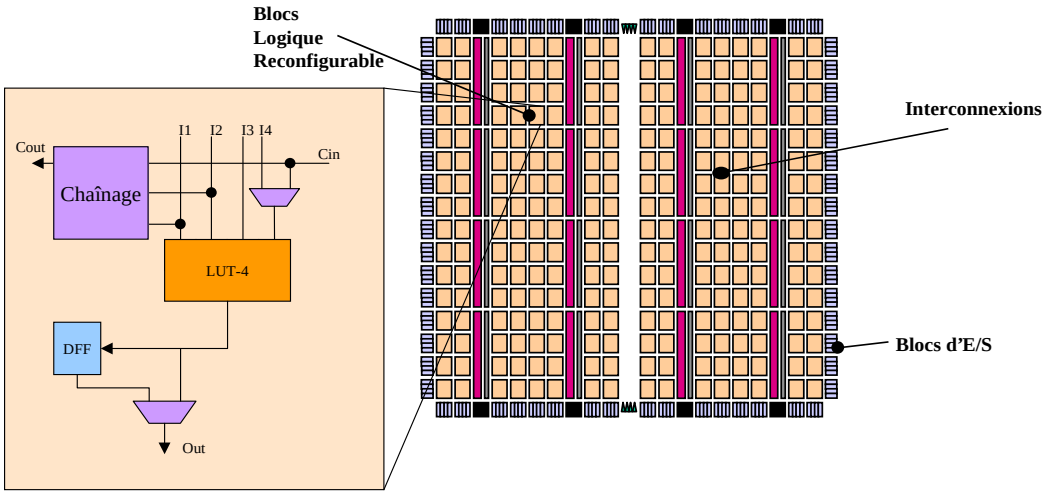

FIG. 1.8 – *Architecture générique des composants reconfigurables au niveau logique. Ils sont constitués d'éléments de calcul reconfigurables, assemblés par le biais d'un réseau d'interconnexions complexe véhiculant des signaux binaires.*

Les ressources de calcul

La cellule de base de ces architectures est un élément mémoire associé à un module combinatoire. Construire un système consiste à associer ces cellules par le biais d'un réseau d'interconnexions programmable. Dans les composants FPGA, la logique combinatoire est implantée sous forme de LUTs (Fig. 1.9). Ce module combinatoire est associé à un élément de mémorisation au sein de la primitive de calcul : le bloc logique configurable (CLB) pour la famille Xilinx et l'élément logique (LE) pour Altera. Ces briques de base intègrent également des fonctionnalités intéressantes telles que le chaînage de retenue pour accélérer l'opération d'addition ou de petits modules mémoire (Fig. 1.8). Les dernières générations de FPGA, le VIRTEX pour Xilinx [19] et l'APEX pour Altera [20], offrent de très fortes capacités d'intégration (4,000,000 portes équivalentes) et une fréquence de fonctionnement élevée.

Dans les nouvelles architectures, les fabricants ont rajouté un niveau de hiérarchie en regroupant les cellules logiques au moyen d'une matrice d'interconnexions locales qui accélère les communications entre proches voisins. Les *versa-blocs*, pour Xilinx, regroupent 4 cellules de base alors que les *LAB* d'Altera intègrent de 8 à 10 *LE*. Les architectures Altera intègrent par ailleurs un autre type de ressource connectées au reste du composant à la manière d'un *LAB*. Ces ressources, baptisées *EAB* (Embedded Array Block) ou *ESB* (Embedded System Block), exploitent un module mémoire piloté par de la logique séquentielle programmable. Elles peuvent être utilisées soit comme de simples mémoires, soit comme de grosses tables de scrutation pour spécifier des fonctions complexes telles qu'un multiplieur 4×4.

Certaines recherches, principalement menées dans les milieux universitaires, étudient par ailleurs l'intérêt de solutions dans lesquelles les ressources de calcul sont de granularité plus

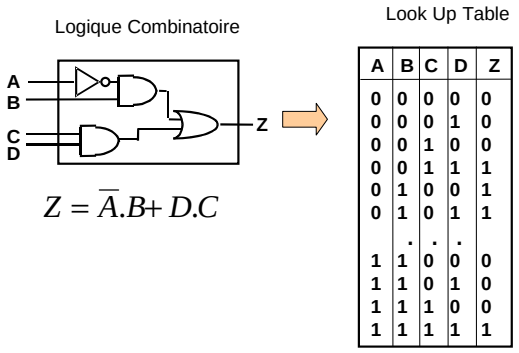

FIG. 1.9 – *Implantation de la logique combinatoire sous forme de LUT : La LUT constitue une mémoire stockant la table de vérité de la fonction à réaliser. Ses entrées permettent d'adresser cette table de vérité qui fourni dès lors la sortie souhaitée.*

épaisse. À titre d'exemple, les projets PipeRench [21, 22], Chess [23], DP-FPGA [24] ou DReAM [25] peuvent être cités. Les modules combinatoires constituant la cellule de base de ces architectures ne sont plus implémentés sous la forme de LUT mais sous la forme de composants multi-fonctions travaillant sur des données de plusieurs bits. Le nombre de cellules nécessaire à la construction d'opérateurs complexes est dès lors limité et, par voie de conséquence, les performances sont accrues lors des traitements arithmétiques.

Les interconnexions programmables

Les composants reconfigurables au niveau logique exploitent des réseaux d'interconnexions très complexes, où chaque bit doit pouvoir être orienté individuellement. Pour les composants Xilinx les interconnexions transitent à travers différents types de ligne suivant la distance, le temps de propagation et le nombre de portes connectées. Des lignes prédéfinies, dites directes, connectent efficacement les proches voisins. Des lignes, dites générales, parcourent l'ensemble du circuit et sont organisées suivant un motif de type *mesh* généralisé. Des buffers sont intégrés aux connexions longues distances. Depuis 1991, les lignes générales sont couplées avec des lignes *double distance* afin de limiter le nombre de passages par des matrices de commutation. Pour les signaux dont la propagation doit être uniforme (signal d'horloge par exemple), des lignes basses impédances les distribuent simultanément à tous les blocs du circuit. Ces dernières disparaissent dans le VIRTEX au profit de lignes routant les signaux tous les 3 ou 6 blocs, l'horloge disposant quant à elle d'un réseau de routage dédié organisé en arbre. Les réseaux d'interconnexions des circuits Altera ont relativement peu évolués depuis le Flex8K [26] et se basent sur un maillage de pistes horizontales et verticales. Les pistes horizontales alimentent les LABs en données qu'ils renvoient indifféremment sur ces mêmes pistes ou sur les lignes verticales.

Dans le cadre des architectures disposant de ressources de calcul travaillant sur des données plus larges, les topologies exploitées sont du même type que celles évoquées précédemment. En limitant le nombre d'interconnexions, ces composants réduisent le surcoût associé aux interconnexions au prix d'une réduction de leur flexibilité.

Modes d'utilisation des architectures reconfigurables au niveau logique

Les architectures reconfigurables au niveau logique ont deux principaux modes d'utilisation : les systèmes de prototypage/émulation et l'accélération de calcul.

Historiquement, ce type d'architecture à tout d'abord été exploité afin de remplacer des composants discrets ou de valider des systèmes destinés à être intégrés dans des circuits de type ASIC [12]. Dans ce contexte, ces circuits sont configurés à l'initialisation et cette configuration est maintenue pendant toute la phase de validation. L'aspect statique de la reconfiguration n'implique dès lors aucune perte d'efficacité et les contraintes de couplage avec le système hôte sont très limitées. Les circuits FPGAs commerciaux sont, aujourd'hui encore, principalement utilisés à cet effet.

Ces architectures peuvent par ailleurs être utilisées à des fins d'augmentation de performance en déchargeant le système hôte de toute ou partie des traitements intensifs pour lesquels il est mal adapté. De nombreuses applications en traitement du signal, de l'image, en cryptographie, en biologie moléculaire, ..., ont donné lieu à des architectures massivement parallèles, et ont démontré que des facteurs d'accélération de plusieurs dizaines, voire plusieurs centaines, d'ordres de grandeur peuvent être obtenus par le biais de cette approche. Dans ce cadre, trois modes d'utilisations peuvent être distingués.

Les cartes accélératrices Ces cartes, contenant un ou plusieurs FPGAs et de la mémoire vive, peuvent se connecter à un processeur via des bus périphériques (e.g. PCI, VME). La machine PeRLe-1, fût l'une des premières réalisations de ce type (1988). Celle-ci se base sur l'architecture PAM (Programmable Active Memory) [27] et est composée de 16 circuits FPGA Xilinx 3090, de mémoire vive et de 7 circuits FPGAs utilisés pour gérer les échanges avec la station hôte. Outre ce type d'assemblage bi-dimensionnels de circuits FPGAs, d'autres systèmes considèrent des assemblages linéaires tels que les systèmes SPLASH-2 [28] ou ArMen [29]. Compte tenu de l'augmentation des capacités d'intégration, les recherches sur ce type de machines sont devenues nettement moins attractives. Désormais, la plupart de ces machines peuvent être intégrées au sein d'un seul et même composant.

Les Co-Processeurs Plus récemment, de nombreuses études ont été menées afin d'étudier des couplages plus étroits entre la ressource reconfigurable et le processeur hôte. À titre d'exemple, le projet GARP proposé par le groupe de recherche BRASS (Berkeley Reconfigurable Architectures, Systems and Software) à Berkeley [30], et le projet NAPA proposé par National Semiconductor [31] peuvent être considérés comme représentatifs, sachant que des projets industriels visent aussi à étudier ce type de couplage [32]. Les deux premières solutions exploitent des blocs reconfigurables conçus au sein même des laboratoires alors que la troisième exploite un composant commercial, le FPGA embarqué M2000 [33].

L'objectif consiste ici à décharger le processeur hôte des traitements pour lequel il n'est pas adapté. Il ne s'agit plus ici de configurer, à l'initialisation, le bloc reconfigurable pour un traitement particulier mais de modifier sa configuration à chaque fois que le processeur hôte rencontre un traitement critique. La reconfiguration de l'accélérateur intervient donc au cours de l'exécution du programme sur le processeur hôte. La distinction entre configuration et reconfiguration apparaît donc ici clairement, de même que l'aspect critique du temps nécessaire à la reconfiguration du bloc reconfigurable.

Ce type de système peut être modélisé par la figure 1.10, sachant que chaque architecture se distingue par la quantité et la qualité des différentes connexions apparaissant sur ce schéma. Dans ce type d'architecture, un processeur de type RISC se charge de l'exécution des traitements de faible complexité. Lorsque celui-ci rencontre un traitement critique (généralement reconnu par une directive assembleur), il configure le bloc reconfigurable afin de se décharger de ce traitement. Certains processeurs reprennent alors l'exécution du programme, sous

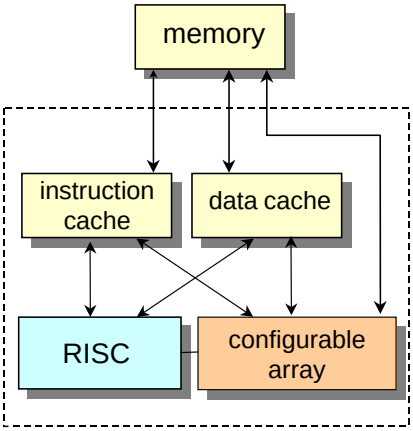

FIG. 1.10 – *Couplage entre un processeur hôte et un bloc reconfigurable de grain fin. Un processeur généraliste contrôle les configurations du bloc reconfigurable de même que ses accès aux mémoires de données.*

réserve qu'il n'y ait pas de dépendances de données, et laissent une complète autonomie au bloc reconfigurable, qui doit dès lors disposer d'un accès direct à la mémoire du processeur. C'est par exemple l'un des modes de fonctionnement privilégié de NAPA qui autorise ainsi une exécution de type *multi-thread*. Dans d'autres projets en revanche, tel que GARP, l'approvisionnement en données du bloc reconfigurable est effectué sous la forme de transferts explicites, sous le contrôle du processeur hôte. Dans ce cas, les exécutions du programme sur le processeur et sur le bloc reconfigurable sont mutuellement exclusives.

Un des facteurs critique pour évaluer la qualité de la solution proposée devient dès lors le temps nécessaire aux reconfigurations. En conséquence, de nombreux mécanismes ont été envisagés dans le cadre de sa réduction. À ce niveau de granularité les deux principales options sont l'utilisation de circuits reconfigurables multi-contextes [34] et la reconfiguration partielle. La reconfiguration multi-contextes consiste à mémoriser plusieurs configurations pour le bloc reconfigurable. Le passage d'une configuration à une autre ne nécessite alors qu'une commutation entre les différents plans de configuration du système. Les performances sont alors très sensiblement améliorées mais ceci se paie par une explosion du coût du circuit. Afin de limiter ce surcoût, une autre solution, utilisée dans le cadre de GARP, consiste à stocker un certain nombre de plans de configuration dans des mémoires caches. Le rapprochement entre les données de configuration et la ressource reconfigurable permet ainsi de réduire les temps de reconfiguration. À titre d'exemple, le projet GARP qui utilise cette technologie ne consomme que quelques cycles pour spécifier une configuration de taille moyenne (500 blocs logiques).

La seconde méthode permettant de limiter les temps de reconfiguration consiste à autoriser une reconfiguration partielle de la ressource reconfigurable. Par reconfiguration partielle, nous entendons ici la propriété de pouvoir faire fonctionner une partie du circuit pendant qu'une autre est en cours de reconfiguration. Cette méthode, exploitée dans le cadre des projets GARP et NAPA, nécessite un découpage fonctionnel et logique de l'architecture mais ouvre en contrepartie des perspectives d'optimisation très intéressantes. Cette technique a également été exploitée dans le cadre du projet ARDOISE, dans lequel des ressources reconfigurables (ATMEL AT40K[35]) sont reconfigurées dynamiquement et partiellement [36].

Les unités fonctionnelles reconfigurables Afin d'améliorer plus encore la qualité du couplage entre le processeur et la ressource reconfigurable, certains projets étudient la possibilité d'insérer des blocs reconfigurables au sein même du chemin de données du processeur. L'alimentation en données de la ressource reconfigurable est alors extrêmement efficace et assurée par le biais des registres généraux du processeur. L'intérêt principal de ce type d'approche est que l'extension du jeu d'instructions du processeur hôte autorise l'utilisation de méthodes classiques de compilation. Ainsi l'utilisateur n'a pas à se soucier de l'architecture ciblée et l'utilisation de langages procéduraux tels que le C est satisfaisante. En revanche, la latence des opérations exécutées sur la ressource reconfigurable impose l'emploi d'instructions multi-cycles et limite les gains qui peuvent être attendus.

Les études menées par le passé confirment cette tendance. À titre d'exemple, des expériences menées autour de l'intégration d'unités de calcul reconfigurables supportant aussi bien les traitements entiers que flottants n'ont mis en évidence que des gains en performance de l'ordre de 20% [14]. Dans le même ordre d'idée, l'extension du jeu d'instructions d'un processeur RISC par le bloc logique reconfigurable CHIMAERA ne se traduit que par un même gain moyen en performance de 20% [37].

Ces résultats mitigés illustrent la difficulté de la tâche consistant à intégrer un accélérateur matériel avec un processeur hôte. Dans [38], les auteurs mettent en avant les aspects devant être pris en compte afin de définir un système performant. Leur étude extrait deux principaux critères, a priori antinomiques, influençant l'efficacité du système.

- Les traitements implémentés doivent être relativement complexes : dédier un bloc reconfigurable à des instructions particulières semble donc inadapté. Les auteurs de [38] préconisent l'exécution de traitements critiques nécessitant, au minimum, plusieurs centaines de cycles d'exécution sur le processeur hôte.
- Le couplage entre le processeur hôte et l'accélérateur doit être très étroit. Idéalement, des connexions directes doivent permettre un approvisionnement en données efficace.

Associer ces deux aspects semble dès lors paradoxal puisqu'il nécessite à la fois des connexions directes entre le processeur hôte et l'accélérateur et des bandes passantes extrêmement importantes pour assurer une distribution efficace des données et des configurations. Bien qu'amorcées depuis de nombreuses années, les recherches sur le couplage entre processeur hôte et accélérateur matériel (qu'il soit dédié ou reconfigurable) sont aujourd'hui encore très attractives.

1.2.2 Les architectures reconfigurables au niveau fonctionnel

Les architectures reconfigurables au niveau fonctionnel peuvent être schématisées par la figure 1.11. L'architecture générique représentée sur cette figure consiste en un ensemble d'opérateurs arithmétiques dédiés ou programmables, reliés par un réseau d'interconnexions, qui communiquent vers l'extérieur au moyen de blocs d'entrées/sorties configurables. En fonction de l'aspect programmable ou dédié des ressources de calcul, les modèles de programmation de ces architectures peuvent être très diversifiés et les recherches menées à ce jour pour explorer leur espace de conception sont très actives. Par la suite, nous caractérisons cet espace de conception en se basant sur des réalisations concrètes issues de différents projets (Pléiades [39], RaPiD [40], Chameleon [41], Morphosys [42], FPFA[5] [44], PACT XPP [45], Systolic Ring [46]).

[5]Pour des raisons stratégiques, FPFA a récemment été renommé MONTIUM (Mountain Chameleon) [43].

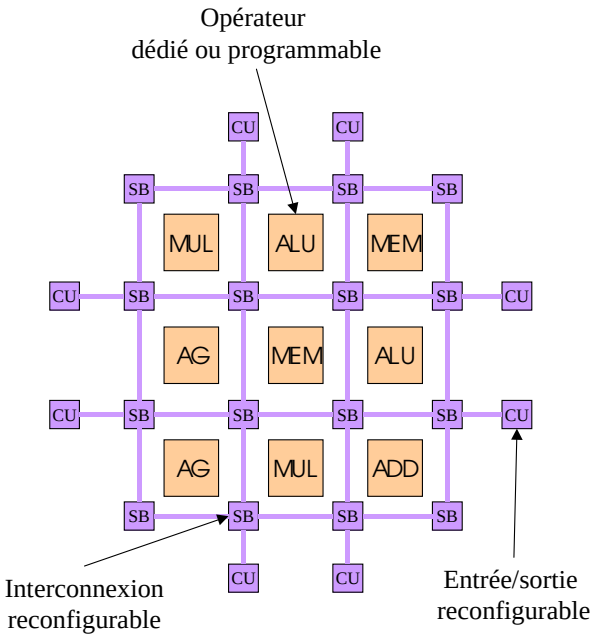

FIG. 1.11 – *Architecture générique des composants reconfigurables au niveau fonctionnel. Elle consiste en un ensemble de ressources de calcul dédiées ou programmables, reliées par un réseau d'interconnexions reconfigurable.*

Les ressources de calcul

Deux approches sont utilisées pour définir les ressources de calcul au niveau fonctionnel. Ces deux approches découlent des modèles de programmation de ces architectures. En effet, dans le cadre de certains projets, les concepteurs ont jugé bon de distribuer le contrôle dans l'architecture, et de définir une hiérarchie dans laquelle le plus bas niveau à pour but de définir la fonctionnalité de l'opérateur. Dans ce cas, les unités de calcul sont relativement complexes et capables de supporter plusieurs opérations. Elles sont typiquement constituées de plusieurs opérateurs (e.g. multiplieur, UAL, registres à décalages, ...) et disposent d'un registre de configuration ou d'un contrôleur local. Ce type d'approche peut par exemple être illustrée par la figure 1.12 représentant la primitive de calcul du Systolic Ring, développé au LIRMM à Montpellier.

L'exploitation de ce type de ressources de calcul complexes, faite dans des projets comme Morphosys ou Chameleon, augmente sensiblement la flexibilité de l'architecture mais se paie par une éventuelle sous-utilisation des ressources et par l'intégration d'un support local de configuration. Ce dernier est typiquement implémenté sous la forme d'un registre de configuration, à la manière de Morphosys ou du Chameleon, ou d'un contrôleur local comme dans le Systolic Ring.

D'autres architectures reconfigurables au niveau fonctionnel, telles que Pléiades, XPP ou RaPiD, se basent en revanche sur des primitives de calcul câblées dont les fonctionnalités sont déterminées en fonction du domaine applicatif. On trouve ainsi dans ces architectures des opérateurs tels que des multiplieurs, des UALs, des générateurs d'adresses, des mémoires, ... et éventuellement certaines fonctions plus spécifiques destinées à accélérer certains traitements, e.g. opérateur ACS (Add-Compare-Select) pour le codage de Viterbi ou l'opérateur SAD (Sum of Absolute Difference) pour l'estimation de mouvement. À l'extrême, certaines

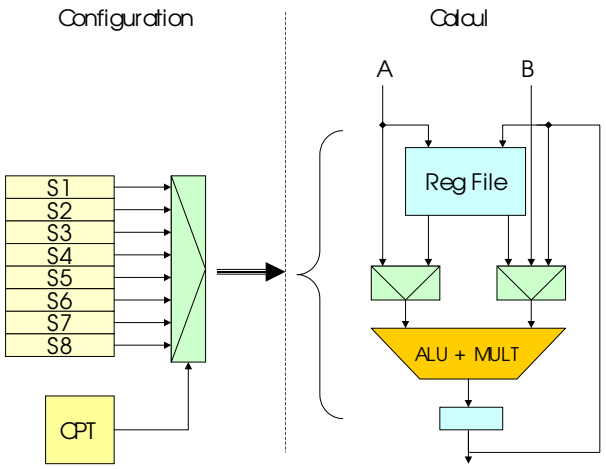

FIG. 1.12 – *Architecture de la ressource de calcul du Systolic Ring, le D-Node. Il consiste en un multiplieur et une UAL câblés travaillant sur des données stockées dans une file de registre. Elle peut être contrôlée localement par le biais d'une mémoire de configuration pouvant stocker jusqu'à huit instructions.*

plateformes, comme Pléiades, définissent tous les opérateurs de l'architecture en fonction du domaine applicatif [47].

Les interconnexions programmables

Les architectures reconfigurables au niveau fonctionnel, compte-tenu du nombre important de ressources de calcul qu'elles intègrent exploitent typiquement l'une ou l'autre des deux topologies d'interconnexions suivantes : les réseaux linéaires ou les réseaux hiérarchiques.

Le réseaux linéaires Les réseaux linéaires permettent de définir des réseaux d'interconnexions relativement simples dans lesquels les communications locales sont très fortement privilégiées. À titre d'exemple, l'architecture RaPiD peut être citée (Fig. 1.13). Dans cette architecture, les communications entre les ressources de calcul se font par le biais d'un réseau segmenté. Des communications directes entre voisins sont assurées et des lignes de plus ou moins longues distances permettent de faire transiter les signaux d'un bout à l'autre du circuit. Ces lignes longues distances réduisent le coût des communications en réduisant la flexibilité du réseau puisque sur ces lignes, les connexions ne sont assurées que tout les 2, 4, 8 ou 16 voisins. Les réseaux linéaires permettent de faire communiquer les ressources de calcul voisines de manière très flexible et efficace. En revanche, les communications entre cellules distantes nécessitent de faire transiter les signaux via plusieurs couches de bus, d'une manière très inefficace. La longueur et la charge de ces connexions se traduisent typiquement par une consommation énergétique élevée.

Ce type de topologie d'interconnexions est particulièrement bien adapté aux applications fortement pipelinées. En effet, dans ces applications, l'essentiel des communications se fait entre des cellules voisines, en exploitant pleinement les connexions locales des réseaux linéaires. En revanche, du fait de l'inefficacité du réseau dans les communications longues distances, l'exploitation des architectures bâties sur ce type de topologie se limite à l'implémentation de pipelines profonds. Les applications disposant d'un parallélisme de grain plus épais (e.g. thread) seront en revanche très inefficacement implémentées.

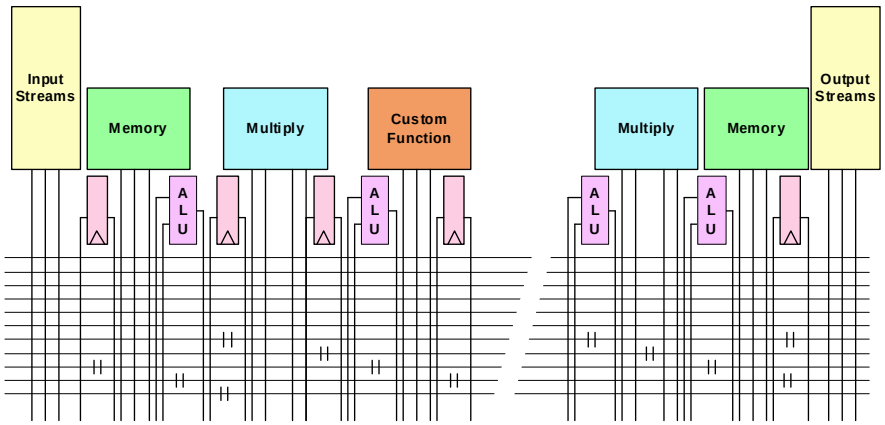

FIG. 1.13 – *Architecture de RaPiD : les unités de calcul communiquent par le biais d'un réseau segmenté. Chaque opérateur peut communiquer avec ses plus proches voisins de manière très flexible. Les communications longues distances sont en revanche beaucoup plus coûteuses et nécessitent la traversée de plusieurs niveaux d'interconnexions.*

Afin d'améliorer l'efficacité de certaines communications longues distances, une solution intéressante a été proposée dans le cadre de l'architecture du Systolic Ring. Elle consiste à définir deux réseaux de communications en anneau. Le premier, dit "direct", sert principalement aux connexions locales et le second, orienté a l'opposé du premier et dit "inverse", est utilisé pour des rebouclages qui sont typiquement caractérisés par des distances de communications importantes. La flexibilité théorique de ce réseau est cependant difficilement exploitable tant il est irrégulier. L'impact de la phase de placement sur la qualité du routage sera en effet dans ce cas déterminante.

Les réseaux hiérarchiques Dès lors que les architectures ciblent des domaines applicatifs plus vastes, les réseaux linéaires deviennent inexploitables. En effet, si la majorité des applications peuvent être décrites de manière à exhiber un pipeline d'exécution profond, d'autres se caractérisent par des niveaux de parallélisme différents et mettent en œuvre des communications globales critiques. Afin de supporter ces applications, un certain nombre de projets se sont basés sur des topologies d'interconnexions hiérarchiques dans lesquelles l'efficacité des connexions locales ne se fait pas au détriment des communications longues distances.

Ces architectures se décomposent en plusieurs niveaux de hiérarchie. Le Chameleon, commercialisé depuis 2000 par *Chameleon Systems*, est un représentant relativement charismatique de ces architectures (Fig. 1.14). Ses 108 unités arithmétiques sont organisées de manière hiérarchique, tout d'abord réunies par groupe de neuf au sein de tuiles, puis regroupées par tranche. Le passage d'un niveau de hiérarchie à un autre se traduit par une réduction de la flexibilité des interconnexions. Dans le même ordre d'idée, les architectures Pléiades et FPFA adoptent également une organisation hiérarchique de leurs ressources de calcul, en se limitant cette fois à deux niveaux de hiérarchie.

Les modes d'utilisation des architectures reconfigurables au niveau fonctionnel

En limitant la reconfiguration aux seules interconnexions, les architectures reconfigurables au niveau fonctionnel ont l'opportunité de réduire très sensiblement le volume de données de configuration. Dès lors, la mise en œuvre de techniques de reconfiguration dynamique ou

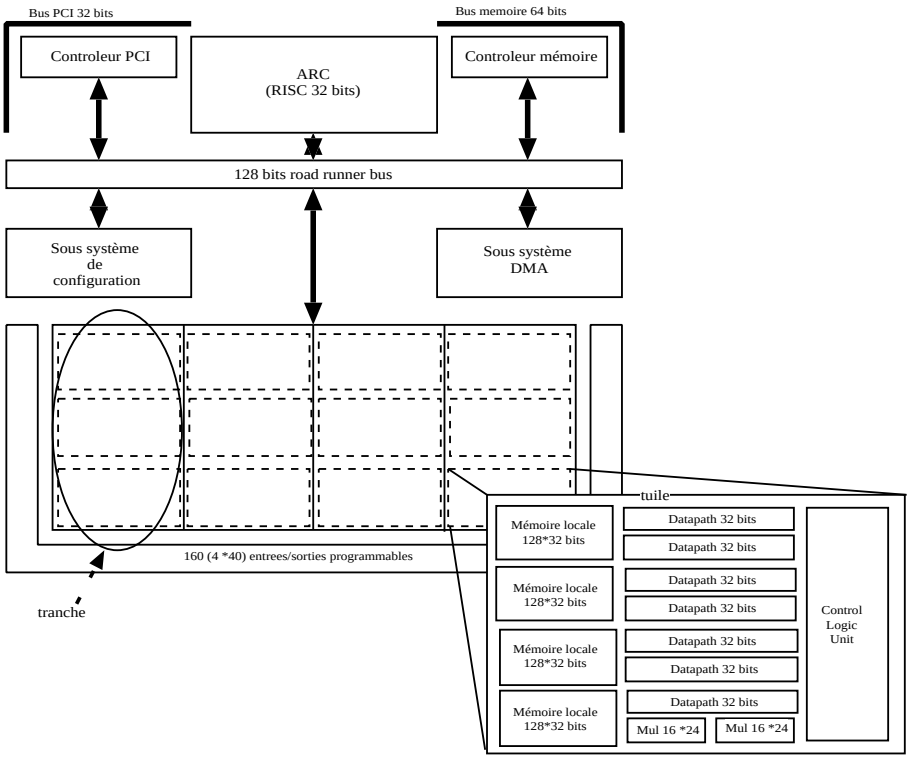

FIG. 1.14 – *Architecture du Chameleon : cette architecture est décomposée en quatre tranches, elles même constituées de 3 tuiles. Au plus bas niveau de la hiérarchie, diverses unités de calcul sont interconnectées avec un réseau totalement flexible. Cette flexibilité s'amenuise à mesure que l'on remonte les différents niveaux de hiérarchie.*

partielle devient beaucoup plus simple.

Les systèmes autonomes Afin d'exploiter pleinement le potentiel de ces architectures, bon nombre de projets associent dans un même système le bloc reconfigurable à un contrôleur chargé de gérer ses reconfigurations. Ce contrôleur est typiquement un processeur généraliste de type RISC. L'exploitation du bloc reconfigurable passe ici par une extension du jeu d'instructions du processeur afin d'autoriser des communications explicites entre la ressource reconfigurable et le processeur hôte.

La définition d'un système autonome impliquant la possibilité de traiter tous les types d'applications (et pas seulement les calculs intensifs), un soin tout particulier doit être apporté à la reconfiguration du bloc reconfigurable. En effet, celle-ci doit être suffisamment rapide pour ne pas perturber l'exécution d'un algorithme par des phases de "gel" trop longues et/ou trop fréquentes. Les recherches menées sur ce thème sont à l'heure actuelle très actives et se traduisent par une grande variété de mise en œuvre de la reconfiguration.

Une première solution, étudiée dans le cadre des FPGAs dits *multi-contextes* et implémentée dans le Chameleon [41], consiste à définir plusieurs plans de configuration. Cette méthode autorise la reconfiguration partielle et dynamique du bloc reconfigurable puisqu'un plan de configuration peut être modifié par le contrôleur pendant qu'un second spécifie la fonctionnalité du bloc reconfigurable. Bien qu'extrêmement performante (la reconfiguration ne prend que le temps d'une commutation sur l'entrée d'un multiplexeur), cette solution est très critiquable

en terme de surface puisqu'elle nécessite la présence sur le circuit de plusieurs mémoires de configuration. Son coût énergétique est par ailleurs très élevé.

Une autre solution destinée à accélérer les temps de reconfiguration consiste à distinguer la configuration matérielle de l'architecture, décrivant la structure du chemin de données, et la configuration logicielle qui la contrôle. La première nécessite un nombre relativement important de données de configuration (quelques centaines, voire milliers de bits) qui sont maintenues pendant toute la durée d'un traitement. La seconde est quant à elle caractérisée par un volume de données de configuration limité (quelques dizaines de bits) mais est amenée à évoluer très souvent. Elle permet par exemple de spécifier un *reset* ponctuel des accumulateurs ou d'initialiser certaines variables avant de rentrer dans une boucle. Cette solution, exploitée notamment dans RaPiD et FPFA, se traduit par une optimisation de l'utilisation des ressources mais se paie par une augmentation du nombre de modèles de programmation du système [48]. L'architecture XPP autorise également un mode de configuration similaire. Sur cette architecture, il est en effet possible de reconfigurer ponctuellement certains blocs particuliers de l'architecture, sans affecter le fonctionnement du reste du circuit. Ce mécanisme, défini sous le terme de *differential configuration* [49], cible principalement le chargement de constantes au sein du chemin de données pour spécifier, par exemple, un *reset* ponctuel des accumulateurs. Il peut par ailleurs être utilisé afin de réduire le temps de configuration.

Enfin, une dernière solution consiste à distribuer le contrôle au même titre que les ressources de calcul. Chaque bloc de calcul doit alors disposer d'un contrôleur gérant les chargements des configurations et les communications avec l'extérieur. Bien qu'efficace, cette solution pose de sérieux problèmes de synchronisation entre les différents éléments de l'architecture. Ce problème est intelligemment réglé dans Pléiades par l'emploi de mécanismes de communication de type Globalement Asynchrone Localement Synchrone (GALS) [50].

Les accélérateurs matériels Bien que la réduction du volume de configuration associé aux architectures reconfigurables au niveau fonctionnel autorise la définition de modèles de programmation relativement simples et par voie de conséquence, la définition de systèmes autonomes, certains projets préfèrent définir des accélérateurs matériels et se décharger de la gestion du système. L'intérêt majeur de cette approche est d'autoriser les équipes de recherche à se concentrer sur l'architecture du bloc reconfigurable. En contre-partie, des mécanismes de communications fiables et efficaces doivent être mis en œuvre afin d'autoriser un couplage simple entre le bloc reconfigurable et l'architecture système qui l'intégrera. Cette solution nécessite également des outils de développement très facilement interfaçables ainsi que des modèles de programmation simples.

1.2.3 Les architectures reconfigurables au niveau système

Les architectures reconfigurables au niveau système sont étudiées depuis de nombreuses années et sont plus généralement appelées les processeurs programmables (Fig. 1.15). Elles se déclinent sous de nombreuses formes en fonction de l'architecture de leur chemin de données (RISC, CISC, DSP) et de la méthode employée pour exploiter le parallélisme de l'application (superscalaire, VLIW).

Les ressources de calcul

Les ressources de calcul intégrées dans les processeurs programmables sont classiquement des unités généralistes telles que des UALs, capables de traiter les opérations les plus courantes.

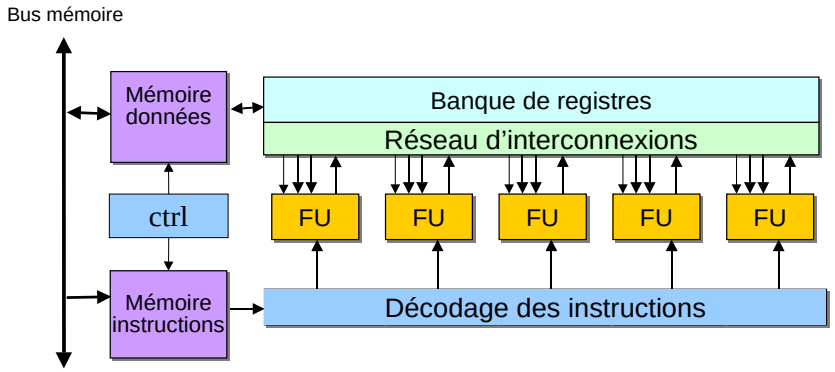

FIG. 1.15 – *Architecture générique des composants reconfigurables au niveau système. Elle est constituée d'un ensemble de ressources travaillant sur une banque de registres centrale. Les opérations sont ordonnancées par un contrôleur unique.*

Certaines architectures disposent également d'unités plus spécifiques, adaptées à des domaines applicatifs bien ciblés. L'exemple le plus connu concerne les multiplieurs et les unités de décalages intégrées dans les chemin de données des DSPs.

Récemment sont apparues des solutions architecturales plus évoluées, distribuées sous la forme d'IP, qui offrent la possibilité aux utilisateurs de spécialiser leur architecture en intégrant des unités de calcul spécialisées [51]. Pour exploiter ces nouvelles ressources de calculs, ces cœurs de processeurs doivent pouvoir modifier leur jeu d'instructions. Les constructeurs proposant ces solutions doivent donc disposer d'outils de conception performants, capables d'adapter le flot de conception à ces nouvelles instructions. Les processeurs CARMEL [52] et ReAL [53, 54], proposés respectivement par Infineon technologies et Philips, sont des exemples d'architecture supportant l'intégration de fonctionnalité dédiées. Ces architectures sont bien souvent citées sous le terme de processeurs configurables [14].

Le nombre d'unités de calcul intégrées dans les architectures reconfigurables au niveau système est particulièrement limité. Depuis les premiers processeurs qui ne disposaient que d'une unique ressource de calcul, un certain nombre de progrès ont cependant été réalisés. Aujourd'hui, les processeurs les plus performants en intègre huit. La gestion de ces ressources de calcul est complexe et nécessite l'exploitation de modèles de programmation nettement plus évolués que celui proposé par John Von neuman [8]. À l'heure actuelle, deux approches sont utilisées.

- L'approche superscalaire qui se traduit par l'intégration d'un support matériel à l'exécution concurrente de plusieurs instructions.
- L'approche VLIW qui se base sur des outils de développement performants pour extraire un maximum de parallélisme au niveau instructions (ILP : Instruction-Level Parallelism) dans l'application.

Cette exploitation du parallélisme au niveau instructions permet d'augmenter sensiblement les performances des processeurs. Cependant, cette augmentation des performances s'accompagne typiquement d'une sous-utilisation des ressources. La levée des dépendances de données et l'extraction du parallélisme d'instructions (dans des descriptions procédurales) sont des tâches aujourd'hui encore trop mal maîtrisées pour exploiter pleinement le potentiel de ces architectures.

Les interconnexions

Contrairement aux architectures présentées dans les sections précédentes, les différents éléments constituant un processeur sont statiquement connectés. Les implémentations réalisées sur ce type d'architecture étant nécessairement de type temporel, tout transfert de données se traduit par le stockage de résultats intermédiaires dans une file de registres.

L'absence de flexibilité des interconnexions se traduit donc par de lourdes pénalités en performances. En contre-partie, très peu de bits suffisent à reconfigurer le chemin de données de ces architectures et ce dernier peu aisément être reconfiguré à chaque cycle. Ceci offre donc à ces architectures une flexibilité optimale puisqu'elles sont capables d'exécuter n'importe quelle application.

Modes d'utilisation des architectures reconfigurables au niveau système

L'un des principaux atouts des architectures reconfigurables au niveau système est de supporter l'exécution de structures de contrôle complexes (branchements conditionnels, gestion d'interruptions, ...). En effet, ce type de traitement nécessite des implémentations temporelles puisque d'un cycle à l'autre, le comportement de l'architecture doit pouvoir être adapté à l'environnement extérieur. Les architectures présentées dans les sections précédentes privilégiant des implémentations spatiales, plus efficaces lors des traitements réguliers, les processeurs programmables ne souffrent d'aucune concurrence dans le cadre du traitement de ces applications.

La flexibilité des processeurs programmables leur ouvre une quantité impressionnante de domaines d'utilisation. Par voie de conséquence, ils sont aujourd'hui intégrés dans la quasi-totalité des systèmes numériques. Que ce soit à des fins de gestion du système ou de traitement, le concepteur dispose d'une palette de solutions lui permettant de répondre à la quasi-totalité des contraintes qu'il est susceptible de rencontrer. Les processeurs restent cependant aujourd'hui encore incapables de répondre conjointement à des contraintes de haute performance et de faible consommation. La principale explication à cet état de fait est qu'il est extrêmement délicat de limiter les gaspillages d'énergie associés à la lecture et au décodage des instructions, de même que ceux associés aux accès aux données dans de larges bancs mémoire.

1.3 Potentiel et limitations des architectures reconfigurables

Dans le cadre des applications mobiles, trois principales contraintes doivent être considérées : la consommation, les performances et la flexibilité. Cette section de conclusion évalue grossièrement ces trois aspects pour les architectures décrites précédemment.

1.3.1 Efficacité énergétique des architectures reconfigurables

Concevoir une architecture faible consommation n'est pas une chose facile. Cette tâche nécessite en effet la prise en compte de ce critère à tous les niveaux de la conception, depuis la définition du système d'exploitation jusqu'à la réalisation physique du circuit. Dans les paragraphes qui suivent, nous présentons les atouts associés aux différentes familles architecturales présentées jusqu'à lors [12]. Il appartient alors aux concepteurs de fournir l'effort (très important) nécessaire à l'exploitation de l'éventuel potentiel de réduction d'énergie de ces architectures.

La principale métrique utilisée dans cette étude pour évaluer cette efficacité énergétique est le MOPS/mW. Celle-ci est définie par l'équation 1.1 [57]. Dans cette équation, N_{OP} représente le nombre moyen d'opérations réalisées par cycle. Le produit $N_{OP}.F_{clk}$ représente donc la puissance de calcul développée par l'architecture considérée et s'exprime en MOPS. Le produit $A_{Chip}.\alpha.C_{SN}.F_{clk}.V_{DD}^2$ représente quant à lui la consommation de puissance sous-jacente à l'exécution de N_{OP} opérations par cycle. La valeur A_{Chip} représente la surface du circuit et C_{SN} la capacité normalisée par unité de surface. L'activité moyenne du composant est finalement prise en compte par l'intermédiaire du paramètre α.

$$\text{Efficacité énergétique} \left(\frac{MOPS}{mW}\right) = \frac{N_{OP}.F_{clk}}{A_{Chip}.\alpha.C_{SN}.F_{clk}.V_{DD}^2} \tag{1.1}$$

La valeur de A_{Chip} est déterminée par l'équation 1.2. Celle-ci fait apparaître la surface des opérateurs, exprimée comme étant la somme du produit du nombre d'opérateurs et de la surface moyenne des opérateurs ($N_{op}.A_{op}$), de la surface des éléments de mémorisation (A_{mem}) et de celle des ressources de contrôle (A_{ctrl}).

$$A_{Chip} = N_{op}.A_{op} + A_{mem} + A_{ctrl} \tag{1.2}$$

De ces deux équations peuvent être extraits les principaux objectifs devant être visés pour obtenir une architecture efficace du point de vue énergétique. Le nombre d'opérations est directement dépendant de la puissance de calcul requise par l'application. Le paramètre C_{SN} est quant à lui un paramètre technologique. Ces deux paramètres ne peuvent donc pas être optimisés. En conséquence, L'optimisation de la consommation d'énergie passe par l'optimisation de la surface moyenne des opérateurs, de la hiérarchie mémoire et des ressources de contrôle, ainsi que par la minimisation de l'activité du composant. Une architecture ne sera donc considérée comme étant efficace du point de vue énergétique que dans la mesure où l'essentiel de la consommation d'énergie est localisée dans les opérateurs et que ces derniers sont optimisés.

Les architectures reconfigurables au niveau logique

Les architectures reconfigurables au niveau logique intègrent énormément de ressources afin de contrôler tous les bits d'une application indépendamment les uns des autres. Une large majorité des applications manipulent cependant des données arithmétiques. De ce fait, construire des opérateurs arithmétiques implique l'utilisation de nombreuses ressources de calcul qui doivent bien sûr être interconnectées. Puisque les réseaux intégrés véhiculent des bits d'une manière extrêmement flexible, les signaux doivent traverser un grand nombre de boites de commutation, ce qui conduit à d'importants gaspillages d'énergie. De nombreux travaux ont porté sur l'analyse de la distribution de la consommation dans les circuits FPGA [58, 59, 60, 61]. Toutes aboutissent à la constatation que la principale source de consommation de ces composants est le réseau d'interconnexions. À titre d'exemple, dans une étude du FPGA Xilinx Xc4003A, Eric Kusse, dans [60], estime le pourcentage d'énergie consommée dans le réseau d'interconnexions à 65%, contre seulement 5% dans les ressources de calcul.

Un autre inconvénient associé à la faible granularité de ces architectures vient du volume de données de configuration qui leur est associé. En effet, les phases de reconfiguration nécessitent la transmission d'une quantité impressionnante de données qui doivent nécessairement être

stockées dans de larges mémoires centrales. Ce volume d'informations dépasse allègrement le million de bits dans les dernières générations de composants FPGA [19, 20]. Lorsque le système nécessite des reconfigurations du bloc reconfigurable (e.g. télécommunications 3G), ces phases se traduisent donc par des pénalités énergétiques très importantes.

Varghese George, à Berkeley, a travaillé avec un certain succès à la réduction de la consommation des FPGAs [62], notamment en limitant la consommation des interconnexions [63]. Cependant, à l'heure actuelle, ces architectures restent très inefficaces d'un point de vue énergétique. Cet état de fait sera vérifié dans le chapitre 4 et apparaît sur le tableau 1.1. Celui-ci fait en particulier apparaître l'influence du volume et des fréquences de reconfiguration sur l'efficacité énergétique des architectures. Les chiffres indiqués dans ce tableau ne visent qu'à donner un ordre de grandeur et n'ont qu'une valeur indicative.

type d'architecture	volume des données de configuration	Fréquence des reconfigurations	Efficacité énergétique	Flexibilité
ASIC	++	++	++	- -
	0 bit	0	≈ 100 MOPS/mW	
Reconfiguration logique	- -	+	-	+
	≈ 1 M bits	application	≈ 1 MOPS/mW	
Reconfiguration fonctionnelle	+	+/-	+	+
	≈ 100 bits	fonction	≈ 10 MOPS/mW	
Reconfiguration système	++	- -	-	++
	≈ 10 bits	cycle	≈ 1 MOPS/mW	

TAB. 1.1 – *Efficacité énergétique des architectures reconfigurables : la première colonne estime le volume des données de configuration et en donne un ordre de grandeur. La seconde se concentre sur la fréquence des reconfigurations et précise si celles-ci interviennent à chaque cycle, à chaque appel de fonction ou à chaque changement d'application. L'efficacité énergétique est approximée dans la troisième colonne et la dernière rappelle la flexibilité associée à chaque architecture.*

Les architectures reconfigurables au niveau fonctionnel

Limiter les cibles de la reconfiguration aux seules interconnexions peut avoir un impact très favorable sur la consommation des architectures reconfigurables au niveau fonctionnel. En effet, se concentrer sur la reconfiguration des interconnexions permet de réduire très sensiblement le volume de données de configuration. Dans le même temps, si les phases de reconfigurations n'interviennent qu'entre les phases de calcul intensif, les étapes de recherche et de décodage de configuration sont très occasionnelles. En effet, la règle des 80/20 affirme que 80% du temps d'exécution est consommé par seulement 20% du code, ce qui signifie que certaines portions du programme sont exécutées pendant de longues périodes [64, 65]. Ainsi, il est possible de réduire le nombre de lectures et de décodages de configuration en ne réalisant ces opérations que lors des appels de fonction (la configuration doit alors permettre l'exécution de toute la fonction). En réduisant à la fois le nombre et le coût des reconfigurations, les architectures reconfigurables au niveau fonctionnel sont donc en mesure de minimiser le gaspillage d'énergie associé à la distribution du contrôle dans le circuit.

Un autre aspect intéressant, sous-jacent à la capacité à reconfigurer le réseau d'interconnexions, est la possibilité d'adapter le parallélisme de l'application à celui de l'algorithme exécuté. En exploitant pleinement le parallélisme intrinsèque de l'algorithme, la fréquence

de fonctionnement peut être réduite de même que la tension d'alimentation et par voie de conséquence, la consommation de puissance et d'énergie du circuit.

Le potentiel des architectures reconfigurables au niveau fonctionnel est malheureusement bien souvent inexploité dans les projets de recherche. En effet, hormis Pléiades, la plupart des études menées sur ce thème se concentrent uniquement sur le gain en performance au prix d'une consommation d'énergie non maîtrisée. Rares sont donc les architectures pour lesquelles une efficacité énergétique peut être extraite de la littérature. En conséquence, l'ordre de grandeur de l'efficacité énergétique des architectures reconfigurables apparaissant dans le tableau 1.1 concerne uniquement les projets pour lesquels un effort conséquent est mené pour optimiser ce paramètre [66].

Les architectures reconfigurables au niveau système

Les architectures reconfigurables au niveau système sont réputées comme étant inefficace d'un point de vue énergétique. Une des principales explications à cet état de fait est que le surcoût associé à la distribution du contrôle dans l'architecture est considérable. En effet, à chaque cycle, un processeur programmable se doit de lire une instruction et de la décoder pour contrôler les unités fonctionnelles. Il est tout de même à noter que certains processeurs embarqués, tels que le DSP16xx de Lucent Technologies [67] ou l'ADSP 21xx d'Analog Devices [68], limitent le coût énergétique de cette opération par l'insertion de buffer mémorisant les dernières instructions lues [69].

Un autre inconvénient associé aux architectures Harvard ou Von Neuman est l'exploitation de larges mémoires centrales. Dans ces architectures, le coût des accès aux données est prohibitif et la dissipation d'énergie dans ces architectures est dominée par la mémoire centrale. Les processeurs souffrent finalement de la quantité limitée de parallélisme pouvant être traitée. Bien que beaucoup de processeurs programmables exploitent un certain niveau de parallélisme (données [70], instructions [71] ou tâches [72]), leur degré de parallélisme est réduit afin de limiter le nombre d'unités fonctionnelles du composant. En effet, un des aspects les plus importants lors de la conception d'un processeur programmable est le coût du circuit. De ce fait, pour avoir une bonne densité de calcul (i.e. ratio entre les performances et la surface), le concepteur doit limiter le degré de parallélisme de l'architecture. En conséquence, puisque ces architectures ne peuvent exploiter que partiellement le parallélisme de l'application, elles doivent fonctionner à des fréquences plus élevées, nécessitant dès lors des arbres d'horloges plus gros et des tensions d'alimentation plus grandes. Bien que, au sein l'espace de conception des processeurs programmables (DSPs, microcontrôleurs, processeurs généralistes, ...), des différences notables existent en terme d'efficacité énergétique, celle-ci est typiquement de l'ordre de quelques MOPS/mW [14, 66].

1.3.2 Performances des architectures reconfigurables

Le critère de performance est sans aucun doute le plus important pour valoriser une architecture. Quelle que soit l'efficacité d'une architecture en consommation ou en surface, elle ne sera en effet jamais utilisée si elle ne dispose pas de la puissance de calcul nécessaire à l'exécution d'une application.

Les architectures reconfigurables au niveau logique

De par leur structure, les architectures reconfigurables au niveau logique sont prédestinées à implémenter spatialement leurs applications. Le niveau de performance est alors directement

lié à la quantité de ressources qu'ils intègrent. L'efficacité de ces architectures dans le cadre de la manipulation de données binaires, voire de petite taille (< 4 bits), est indéniable et ces architectures ne disposent d'aucune réelle concurrence dans ce domaine.

Le problème est cependant tout autre pour les traitements arithmétiques. Comme il a déjà été dit, construire un opérateur arithmétique dans de telles architectures nécessite l'utilisation et l'interconnexion d'un grand nombre de ressources de calcul. La quantité de ressources de routage devant être traversées devient alors prohibitive et les performances des opérateurs ainsi synthétisés sont très nettement dégradées par rapport à des solutions *full-custom*. Bien que l'introduction d'architectures à grain plus épais limite cette tendance, les résultats obtenus restent insatisfaisant pour des traitements critiques. Ainsi, pour atteindre un niveau de performance comparable aux solutions de type *full-custom*, les implémentations doivent accroître leur niveau de parallélisme, ce qui se traduit par des solutions inefficaces du point de vue de la densité de calcul. Pour réduire l'importance de ce problème, les architectures les plus récentes intègrent au sein de leur structure reconfigurable des opérateurs câblés travaillant au niveau arithmétique (e.g. VIRTEX-II [19], Stratix [73]).

Les architectures reconfigurables au niveau fonctionnel

En se basant sur des opérateurs *full-custom*, les architectures reconfigurables au niveau fonctionnel ne souffrent pas du problème évoqué précédemment, tout en maintenant l'avantage d'implémenter spatialement leurs applications. La possibilité d'exploiter le parallélisme à grain épais (données, instructions, tâches) de l'application leur garantit de très hautes performances.

En revanche, ces architectures n'ont pas la possibilité d'exploiter le parallélisme de niveau bit des applications. En effet, dans ce contexte, l'emploi d'opérateurs travaillant au niveau arithmétique, même s'ils autorisent les traitements logiques, est très inefficace puisque avant de pouvoir traiter un bit particulier dans un mot de N bits, il est nécessaire de préalablement l'extraire en lui appliquant un masque. Un traitement logique nécessite donc deux opérations sur une ressource arithmétique. Ce problème est illustré sur la figure 1.16 représentant le traitement logique $S = (A1.B1).(A0 + B0)$ sur une architecture reconfigurable au niveau fonctionnel. Le premier étage est chargé d'extraire les données binaires dans le mot arithmétique et est suivit des deux étages d'exécution. L'utilisation de 7 UALs (de 8 bits dans cet exemple) pour traiter 3 opérations logiques démontre l'inefficacité de ces architectures lorsqu'elles sont utilisées dans de telles conditions.

Pour pallier ce problème, un certain nombre d'architectures, telles que Pléiades ou le Chameleon, intègrent parmi leurs unités de calcul des opérateurs logiques capables de traiter plus efficacement ces opérations.

Les architectures reconfigurables au niveau système

Les architectures reconfigurables au niveau système implémentant leurs applications de manière temporelle, elles sont généralement peu performantes lors des calculs intensifs. L'exploitation du parallélisme des applications se limite typiquement aux données ou aux instructions (VLIW/superscalaires), et l'ILP utilisé en pratique dépasse rarement 8, de manière à assurer une bonne densité de calcul.

Dans [74], Daniel Menard met en évidence un parallélisme d'instructions relativement important (entre 2.8 et 7.5) dans le cadre d'implémentations d'algorithmes de filtrage sur le TMS320C64x (supportant un ILP maximum de 8). Dans le même ordre d'idée, Emmanuel

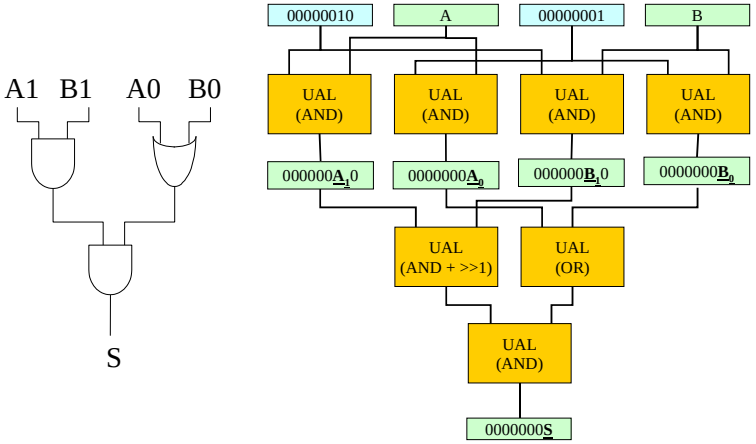

FIG. 1.16 – *(a) Traitement d'une opération logique sur (b) une architecture reconfigurable au niveau fonctionnel : une première étape vise tout d'abord à extraire les bits dans les mots arithmétiques. Les fonctions logique sur 1 bit sont ensuite traitées sur des opérateurs arithmétiques. Sur cet exemple, 7 UALs sont nécessaires au traitement de 3 opérations logiques.*

Gaudry, dans [1] met en avant un ILP variant de 2.3 à 3.4 pour le traitement d'un *Rake Receiver* [75] sur un processeur développé à ST microelectronics, le Lx [76] (disposant d'un ILP maximum de 4). Ceci démontre bien la capacité des architectures reconfigurables au niveau système à proposer de hautes performances. Ceci se fait cependant au prix d'un contrôle très complexe.

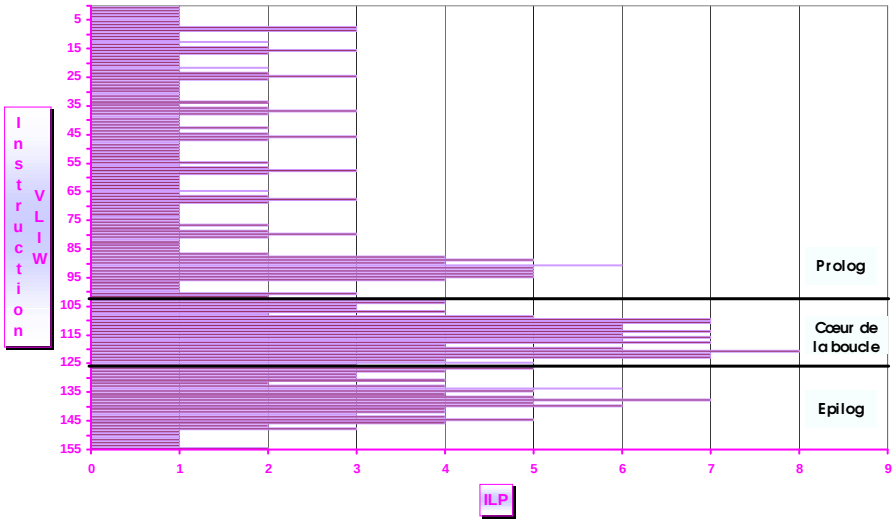

FIG. 1.17 – *Variation de l'ILP lors de l'exécution d'un Complex despreading [77] : celui-ci est très important à l'intérieur du cœur de boucle mais est limité le reste du temps.*

La figure 1.17 décrit le niveau de parallélisme d'un *complex despreading* [77], traitement clé d'un récepteur WCDMA (cf. chapitre 4.1, page 112) . La méthode utilisée pour extraire du parallélisme de l'application est du pipeline logiciel [78] et l'architecture cible est le TMS320C64x [79]. Cette figure fait apparaître un parallélisme important lors de l'exécution du cœur de

boucle, avec un ILP moyen de 5.5. Dans ce cas particulier la quasi-totalité des ressources de l'architecture est utilisée. En revanche, dans le reste du code, l'ILP est très faible et varie typiquement entre 1 et 3 pour augmenter jusqu'à 4 dans les phases de prologue et d'épilogue, propres aux applications pipelinées logiciellement. Ainsi, définir une architecture reconfigurable au niveau système combinant à la fois haute performance et haute densité de calcul est un exercice extrêmement délicat. Compte-tenues des contraintes de coût inhérentes aux marchés ciblés par les processeurs programmables, les concepteurs de ces systèmes privilégient généralement la densité de calcul au détriment des performances.

Ces architectures sont en revanche beaucoup plus performantes que celles évoquées précédemment dans le traitement des applications irrégulières. Ces applications, caractérisées par un très faible niveau de parallélisme et par des comportements non déterministes, nécessitent des implémentations temporelles. En effet, il est très difficile de prédire le comportement de ces algorithmes à un cycle donné, celui-ci dépendant bien souvent du résultat des traitements réalisés au cycle précédent ou de l'environnement extérieur (e.g. arrivée d'interruption). Si les processeurs programmables sont peu performants pour les calculs intensifs, ils sont donc parfaitement adaptés aux traitements irréguliers et aux applications orientées contrôle.

1.3.3 Flexibilité des architectures reconfigurables

Compte tenu du fait que toutes les architectures examinées dans ce chapitre peuvent être reconfigurées pour s'adapter à un traitement donné, elles peuvent être considérées comme optimales du point de vue de la flexibilité. L'efficacité de la reconfiguration vient cependant pondérer cette remarque. En effet, en fonction du coût de la reconfiguration, l'utilisateur fournira un effort plus ou moins important pour limiter le nombre de ces configurations et donc le temps passé à reconfigurer le circuit. Deux aspects sont donc à prendre en compte pour juger de la flexibilité d'une architecture : le volume de données de configuration et l'efficacité de la reconfiguration. Nous considérons ici l'efficacité de la reconfiguration comme étant directement liée au nombre de ces reconfigurations. Didier Demigny, dans [80], formalise ce critère sous le nom de rémanence.

Ce critère de rémanence est défini de la manière suivante. Soit N_a le nombre d'opérateurs utilisés dans une architecture et F_e sa fréquence d'exécution. Soit N_c le nombre d'opérateurs reconfigurés à chaque cycle d'une horloge de configuration de période T_c, la rémanence R est déterminée par l'équation suivante :

$$R = \frac{N_a.F_e}{N_c.F_c} \tag{1.3}$$

En d'autres termes, la rémanence indique le nombre de cycles nécessaires à la reconfiguration complète de l'architecture. Les auteurs de [80] justifient l'intérêt de ce critère par son caractère discriminant. Il donne en effet une idée du nombre de données minimum devant être traitées entre deux reconfigurations pour que l'architecture exploite efficacement ces ressources de calcul : typiquement les configurations doivent être maintenues pendant un nombre de cycles supérieur à 10 R. Les auteurs de [80] font également remarquer que l'inverse de R mesure le caractère dynamique de la reconfiguration. En effet, plus R est faible, plus l'architecture peut être reconfigurée rapidement, donc plus sa reconfiguration peut être considérée comme dynamique.

Compte tenu de ce critère de rémanence, nous pouvons donc conclure que la solution optimale en terme de flexibilité est le processeur programmable ($R = 1$). Ce type d'architecture peut être utilisée pour n'importe quel traitement sans pénaliser l'exécution par les phases de reconfiguration. En pratique, les architectures reconfigurables au niveau fonctionnel ont une rémanence R de quelques dizaines de cycles. Ces architectures doivent donc privilégier l'implémentation d'applications disposant d'un volume de données important, supérieur au millier de données arithmétiques (e.g. traitement du signal). Les architectures reconfigurables à grain fin tels que les FPGAs, ont quant à elles une rémanence de plusieurs milliers (jusqu'à plusieurs millions pour les plus gros FPGA) de cycles. Elles doivent donc cibler des traitements manipulant de très gros volumes de données (e.g. traitement de l'image). Par l'utilisation de cellules de calcul à grain plus épais (e.g. DReAM, Chess) ou par des techniques de reconfigurations partielles (NAPA, GARP), certains projets réduisent néanmoins cette rémanence dans des proportions qui leurs ouvrent des domaines applicatifs plus irréguliers.

1.4 Synthèse

Dans la première section de ce chapitre nous avons délimité l'espace de conception des architectures reconfigurables. Par le biais d'exemples issus de la littérature, nous avons ensuite caractérisé cet espace de conception et dressé un état des solutions architecturales envisageables. Dans la section précédente nous avons finalement évalué ces architectures suivant les trois principaux critères associés aux applications mobiles de prochaines générations.

Dans le cadre des architectures reconfigurables, des efforts conséquents ont été produits afin de réduire le coût de la reconfiguration et ainsi augmenter leur flexibilité. Ainsi, depuis l'échec cuisant du FPGA Xilinx Xc6200 [81] ayant amorcé ces recherches, de nombreux mécanismes architecturaux ont été introduits pour simplifier la gestion de la reconfiguration tout en réduisant son surcoût. Ainsi, grâce à la reconfiguration partielle, à la définition de grain de reconfiguration plus épais, ..., les performances des architectures reconfigurables permettent désormais de supporter de très hauts niveaux de performance.

À l'heure actuelle en revanche, la reconfiguration dynamique n'est pas exploitée dans le cadre de la maîtrise de la consommation. Cette contrainte, bien qu'unanimement reconnue comme l'une des plus critiques pour les systèmes embarqués, n'est quasiment jamais prise en compte lors de la conception des architectures reconfigurables. Pléiades est l'une des rares architectures pouvant justifier cette qualification. En contre-partie, pour assurer cette efficacité énergétique, cette plateforme se doit d'être dédiée à un domaine applicatif bien identifié, à l'image de la spécialisation de l'architecture MAIA pour le codage de la parole [39].

Afin de répondre concurremment à ces contraintes de hautes performance, de flexibilité et de faible consommation, nous avons défini un nouveau modèle architectural. Celui-ci, baptisé DART, est présenté dans le chapitre suivant.

Chapitre 2

DART : une architecture reconfigurable dynamiquement

Sommaire

2.1 Influence des télécommunications 3G sur la définition de l'architecture

Le chapitre introductif a mis en avant un certain nombre de contraintes liées aux applications mobiles. L'état de l'art présenté dans le second chapitre a quant à lui contribué à caractériser les différents paradigmes des systèmes intégrant des structures reconfigurables. Dans cette section, nous allons récapituler les contraintes associées aux télécommunications de troisième

génération et extraire des solutions présentées précédemment, celles qui sont susceptibles d'autoriser leur support [10, 15].

2.1.1 Adéquation Algorithme-Architecture (AAA)

En matière de conception d'architecture, tout est affaire de compromis. Ainsi, l'architecture la plus performante, la plus flexible, la plus simple à programmer, ... n'existera jamais. Il est donc utopiste de tenter de définir une telle architecture généraliste. En revanche, lorsque le domaine applicatif ciblé est bien cadré, il est possible de faire les concessions qui autorisent la définition d'une architecture optimale, au sens où elle répond "au mieux" à son cahier des charges. Il s'agit donc ici de rendre l'architecture en adéquation avec l'application (AAA : Adéquation Algorithme Architecture).

Variété de grains de calcul

Un des paramètres les plus significatifs pour caractériser l'application est le grain de calcul. Dans le cadre de notre domaine applicatif, il est impossible d'identifier un grain de calcul unique. En effet, celui-ci évolue tout au long d'une chaîne de communication. Typiquement, il a tendance à se réduire à mesure que l'on se rapproche du support de transmission. Ainsi, les traitements de haut niveau travaillent principalement sur des données arithmétiques. Il s'agit ici de faire du traitement audio, vidéo, ... D'autres algorithmes, parmi lesquels les codages de canal, travaillent quant à eux sur des données de très petites tailles, éventuellement binaires.

Afin de traiter efficacement toutes les applications intégrées aux télécommunications 3G, DART dispose de deux types de ressource. Les premières sont dédiées aux traitements arithmétiques et travaillent sur des données codées sur 16 bits. Nous avons défini la largeur de ces unités en fonction des formats de données les plus couramment manipulés. Les traitements manipulant des données de petite taille, et caractérisés par un parallélisme massif de niveau bit, sont quant à eux implémentés par le biais de ressources de calcul logiques, basées sur des LUTs.

Variété de tailles de données

Si la notion de "grain de calcul" est traditionnellement réservée à la distinction arithmétique/logique, il est important, dans un domaine applicatif aussi vaste que le notre, d'étudier plus en détail les données qui sont manipulées. En particulier, toutes les applications arithmétiques, loin s'en faut, ne manipulent pas des données 16 bits. À titre d'exemple, parmi les traitements multimédia, cohabitent des applications travaillant sur des données 8 bits (image), 13 bits (parole), 16 bits (audio).

Supporter chacune de ces tailles de données implique la conception d'unités de calcul extrêmement flexibles, s'accompagnant des mêmes travers que ceux évoqués pour les architectures reconfigurables au niveau logique. Une solution alternative est de ne supporter qu'un sous-ensemble de ces tailles de données. Dans le cadre de notre architecture nous avons choisi de supporter deux tailles de données : 8 et 16 bits. Cette limitation est transparente pour une très grande majorité d'applications et à l'avantage de permettre la mise en œuvre d'opérateurs SWP (Sub-Word Processing). Cette technique consiste à diviser un opérateur manipulant des données de largeurs N afin de pouvoir exécuter en parallèle deux opérations sur des fractions de mots de largeur N/2. Ce concept est illustré par la figure 2.1 pour les opérations de multiplication et d'addition/soustraction. Compte tenu de son faible impact sur les perfor-

mances lors des traitements 16 bits, l'utilisation du *SWP* est de plus en plus courante dans les processeurs de traitement du signal (e.g. TigerSHARC [84], TMS320C64x [79]).

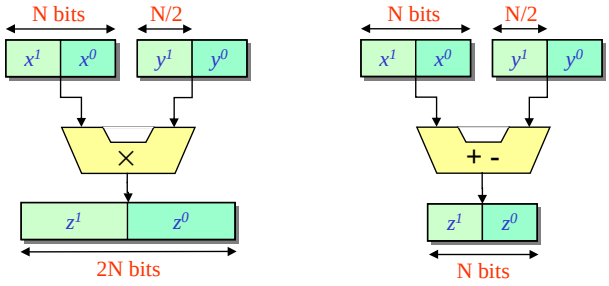

FIG. 2.1 – *Opérateurs SWP. La division d'un opérateur de largeur N permet de traiter en parallèle 2 opérations de largeur N/2.*

Par l'emploi de ces techniques *SWP*, l'architecture est donc en mesure d'augmenter sa puissance de calcul lors des traitements qui manipulent des données 8 bits et qui disposent d'un fort parallélisme de données (e.g. traitements vidéo) [70]. Ceci ce fait cependant au prix d'un placement des données en mémoire relativement délicat à mettre en œuvre puisque les données doivent impérativement être alignées les unes par rapport aux autres. Cette tâche est dévolue aux outils de développement associés à DART.

Variété de motifs de calcul

Cibler un domaine applicatif aussi vaste que les télécommunications 3G se traduit inévitablement par une très grande variété de calcul. Afin de rester en adéquation avec l'algorithme, l'architecture doit donc être en mesure d'adapter son chemin de données à chaque modification du motif de calcul. Compte-tenu de la fréquence de ces modifications dans une chaîne de communication 3G, ces modifications doivent par ailleurs être dynamiques afin de ne pas perturber l'exécution par des phases de reconfigurations trop longues et/ou trop fréquentes.

Dans cette optique de reconfiguration dynamique, nous avons opté pour une reconfiguration au niveau fonctionnel des chemins de données de notre architecture. Afin de limiter le temps de reconfiguration, sans pour autant réduire la flexibilité de l'architecture (et donc son potentiel d'optimisation), nous avons par ailleurs tenté de réduire le volume de données nécessaire à cette opération.

Les télécommunications 3G n'étant pas uniquement constituées de calculs intensifs et réguliers, nous avons par ailleurs autorisé une reconfiguration systématique de l'architecture. En reconfigurant ses chemins de données à chaque cycle (à la manière d'un processeur) notre architecture est donc en mesure d'implémenter certaines portions de code de manière temporelle. La flexibilité de notre architecture est ainsi améliorée au prix de l'intégration de mécanismes architecturaux relativement simples et de la définition de plusieurs modèles de programmation.

2.1.2 Performance et parallélisme

L'une des contraintes les plus sévères inhérentes aux télécommunications 3G est sans aucun doute le niveau de performance requis. Celui-ci, estimé à 12 milliards d'opérations par seconde, est principalement issu des traitements multimédia et des techniques d'accès par

code (WCDMA : Wide-Band Code Division Multiple Access). Supporter un tel niveau de performance passe inexorablement par une exploitation optimale du parallélisme du domaine applicatif.

Outre le parallélisme de données précédemment évoqué, une chaîne de communication de troisième génération dispose également de trois autres types de parallélisme : le parallélisme d'opérations, de tâches et d'applications.

Le parallélisme d'opérations

Le parallélisme d'opérations, ou d'instructions (ILP : Instruction Level Parallelism), est inhérent à tous les algorithmes de calcul. Bien que contrainte par les dépendances de données, son exploitation est généralement relativement simple. Elle nécessite l'introduction de multiples ressources de calcul susceptibles de travailler indépendamment les unes des autres. Pour exploiter ce parallélisme, le contrôleur de l'architecture doit être en mesure de spécifier concurremment à plusieurs opérateurs, la fonction devant être implémentée.

Dans le même temps, les unités fonctionnelles doivent être approvisionnées en données. Il est ainsi indispensable de disposer de très fortes bandes passantes avec la mémoire de données. Afin d'assurer une bande passante suffisante, nous avons décidé de distribuer les données entre de multiples unités de mémorisation. Notre architecture dispose ainsi de nombreuses mémoires de petite taille (256 mots). Notre solution est donc en complète opposition avec les approches programmables traditionnelles où une mémoire unique de forte capacité approvisionne simultanément plusieurs unités fonctionnelles via de multiples ports d'entrée/sortie. Ainsi, nous décidons de risquer un éparpillement (contrôlé) des données afin de maîtriser la consommation d'énergie de la hiérarchie mémoire. En effet, les études montrent que la consommation énergétique par accès croît de manière linéaire avec l'augmentation du nombre de ports d'entrée/sortie des mémoires.

Par ailleurs, afin de limiter la complexité du contrôleur de l'architecture, nous distribuerons la tâche de génération d'adresses à des contrôleurs locaux. Chaque mémoire sera donc associée à un générateur d'adresses dédié qui se chargera de gérer ses accès.

Le parallélisme de tâches

Le parallélisme de tâches (TLP : *Thread Level Parallelism*) décrit le nombre de traitements qui peuvent être exécutés concurremment pour implémenter un algorithme particulier. À titre d'exemple, un filtrage adaptatif est constitué de deux *threads* : l'un concerne l'adaptation des coefficients et le second exécute le filtrage. Ce TLP est nettement plus délicat à exploiter que le parallélisme d'instructions. En effet, le degré de TLP varie très fortement d'une application à une autre, et plus encore entre deux descriptions d'une même application.

Ainsi, pour exploiter ce parallélisme tout en garantissant une densité de calcul optimale, y compris lorsque la description de l'application exhibe un TLP nul, il est nécessaire de définir une architecture capable d'adapter l'organisation de ces ressources de calcul [85]. DART a donc été conçue de manière à autoriser divers compromis entre ILP et TLP. Pour ce faire, nous avons organisé notre architecture en une hiérarchie dont le plus bas niveau est un regroupement d'unités de calcul, appelé DPR (DataPath Reconfigurable). Chaque DPR pouvant être contrôlé indépendamment de ces homologues, il peut implémenter un *thread* particulier de l'application. À l'inverse, lorsque le TLP de l'application est nul, ces DPRs peuvent être interconnectés les uns avec les autres et être considérés par le contrôleur de l'architecture comme une ressource de calcul unique, afin de profiter au maximum de l'ILP de l'application. Entre

ces deux extrêmes, il existe bien sûr diverses possibilités permettant d'obtenir un compromis optimal entre ILP et TLP.

Le parallélisme d'applications

Le parallélisme d'applications peut être considéré comme une extrapolation du TLP. Il s'agit ici d'identifier les applications qui peuvent être exécutées en parallèle. À la différence du parallélisme de tâches, les applications exécutées en parallèle travaillent sur des jeux de données bien distincts. Ce parallélisme d'applications apparaît par exemple sur la figure 1 (page 2). Sur cette figure, il apparaît clairement que les traitements de haut-niveaux peuvent être exécutés concurremment avec les codeurs de source ou de canal par exemple.

Afin d'implémenter concurremment ces applications, un second niveau de hiérarchie a été rajouté à DART. Ainsi, notre architecture a été divisée en *cluster* qui traitent simultanément différentes applications. Puisque ces applications sont totalement différentes, temps du point de vue des motifs de calcul que des données manipulées, ces *clusters* sont autonomes et disposent de leurs propres ressources de contrôle, de stockage et de calcul.

2.1.3 Programmabilité et efficacité énergétique

L'organisation hiérarchique de DART a par ailleurs d'autres bienfaits. En particulier, celle-ci simplifie grandement la définition d'outils de développement et permet de maîtriser totalement la consommation d'énergie.

Maîtrise de la consommation

La maîtrise de la consommation est un enjeu majeur dans le domaine de l'embarqué. C'est également une contrainte particulièrement difficile à satisfaire compte-tenu du nombre de paramètres devant être pris en compte lors de la définition de l'architecture. Concevoir une architecture faible consommation nécessite donc de considérer cette contrainte comme primordiale et ce, à tous les niveaux de conception [86, 12].

Maîtrise de la consommation au niveau système Au niveau système, la maîtrise de la consommation passe par l'adaptation du compromis énergie consommée/puissance de calcul aux besoins de l'application.

Le temps de traversée d'un transistor, donné pour un modèle à canal court par l'équation 2.1, est en effet directement lié à sa tension d'alimentation. Dans le même temps, l'énergie consommée croît de manière quadratique avec cette même tension d'alimentation (Eq. 2.2). Dès lors, en réduisant la tension d'alimentation, l'énergie consommée dans le circuit est réduite. Cette réduction se traduisant par une augmentation de la latence des composants elle doit cependant être accompagnée d'une diminution de la fréquence de fonctionnement et donc, des performances.

$$Delay \cong \frac{C_L.V_{DD}}{k_v.W.(V_{DD} - V_T - V_{DSAT})^2} \propto \frac{V_{DD}}{(V_{DD} - V_{th})^2} \tag{2.1}$$

$$Energy \propto V_{DD}^2 C_{eff} \tag{2.2}$$

Dans le cadre des télécommunications mobiles de troisième génération, de même que dans la majorité des domaines d'applications embarqués, la puissance de calcul requise par le système varie de manière très significative au cours du temps. Si les définitions de la tension

d'alimentation et de la fréquence de fonctionnement doivent être faites dans le pire cas (i.e. pour un taux d'occupation maximal de l'architecture), il est en revanche possible de réduire ces deux paramètres lors de l'exécution de traitements moins critiques. La mise en œuvre de ce concept fait appel aux techniques DVS (Dynamic Voltage Scaling) [87, 88]. Afin d'adapter la consommation d'énergie de DART à sa charge, ces techniques de DVS peuvent être implémentées dans DART. Dans DART, la gestion de la tension d'alimentation est dévolue au système d'exploitation étudié dans le cadre de la thèse d'Imène BenKermi[89].

Maîtrise de la consommation au niveau architectural Au niveau architectural, d'importants gains peuvent être obtenus en réduisant les gaspillages d'énergie liés au contrôle de l'architecture et à l'accès aux données. Dans DART, nous avons donc tenté de réduire le coût de recherche et de décodage des instructions. Pour cela, un effort particulier à été apporté à la réduction du volume de configuration et des fréquences des reconfigurations. Ces efforts se sont traduits par l'exploitation de la régularité des traitements par le biais d'un nouveau concept : le SCMD (Single Configuration Multiple Data).

Réduire le coût des accès aux données est un exercice tout aussi délicat que critique dans un contexte de faible consommation. Il faut à la fois limiter le nombre d'accès aux mémoires de données et le coût de ces accès. Afin de maîtriser le coût d'un accès, la hiérarchie mémoire a été conçue consciencieusement, en limitant au maximum la taille des mémoires du plus bas niveau de la hiérarchie. La réduction du nombre d'accès aux mémoires de données est quant à elle obtenue sur DART par le chaînage des opérateurs. En effet, en éliminant le coût des accès aux données volatiles (correspondant à un accès à une file de registre sur un processeur programmable), un gain substantiel en énergie peut être obtenu. Par ailleurs, les traitements flots de données étant relativement fréquents dans les applications embarquées, nous avons introduit dans notre architecture la possibilité de créer des chaînes de retard. Ces chaînes de retard permettent de réduire drastiquement le nombre d'accès aux mémoires de données lorsque, dans une application, plusieurs échantillons d'un même vecteur sont manipulés concurremment. Ceci est illustré par la figure 2.2 qui représente un traitement flot de données où à chaque cycle, les échantillons i et $i-1$ du vecteur X doivent être lus.

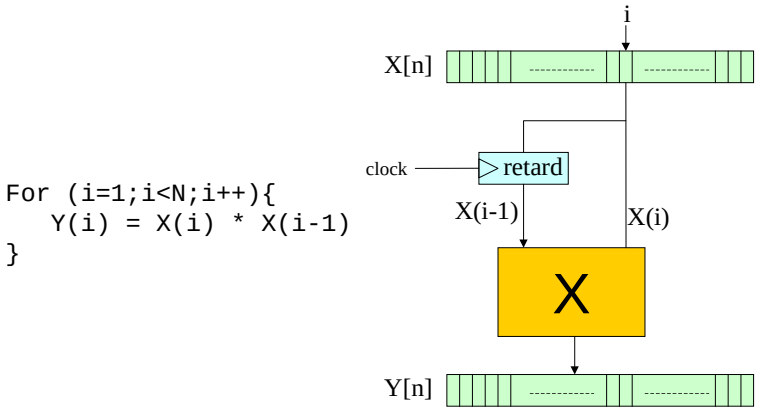

FIG. 2.2 – *Exemple d'utilisation d'une chaîne de retard. Afin d'éviter d'accéder deux fois au vecteur X (aux positions i et i-1), il est possible d'utiliser une chaîne de retard. La valeur de X(i-1) est déterminée par la valeur de X(i) au cycle précédent.*

Maîtrise de la consommation au niveau logique Au niveau logique, la maîtrise de la consommation passe par la réduction du nombre de transistors commutant inutilement. Pour cela, deux techniques ont été appliquées dans DART. La première consiste à isoler les différentes fonctionnalité des unités de calcul, par le biais de *latch*. En effet, l'intégration d'unités fonctionnelles généralistes implique l'introduction de nombreux transistors qui ne trouveront leur utilité que dans certains cas particulier. À titre d'exemple, une UAL est capable de faire des additions, des soustractions, des décalages, ... En réalisant un découpage logique de ces unités et en insensibilisant aux entrées, par le biais de *latchs*, les sous-ensembles du circuit non utiles au traitement en cours, seuls les transistors utiles au traitement de l'addition, par exemple, seront utilisés lorsque cette opération sera exécutée.

Une autre méthode bien connue pour optimiser la consommation au niveau logique est l'utilisation d'horloges gardées. Cette technique consiste à déconnecter l'horloge des éléments de calcul qui n'ont pas d'utilité pour le traitement en cours. Compte tenu de l'évolution de la puissance de calcul requise par une chaîne de radiocommunication mobile au cours du temps, l'intégration de cette fonctionnalité dans DART est indispensable. En effet, cette architecture se doit de supporter les traitements les plus critiques et pour cela, intègre un grand nombre de ressources de calcul et de mémorisation. Sur les traitements moins critiques, le taux d'utilisation de ces ressources n'est cependant pas optimal et un certain nombre d'unités de calcul peuvent ne pas être utilisées. Afin d'éliminer tout gaspillage d'énergie, nous déconnectons donc l'horloge de ces ressources par le biais d'un signal de contrôle appelé "garde".

Maîtrise de la consommation au niveau physique Au niveau physique la principale source de dissipation d'énergie est l'arbre d'horloge. Il convient donc de l'optimiser. Pour cela, nous en avons réduit la longueur, par le biais de la découpe en *cluster*, et la dimension, en travaillant à des fréquences "raisonnables" (130MHz). Par ailleurs, par une organisation hiérarchique des interconnexions, nous limitons dans le même temps leur coût énergétique. En effet, celles-ci sont relativement peu chargées et courtes.

La maîtrise de la consommation au niveau physique est très délicate et nécessite des techniques bien particulières qui doivent, la plupart du temps, être appliquées au moment du placement/routage de l'architecture. Dans cette étude ces étapes de placement/routage n'ont pas été réalisées. De ce fait, bon nombre d'optimisations au niveau physique sont en suspend. Il est à noter que ces dernières sont susceptibles de conduire à la définition d'une fréquence d'horloge plus élevée, si un meilleur compromis *temps-puissance* peut être obtenu.

Outils de développement de haut-niveau

Une architecture n'étant rien sans outils capables de l'exploiter, un soin particulier doit être apporté à la conception de ces derniers. Pour cela, nous avons œuvré, dans cette étude, à la définition de modèles de programmation simples. Ces modèles de programmation sont directement issus des différents styles de traitements qui peuvent cohabiter dans une description de haut niveau d'une application. En particulier, nous avons isolé les calculs d'adresses et les calculs intensifs du reste du code source, et défini deux modèles de programmation adaptés à ces traitements. Un troisième modèle de programmation a par ailleurs été défini pour supporter les traitements irréguliers. Bien que l'aspect procédural du *C* complique la tâche d'extraction du parallélisme [90], nous avons choisi ce langage comme point d'entrée de notre chaîne de développement. Ce choix se justifie par les contraintes sévères de portabilité qui s'appliquent aux descriptions de systèmes aussi complexes que les télécommunications 3G.

En distinguant plusieurs modèles de programmation, nous sommes en mesure de définir des outils adaptés à chacun d'eux. Compte tenu de la simplicité de ces modèles, les outils définis ont une complexité raisonnable, autorisant le développement d'une chaîne de développement efficace. Au final, la chaîne de développement associée à DART est basée sur l'utilisation conjointe d'une partie frontale permettant la transformation et l'optimisation de codes C, d'un compilateur reciblable et d'un outil de synthèse de haut-niveau. La description de cette chaîne de développement fait l'objet du chapitre 3.

Dans cette section, les principaux concepts exploités lors de la définition de DART ont été introduits. Les sections qui suivent décrivent la mise en œuvre de ces concepts dans l'architecture DART. Afin de décrire cette architecture, nous avons adopté une structure hiérarchique proche de celle qui la caractérise. Nous débuterons ainsi par la description de l'architecture système pour aboutir à la description des opérateurs.

2.2 Organisation de l'architecture

2.2.1 Architecture système

Le plus haut niveau de la hiérarchie de DART peut être schématisé par la figure 2.3. Cette architecture système est bâtie autour d'un contrôleur de tâches, de ressources de mémorisation et de ressources de calcul : les *clusters*.

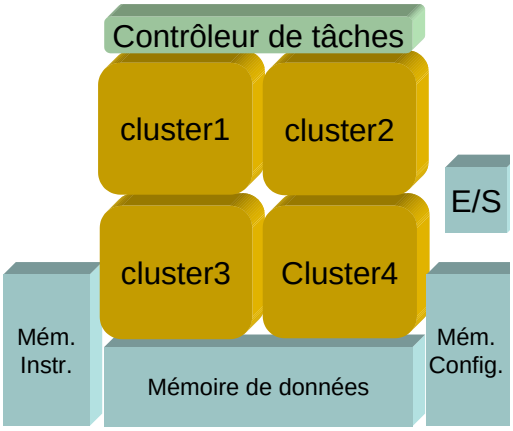

FIG. 2.3 – *Vue système de DART. Cette architecture se compose d'un contrôleur de tâches chargé d'assigner aux clusters les différents traitements devant être exécutés. Une fois configuré, chaque cluster travaille alors de manière autonome et gère ses propres accès à la mémoire de données.*

Le contrôleur de tâches est chargé de la gestion de l'architecture. Il a pour but d'assurer que les traitements sont effectués en temps et en heure. Pour cela, il se doit de supporter un système d'exploitation temps réel (RTOS : Real-Time Operating System), étudié dans le cadre de la thèse d'Imène Benkermi [89], chargé de décider quelles sont les applications qui doivent être exécutées à un instant donné. Cette décision se base sur des informations telles que l'urgence de la tâche, la disponibilité des ressources, ...

Spécifier une tâche aux ressources de calcul nécessite de les reconfigurer en chargeant leur mémoire de configuration. Le chargement de ces mémoires nécessite alors plusieurs centaines de cycles qui doivent être pris en compte lors de l'assignation des tâches. Une fois spécifiée, ces tâches s'exécutent cependant sans nécessiter la moindre intervention du contrôleur de tâches. Les *clusters* se chargent alors de l'exécution du traitement qui leur est assigné et des éventuels accès à la mémoire de données. L'autonomie des *clusters* permet d'exploiter pleinement le parallélisme d'applications du système.

L'architecture système de DART dispose par ailleurs d'un module d'entrée/sortie permettant d'assurer des communications efficaces entre DART et un éventuel environnement extérieur. Celui-ci permet notamment d'assurer la connectique de l'architecture avec des bus standards tels que le ST Bus, l'OCCN[91] ou encore l'AMBA[92].

2.2.2 Architecture des clusters

L'architecture d'un *cluster* de DART peut être représentée par la figure 2.4. Celle-ci s'organise autour de trois composantes : des ressources de calcul, des ressources de mémorisation et un contrôleur.

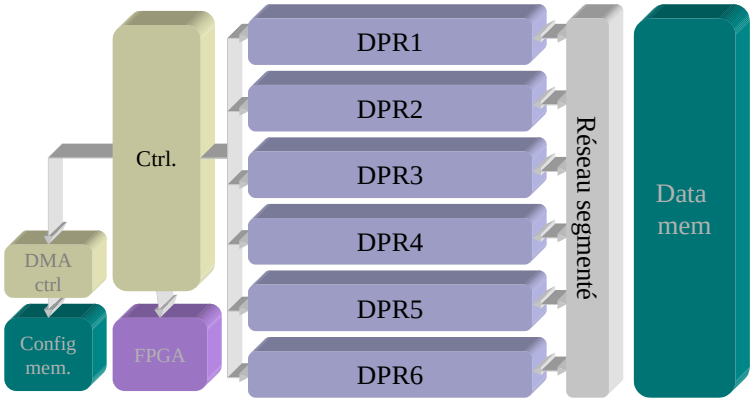

FIG. 2.4 – *Architecture d'un cluster de DART. Elle se compose d'un cœur de FPGA, de DataPath Reconfigurables pouvant être interconnectés via un réseau segmenté et d'un contrôleur gérant ces ressources de calcul. Les DPRs et le cœur de FPGA manipulent des données stockées dans une mémoire partagée.*

Les ressources de calcul

Sachant que plusieurs grains de calcul cohabitent dans une chaîne de traitement de troisième génération, chaque *cluster* de DART intègre deux types de ressources de calcul : des DPRs et un cœur de FPGA. Les DPRs, qui seront présentés plus en détails dans la section suivante, se chargent des traitements arithmétiques, et le cœur de FPGA des traitements logiques. Par l'intégration des DPRs nous souhaitons ainsi être en mesure d'offrir un très haut niveau de performance et par celle du cœur de FPGA nous souhaitons disposer d'une flexibilité à la hauteur de nos besoins, sachant que l'efficacité des opérateurs arithmétiques est mis à mal par l'intégration d'un codeur de Huffman [93] par exemple.

Dans cette architecture, le cœur de FPGA est considéré comme une IP (Intellectual Property). De ce fait son architecture, de même que son dimensionnement (nombre de portes

logique équivalentes), est a priori indéterminée[1]. Que ce soit du point de vue architectural ou logiciel, ce bloc est donc traité d'une façon particulière. Au niveau architectural, l'intégration de ce bloc est relativement simple. Il s'agit en effet de rajouter la logique contrôle nécessaire à sa gestion. Les transmissions de données entre cet IP et les DPRs se font quant à elles via la mémoire de données du *cluster*. Ce mécanisme n'implique dès lors aucun surcoût matériel, si ce n'est l'ajout de connexions directes entre la mémoire et l'IP. Le développement des applications devant être exécutées sur ce bloc sera quant à lui réalisé par le biais des outils proposés par les fournisseurs de l'IP, où par le biais d'outils génériques [95, 96].

Les ressources de mémorisation

Toutes les ressources de calcul du *cluster* accèdent à une même mémoire de données. Celle-ci a été dimensionnée à 16 kmots de 16 bits afin d'assurer une certaine autonomie aux *clusters*. Cette mémoire est accompagnée d'un contrôleur en charge de distribuer les données aux mémoires internes des DPRs ainsi qu'au FPGA. Ces mouvements de données sont statiquement programmés et déterminés au moment de la compilation. Ainsi, chaque reconfiguration du *cluster* intégrera également une reconfiguration de ce contrôleur mémoire afin qu'il soit en mesure de déterminer à quel cycle telle ou telle donnée doit être lue/écrite depuis/vers telle ou telle mémoire des DPRs. Son comportement est donc totalement déterministe ce qui permet de simplifier les mécanismes d'accès aux données. Dès lors, DART s'affranchi des mécanismes de gestion de mémoire virtuelle, très consommateurs en énergie et en temps. Ceci se paye cependant par un effort important au moment de la compilation. Ce contrôleur est examiné plus en détail dans le paragraphe 2.4.3.

Chaque *cluster* de DART intègre également une mémoire de configuration dédiée au FPGA. Son dimensionnement dépend directement du choix de l'IP FPGA.

Le contrôleur

Le contrôleur du *cluster* est chargé de la gestion des ressources de calcul internes au *cluster*. La configuration du FPGA se fait de manière série par le biais d'un contrôleur DMA (Direct Memory Access). Le rôle du contrôleur du *cluster* dans la reconfiguration du FPGA est donc très limité puisqu'il se contente de spécifier une adresse de début et de fin de configuration au contrôleur DMA. Celui-ci se charge alors de transmettre en série les données depuis la mémoire de configuration du *cluster* vers le FPGA. Compte-tenu du temps de reconfiguration[2] du FPGA, celle-ci devra être anticipée. Une fois encore, la prédiction des dates de reconfiguration est réalisée statiquement pendant la compilation.

Le rôle du contrôleur de *cluster* dans la reconfiguration des DPRs est en revanche beaucoup plus important. Les reconfigurations de ces derniers sont réalisées sous la forme de flots d'instructions de configuration (cf. §2.5). Ainsi, la principale tâche de ce contrôleur est de séquencer ces instructions de configuration. Son architecture est donc comparable à celle de tout contrôleur de processeur programmable, à ceci près qu'il n'a pas à gérer des mécanismes complexes d'interruptions et qu'il ne séquence pas des instructions (au sens instructions microprocesseur), mais des configurations. Ainsi, il n'y a pas lieu de systématiser de coûteux accès à la mémoire d'instructions et des décodages d'instructions non moins coûteux. En effet, ces accès n'interviendront que lors des reconfigurations et seront donc très occasionnels. Des mécanismes de mise en attente (instructions *WAIT N cycles*) ont donc été mis en place afin de

[1] Une étude menée au laboratoire vise cependant à définir une architecture de cœur de FPGA susceptible d'être utilisée au sein de DART [94].

[2] Bien que le FPGA ne soit pas dimensionné, celui-ci peut être considéré comme étant de l'ordre de 100μs.

minimiser le nombre de lectures et de décodages d'instructions. Ceci permet de réduire considérablement la consommation d'énergie, d'autant que la taille de la mémoire d'instructions a pu être réduite à 2 kmots.

2.3 Architectures des DPRs

Les DPRs constituent le dernier niveau de la hiérarchie de DART et sont représentés sur la figure 2.5. Ces DPRs sont constitués d'unités fonctionnelles interconnectées suivant un motif totalement flexible. Le nombre de ces DPRs, de même que celui des unités fonctionnelles qu'ils intègrent, est un paramètre de l'architecture. Ces paramètres ont, dans un premier temps, été déterminés afin de dimensionner l'architecture en fonction des besoins en calcul du domaine applicatif. Les 12 GOPS requis par les télécommunications 3G se sont ainsi traduits par l'intégration de 6 DPRs intégrant chacun 4 unités fonctionnelles. Nous vérifierons par la suite (cf. §4.4.3) que ce choix n'influe pas sur la viabilité des concepts évoqués dans ce mémoire.

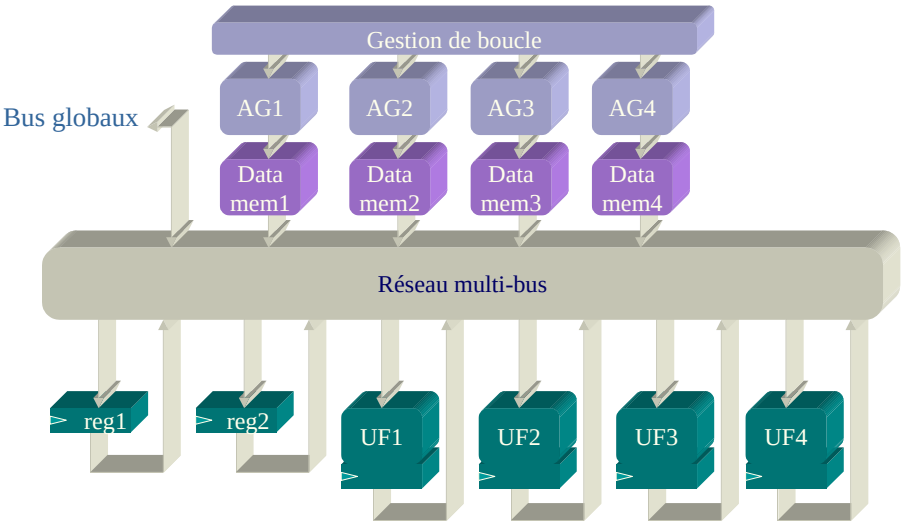

FIG. 2.5 – *Architecture d'un DPR de DART. Chaque DPR est constitué d'unités fonctionnelles interconnectées suivant un motif totalement flexible de type multi-bus. Elles travaillent sur des données stockées dans des mémoires locales auxquelles sont associées des générateurs d'adresses (AG).*

2.3.1 Les opérateurs

Chaque DPR intègre 4 unités fonctionnelles, à savoir 2 multiplieurs/additionneurs (UF1 et 3) et 2 UALs (UF2 et 4). Ces unités ont été optimisées suivant des contraintes de performance et de consommation par Taofik Saïdi [97]. Toutes ces unités sont dynamiquement reconfigurables et supportent les traitements SWP. Ces unités permettent par ailleurs la réalisation de décalages en parallèle avec les traitements arithmétiques (1 en sortie de chaque multiplieur ainsi qu'en entrée et en sortie des UALs). Par soucis d'accroissement de la dynamique de calcul, les UALs travaillent avec 8 bits de garde. Dans leur version actuelle, ces opérateurs supportent les opérations figurant dans le tableau 2.1.

Opérateur	ES	SWP	recadrage en entrée	opérations	garde	SAT	recadrage en sortie
UF1 UF3	16x16 ->32	8 bits	NON	MUL,ADD	NON	NON	<< 1 à << 8
UF2, UF4	32x40 ->40	8,16 bits	<< 4 à >> 4	ADD,SOUS, MIN MAX,ABS,ET OU,XOR,NOP	8 bits	OUI	>> 1 à >> 8

TAB. 2.1 – *Opérations supportées par les opérateurs des DPRs : la première colonne spécifie le format des entrées et de la sortie. La seconde précise, pour chaque unité, les tailles de données supportées lors les traitements SWP. Les suivantes décrivent les différents traitements pouvant être exécutés en parallèle sur les unités : décalage en entrée, opération, saturation et recadrage en sortie.*

Architecture du multiplieur

Le multiplieur est une unité critique dans le DPR, tant du point de vue des performances que de la consommation. Cette unité est centrée sur un module supportant les multiplications de nombres signés codés sur 16 bits dans une arithmétique double précision (i.e. la sortie est codées sur 32 bits, doublant ainsi la précision). Pour optimiser cet opérateur, un codage de Booth modifié a été associé à une structure de Wallace. Le travail d'optimisation lié à cet opérateur est décrit en annexe A.

Support SWP Afin d'introduire un support pour les opérations SWP, il est possible de modifier l'architecture de l'unité afin de définir des fonctionnalités 16 bits par l'association de fonctionnalités 8 bits. Pour un multiplieur, on parle dès lors de décomposition 4M [98]. Par cette décomposition, le multiplieur 16 bits est construit autour de 4 multiplieurs 8 bits. Cette solution offre l'avantage d'être très flexible et d'autoriser des décompositions successives des opérateurs pour introduire des opérations 4 bits, 2 bits ... [99]. En revanche, le support matériel devant être introduit pour assurer la compatibilité entre les différents étages de l'opérateur est trop complexe pour un multiplieur 16 bits [97] et conduit à des performances relativement médiocres.

Une autre solution consiste à intégrer dans l'unité plusieurs multiplieurs. Le support d'opérations SWP 8 bits dans une unité 16 bits nécessite dès lors l'introduction d'un multiplieur 16 bits et de 2 multiplieurs 8 bits. Bien que se traduisant par un surcoût notable en surface, cette solution offre de nombreux avantages. En particulier, seuls un multiplexeur et un démultiplexeur viennent rallonger le chemin critique de l'unité. Les performances sont dès lors quasiment identiques à celles d'un opérateur dédié à des opérations 16 bits. L'autre avantage notable de cette solution concerne le gain en énergie. Ce gain s'explique par la possibilité d'isoler les différents blocs de l'unité en insérant des *latchs*. En effet, une décomposition 4M impose l'utilisation de tous les transistors de l'architecture, que ce soit pendant les traitements 16 bits ou les traitements 8 bits. L'introduction d'unités spécialisées permet en revanche d'isoler les opérateurs 8 bits lors des opérations 16 bits et, inversement, d'isoler l'opérateur 16 bits lors des traitements 8 bits.

Au final, pour une technologie *umc* 0.18μm et une tension d'alimentation de 1.8V, notre opérateur se caractérise par une augmentation de 10% de la surface occupée par rapport à un multiplieur 4M. Cet inconvénient est cependant occulté par le gain en performance qui atteint 30%, et par le gain en énergie qui s'élève à 35% lors des traitements 16 bits.

Synthèse L'efficacité des structures Booth-Wallace [100] justifie leur utilisation au sein de notre unité (cf. Annexe A). L'architecture de cette dernière peut être représentée par la figure 2.6. L'insertion de *latch* dans cette unité permet d'isoler les différents blocs la constituant. Bien que se traduisant par une augmentation de 6% de la surface et de 12% de la latence du composant, leur insertion est justifiée par les gains importants quelles procurent du point de vue de l'énergie. En effet, ces *latchs* autorisent une réduction de l'énergie consommée pendant les traitements 16 bits de 32%, et de 61% lors des traitements 8 bits en mode SWP (cf. tableau 2.2).

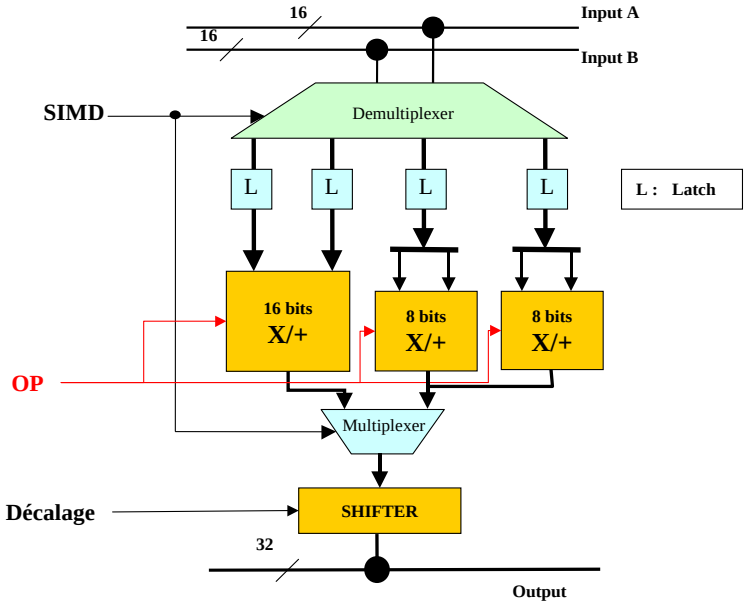

FIG. 2.6 – *Architecture du multiplieur SWP de DART. Cette unité intègre des multiplieurs 16 bits et 8 bits isolés par le biais de latchs. Une commande en entrée de cette unité dirige les données vers l'opérateur adéquat en contrôlant la logique d'interconnexion. Une commande supplémentaire (OP) permet par ailleurs d'utiliser l'arbre de Wallace de ces multiplieurs pour réaliser l'addition de deux nombres*

Architecture	**Surface(μm^2)**	**Latence (ns)**	**Consommation 8 bits(pJ)**	**Consommation 16 bits(pJ)**
Avec *latchs*	36786	5	26.7	46.6
Sans *latchs*	34616	4.4	68.6	68.6

TAB. 2.2 – *Caractéristiques des multiplieurs avec et sans latch : l'augmentation de la surface et de la latence de l'opérateur induite par l'insertion des latchs est largement occultée par les gains en consommation d'énergie pouvant atteindre 60% en mode SWP.*

Dans une chaîne de communication 3G, l'emploi de multiplications n'est pas systématique. En conséquence, dédier un opérateur à cette fonctionnalité se traduit inexorablement par une sous-utilisation de cette ressource. Par ailleurs, les multiplieurs de DART décrits précédemment exploitent un additionneur pour combiner les différents produits partiels. Il est donc possible de définir des opérateurs multi-fonctions avec un surcoût matériel négligeable. Ainsi, nous avons rajouté à ces unités une commande (entrée *OP* de la figure 2.6) permettant de

spécifier aux multiplieurs qu'ils seront utilisés pour réaliser une addition. Dans le cas du traitement d'une addition, toute la partie de l'opérateur permettant de générer et de combiner les produits partiels est alors court-circuitée par le biais de multiplexeurs, afin de limiter la consommation d'énergie induite par cette opération.

Architecture de l'unité arithmétique et logique

Le chemin critique de l'unité arithmétique et logique de DART est principalement influencé par l'efficacité de l'additionneur. Celui-ci a donc été conçu avec une attention toute particulière. L'optimisation de cet opérateur, présenté en annexe B, s'est traduite par la définition d'un additionneur de Sklantsky.

Structure de l'UAL L'UAL étant susceptible d'être chaînée à la sortie d'un multiplieur, elle se doit de supporter des entrées de 32 bits. Pour accroître la dynamique des signaux lors des opérations d'accumulation, nous avons étendu la largeur des UALs à 40 bits, autorisant ainsi l'emploi de 8 bits de garde. Les contraintes de précision étant déterminantes dans les algorithmes de traitement du signal, nous avons par ailleurs facilité l'emploi de l'arithmétique virgule fixe en insérant des décaleurs en entrée et en sortie de l'UAL.

La soustraction $A-B$ étant équivalente à l'addition $A+(-B)$, l'insertion de cette opération dans l'UAL est triviale. En effet, sachant que $-B = \overline{B}+1$, le traitement d'une soustraction sur un additionneur ne nécessite que la mise à 1 de la retenue entrante, et l'inversion de B. Le surcoût matériel est alors négligeable puisqu'il ne correspond qu'à un étage de portes XOR entre l'entrée B et l'additionneur. Une commande de soustraction (cde) vient contrôler la valeur envoyée à l'additionneur. Cette commande vaut '0' lors d'une addition et '1' lors d'une soustraction. Ainsi la quantité $B_i \bigoplus cde$ vaut B_i pour une addition et $\overline{B_i}$ pour une soustraction.

Dans le même ordre d'idée, l'introduction de l'opération de valeur absolue dans l'UAL est très simple. Il s'agit ici de réaliser l'opération $0-B$ lorsque B est négatif et $0+B$ lorsque B est positif. Le signe de B étant déterminé par son bit de poids fort, il s'agit donc de considérer ce dernier comme le bit de commande de l'additionneur/soustracteur.

Cette UAL est par ailleurs en mesure de traiter des opérations déterminant la plus grande ou la plus petite valeur entre deux nombres. L'introduction de ces deux fonctionnalités est relativement aisée et ne nécessite pas la définition d'un nouvel opérateur. Il s'agit en premier lieu de soustraire les deux entrées A et B. La sélection de la valeur la plus grande ou la plus petite est alors conditionnée par le signe du résultat. En effet, si $A-B>0$ *alors* $A>B$ et inversement, si $A-B<0$ *alors* $A<B$.

La dernière fonctionnalité devant être insérée dans l'additionneur/soustracteur est la possibilité de supporter les traitements SWP. La mise en place de ce concept est triviale dans cet opérateur. Il s'agit en effet de considérer cet opérateur comme une cascade d'unités 8 bits. La mise en cascade se faisant par l'intermédiaire des retenues, l'implantation du SWP se traduit par la nécessité de ne pas propager la retenue entre les unités lorsqu'un tel traitement est demandé. Ce principe est illustré par la figure 2.7 et est mis en œuvre par le biais de multiplexeurs.

Synthèse L'opérateur synthétisé peut être représenté par la figure 2.8. Il est centré sur deux entités : l'unité arithmétique présentée précédemment et l'unité logique traitant les opérations *AND, OR et XOR*. Par l'insertion de *latchs*, la consommation est réduite dans la cadre des

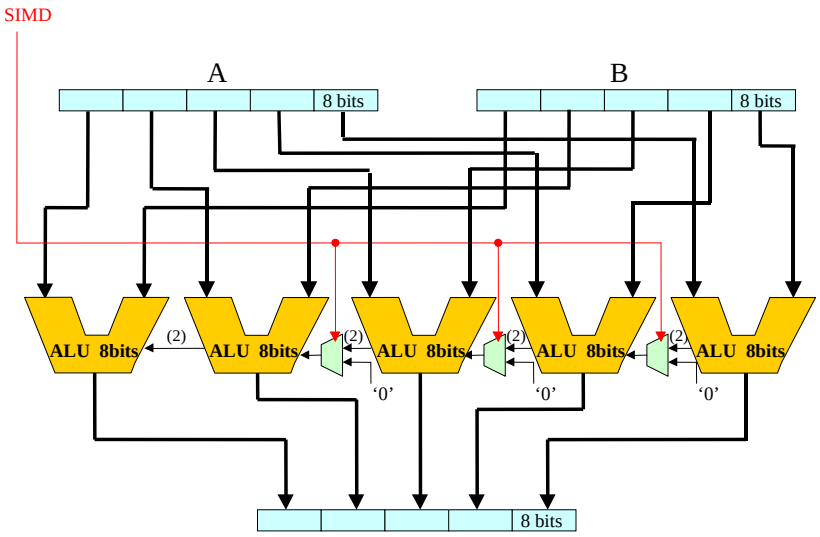

FIG. 2.7 – *Structure d'un additionneur SWP sur 40 bits. L'additionneur est décomposé en 5 modules de 8 bits cascadés par leur retenue. Une commande réalise la connexion ou la déconnexion des différents modules en fonction du mode SWP souhaité (8,16 ou 32/40 bits).*

traitements logiques. Dans ce cas, le bloc arithmétique est en effet insensibilisé aux entrées. L'unité arithmétique est par ailleurs suivie d'un bloc de saturation permettant de faire des additions/soustractions sans avoir à se soucier des éventuels débordements. Deux décaleurs sont finalement intégrés dans l'UAL. Le premier, en entrée du bloc arithmétique, supporte des décalages à gauche ou à droite d'une amplitude pouvant aller jusqu'à 4. Un décaleur en sortie permet quant à lui de recadrer les résultats par des décalages à droite d'une amplitude pouvant aller jusqu'à 8. La configuration de cet opérateur nécessite 11 bits : 3 pour spécifier la fonctionnalité de l'opérateur, 6 pour les décalages et 2 pour les opérations SWP. Au final, pour une technologie umc $0.18\mu m$ alimentée en 1.8V, le chemin critique de cette unité est de 5.21ns. Elle occupe $36290\mu m^2$ et consomme dans le pire cas (Addition +saturation +décalage) 36pJ par accès.

2.3.2 Les interconnexions

Intra-DPR

Ces opérateurs, de même que les mémoires et les registres, sont interconnectés au sein du DPR par le biais d'un réseau multi-bus totalement flexible. Ainsi, toute unité peut être connectée à n'importe quelle autre dans le DPR et la sortie d'une unité peut simultanément être connectée avec toutes les autres. La sortie d'une mémoire pourra par exemple être simultanément connectée aux entrées des 4 unités fonctionnelles du DPR tout en alimentant les unités fonctionnelles d'autres DPRs via des bus globaux. Ces connexions de type "un vers tous" permettent de réduire très significativement le nombre d'accès mémoire.

Afin d'autoriser ces communications "un vers tous", chaque unité du DPR dispose d'un bus dont il est le seul maître. Le DPR est ainsi traversé par 18 bus, chacun correspondant à une ressource particulière du DPR : 4 d'entre eux sont alloués aux mémoires (16 bits), 2 aux

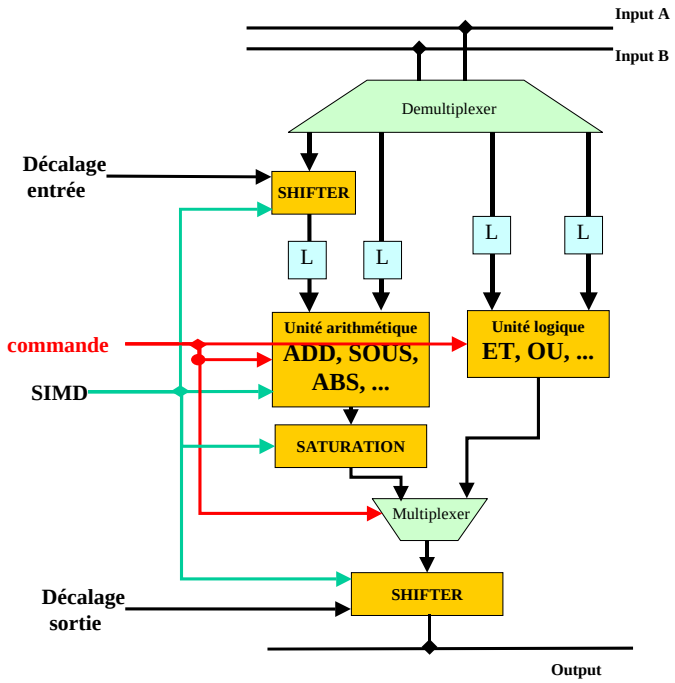

FIG. 2.8 – *Structure de l'UAL de DART. Elle est construite autour d'une unité arithmétique et d'une unité logique isolées par le biais de latchs. Un module de saturation a par ailleurs été introduit en sortie de l'unité arithmétique. Des décaleurs permettent de gérer efficacement la précision et la dynamique des données codées en virgule fixe.*

registres (32 bits), 4 aux unités fonctionnelles (32 et 40 bits). Les 8 derniers bus permettent d'accéder aux données générées dans d'autres DPRs (32 bits).

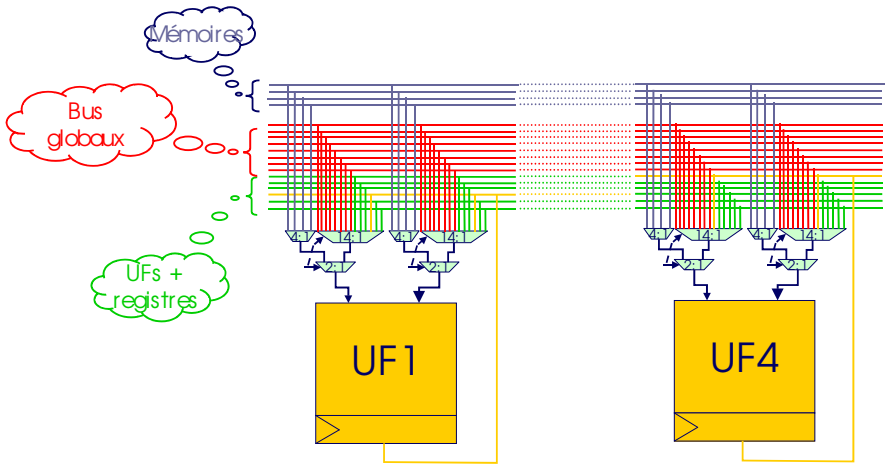

FIG. 2.9 – *Connexion des UF1 et 4 sur le réseau multi-bus des DPRs. Chaque unité renvoie son résultat sur le bus qui lui est dédié. Toutes les unités du DPR (y compris elle-même) peuvent alors considérer ce bus comme une entrée, en contrôlant la logique de multiplexage. Chaque entrée d'une unité a ainsi accès aux sorties de toutes les autres unités du DPR ainsi qu'aux données présentes sur les 8 bus globaux.*

Chaque entité du DPR est dès lors susceptible d'accéder à l'un des 18 bus pré-cités. La sélection du bus considéré comme une entrée par une unité fonctionnelle se fait par l'intermédiaire de logique de multiplexage. En particulier, chaque entrée des deux unités fonctionnelles représentées sur la figure 2.9 accède au bus désiré par l'intermédiaire de 3 multiplexeurs.

1. Un multiplexeur 4 vers 1 sélectionne parmi les 4 données issues des mémoires celle qui est susceptible d'être manipulée ;
2. Un multiplexeur 14 vers 1 sélectionne parmi les données issues des 2 registres, des 4 UFs et des 8 bus globaux, celle qui est susceptible d'être manipulée ;
3. Le multiplexeur 2 vers 1 sélectionne finalement parmi les données extraites par les deux premiers multiplexeurs celle qui doit être envoyée en entrée de l'unité.

Cette logique de multiplexage distingue donc les configurations où les données sont issues des mémoires, de celles donnant lieu à des chaînages d'opérateurs. Si les configurations des multiplexeurs 4 vers 1 et 14 vers 1 sont directement issues des données de configuration, celle du multiplexeur 2 vers 1 est obtenue par décodage. Les entrées de ces multiplexeurs étant de tailles variables, ils gèrent par ailleurs l'alignement de données et les éventuels décalages.

Il a été retenu comme convention que si le multiplexeur 14 vers 1 est utilisé, alors sa commande sélectionne l'une des entrées situées entre les adresses 0 et 13. Dès lors que l'adresse de ce multiplexeur est supérieure à 13, le multiplexeur 2 vers 1 considère que l'entrée de l'UF est issue d'une des mémoires et sélectionne la sortie du multiplexeur 4 vers 1. Dans le cas contraire, la sortie du multiplexeur 14 vers 1 est envoyée vers l'entrée de l'UF. Cette distinction entre les signaux issus des mémoires et les autres signaux sera justifiée ultérieurement dans le §2.5.3.

Les différences de format entre les entrées des multiplexeurs 14 vers 1, ainsi que la possibilité de traiter des données en mode SWP se traduit une logique de multiplexage complexe. Outre la sélection des entrées, ces modules sont ainsi en charge de réaliser d'éventuelles extensions de signes, d'aligner des données, etc. En conséquence, cette logique de multiplexage a un temps de traversée non négligeable ($\sim 0.8ns$) et influe significativement sur l'énergie consommée par DART ($\sim 4pJ$). Ces modules réalisant des traitements indispensables au bon déroulement du calcul, nous les considérerons par la suite comme partie intégrante des opérateurs.

Inter-DPR

Sachant que les degrés de parallélisme des applications 3G sont très variables, les DPRs sont interconnectés suivant un réseau segmenté (Fig. 2.10). Grâce à ce réseau, plusieurs DPRs peuvent être interconnectés pour implémenter une application massivement parallèle et ainsi augmenter le degré de parallélisme au niveau opération (ou instruction). Les communications entre ces DPRs peuvent alors être globales, e.g. une mémoire approvisionne en données des unités fonctionnelles appartenant à chacun des DPRs interconnectés, ou locales, e.g. pour transmettre un résultat temporaire entre deux unités fonctionnelles appartenant à deux DPRs adjacents. À l'inverse, pour les applications disposant de très peu de parallélisme, chacun des DPRs peu travailler indépendamment de ses homologues sur différentes tâches. Cette indépendance des différents DPRs est assurée par une déconnexion des bus globaux. Par le biais de ce réseau, il est donc possible de définir un grand nombre de compromis entre parallélisme d'instructions et parallélisme de tâches. Ce dernier doit être défini à la compilation, en fonction des besoins de l'application.

Chaque unité du DPR doit avoir accès à ces bus globaux. En effet, les mémoires doivent être en mesures d'approvisionner en données des unités présentes dans d'autres DPRs, ceci afin

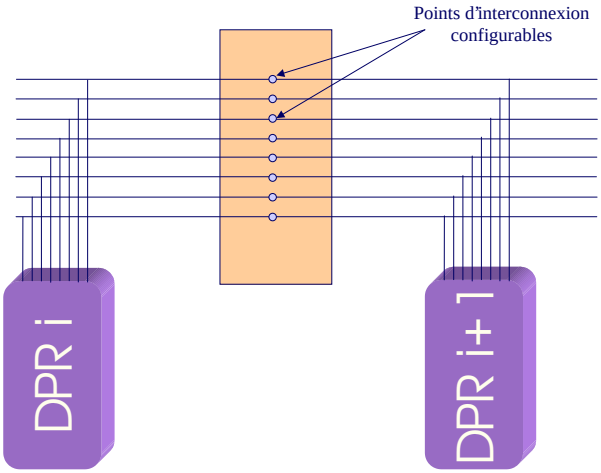

FIG. 2.10 – *Structure du réseau segmenté des clusters. Les DPRs ont la possibilité de se transmettre des données par l'intermédiaire de 8 bus globaux. La déconnexion des différentes lignes de ce réseau segmenté permet, à l'inverse, d'isoler les différents DPRs du cluster*

d'éviter la duplication des données dans des mémoires présentes au sein de différents DPRs. Les registres doivent également pouvoir être chaînés de manière à créer des retards supérieurs à 2 cycles. La possibilité de chaîner les opérateurs doit finalement se traduire par un accès aux bus globaux des unités fonctionnelles. Au final, 10 éléments du DPR sont susceptibles d'accéder à 8 bus globaux. Afin d'éviter tout conflit sur ces bus, une mise en haute impédance doit en outre être possible.

Cette connectique est assurée par le biais d'une logique de multiplexage. Initialement, à chaque bus global fût associé un multiplexeur 11 vers 1, sélectionnant entre les 10 unités du DPR et la mise en haute impédance la valeur devant être transmise sur les bus globaux. Les contraintes de faible consommation et une constatation simple nous ont cependant conduits à réduire de moitié le nombre de ces multiplexeurs. En effet, lors des communications inter-DPRs, les sources et les destinations sont équitablement distribuées dans le *cluster*, i.e. au sein d'un DPR, le nombre de lectures et d'écritures sont du même ordre. En conséquence, la probabilité qu'un DPR réalise plus de 4 écritures sur les bus globaux est quasi nulle. Ainsi, 4 commandes de multiplexeurs 11 vers 1 (16 bits) sont systématiquement inexploitées. Au niveau du *cluster*, 96 bits de configurations sont donc inutilement lus et décodés, conduisant à un gaspillage d'énergie préjudiciable à l'efficacité de l'architecture.

Afin de limiter à 4 le nombre de multiplexeurs, tout en optimisant la flexibilité du réseau segmenté, nous avons autorisé chaque DPR à n'écrire que sur 4 bus globaux : les bus pairs ou les bus impairs. La décision de connecter un DPR aux bus pairs ou impairs est réalisée au moment de la compilation. Elle se traduit par l'ajout d'un bit de configuration aux 16 bits de commandes des multiplexeurs. Ces 17 bits permettent dès lors de contrôler l'accès d'un DPR aux bus globaux, par le biais de la logique de multiplexage représentée sur la figure 2.11. En contrôlant la décision d'accéder aux bus pairs ou impairs indépendamment pour chaque DPR, la flexibilité du réseau d'interconnexions global est quasi-optimale tout en minimisant le volume de données de configuration.

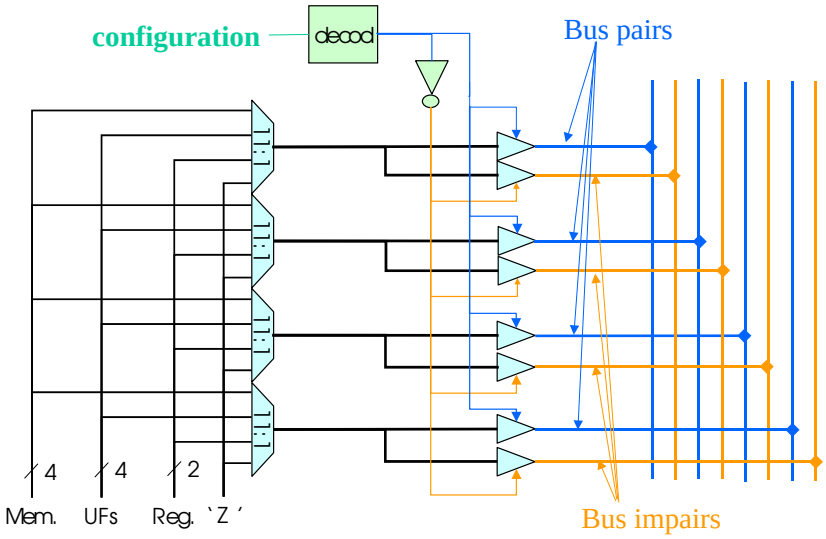

FIG. 2.11 – *Mécanisme d'accès aux bus globaux. Chaque élément d'un DPR accède aux bus globaux via les 4 multiplexeurs. La sortie de chacun des 4 multiplexeurs est orientée vers deux tri-states qui conditionnent l'accès aux bus globaux. Le premier tri-state gère l'accès à un bus pair et le second, l'accès à un bus impair. Les commandes de ces deux tri-states étant inversées, l'un au moins de ces deux tri-states est mis en position de haute impédance et libère le bus global qu'il contrôle. La commande bus pair/bus impair est commune à tout le DPR.*

2.4 La hiérarchie mémoire

2.4.1 Les ressources de mémorisation

Les ressources de mémorisation d'un *cluster* de DART sont distribuées en deux niveaux de hiérarchie. Le premier est constitué d'une mémoire unique de 16 kmots de 16 bits. Cette mémoire, bien que locale, peut être assimilée à la mémoire cache d'un microprocesseur. Elle procure une solution intermédiaire, en terme de coût énergétique et de capacité de stockage, entre les mémoires locales et la mémoire de masse du système (externe aux *clusters*). Contrairement aux mémoires caches, la détermination des accès mémoires est déterministe au sens où, lors de la compilation, chaque donnée est placée physiquement en mémoire et les dates de chaque accès aux données sont connues. Le séquencement de ces accès et les déplacements de données entre cette mémoire et les mémoires locales sont assurés par le contrôleur mémoire décrit dans le section 2.4.3. C'est via cette mémoire que se font les éventuels échanges de données entre les DPRs et le cœur de FPGA.

Le second niveau de hiérarchie intègre 24 mémoires locales distribuées équitablement entre les 6 DPRs. Ces mémoires, d'une capacité de 256 mots de 16 bits chacune, stockent les données manipulées au sein des DPRs. Leur dimensionnement satisfait un compromis entre efficacité énergétique et autonomie des DPRs. En effet, la faible profondeur de ces mémoires en fait une solution efficace en terme énergétique puisqu'un accès mémoire dissipe relativement peu d'énergie. Dans le même temps, les 4 mémoires de chaque DPR permettent de limiter le nombre d'accès aux données externes aux DPRs et ainsi, réduisent la charge du contrôleur mémoire. Chacune de ces mémoires est contrôlée par un générateur d'adresses local.

2.4.2 Les générateurs d'adresses

Architecture et principe de fonctionnement des générateurs d'adresses

Les générateurs d'adresses constituent le dernier niveau de la hiérarchie du contrôle dans DART. Chaque *cluster* en intègre 24 de manière à contrôler l'accès aux données indépendamment sur chaque mémoire. Ces derniers doivent donc être autonomes puisqu'une fois une configuration de chemin de données spécifiée, ils sont les seuls garants du bon fonctionnement du circuit. Afin d'assurer cette indépendance, ces générateurs d'adresses (AG : Address Generator) disposent d'une architecture comparable à des processeurs RISC simplifiés. Celle-ci, centrée sur une file de registres, peut être représentée par la figure 2.12.

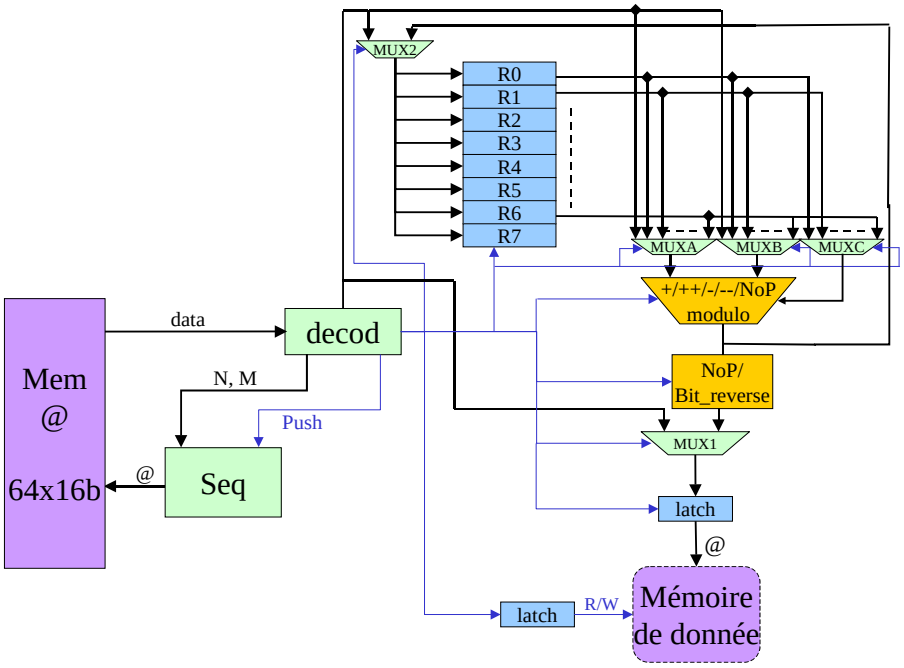

FIG. 2.12 – *Architecture des générateurs d'adresses internes aux DPRs. Cette architecture est centrée autour d'une file de registres alimentant une UAL chargée des calculs d'adresses. Une mémoire de petite taille contient les instructions de génération d'adresses qui doivent être décodées pour contrôler le chemin de données. L'accès à cette mémoire est contrôlé par un séquenceur.*

Cette architecture s'organise autour d'un séquenceur (SEQ) spécifiant à la mémoire d'instructions l'adresse de celle qui doit être décodée. La sortie de ce décodeur contrôle alors un chemin de données centré sur une file de 8 registres et une UAL. Afin de traiter efficacement un grand nombre d'applications de traitement du signal, ces AGs supportent une grande diversité de modes d'adressage parmi lesquels les adressages bit-reverse, modulo, ... Dès lors, l'UAL est capable d'additionner ou de soustraire la valeur de deux registres (e.g. base et offset), de réaliser des post-incrémentations et des post-décrémentations. Cette UAL est suivie d'un module capable de transformer une adresse binaire en une adresse bit-reverse sur 6,7 ou 8 bits. La sortie de ce module attaque alors un multiplexeur qui déterminera si l'adresse est issue de ce calcul ou bien est une valeur immédiate issue du décodeur. Par souci de minimisation

d'énergie, une *latch* sépare la sortie de ce multiplexeur de l'entrée d'adresse de la mémoire afin de ne pas faire commuter inutilement le bus d'adresse de la mémoire lorsqu'aucune donnée n'a à être lue.

La gestion de boucle

Un processeur n'intégrant pas de mécanisme de gestion de boucle se doit d'initialiser une variable puis de la mettre à jour régulièrement afin de déterminer à quelle itération de la boucle il se trouve. Lorsque la dernière instruction d'un cœur de boucle est lue il faut alors retourner à la première instruction de cette boucle. Ce saut dans la mémoire d'instructions est cependant conditionné par le numéro de l'itération de la boucle. Le saut ne sera pas réalisé si toutes les itérations de la boucle ont été traitées. Si ce contrôle ne se traduit que par quelques cycles supplémentaires à chaque itération, ce surcoût devient inacceptable pour les traitements intensifs où le nombre d'itérations est très important.

La puissance de calcul de DART étant principalement exploitée durant les boucles, des mécanismes efficaces ont du être intégrés pour les gérer [101]. Ceux-ci autorisent le support de boucles imbriquées (jusqu'à 4 niveaux d'imbrication) multiples (jusqu'à 15 instructions par cœur de boucle) sans consommer aucun cycle pour l'initialisation du compteur, sa mise à jour, ... Puisque les unités fonctionnelles travaillent toutes à l'exécution d'un unique cœur de boucle, cette gestion est commune aux quatre AGs de chacun des DPRs (Fig. 2.13). Le module chargé de cette gestion nécessite alors trois entrées, issues du décodage de l'instruction de génération d'adresses *REPEAT N, M* par les AGs :

1. une commande *push* permettant de spécifier que l'on entre dans une boucle ;
2. une donnée N correspondant au nombre d'itérations de la boucle (codée sur 10 bits) ;
3. une donnée M correspondant au nombre d'instructions dans le cœur de boucle (codée sur 4 bits).

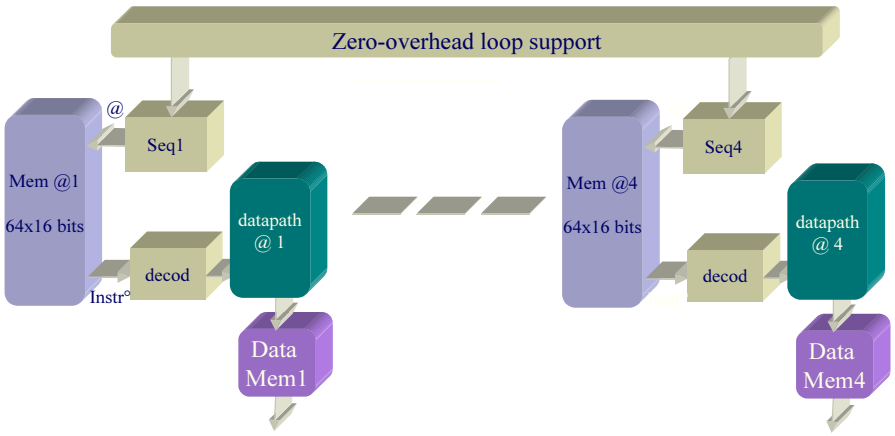

FIG. 2.13 – *Insertion du module de gestion de boucle dans le DPR. Toutes les unités d'un DPR travaillant à l'exécution d'une boucle unique, ce module de gestion de boucle est commun aux 4 AGs.*

L'objectif est alors de gérer l'évolution du Compteur de Programme (PC) du séquenceur (Fig. 2.14) de manière à ce qu'il retourne l'adresse de la première instruction du cœur de

boucle dès lors que la dernière des instructions de ce cœur a été lue. Ceci est illustré, dans le cas d'une boucle simple par l'exemple suivant :

```
for (i=1;i<=N;i++){
    inst_i;
    inst_i+1;
    inst_i+2;
    inst_i+3;
};
inst_i+4;
```

Ce programme doit se traduire au niveau des générateurs d'adresses par :

```
REPEAT N,M||PC++     ||PC++;      // M=4
    read(inst_i)     ||PC++;
    read(inst_i+1)   ||PC++;
    read(inst_i+2)   ||PC++;
    read(inst_i+3)   ||PC++-M;
read(inst_i+4)       ||PC++;
```

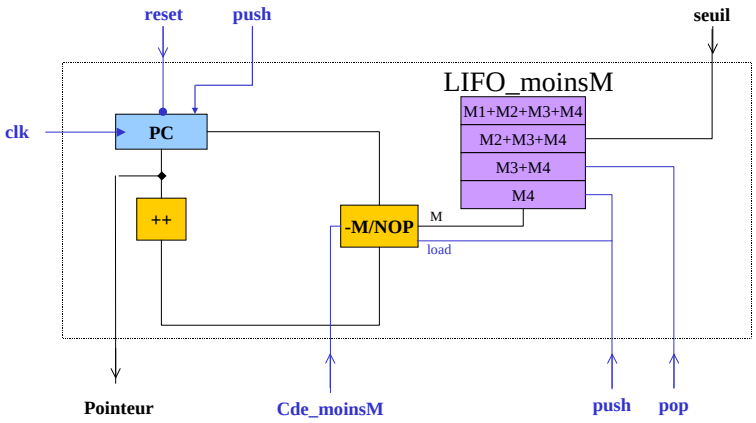

FIG. 2.14 – *Architecture du séquenceur des générateurs d'adresses. Le registre PC détermine l'adresse de l'instruction en cours d'exécution. Sa mise à jour est effectuée en fonction des signaux issus du module de gestion de boucle.*

Principe de fonctionnement Le module de gestion de boucle (Fig. 2.15) s'organise autour d'un compteur déterminant le nombre de cycles passés dans une boucle : le compteur de boucle (CPT). Pour conditionner le saut de la dernière vers la première instruction d'un cœur de boucle, le reste de la division du compteur de boucle et du nombre d'instructions de la boucle (M) est calculé. Si CPT est un multiple de M, la soustraction du compteur de programme des AGs par M et validée (Fig. 2.14). PC pointe dès lors sur la première instruction de la boucle.

Ce comportement perdure tant que toutes les itérations d'une boucle n'ont pas été traitées. Lors de la $N^{ième}$ itération, l'instruction suivant la $M^{ième}$ du cœur de boucle sera l'instruction suivant la boucle (inst_i+4 sur l'exemple précédent). Lorsque cette dernière itération est détectée, il faut donc invalider la soustraction de PC et de M. Dans le cadre de cette détection, puisque CPT stocke le nombre de cycles passés dans la boucle, cette valeur doit être comparée

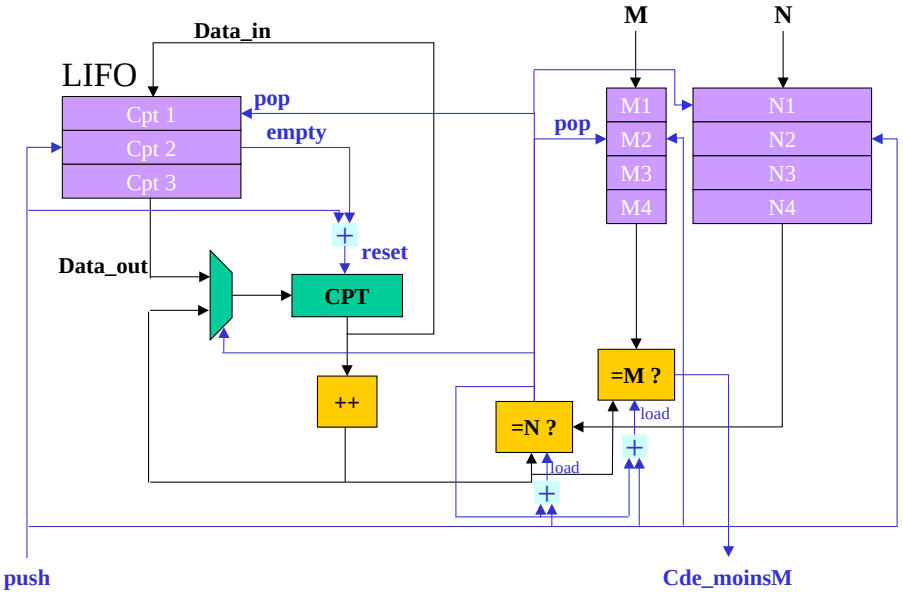

FIG. 2.15 – *Architecture du module de gestion de boucle. Le registre CPT stocke le nombre de cycles passés dans la boucle courante. Sa valeur est comparée à M et N afin de détecter les fins d'itération et les fins de boucle. 3 piles de type LIFO permettent de stocker ces différents paramètres pour les 4 niveaux d'imbrications supportés par ce module.*

au produit de N et de M, et non au nombre d'itérations seul. Il faudra en effet passer $N \times M$ cycles dans la boucle pour l'exécuter entièrement.

Cas des boucles multiples Il est en outre envisageable que l'une des instructions d'un cœur de boucle soit elle-même une instanciation de boucle. Dans ce cas, les mécanismes de gestion du PC sont sensiblement plus complexes et nécessitent de mémoriser la position dans la boucle courante ainsi que celles des boucles de niveaux supérieurs. Afin de maintenir la possibilité d'avoir 15 instructions dans un cœur de boucle et ce, quel que soit le niveau d'imbrication de cette boucle, les instructions du cœur de boucle du niveau le plus faible ne sont pas considérées comme des instructions par les boucles de niveaux supérieurs. Ceci permet de ne pas limiter la complexité (nombre d'instructions) des boucles imbriquées. Dans l'exemple suivant, M1 sera égal à 3 et M2 à 4.

```
for (i=1;i<=N1;i++){
  inst_k;
  for (j=1;j<=N2;j++){
    inst_k+2;
    inst_k+3;
    inst_k+4;
    inst_k+5;
  };
  inst_k+6;
};
inst_k+7;
```

Ce code sera traduit par :

```
REPEAT N1,M1          || PC++       //M1=3
  read(inst_i)        || PC++;
  REPEAT N2,M2        || PC++       //M2=4
    read(inst_i+2)    || PC++;
    read(inst_i+3)    || PC++;
    read(inst_i+4)    || PC++;
    read(inst_i+5)    || PC++-M1;
  read(inst_i+6)      || PC++-(M2+M1-1);
read(inst_i+7)        || PC++;
```

Cet exemple fait par ailleurs apparaître le fait que pour la dernière instruction de la boucle de niveau 1 (*REPEAT N1,M1*), il faut soustraire *M1+M2* et non pas simplement *M1* au *PC*. Ceci s'explique par le fait que les instructions de la boucle de niveau 2 (*REPEAT N2,M2*) ne sont pas considérées comme telles par la boucle de niveau 1 qui ne "voit" que l'instruction *REPEAT N2,M2*. Pour correctement mettre à jour le *PC* il faut cependant prendre en compte les incrémentations du *PC* réalisées pendant le traitement de la boucle de niveau 2. Ces mécanismes complexes se sont traduits par l'architecture représentée sur la figure 2.15.

Jeu d'instructions et modèle de programmation

Le modèle de programmation des générateurs d'adresses de DART est comparable à celui des processeurs programmables actuels. Les AGs forment alors de bonnes cibles pour des outils de compilation. Du point de vue de la reconfiguration, ils ne nécessitent que le chargement de leur mémoire d'instructions (64 mots de 16 bits). Ils sont reconfigurés en même temps que le contrôleur du *cluster*. Il est cependant possible d'envisager une évolution de DART dans laquelle leur reconfiguration pourrait intervenir en cours de traitement afin, par exemple, de supporter l'implémentation d'applications disposant de branchements conditionnels.

Comme il a été dit plus tôt, les modes d'adressage supportés sont relativement nombreux. De ce fait, le jeu d'instructions de ces AGs est conséquent et peut être résumé par les opérations listées ci-dessous.

REPEAT Ces instructions permettent de spécifier l'entrée dans une boucle. Elles spécifient par ailleurs le nombre d'itérations de la boucle et le nombre d'instructions du cœur de boucle.

NOP Ces instructions permettent de ne pas systématiser les accès à la mémoire d'instructions. En effet, faire suivre une instruction *REPEAT* par un *NOP* permet d'émuler les instructions *WAIT* utilisées au niveau du contrôleur de *cluster*.

LOAD Diverses déclinaisons des instructions de chargement de registre existent. Celles-ci permettront par exemple de réaliser concurremment un adressage immédiat et un chargement de registre.

MODULO DART supporte à la fois les adressages modulo en mode direct et en mode inverse. En mode direct, une valeur de pointeur est incrémentée avant d'être comparée à l'adresse de fin d'un tableau afin de détecter un éventuel débordement qui devra se traduire par l'affectation du pointeur à la valeur de l'adresse de début du tableau. En mode inverse, le pointeur est décrémenté et comparé à l'adresse de début du tableau. En cas de débordement, le pointeur se voit affecter la valeur de l'adresse de fin du tableau.

ADDITION/SOUSTRACTION Diverses déclinaisons des instructions d'addition et de soustraction existent. En particulier, les incrémentations et les décrémentations ont été distinguées des opérations classiques d'additions et de soustractions afin d'en limiter

la consommation. La sortie de l'UAL étant par ailleurs connectée à l'entrée de la file de registre (Fig. 2.12), il est possible de réaliser des pré-incrémentations ou des pré-décrémentations. Ces opérations traiteront des données immédiates de la même façon que des données issues des registres, et le rangement du résultat d'un calcul dans un registre pourra être concurrent à son envoi vers l'adresse de la mémoire de données.

BIT REVERSE Toutes les opérations arithmétiques pourront également être suivies d'un codage bit-reverse, susceptible d'opérer sur 6, 7 ou 8 bits.

2.4.3 Le contrôleur mémoire

Le contrôleur mémoire du *cluster* permet de faire transiter les données entre la mémoire du *cluster* et les mémoires internes aux DPRs. Pour cela, il se doit de générer les adresses et les signaux de contrôle nécessaires à l'orientation des données, soit depuis la mémoire du *cluster* vers les mémoires locales, soit dans le sens inverse (Fig. 2.16). Ces mouvements de données étant statiquement programmés et déterminés au moment de la compilation, le rôle de ce contrôleur est de séquencer et de décoder des instructions de transferts de données. Son architecture est donc similaire à celle des générateurs d'adresses présentés dans le paragraphe précédent. Il intègre également une structure de gestion matérielle des boucles.

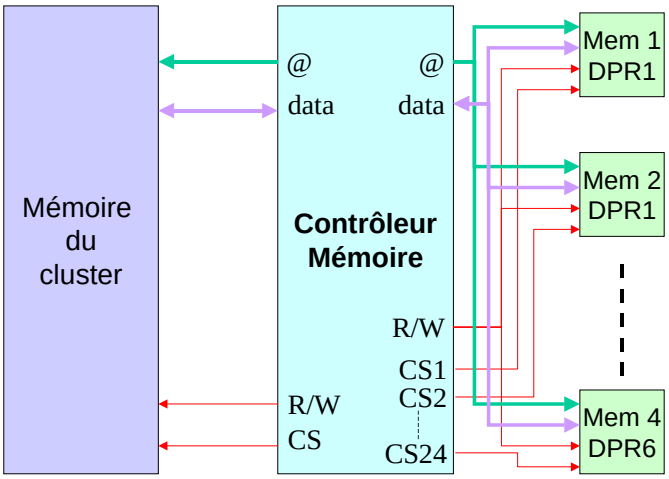

FIG. 2.16 – *Synoptique du contrôleur mémoire d'un cluster : il se comporte comme une interface programmable entre la mémoire de masse et les mémoires internes aux DPRs. Il se charge de gérer les transferts de données entre ces différents éléments en générant les adresses et les signaux de contrôle statiquement déterminés au moment de la compilation.*

Tous les modes d'adressage supportés par les générateurs d'adresses des DPRs n'ont pas lieu de l'être par le contrôleur mémoire du *cluster*. En effet, la génération d'adresses, au niveau du DPR, consiste typiquement à parcourir les échantillons d'un vecteur ou d'un bloc de données en suivant un motif de calcul propre à l'algorithme implémenté (e.g. modulo, bit-reverse). La tâche du contrôleur mémoire du *cluster* se limite quant à elle à la mise à jour du vecteur ou du bloc de données stocké dans les mémoires locales. Ce principe peut être illustré par l'exemple d'un filtre FIR :

```
for (n=0;n<Taille_Paquet;n++){  //Pour chaque échantillon du paquet
```

```
    for(j=1;j<N;j++)            //Viellissement du vecteur X[]
    {
        X[j]=X[j-1];
    }
    X[0]=paquet[n];             //Mise à jour du vecteur X[]
    for(j=0;j<N;j++             //Filtrage
    {
        y[n]+=X[j]*H[N-j];
    }
}
```

La boucle supérieure de cet exemple permet de parcourir un ensemble de données devant être filtrées. À chaque itération de cette boucle, l'échantillon n du vecteur *Paquet* est lu et écrit dans le vecteur X. C'est sur ce vecteur X que s'applique le filtrage. Concrètement, le vecteur X sera stocké dans les mémoires locales des DPRs et le vecteur *Paquet* dans la mémoire du *cluster*. Les générateurs d'adresses des DPRs se chargeront de faire évoluer la variable i, lorsque dans le même temps, le contrôleur mémoire se chargera de la gestion du pointeur n. De cet exemple peuvent être extraites quelques constations.

1. Les mises à jour ne sont qu'occasionnelles. Typiquement, ces mises à jour sont réalisées à chaque itération de la boucle supérieure et non à chaque itération du cœur de boucle implémenté sur le DPR. Dans l'exemple précédent, les mises à jour n'interviennent qu'entre chaque filtrage d'échantillon.
2. Les transferts de données se font soit un à un, soit par paquet. Dans les deux cas, un adressage direct avec pré- ou post-incrément est suffisant.
3. Plusieurs mises à jour peuvent être réalisées concurremment. En effet, si une donnée est dupliquée dans plusieurs mémoires locales, il est possible de réaliser les mises à jour de tous les vecteurs en mémoire de manière concurrente. Pour cela, il suffit d'envoyer simultanément les mêmes adresses et les mêmes données à plusieurs mémoires locales. En pratique, il s'agira ici d'activer simultanément plusieurs *Chip Select* (*CSi* sur la figure 2.16).

L'architecture du contrôleur mémoire se base donc sur celle des générateurs d'adresses. À la différence de ces derniers, il n'intègre cependant qu'un nombre limité de modes d'adressage. Outre les adresses des données manipulées, ce module génère également l'ensemble des signaux de contrôle nécessaires à l'identification des mémoires ciblées par le transfert de données (e.g. *Chip Select*).

2.5 La reconfiguration dynamique

2.5.1 Introduction

Si la flexibilité introduite au sein des DPRs autorise la définition d'un chemin de données en adéquation avec le motif de calcul à implémenter, elle se traduit également par un volume de données qui, bien que sans commune mesure avec celui des architectures reconfigurables de grain fin, est conséquent. Les différentes cibles de la reconfiguration, et le volume de données qu'elles requièrent, sont listées dans le tableau 2.3.

Cible de la reconfiguration	**Nb Bits /ressource**	**nb ressources /DPR**	**nb bits /DPR**	**nb bits /cluster**
Switch Box (SB)	46			46
écriture mémoire	4	4	16	96
écriture bus globaux	4	4	16	96
entrées UF (depuis UF)	4	10	40	240
gardes des générateurs d'adresses (AG)	1	4	4	24
entrées UF (depuis mém.)	2	10	20	120
gardes registres	1	6	6	36
Multiplieurs/additionneurs	3	2	6	36
UALs	11	2	22	132
total				**826**

TAB. 2.3 – *Distribution du volume de configuration entre les différents éléments du cluster. À chaque cible de la reconfiguration est associé un certain nombre de bits. La seconde colonne précise le nombre de bits nécessaires à la configuration d'un élément. La colonne suivante précise le nombre de ressources par DPR. Les colonnes 4 et 5 précisent finalement le nombre de bits nécessaires à la configuration de ces cibles, respectivement dans un DPR et dans le cluster complet.*

Il est à noter dans ce tableau que 72% du volume de configuration est utilisé pour les interconnexions. Ces 598 bits sont en particulier utilisés pour spécifier de quelles mémoires sont issues les données, de quelle façon sont interconnectées les unités fonctionnelles ou les DPRs et finalement dans quelles mémoires seront écrits les résultats. Dès lors, malgré le haut niveau de flexibilité des unités fonctionnelles, et notamment des UALs, le volume de configuration qu'elles requièrent est limité (20%) comparativement au reste du *cluster*. Il est par ailleurs à noter que l'intégration d'horloges gardées est aussi peu problématique du point de vue de la configuration (7%) qu'elle est intéressante du point de vue de la consommation d'énergie [6].

La flexibilité étant un enjeu majeur des télécommunications 3G, DART doit être en mesure de traiter tout type d'applications, y compris les applications irrégulières dans lesquelles les motifs de calculs se succèdent sans ordre particulier et de manière non répétitive. Nous devons donc être en mesure de reconfigurer les ressources de calcul d'un *cluster* en 1 cycle. Cependant, la contrainte de faible consommation nous impose d'utiliser des instructions de taille suffisamment faible pour que l'essentiel de la consommation d'énergie ne soit pas issue de la lecture des instructions mais bien des opérations de calcul. En effet, une architecture ne peut être qualifiée de faible consommation qu'à la condition que l'énergie consommée durant un traitement est utile à l'obtention des résultats (opérations, lecture de données, ...). À ce titre, les lectures et les décodages d'instructions sont donc perçus comme des gaspillages d'énergie.

Afin de réduire la taille des instructions (initialement de 826 bits), deux méthodes ont été employées. Toutes deux dérivent de la règle des 80/20. Cette règle, validée par la pratique dans [65] et [103], affirme que 80% du temps d'exécution d'un programme est consommé par 20% du code, et que seul les 20% du temps d'exécution restant sont consommés par le reste des instructions du code source. Cette observation nous a conduit à distinguer les 20% de codes qui consomment la plus grande part du temps d'exécution, que nous qualifions de réguliers, des 80% restants, qualifiés d'irréguliers, qui sont exécutés d'une manière très

sporadique. Par cette distinction, nous avons pu définir deux modes de reconfiguration, la reconfiguration logicielle (SW) et la reconfiguration matérielle (HW), chacun adapté à l'un ou l'autre de ces types de traitements. Nous avons par ailleurs pu valider un nouveau concept : le SCMD (*Single Configuration Multiple Data*).

*2.5.2 Le SCMD (*Single Configuration Multiple Data*)*

Introduction

Les traitements réguliers et les traitements irréguliers ont des caractéristiques radicalement opposées. En particulier, les traitements irréguliers disposent d'un parallélisme extrêmement réduit. En effet, dans les portions de codes irrégulières, les instructions se succèdent sans ordre particulier et d'une manière non répétitive. Les nombreuses dépendances de données inhérentes à ces portions de code limitent par ailleurs leur parallélisme d'opérations. Du fait de leur faible complexité, ces portions de code ont une influence minime sur le temps d'exécution de l'application.

À l'inverse, les traitements réguliers sont caractérisés par un fort degré de parallélisme d'opérations. Dans le cadre de leur implémentation, les mêmes séquences d'opérations sont répétées un grand nombre de fois avec des dépendances de données quasi-inexistantes. Celles-ci sont en effet typiquement limitées aux accumulations et sont donc très occasionnelles. À cette régularité et à ce fort degré de parallélisme s'ajoutent par ailleurs de fortes contraintes de performance. L'efficacité de leur exécution a donc un impact décisif sur la qualité de l'implémentation.

Compte-tenu de l'antinomie de ces deux types de traitements, leurs implémentations sont également très différentes.

- Les traitements irréguliers sont implémentés sur un seul DPR. Étant donné leur faible niveau de parallélisme, ce nombre réduit d'unités fonctionnelles utilisables (4) est en effet amplement suffisant.
- Les traitements réguliers sont quant à eux implémentés sur autant de DPRs que possible, afin d'exploiter au mieux leur parallélisme et de minimiser leur temps d'exécution. Les méthodes d'extraction de parallélisme étant typiquement basées sur des techniques de déroulage de boucle et de pipeline logiciel, ces implémentations sont par ailleurs caractérisées par une régularité importante. De ce fait, les configurations des différents DPRs exploités seront très similaires.

Afin de reconfigurer les *clusters* en utilisant un minimum de données, nous avons tiré parti de ces caractéristiques pour définir un nouveau concept, le SCMD. Ce dernier, acronyme de *Single Configuration Multiple Data*, est une évolution du concept de SIMD dans lequel plusieurs opérateurs exécutent une même opération sur des jeux de données différents. Dans le cadre du SCMD, le partage des configurations n'est plus limité aux seuls opérateurs mais est étendu aux DPRs.

Par le bais de ce concept, les 826 bits initialement indispensables à la configuration de 6 DPRs d'un *cluster* peuvent être réduit à 156, soit un facteur de réduction supérieur à 5. Cette minimisation du volume de configuration n'est cependant possible que dans la mesure où la régularité de l'algorithme peut être reproduite au niveau de l'architecture. Dans le cas contraire, il est indispensable de maintenir la possibilité de contrôler chaque bit de configuration d'un DPR indépendamment des configurations des autres DPRs. Entre ces deux extrêmes, il existe

bien sûr de nombreuses possibilités, où seule une portion de la régularité de l'application peut être extraite. Il est donc souhaitable de partitionner le flot de configuration, afin d'autoriser l'extraction de tout ou partie de ce dernier en vue de partager certaines informations de configuration entre plusieurs DPRs. À cette fin, la configuration a été découpée en un flot d'instructions qui sont distribuées aux DPRs par le biais du contrôleur du *cluster*.

Mise en œuvre du SCMD

Afin d'autoriser la reconfiguration simultanée de plusieurs DPRs tout en maintenant la possibilité de les reconfigurer indépendamment les uns des autres, un champ supplémentaire de 6 bits a été introduit au sein des instructions de configuration. Dans ce champ, chaque bit est considéré comme un *DPR_ select*, validant ou non les informations de configuration pour un DPR donné. Ce principe est illustré par la figure 2.17.

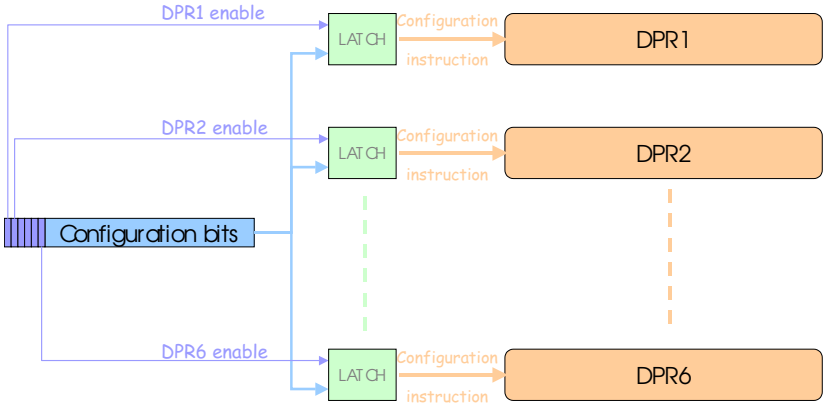

FIG. 2.17 – *Mise en œuvre du SCMD. Un champ de 6 bits est concaténé aux instructions de configuration. Chaque bit rajouté est dédié au contrôle d'un latch afin de valider ou non une instruction de configuration pour un DPR donné. Lorsque plusieurs gardes sont simultanément actives, l'instruction de configuration est distribuée simultanément à plusieurs DPRs.*

La mise en œuvre du concept de SCMD se traduit par un surcoût matériel correspondant à une extension de 6 bits de la largeur de la mémoire d'instructions du contrôleur, et à l'insertion de quelques *latchs* au sein des DPRs. Ce surcoût est donc extrêmement limité comparativement à la réduction du volume de données de configuration sous-jacente. Les différentes implémentations réalisées démontrent en effet des facteurs de réduction du volume de configuration variant entre 1.5 et 5, en fonction de la régularité du traitement implémenté (cf. chapitre 4).

2.5.3 La Reconfiguration logicielle (SW)

Principe de la reconfiguration logicielle

Le mode de reconfiguration logicielle a été conçu afin de répondre aux besoins inhérents aux zones de calcul irrégulières, où les motifs de calculs se succèdent très rapidement, sans ordre particulier et de manière non répétitive. La principale caractéristique de ce mode de reconfiguration est d'autoriser une reconfiguration des DPRs en un cycle. Afin d'autoriser cette reconfiguration en un cycle, éventuellement systématique des DPRs, la flexibilité de ces

derniers a été limitée en adoptant un motif de calcul de type *Read-Modify-Write*. Dans ce cas, le modèle de calcul est donc comparable à celui des DSPs VLIW conventionnels. À chaque cycle les données sont lues, traitées, puis les résultats sont rangés en mémoire. Ceci justifie la qualification de logiciel de ce mode de reconfiguration. Il n'est ici en aucun cas possible de chaîner des opérateurs ou de réaliser des traitements SWP.

Les cibles de la reconfiguration sont, dans ce mode de reconfiguration, en nombre limité puisqu'il s'agit de préciser de quelles mémoires sont issues les données manipulées, et quelles sont les opérations à réaliser. Pour limiter plus encore le nombre de bits nécessaires à cette reconfiguration, nous imposons que chaque unité écrive toujours son résultat dans la même mémoire, e.g. le 1^{er} multiplieur écrit dans la mémoire 1 et le 2^{nd} dans la mémoire 3, ... Ce mode de reconfiguration est illustré par la figure 2.18.

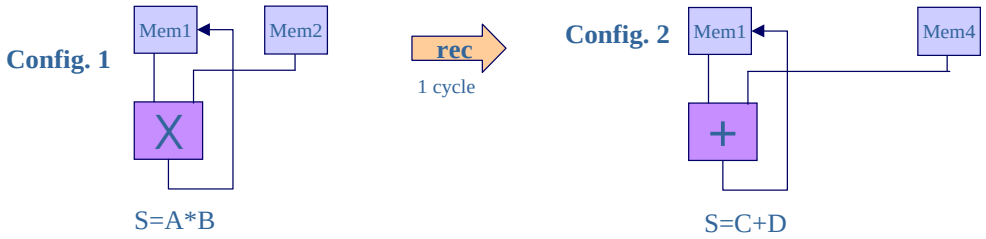

FIG. 2.18 – *Exemple de reconfiguration logicielle : passage d'une multiplication de deux nombres stockés dans les mémoires 1 et 2 à l'addition de deux nombres stockés dans les mémoires 1 et 4. Durant les deux cycles que durent ces deux opérations, les résultats seront rangés systématiquement dans la mémoire 1.*

Les instructions de reconfiguration SW

Ces reconfigurations logicielles sont spécifiées par le biais d'instructions de 52 bits, issues du contrôleur de *cluster* (Fig. 2.19). Ces configurations sont identifiées par le biais d'un champ de 2 bits dans l'instruction. Elles ont pour but de définir une structure de chemin de données autorisant le traitement des instructions de type *Read-Modify-Write*. Chaque étape de ce calcul nécessite donc la commande de certains éléments du DPR.

Read Sachant que les opérandes ne sont issues que des mémoires internes au DPR, il faut commander :
- les multiplexeurs 4 vers 1 (MUX 4 :1) en entrée des UFs,
- les MUX 2 :1 afin d'envoyer les sorties des MUX 4 :1 vers les entrées des UFs. Cette commande est issue d'un décodage portant sur le type d'instruction (si instruction SW alors commande MUX2 :1='0'),
- les gardes des générateurs d'adresses.

Puisqu'il n'y a pas de chaînage des opérateurs, il est inutile de commander les MUX 14 :1 des différentes unités fonctionnelles. Les registres ne sont par ailleurs pas utilisés lors des reconfigurations logicielles.

Modify Le traitement des données nécessite de spécifier :
- les gardes des registres en sortie des UFs,
- les décalages en entrée des UALs et en sortie des multiplieurs/additionneurs,
- les opérations des UFs.

Write Chaque UF écrit dans la mémoire qui lui est associée. La commande des MUX 10 :1 en entrée des mémoires est donc implicite et issue d'un décodage portant sur le type d'instruction (si instruction SW alors UF_i écrit dans la mémoire i $\Rightarrow$ commande $MUX10:1$ MEM_i = i).

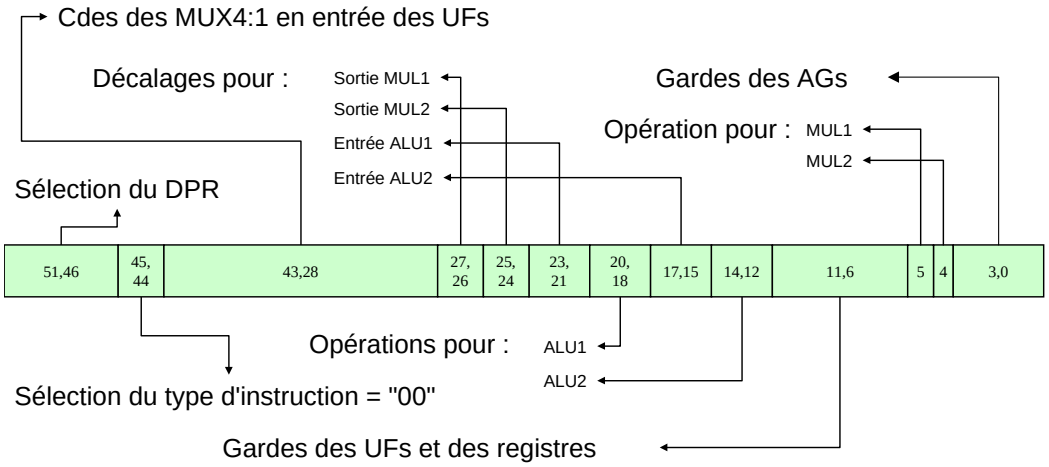

FIG. 2.19 – *Format des instructions SW*

2.5.4 La Reconfiguration matérielle (HW)

Principe de la reconfiguration matérielle

Outre la reconfiguration logicielle, adaptée aux traitements irréguliers, un second mode de reconfiguration a été défini afin de répondre aux besoins des traitements réguliers, tels que les cœurs de boucles, dans lesquels un même motif de calcul est utilisé pendant de longues périodes de temps. Ce mode de reconfiguration est qualifié de Hardware car il se traduit par un potentiel d'optimisation comparable à celui des solutions matérielles. En effet, il s'agit ici d'assurer une totale flexibilité au sein du DPR afin d'autoriser l'optimisation du chemin de données en fonction du motif de calcul. En contrepartie du fort potentiel d'optimisation de ce mode de reconfiguration, le passage d'un motif de calcul à un autre nécessite une phase de reconfiguration dont la durée varie typiquement de 3 à 19 cycles[3], en fonction de la régularité de la configuration (i.e. de l'efficacité du SCMD) et du nombre de DPRs utilisés.

Cette reconfiguration HW consiste à spécifier une configuration par le biais d'un flot d'instructions, puis d'adopter un modèle de calcul de type *dataflow* dans lequel la structure du chemin de données est figée. Une fois cette configuration spécifiée, le contrôleur du *cluster* est donc libéré du contrôle des DPRs et n'a plus à accéder à la mémoire d'instructions et à décoder de nouvelles instructions. Les informations de configuration sont alors stockées dans des *latchs*, au sein des DPRs. Avec ce modèle de calcul il ne reste plus qu'à approvisionner les unité fonctionnelles en données. Le contrôle est ainsi dévolu aux unités de génération d'adresses.

Ce mode de reconfiguration est illustré par la figure 2.20. Sur cette figure, une configuration de filtrage est tout d'abord spécifiée sur un DPR (partiellement représenté), puis un modèle

[3]Le contrôleur du *cluster* distribue une instruction par cycle.

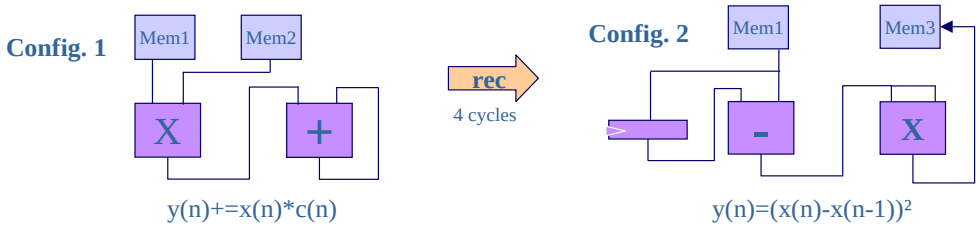

FIG. 2.20 – *Exemple de reconfiguration matérielle. L'architecture est dans un premier temps en adéquation avec une application de filtrage. Après une phase de reconfiguration de 4 cycles, le motif de calcul est optimisé pour implémenter le calcul de* $(x(n) - x(n-1))^2$

de calcul de type flot de données (*dataflow*) est adopté. Le temps du filtrage, aucune autre instruction n'a à être lue par le contrôleur du *cluster* et les générateurs d'adresses se chargent seuls de gérer l'envoi des données sur le multiplieur. Une fois le filtrage terminé, le contrôleur du *cluster* "reprend la main" de manière à spécifier une nouvelle configuration visant à optimiser le DPR pour le calcul de $(x(n) - x(n-1))^2$.

Le flot d'instructions de reconfiguration HW

Afin de reconfigurer entièrement un DPR, une suite de 4 instructions de 52 bits est nécessaire. Celle-ci débute par une instruction SW qui spécifie les fonctions des opérateurs, ainsi que les éventuelles lectures des mémoires (cf. §2.5.3). Elle est suivie de deux instructions HW (Fig. 2.21 et 2.22) qui reconfigurent tout ce qui ne l'a pas été par l'instruction SW au sein du DPR. Finalement, une instruction d'interconnexion spécifie les éventuelles connexions entre DPRs. Si plusieurs DPRs doivent être reconfigurés, ils partagent cette instruction d'interconnexion. En fonction du nombre N de DPRs à configurer, si l'on n'exploite pas le concept SCMD, $3 \times N + 1$ instructions sont donc nécessaires à la spécification de la configuration. Si le réseau segmenté n'est pas utilisé (i.e. si chaque DPR travaille indépendamment de ces homologues), ce nombre d'instructions se réduit à $3 \times N$.

Les instructions HW ont à définir :
- les commandes des MUX 10 :1 en entrée des UFs et des registres. À partir de ces commandes, de la logique générera les commandes des MUX 2 :1 pour déterminer si les entrées des opérateurs sont issues des mémoires ou d'autres UFs (si cdeMUX14 :1≥14 alors cdeMUX2 :1='0' sinon cdeMUX2 :1='1'),
- les commandes des MUX4 :1 en entrée des registres,
- les commandes des MUX14 :1 en entrée des mémoires,
- les commandes des MUX11 :1 contrôlant les accès aux bus globaux,
- les décalages en sortie des UALs,
- le format des données (pour l'éventuel fonctionnement SWP).

Toutes ces informations sont codées dans les instructions HW-1 et HW-2, respectivement représentées sur les figures 2.21 et 2.22. Ces instructions disposent à nouveau d'un champ de 2 bits autorisant leur identification et d'un champ de 6 bits autorisant l'exploitation du SCMD.

Une dernière instruction vient conclure la reconfiguration HW. Celle-ci, représentée sur la figure 2.23, vise à configurer les Switch Box (SB) du réseau segmenté, et à déterminer, pour chaque DPR, si les écritures se font sur les bus pairs ou les bus impairs. De même que pour les instructions SW et HW, ces instructions intègrent un champ de 2 bits permettant de les identifier.

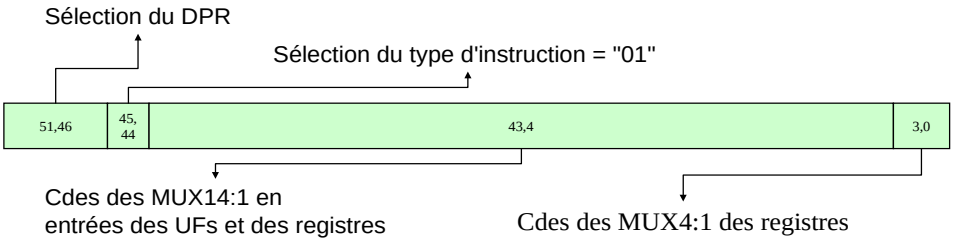

FIG. 2.21 – *Format des instructions HW-1*

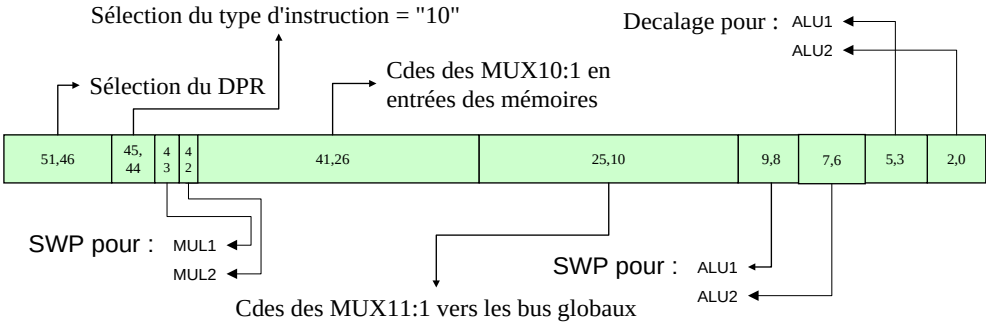

FIG. 2.22 – *Format des instructions HW-2*

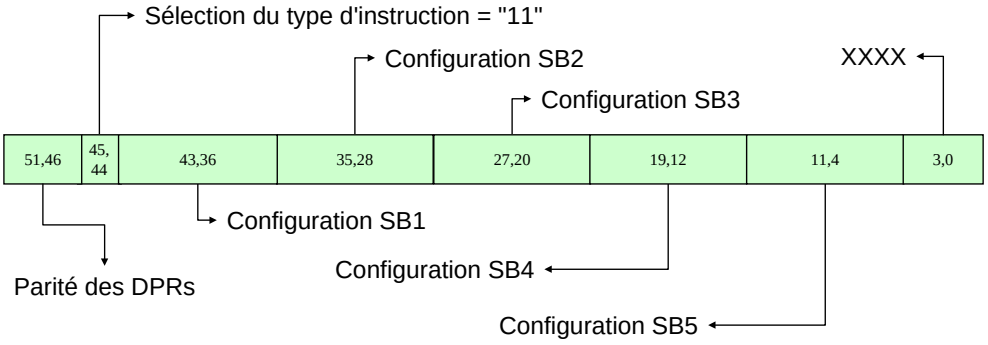

FIG. 2.23 – *Format des instructions d'interconnexion*

2.6 Conclusions

Dans ce chapitre, l'architecture DART a été présentée. Celle-ci vise à répondre aux besoins inhérents aux applications mobiles de prochaines générations : hautes performances, faible consommation, flexibilité et faible coût. Les concepts définis ou exploités pour arriver à ces fins ont été exposés. Leur mise en œuvre s'est traduite par la définition d'une architecture hiérarchisée, capable de s'adapter aux caractéristiques de l'application implémentée (motif de calcul, degré de parallélisme, ...).

En minimisant le volume des données de configuration et en adaptant les mécanismes de distribution du contrôle au travers de l'architecture à l'application exécutée, des modèles de programmation simples ont par ailleurs été extraits. L'exploitation de ces modèles fait l'objet du chapitre 3.

Cette architecture peut par ailleurs être considérée comme une plateforme matérielle [104], au sens où les ressources intégrées dans DART peuvent être modifiées sans limiter l'efficacité des concepts précédemment évoqués. Dans cette étude, nous avons abordé ce problème d'une manière pragmatique, en dimensionnant l'architecture aux besoins des télécommunications 3G. Ainsi, les 12 GOPS requis par ces applications ce sont traduits par une découpe en 4 *clusters* qui intègrent chacun 6 DPRs de 4 unités fonctionnelles. La modification de ces paramètres n'a que peu d'impact sur l'architecture et les outils mais la détermination de ces derniers est en revanche un exercice délicat. Il apparaît ainsi une perspective intéressante à ce travail. La définition de méthodologies permettant d'explorer l'espace des solutions proposées par DART et de métriques permettant d'en extraire l'optimale, pour un domaine applicatif donné, semble en particulier très motivant.

Chapitre 3

Flot de compilation pour DART

Sommaire

3.1 Introduction

Le chapitre 2 a mis en exergue les différents modèles de programmation de DART. Les divergences conceptuelles entre la génération d'adresses ou les reconfigurations SW, et les

reconfiguration HW peuvent dès lors être clairement identifiées. Les premières affichent un modèle de programmation comparable à celui des processeurs programmables. À l'opposé, les reconfigurations HW traduisent un modèle d'exécution similaire à celui des solutions matérielles (ASIC, FPGA). La conception d'un outil de développement, traduisant un code de haut niveau en un code binaire exécutable, qui soit en mesure de cibler deux modèles d'exécution aussi différents que peuvent l'être les modèles d'exécution spatiaux et temporels, est inconcevable pour DART.

En effet, les outils ciblant ces deux modèles d'exécution varient singulièrement dans leur manière de traiter une application. Dans le cadre d'une implémentation spatiale, il s'agit de transformer un motif de calcul de manière à l'implémenter d'une manière optimale, sur un nombre suffisant de ressources de calcul et de stockage, sous contrainte de coût ou de latence. À l'inverse, pour une implémentation temporelle, l'objectif est d'ordonnancer les traitements à réaliser pour optimiser le partage d'un faible nombre de ressources, de calcul et de mémorisation, entre les différents traitements constituant l'application. En conséquence, les outils générant ces programmes où ces configurations exploitent des méthodes très différentes. Malgré un rapprochement certain des mondes de la conception logicielle et matérielle, leur association reste aujourd'hui encore très difficile.

De ce fait, l'approche utilisée pour DART n'a pas été d'envisager une chaîne de développement monolithique mais au contraire de la décomposer de manière à traiter efficacement les différents modèles de programmation (reconfiguration HW, SW ou génération d'adresses). L'un des aspects clés de notre méthodologie de développement consiste donc à partitionner l'application. Dans la littérature, diverses approches ont été proposées. La plupart d'entre-elles se limitent à la distinction des codes devant être exécutés sur un processeur et ceux devant être implémentés sur des accélérateurs matériels. Ce partitionnement peut alors être manuel ou automatique.

3.1.1 Le partitionnement manuel

La complexité de la tâche de partitionnement conduit souvent les concepteurs à privilégier une approche manuelle. Cette approche se traduit typiquement par la définition d'un langage propriétaire ou par l'extension d'un langage courant par des directives ou des primitives particulières. Le langage OCCAM [105], développé dans le cadre de l'exploitation des *transputers* [106], fût l'un des premiers langages définis à ce titre. Par des primitives *PAR* ou *SEP* celui-ci permettait à l'utilisateur de préciser explicitement si une portion de code devait être exécutée en parallèle sur plusieurs processeurs élémentaires, ou d'une manière séquentielle sur une ressource unique. D'autres primitives traduisaient dans le même temps des points de synchronisation et de communication.

Dans le cadre du calcul reconfigurable, cette approche est également exploitée dans de nombreux projets [107]. Le langage RaPiD-C est par exemple utilisé pour décrire les applications devant être implémentées sur l'architecture RaPiD [48]. Ce langage, proche du C dans sa syntaxe, permet d'expliciter le partitionnement et le parallélisme de l'application, ainsi que les mouvements de données, souhaités par le programmeur. La description de l'algorithme prend la forme de nids de boucles dont les plus bas niveaux sont traduits en configurations matérielles. Les boucles supérieures, quant à elles, permettent de générer les informations de contrôle du chemin de données (e.g. reset, vieillissement, ...).

Le langage *NAPA C* est un autre exemple de langage défini à partir du C. La modification est cependant cette fois plus réduite et moins contraignante puisqu'elle se limite à l'intégration de pragmas. Ces derniers autorisent l'utilisateur de NAPA à spécifier si une section de code doit être exécutée sur le processeur à jeu d'instruction fixe (FIP) ou sur le bloc reconfigurable (ALP). L'intérêt majeur de cette approche, consistant à minimiser les modifications faites sur la définition du langage de description originel, est de maintenir la possibilité d'utiliser des outils commerciaux. L'effort de développement, au niveau du *front-end* notamment, est ainsi fortement réduit.

3.1.2 Le partitionnement automatique

Des contraintes de portabilité de code poussent cependant bien souvent les concepteurs à privilégier des techniques de partitionnement entièrement automatisées. Cette automatisation autorise dès lors l'emploi de tout type de langage d'entrée, y compris les langages procéduraux tels que le C. L'automatisation du partitionnement est typiquement basée sur du profiling. Cette étape a pour but d'étudier la distribution de la complexité au travers de la description de l'application. Elle nécessite une connaissance a priori des jeux de données manipulés par l'application.

À titre d'exemples, les projets PICO [108], Pleiades [109] ou GARP [30] peuvent être cités. Dans chacun de ces projets, les codes réguliers sont implémentés par des outils dédiés sur des architectures massivement parallèles. Ces derniers sont des réseaux systoliques dans le cadre du projet PICO (Program-In Chip-Out), développé à HP labs, et des circuits reconfigurables au niveau fonctionnel et au niveau porte respectivement pour les projets Pleiades [110] et GARP [111] développés à Berkeley. Les codes irréguliers sont quant à eux implantés sur des processeurs programmables (de type RISC dans les projets Pleiades et GARP et de type VLIW dans le projet PICO) via une chaîne de compilation plus traditionnelle.

L'approche adoptée pour DART est d'automatiser cette étape de partitionnement. Si les méthodes de partitionnement ont traditionnellement été appliquées à la distinction entre les traitements intensifs et les codes irréguliers, nous leur avons ajouté une troisième cible. En effet, compte tenu des spécificités architecturales sous-jacentes à la génération d'adresses, les manipulations de données sont transformées en codes exécutables via des techniques de compilation qui leurs sont propres.

Dans ce chapitre, nous présentons le flot de développement de DART. La section 3.1.3 présente un aperçu de ce flot. Les différents modules qui le composent sont détaillés dans les sections suivantes. Malgré la complexité de la tâche d'extraction de parallélisme au sein de descriptions procédurales [90], les contraintes de portabilité de code nous ont conduit à utiliser le langage C comme point d'entrée de notre chaîne de développement. Ce choix a par ailleurs l'avantage d'offrir une large gamme d'outils, libres ou commerciaux, pour l'analyse et la manipulation des applications décrites dans ce langage.

3.1.3 Méthodologie de développement de DART

Le flot de développement de DART est principalement centré sur le résultat du partitionnement évoqué précédemment. Cette chaîne de développement est basée sur l'utilisation conjointe d'un *front-end* permettant la transformation et l'optimisation de codes C (Suif de

Stanford [112]), d'un compilateur reciblable (outil CALIFE de l'IRISA [113]) et d'un outil de synthèse de haut niveau (outil BSS de l'ENSSAT [114]).

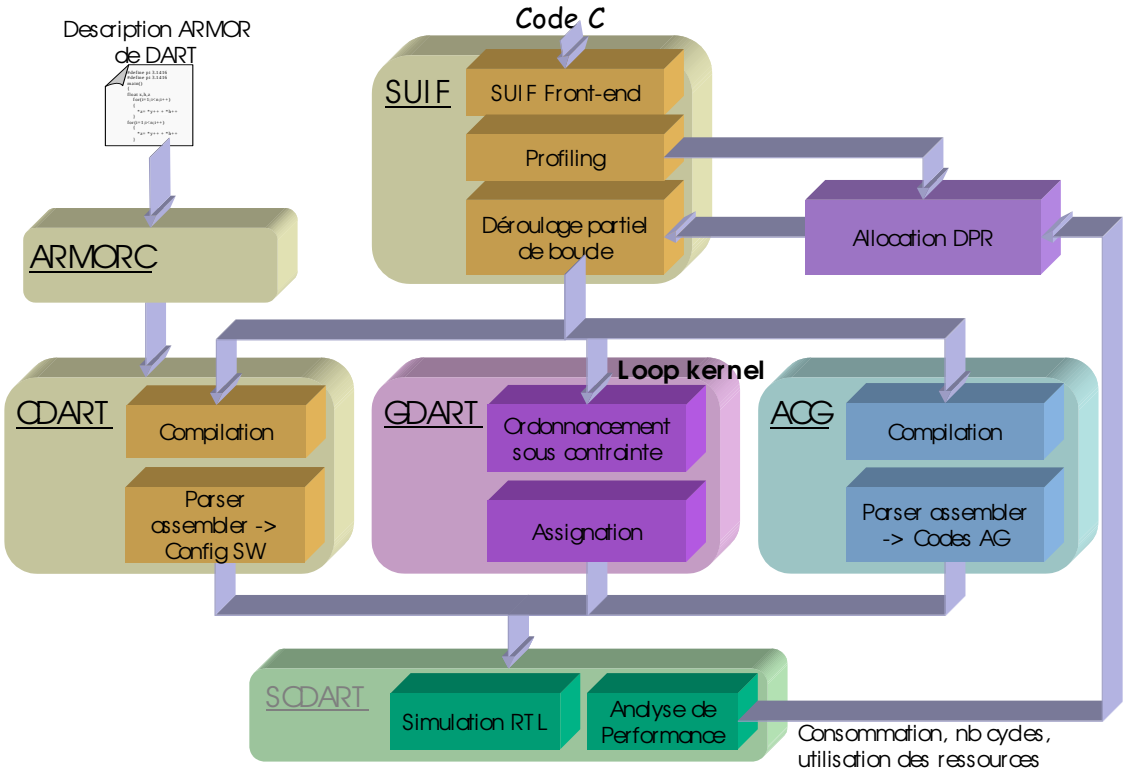

FIG. 3.1 – *Flot de conception de DART. Le code source est tout d'abord manipulé par un front-end SUIF. Celui-ci permet d'optimiser le code à un haut niveau et d'extraire le parallélisme de l'application. Une passe de partitionnement permet ensuite de distinguer les traitements irréguliers et les manipulations de données, des traitements réguliers. Les deux premiers types de code seront traduits en codes binaires via des passes classiques de compilation (issues de l'environnement CALIFE), respectivement dans les outils cDART et ACG. Les traitements réguliers sont quant à eux transformés en configurations de DPRs, par le biais de techniques issues de la synthèse comportementale de circuits dédiés (BSS), dans l'outil gDART. Les codes ainsi générés peuvent finalement être simulés par le module SCDART afin de vérifier l'efficacité de l'implémentation.*

La chaîne de développement associée à DART est modélisée sur la figure 3.1. À partir d'une application décrite en C, l'utilisation de Suif permet de générer un graphe de données et de contrôle (CDFG) et de réaliser un certain nombre de transformations automatiques. Pour ce problème sont développées des passes spécifiques de déroulage de boucles et d'extraction de codes réguliers. En sortie, la séparation entre les phases de configuration HW, de configuration SW et de génération des adresses à travers plusieurs spécifications C permet de traiter, par la compilation et la synthèse architecturale, l'ensemble d'une application. Une fois générés, les codes binaires exécutables sur DART (configurations HW, SW, codes des générateurs d'adresses) peuvent être simulés au moyen d'un outil développé en systemC [115]. Celui-ci permet en outre d'évaluer la qualité de l'implémentation, sur des critères de performance, de taux d'occupation des ressources ou d'énergie consommée.

Dans la suite de ce chapitre, les différents modules apparaissant sur la figure 3.1 et s'intégrant

dans notre flot de développement sont présentés. La section suivante se concentre sur la partie frontale de notre outil, chargée de mettre en forme l'application de manière à pouvoir attaquer les modules plus spécifiques de transformation de code qui seront décrits par la suite.

3.2 Partie frontale du flot

3.2.1 Généralités

Le système SUIF (Stanford University Intermediate Format) est un environnement conçu par l'Université de Stanford pour le développement et l'évaluation de nouvelles techniques de compilation. Cet environnement met à la disposition des utilisateurs une partie frontale compatible avec différents langages (C, Fortran, C++, Java, ...), ainsi qu'une représentation intermédiaire (IR). Diverses bibliothèques facilitent la manipulation et la transformation de cette IR. Deux systèmes ont été développés à ce jour, SUIF1 [112] et SUIF2 [116]. Le second diffère du premier par un *front-end* autorisant le traitement de codes écrits en langage objet (C++ ou Java). Ce qui sera présenté dans la suite du présent document concernera uniquement la première déclinaison de SUIF. Tous les concepts et développements évoqués pourront cependant aisément être transposés dans SUIF2.

Le système SUIF est organisé comme un ensemble de passes de compilation. À chaque passe, une analyse ou une transformation du code est effectuée et le résultat est écrit dans un fichier respectant le format de celui d'entrée. Ainsi, puisque toutes les passes manipulent une même structure de données, elles peuvent être réorganisées de manière très simple, par permutation des ordres d'exécution. De même, de nouvelles passes de compilation peuvent être insérées ou éliminées, sans influencer le reste du traitement.

À chaque fichier source correspond un objet SUIF qui n'est autre que sa représentation intermédiaire. SUIF en fournit une implémentation orientée objet. Les bibliothèques définissent alors des classes permettant de représenter les différents éléments du format intermédiaire et d'effectuer diverses opérations et analyses sur ces objets. Un des intérêts de SUIF est alors le niveau d'abstraction de son format intermédiaire qui conserve toutes les informations contenues dans le code source. Que ce soit au niveau du typage des données, de leur localisation (numéro de ligne) ou encore au niveau des structures de contrôle, aucune information n'est éclipsée.

L'objet intégrant la représentation intermédiaire est la classe de base sous SUIF. De cette classe dérive plusieurs autres telle que la table des symboles qui contient les variables globales définies dans le programme (constantes, tableau, ...) ou celles associées à chacune des procédures du code source. Cette représentation intermédiaire est illustrée par la figure 3.2. Dans cette représentation, le code source (*fileset*) est représenté par une liste de fichiers dont chaque élément (*fileset entry*) pointe sur la liste des procédures qu'il intègre. Chacune de ces procédures (*tree_proc*) est alors considérée comme une somme d'instructions décrites dans une liste dont chaque élément (*tree_node*) est un arbre d'expression représentant cette instruction et pointant sur son successeur.

Il est ainsi possible d'analyser finement un programme C, via sa représentation intermédiaire, en parcourant sa structure arborescente orientée objet. En outre, SUIF met à disposition de l'utilisateur un certain nombre de méthodes permettant de mettre en œuvre des transformations sur les objets précédemment cités (suppression, insertion, ...) et par voie de

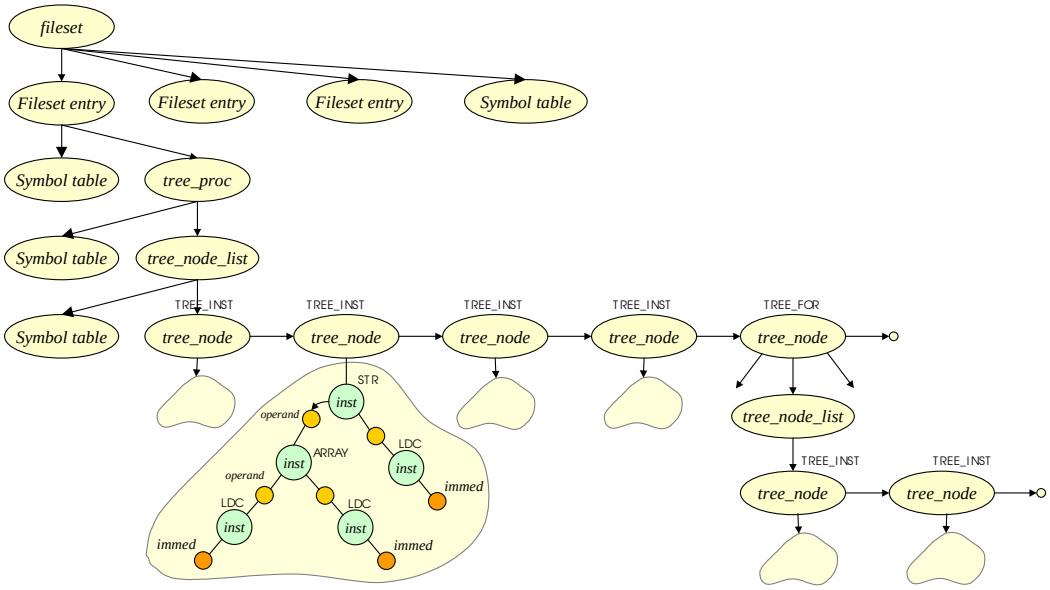

FIG. 3.2 – *Représentation interne de SUIF. Le code source (fileset) est représenté par une liste de fichiers dont chaque élément (fileset entry) pointe sur la liste des procédures qu'il intègre. Chaque procédure (tree_proc) est considérée à son tour comme une liste d'instructions dont chaque élément (tree_node) est un arbre d'expression.*

conséquence, sur le code source. Suite à ces transformations et/ou ces analyses, appliquées sur la représentation intermédiaire, une passe de SUIF permet finalement de sauvegarder les résultats sous la forme d'un nouveau code C.

3.2.2 L'extraction de code

La principale fonction du *front-end* SUIF est de réaliser le partitionnement évoqué dans l'introduction de ce chapitre. Dans le flot de développement de DART, ce partitionnement est automatique. La description C de l'application doit ainsi être analysée afin de déterminer les portions de code critiques en terme de temps d'exécution. Les informations générées par cette étape, dite de profiling, sont transmises à SUIF sous la forme d'annotations du code source. En fonction de ces annotations, une passe SUIF extrait les traitements réguliers du code dans de nouveaux fichiers C. À chaque calcul intensif est associé un fichier particulier, avec le prototype d'une fonction *main*, les déclarations de variables, ... indispensables à sa validité syntaxique.

Par soucis de réduction de l'effort de conception, seule une version simplifiée de cette approche a été développée. Dans l'état actuel de notre outil, seuls les cœurs de boucles sont considérés comme des zones de code régulières. L'intérêt principal de cette limitation est de simplifier l'analyse du code permettant d'identifier les traitements critiques. Afin d'identifier

les cœurs de boucle, une passe spécifique a été développée sous SUIF [117]. Celle-ci consiste à parcourir les *tree_ nodes* de la représentation intermédiaire de SUIF et de rechercher ceux correspondants à des boucles *FOR*. Lorsqu'une boucle FOR est trouvée, le cœur de cette boucle est recopié dans un nouveau fichier et remplacé dans la représentation intermédiaire du code source par une succession de lectures et d'écritures correspondant aux accès mémoires qui seront réalisés durant l'exécution de cette boucle (Fig. 3.3). Dans le cas de boucles imbriquées, seule la boucle de plus bas niveau sera candidate à une implémentation HW.

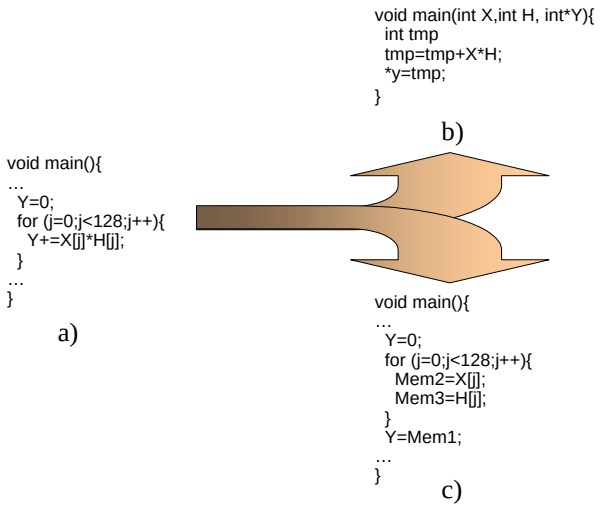

FIG. 3.3 – *Extraction des cœurs de boucle. À partir d'un fichier C décrivant l'application (a), deux fichiers sont générés. Le premier (b) contient le motif de calcul caractérisant la boucle extraite. Le second (c), contient tous les traitements irréguliers et les manipulations de données. Dans ce fichier, la boucle ne contient désormais plus que les accès aux variables.*

La nouvelle représentation SUIF du code source ne respecte donc plus la fonctionnalité de l'application. Le code résultant de cette transformation n'étant pas utilisé dans la suite de notre flot de conception à des fins de simulation, cette caractéristique est néanmoins sans effets.

En sortie de cette passe d'extraction, si une boucle doit être extraite, deux fichiers sont générés. Le premier contient le cœur de boucle devant être implémenté sous la forme d'une reconfiguration matérielle. Il sera transmis au module gDART et ne contient que les informations décrivant le motif de calcul. Toutes les informations relatives au contrôle de la boucle ou à la gestion des données y sont absentes. Ces dernières sont réunies au sein du second fichier généré par cette passe. Celui-ci est issu de la modification du fichier source. Il contient l'ensemble des traitements irréguliers (i.e. tous les traitements qui n'ont pas été extraits avec le cœur de boucle), ainsi que les informations de contrôle. Les manipulations de données réalisées au sein de la boucle apparaissent désormais explicitement, sans faire intervenir les opérations arithmétiques sous-jacentes.

Lorsque N boucles sont candidates à une implémentation HW, $N+1$ fichiers sont générés. À chaque cœur de boucle est associé un fichier décrivant son motif de calcul. Le dernier fichier contient quant à lui les informations de contrôle, les traitements irréguliers et les manipulations de données relatives à l'ensemble de l'application. C'est par le biais de ce fichier que pourront, en sortie des modules de génération de code, être synchronisées les différentes configurations (SW ou HW) des DPRs, avec les générateurs d'adresses.

3.2.3 Le déroulage de boucle

Par l'extraction des cœurs de boucle, l'essentiel des informations relatives au parallélisme du calcul sont supprimées. En effet, les différents fichiers issus de l'extraction des traitements réguliers ne contiennent que les motif de calcul utilisés dans les boucles. Ainsi, l'outil gDART qui est en charge de transformer ces motifs de calculs en configurations de DPRs n'a aucune visibilité sur le nombre d'itérations de la boucle et donc sur son degré de parallélisme d'opérations. Ainsi, dans l'exemple précédent (Fig. 3.3 b), gDART ne manipule qu'une seule multiplication-ACcumulation. Ce module est incapable de déterminer qu'il est possible d'en réaliser jusqu'à 128 en parallèle. Pour que cette implémentation soit efficace il est donc nécessaire d'extraire le parallélisme intrinsèque de cette boucle. Ceci est réalisé en la déroulant partiellement. Pour illustrer cette technique nous pouvons reprendre l'exemple du filtrage numérique précédent :

```
for (i=0;i<128;i++){
        tmp = tmp + x[i]*h[i];
}
```

LIST. 3.1 – *Boucle principale d'une description de filtre FIR sur 128 points.*

Si nous appliquons un facteur de déroulage de 2, nous obtiendrons la boucle transformée suivante :

```
for (i=0;i<128;i+=2){
        tmp = tmp + x[i]*h[i];
        tmp = tmp + x[i+1]*h[i+1];
}
```

LIST. 3.2 – *Boucle principale d'une description de filtre FIR sur 128 points déroulée 2 fois.*

Ainsi, par ce déroulage d'ordre 2, nous avons augmenté le niveau de parallélisme du cœur de boucle transmis à gDART puisque cette fois, 2 Multiplications-ACcumulation pourront être exécutées simultanément. Ce principe peut bien entendu être étendu pour envisager des facteurs de déroulage plus élevés, qui exhibent des niveaux de parallélisme plus importants. L'ordre du déroulage dépend du nombre de DPRs alloués pour exécuter le traitement. La quantité de DPRs exprime en effet le nombre maximal d'opérations pouvant être traitées en parallèle pour une configuration donnée. Cette quantité est spécifiée par l'utilisateur. Dans l'état actuel de la chaîne de développement, aucun outil ne vient simplifier la tâche de l'utilisateur pour cette prise de décision. Il apparaît donc ici une perspective de recherche intéressante, à savoir la conception d'un outil déterminant le nombre de DPRs qui se traduira par une implémentation optimale de l'application, suivant un critère énergie ou de performance.

Ce déroulage est implémenté sous SUIF en deux phases. La première consiste à dupliquer les instructions présentes au sein de cette boucle. La seconde vise à modifier les indices utilisés dans ces instructions dès lors qu'ils dépendent du compteur de boucle (e.g. i dans l'exemple précédent). À chaque duplication du cœur de boucle initial, l'indice est incrémenté de 1. Le pas de la boucle doit être incrémenté de la même façon. Pour que ce déroulage soit viable,

la borne supérieure de la boucle doit être une valeur numérique pouvant être déterminée au moment de la compilation (i.e. indépendante de la valeur des entrées).

Si dans l'exemple précédent, le déroulage partiel est un traitement trivial, il se complique cependant lorsque l'ordre du déroulage n'est pas un multiple de la borne supérieure. Ainsi, si nous souhaitons réaliser un déroulage partiel d'ordre 3 de la boucle principale du filtre FIR précédemment évoqué, nous devons obtenir le résultat présenté dans le listing 3.3.

```
for (i=0 ;i<126 ;i+=3){
        tmp = tmp + x[i]*h[i] ;
        tmp = tmp + x[i+1]*h[i+1] ;
        tmp = tmp + x[i+2]*h[i+2] ;
}
tmp = tmp + x[126]*h[126] ;
tmp = tmp + x[127]*h[127] ;
```

LIST. 3.3 – *Boucle principale d'une description de filtre FIR sur 128 points déroulée 3 fois.*

Outre la duplication de l'ensemble des instructions du cœur de la boucle il est désormais nécessaire de gérer l'ajout d'instructions supplémentaires, ici au nombre de 2, de manière à maintenir la fonctionnalité de la boucle. Par ailleurs, les bornes de la boucle devront être modifiées afin de tenir compte des instructions qui seront exécutées en dehors de celle-ci. Cette spécificité peut être généralisée en considérant les variables suivantes :

- F : Facteur de déroulage ;
- BS : Borne Supérieure ;
- BI : Borne Inférieure ;
- $STEP$: Pas de la boucle ;
- A : Nombre d'instructions à rajouter ;
- NI : Nombre d'Itérations ;
- NBS : Nouvelle Borne Supérieure ;
- NBI : Nouvelle Borne Inférieure ;
- $NSTEP$: Nouveau Pas de la boucle.

Le nombre d'itérations de la boucle, NI, devient $\frac{(BS-BI)}{STEP}$. Si NI est un multiple de F alors il n'est pas nécessaire d'ajouter des instructions ($A=0$). Dans le cas contraire, il faut diviser NI par F afin d'obtenir Q (quotient entier). La nouvelle borne supérieure, NBS, est alors égale au produit de Q et de F. Enfin, pour obtenir le nombre d'instructions à rajouter, on effectue la soustraction de NBI par NBS. Le pas de la boucle doit alors être modifié de manière à obtenir $NSTEP$ par le produit de $STEP$ et de F.

3.2.4 Les transformations de code non développées

Les optimisations de boucle

Le *front-end* de notre flot de développement peut également être exploité afin de réaliser des transformations de haut niveau sur le code source de manière à l'optimiser suivant des critères d'occupation mémoire, de performance ou encore de consommation. En particulier, dans le cadre de la thèse d'Antoine Fraboulet [118], un certain nombre de transformations de code ont été développées afin de réduire la consommation des systèmes embarqués. L'approche utilisée

consiste à minimiser le nombre d'accès aux mémoires de masse en augmentant la localité temporelle des données. Par localité temporelle, nous entendons ici la propriété caractérisant le temps séparant deux accès consécutifs à un même emplacement mémoire. Elle peut être améliorée en utilisant diverses techniques telles que celles représentées sur la figure 3.4 [119, 120, 121]. Par le biais d'une collaboration, ces techniques sont en cours d'intégration dans notre *front-end*.

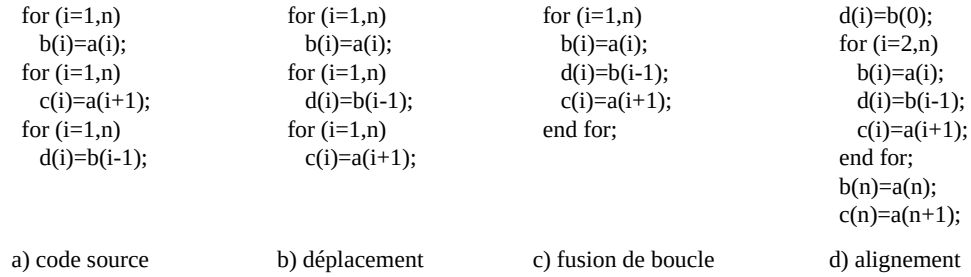

FIG. 3.4 – *Illustration de méthodes d'optimisation de boucle permettant d'augmenter la localité temporelle des données. La méthode de déplacement (b) vise à rapprocher les dates de production et d'utilisation des variables afin d'éviter le stockage de données intermédiaires en mémoire. Cet objectif est également ciblé par la technique de fusion de boucle (c) qui réunit plusieurs boucles dans un même espace d'itérations. L'alignement de boucle (d) minimise finalement le nombre d'accès mémoire en uniformisant les indices de boucle.*

Création d'une chaîne de retard

SUIF peut finalement être utilisé à des fins de pré-traitement. En effet, certaines informations ne peuvent être obtenues qu'à la suite d'une analyse fine de l'application. Compte tenu de la richesse de la représentation intermédiaire de SUIF, cet environnement est parfaitement adapté à cet effet. En particulier, l'extraction des informations de retard est problématique à partir de la seule étude du cœur de boucle. De ce fait, la création d'une chaîne de retard, à partir des registres insérés dans les chemins de données de DART, ne peut être faite que par le biais d'une intervention manuelle de l'utilisateur. À l'heure actuelle, le programmeur est en charge de spécifier si deux échantillons d'un vecteur peuvent être obtenus par un accès mémoire unique. Ceci se fait via un appel à une fonction *RETARD()*.

Ce mécanisme de création de chaîne de retard est illustré par la figure 3.5. La contrainte d'écriture est dès lors relativement importante et il peut être souhaitable de s'en affranchir. Pour cela, l'analyse du code doit permettre d'identifier la redondance des accès à un vecteur, et traduire cette redondance par l'insertion de fonctions *RETARD()* dans le programme. L'analyse du code source, par une passe SUIF, doit ainsi déterminer que dans un cœur de boucle, *X(n)* et *X(n-k)* sont des accès redondants, dès lors que :

- la distance k séparant ces deux accès au vecteur est fixe ;
- le vecteur X est parcouru sur toute sa longueur et d'une manière linéaire. En d'autres termes, l'adresse des échantillons lus dépend du compteur de boucle et la mise à jour de ce dernier se fait par un incrément positif constant.

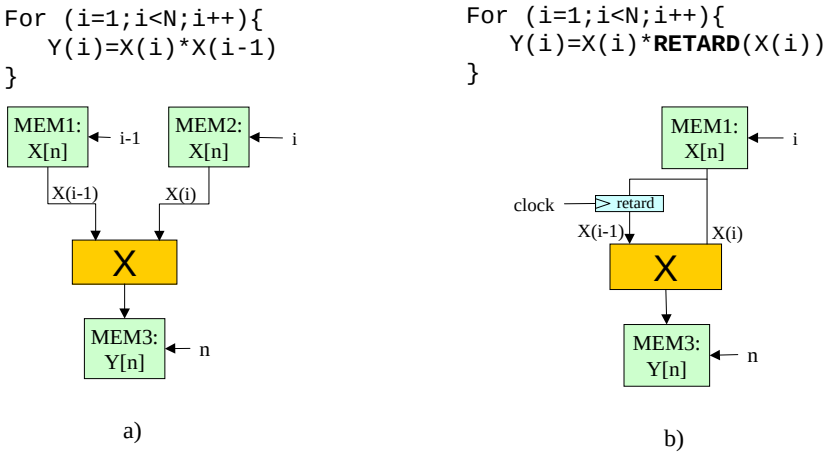

FIG. 3.5 – *Illustration de la méthode de réduction du nombre d'accès mémoire par le biais de l'appel à la fonction RETARD(). Le code source (a) ne donnant aucune indication quant à la possibilité de créer une chaîne de retard, il conduit à la définition d'une architecture dans laquelle le vecteur X est dupliqué dans deux mémoires. Par l'appel à la fonction RETARD() le code (b) permet de réduire le nombre de mémoires utilisées en déduisant la valeur de l'échantillon X(n-1) de la valeur de l'échantillon X(n) au cycle précédent.*

3.3 Génération des configurations HW : gDART

L'outil gDART est le premier des trois modules en charge de générer des codes binaires exécutables sur DART. Son point d'entrée correspond aux fichiers C intégrant les cœurs de boucle extraits par le *front-end* SUIF. Ce module se base sur l'infrastructure de l'outil de synthèse comportementale BSS (Breizh Synthesis System) [114], développé au laboratoire, pour transformer le code C d'entrée en un graphe flot de données (DFG : Data Flow Graph). Puisque l'extraction du parallélisme a été réalisée durant les transformations SUIF, la tâche de ce module se limite à déterminer la structure du chemin de données qui implémentera au mieux ce DFG, et à transformer cette structure en une configuration matérielle des DPRs.

3.3.1 Réduction de boucle

La tâche est cependant plus ardue qu'il n'y parait puisque malgré le très fort degré de flexibilité de DART, cette architecture souffre de quelques limitations. En particulier, la principale contrainte lorsqu'il s'agit d'ordonnancer un graphe flot de données en vue de l'implémenter sur DART est qu'il est préjudiciable de maintenir des résultats intermédiaires sur plus d'un cycle. Cette limitation, motivée par la nécessité de consommer peu d'énergie, s'explique par l'absence de files de registres au sein des DPRs. Maintenir une donnée intermédiaire dans de telles conditions nécessite de la stocker dans une mémoire locale. Ceci se traduit donc par un gaspillage d'énergie nuisible à l'efficacité énergétique de DART.

Dans le cas particulier d'un filtre FIR, partiellement déroulé d'un facteur 4, ce problème apparaît clairement au niveau de l'accumulation dans la figure 3.6, représentant son DFG après un ordonnancement au plus tôt (ASAP : As Soon As Possible). Le rebouclage de la variable d'accumulation crée en effet ici une boucle, dite critique, d'une latence de 4 cycles. Le problème se résout cependant ici de manière relativement simple puisque la permutation

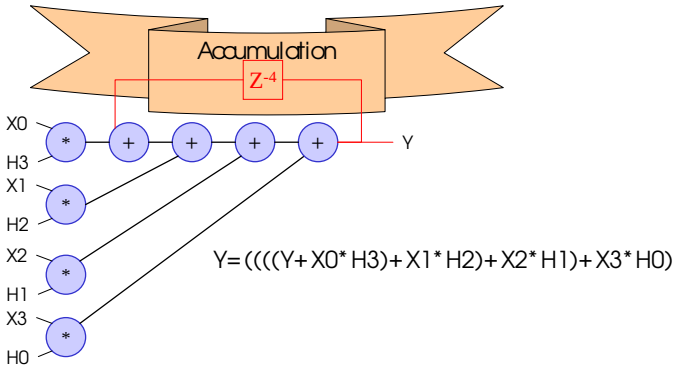

FIG. 3.6 – *DFG d'un filtre FIR déroulé 4 fois. Le rebouclage de la variable d'accumulation entre les étages 5 et 2 du pipeline d'exécution modélise une boucle critique d'une latence de 4 cycles. Un stockage de cette donnée dans une mémoire locale est dans ce cas indispensable.*

des additions permet de réduire cette latence à un cycle. Le résultat de la réduction de cette boucle critique correspond alors à la figure 3.7.

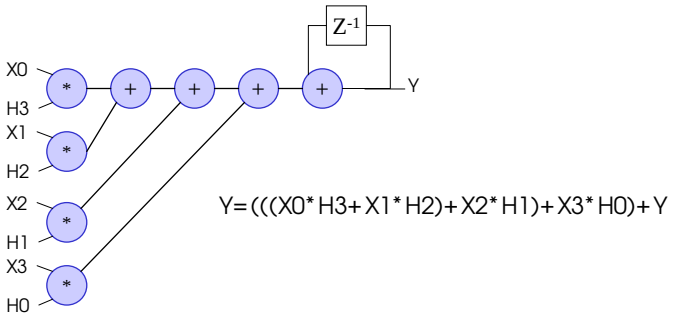

FIG. 3.7 – *DFG d'un filtre FIR déroulé 4 fois après la réduction de la boucle critique. Suite à la permutation des opérations d'addition, la latence de la boucle critique est réduite à un cycle.*

Cette méthode peut se généraliser en cassant les boucles critiques par permutation d'opérateurs [122]. Ces permutations doivent cependant garantir le maintien de la fonctionnalité du DFG. Il faut donc respecter les règles d'associativité et de commutativité relatives aux opérateurs arithmétiques :

1. l'addition et la soustraction sont associatives ;
2. la multiplication est associative et distributive.

Certaines boucles critiques ne peuvent en revanche pas être réduites, sous peine de modifier la fonctionnalité du graphe. Ce cas de figure peut être illustré par la figure 3.9 dans laquelle la permutation des opérateurs ne réduit pas la latence de la boucle critique. Dès lors, l'implémentation de tels DFGs implique nécessairement l'utilisation des mémoires locales pour stocker les résultats intermédiaires. Si la latence de la boucle critique est faible, ce problème peut également être résolu en utilisant les registres intégrés dans le chemin de données de DART.

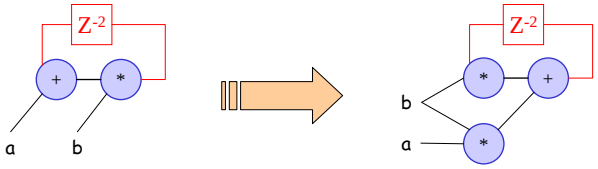

FIG. 3.8 – *Exemple d'échec de réduction de boucle critique. L'exploitation de la propriété de distributivité associée à l'opération de multiplication ne permet pas de réduire la latence de la boucle critique.*

3.3.2 Réduction de la profondeur du graphe

Bien que le DGF obtenu suite à la passe de réduction de boucle soit implémentable sur DART, la figure 3.7 fait apparaître une latence du pipeline d'exécution relativement importante. Dès lors, dans le cadre de boucles imbriquées, si le nombre d'itérations de la boucle de plus bas niveau est faible, cette caractéristique peu devenir extrêmement préjudiciable. En effet, sur l'exemple précédent, la figure 3.7 montre que 5 cycles sont nécessaires à l'amorçage du pipeline d'exécution. Si ce DFG correspond au cœur du filtrage 16 points (partiellement déroulé 4 fois) du filtre adaptatif LMS (Least Mean Square) décrit dans le listing 3.4, le temps d'exécution du traitement complet sera consommé à 64% par les amorçages successifs des pipelines d'exécution de la mise à jour des coefficients puis du filtrage.

```
for (i=0 ; i<1024 ; i++){
        for (j=0 ; j<16 ;i++) {          //filtrage
                y[i]+=x[j]*h[15-j]
        }
        adapt := cte * (yd - y[i]) ;     //mise à jour
        for (k=0 ; k<16 ; k++){
                h[k] := h[k] + adapt * x[N-k-1] ;
        }
}
```

LIST. 3.4 – *Algorithme de filtrage adaptatif LMS.*

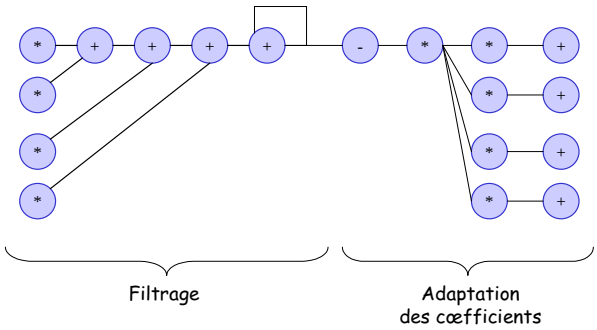

FIG. 3.9 – *DFG d'un filtrage adaptatif LMS.*

Ces 64% de temps consommés pour les amorçages de pipeline motivent alors la poursuite

de l'effort d'optimisation du DFG. En particulier, une passe de parallélisation du DFG a été développée dans le but d'en réduire la profondeur. Cette parallélisation est basée sur les propriétés de commutativité propres aux opérations d'additions et de multiplications. L'algorithme est centré sur une fonction de recherche de suites d'additions ou de multiplications exécutées séquentiellement et candidates à la parallélisation. Les candidatures sont ensuite évaluées suivant des critères de dates d'exécution. Une fonction de parallélisation dont le fonctionnement est illustré par la figure 3.10 est alors appliquée à la séquence d'opérations.

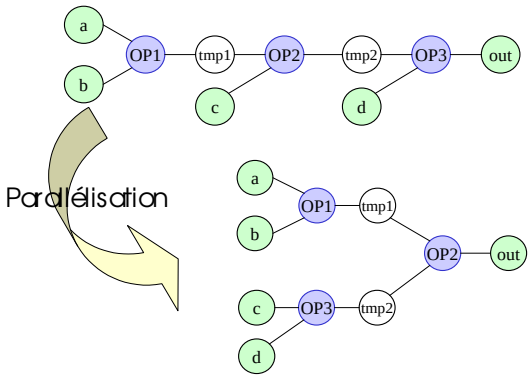

FIG. 3.10 – *Illustration de l'algorithme de parallélisation de séquences d'opérations. Celui-ci se base sur l'égalité* $((a \dot{+} b) \dot{+} c) \dot{+} d = (a \dot{+} b) \dot{+} (c \dot{+} d)$*, caractérisant les opérations commutatives.*

Cette parallélisation, autrement connue sous le nom de *tree height reduction* [122], génère pour l'exemple précédent le DFG représenté sur la figure 3.11. Le nombre de cycles nécessaires à l'amorçage de son pipeline d'exécution a été réduit à 4 cycles. Le temps d'exécution n'est plus proportionnel au nombre d'accumulations mais à son logarithme en base 2.

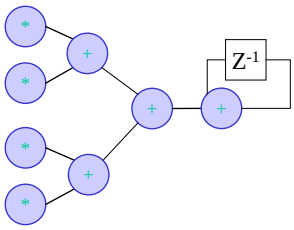

FIG. 3.11 – *DFG du filtre FIR, partiellement déroulé 4 fois, après optimisation. La réduction de la profondeur du graphe, obtenue en exécutant en parallèle plusieurs additions, permet de limiter le coût de son amorçage.*

Le DFG ainsi optimisé en vue de son implémentation sur DART, doit ensuite être transformé en une configuration matérielle. Pour cela, les opérations doivent être assignées aux opérateurs et les entrées/sorties doivent être assignées à des mémoires. À ce stade de la génération de la configuration, le DFG peut être observé suivant différents formats. L'outil développé dispose en effet d'une sortie textuelle et de deux sorties graphiques. Un premier format de représentation graphique du DFG peut être visualisé par le biais d'un outil développé dans le cadre de BSS (VisuGF). Une sortie plus standard permettant d'utiliser l'outil libre XVCG [123] est en outre disponible (Fig. 3.12). La version textuelle du DFG optimisé est quant à elle réservée à des fins de déboggage.

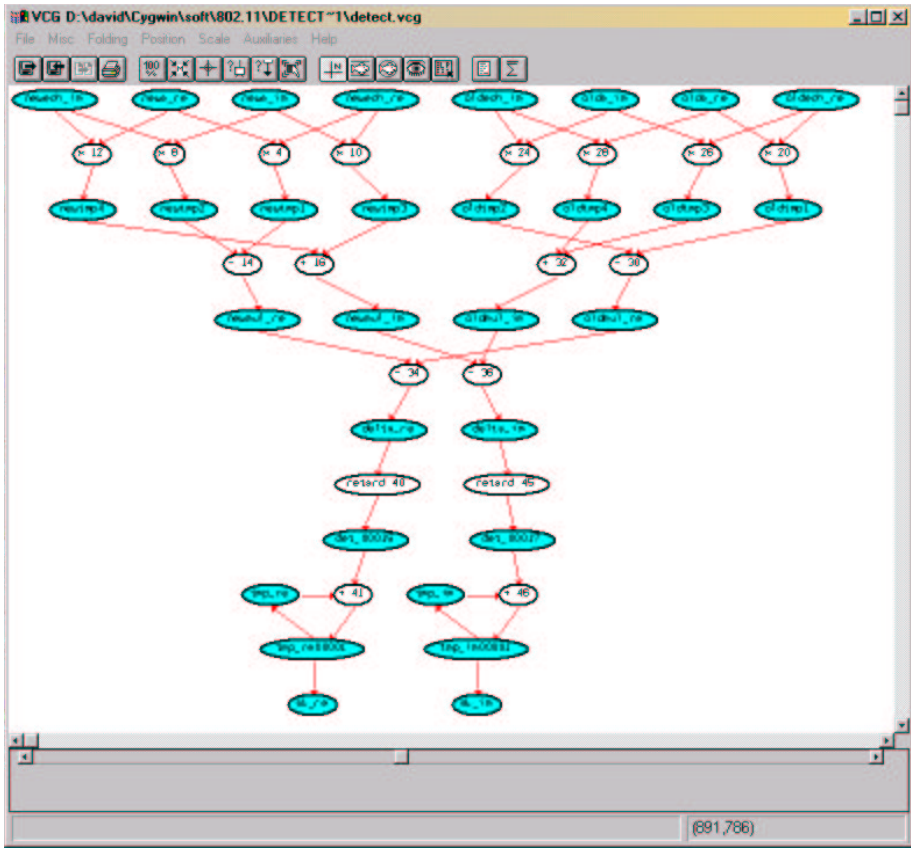

FIG. 3.12 – *Capture d'écran de l'outil Xvcg : Le DFG visualisé correspond à l'algorithme de synchronisation de la norme 802.11a, après sa transformation par gDART.*

3.3.3 L'allocation mémoire

Suite à l'optimisation du DFG, les opérations le constituant doivent être assignées à des opérateurs, et les variables doivent être assignées aux mémoires qui les stockent. La principale contrainte devant être supportée par la phase d'allocation mémoire est l'interopérabilité des configurations. Celle-ci ne peut être garantie que dans la mesure où toutes les phases du traitement manipulent les mêmes données. En effet, les différentes configurations constituant un algorithme communiquent exclusivement par l'intermédiaire des données. Il est donc nécessaire que leurs producteurs et leurs consommateurs considèrent les mêmes emplacements mémoire. Deux cas de figures peuvent être distingués selon la durée de vie d'une variable, i.e. le temps séparant sa date de production de sa date de consommation.

- Si la durée de vie d'une variable est supérieure au seuil de localité, celle-ci sera stockée dans la mémoire de données du *cluster* ;
- Si la durée de vie d'une variable est inférieure au seuil de localité, celle-ci sera stockée dans une des mémoires de données locale aux DPRs.

La notion de seuil de localité représente ici le temps maximum pendant lequel une donnée peut être stockée dans les mémoires locales. Idéalement, ce temps tend vers l'infini et toutes les variables sont stockées au plus près des opérateurs. La taille réduite des mémoires internes aux DPRs interdit cependant ce type d'approche et implique le stockage de certaines données

dans la mémoire du *cluster*. En pratique, ce temps n'a pas de signification physique, et la décision de maintenir une donnée dans une mémoire locale ou de la "rapatrier" dans la mémoire du *cluster* est prise par l'outil ACG, examiné dans la section 3.4.4, qui gère les accès aux mémoires. Les informations relatives au placement des données en mémoire transitent entre les outils gDART et ACG par le biais d'un tableau de correspondance tel que le tableau 3.1.

Variable	**Mémoire**
X_0	0
H_0	1
X_1	2
⋮	⋮
H_3	7
Y	0

TAB. 3.1 – *Exemple de tableau de correspondance. À chaque variable est associé le numéro de la mémoire qui la stocke. Il n'est pas fait ici de distinction entre les données lues et les données écrites.*

Les variables stockées dans la mémoire du *cluster* n'apparaissent pas dans ce tableau de correspondance. Pour de telles variables, l'outil d'assignation a donc une totale autonomie pour décider de l'allocation mémoire la plus pertinente. Dans ce cas, c'est donc gDART qui décidera de la mémoire locale stockant une variable donnée. L'outil ACG devra alors faire en sorte de transférer cette donnée dans la bonne mémoire avant le début du calcul. En revanche, si la variable est déjà en mémoire locale, l'outil gDART doit tenir compte de ce placement des données et réaliser son assignation en conséquence.

3.3.4 L'assignation des opérateurs

L'assignation consiste à affecter chaque nœud du DFG à un opérateur de DART. La méthode d'assignation diffère selon que les données manipulées sont présentes dans la mémoire du *cluster* ou dans les mémoires locales. En conséquence, deux algorithmes d'assignation ont été développés.

Assignation directe

Le premier mode d'assignation est adapté au cas où les données sont stockées dans la mémoire du *cluster*. Il n'y a dans ce cas pas de contraintes sur le placement des données et gDART décide seul du placement le plus avantageux. Le déroulage des boucles ayant été fait en tenant compte de la quantité de DPRs alloués, le nombre d'opérateurs disponibles est nécessairement suffisant à l'implémentation du cœur de boucle. La seule difficulté devant être surmontée concerne donc le nombre limité de bus globaux, qui impose une minimisation des communications inter-DPR.

Afin de limiter ces communications inter-DPR, l'assignation en mode direct tente d'assigner chacune des opérations sur un opérateur situé au plus près de l'opérateur qui consommera son résultat. Ceci implique donc de décider a priori d'un placement en mémoire des sorties de la boucle puis de parcourir le graphe depuis ses sorties jusqu'à ses entrées. Une fois les opérations assignées aux opérateurs, et les variables assignées aux mémoires, suivant un schéma décrit par le listing 3.5, un second parcours du DFG permet alors de générer la configuration du

chemin de données. Il faut pour cela déterminer les adresses des multiplexeurs, allouer des bus globaux aux communications inter-DPRs, et spécifier les opérations aux opérateurs tout en validant la garde des UFs, des registres et des AGs utilisées. Le résultat du placement des données en mémoire doit par ailleurs apparaître dans le tableau de correspondance afin d'être pris en compte lors des générations de configuration futures.

```
Fonction AssignationDirecte(*NoeudConfig s)
        Si s==NoeudDeb Alors
                return OK;
        Fsi;
        Si Suivant(s)==NoeudFin Alors
                s->NumOp=PremMemLibre();
                s->NumDPR=s->NumOp/4;
        Sinon
                DPRSuivant=Suivant(s)->NumDPR;
                Si OpLibre[TypeNoeud(s)][DPRSuivant]==TRUE;
                        Affecter(s,DPRSuivant);
                Sinon
                        j=0;
                        Affecte=FALSE;
                        Tant Que j<NbDPR et non Affecte Faire
                                Si OpLibre[TypeNoeud(s)][j]==TRUE;
                                        Affecter(s,j);
                                        Affecte=TRUE;
                                Sinon
                                        j++;
                                Fsi;
                        Fait;
                Fsi;
        Fsi;
        Precedent=s->DebPred;
        Tant Que Precedent!=NULL Faire
                AssignationDirecte(Precedent->s);
                Precedent=Precedent->Suivant;
        Fait;
```

LIST. 3.5 – *Algorithme d'assignation en mode direct. Cette fonction est appelée depuis la fonction main avec le nœud fin en paramètre (i.e. le nœud dont la date ASAP est la plus grande). La fonction Suivant(*NoeudConfig s) renvoie le nœud de configuration suivant le nœud courant s. La fonction PremMemLibre() renvoie le numéro de la première mémoire accessible. La fonction TypeNoeud(*NoeudConfig s) renvoie le type d'opération représenté par le nœud de configuration s. L'examen du tableau à deux dimensions OpLibre[][][] permet de déterminer pour chaque opérateur s'il peut être assigné ou non. La fonction Affecter(*NoeudConfig s,int i) assigne l'opération s au premier opérateur libre du DPR i.*

Assignation sous contrainte de placement mémoire

La méthode d'assignation diffère sensiblement lorsque les données ont préalablement été placées en mémoires. En effet, le module d'assignation n'a cette fois plus une totale liberté pour définir une configuration minimisant les communications inter-DPR. Il est donc nécessaire de placer les opérateurs au plus près de ceux ayant générés les données qu'ils consomment.

L'algorithme développé décompose le DFG de l'application en étapes de calcul. Chaque étape de calcul représente ainsi un cycle d'exécution de ce DFG (Fig. 3.13).

L'algorithme d'assignation, décrit par le listing 3.6, vise à assigner simultanément un opérateur ainsi que ceux ayant générés les entrées qu'il manipule. Cette approche permet par exemple de garantir que les multiplieurs apparaissant sur le DFG de la figure 3.13 seront assignés à des ressources appartenant au même DPR que les mémoires stockant les données X et H qu'ils manipulent. Ainsi, cet algorithme assure que la régularité de l'algorithme sera retranscrite au niveau de l'architecture générée dès lors que le placement des données en mémoire est judicieux. Par l'emploi de cette méthode d'assignation, il est ainsi possible d'influencer la qualité, i.e. la régularité, de l'architecture générée. Elle affiche en revanche ces limites lorsque les données manipulées par un opérateur sont stockées dans des DPRs différents, i.e. lorsque le placement des données est de mauvaise qualité.

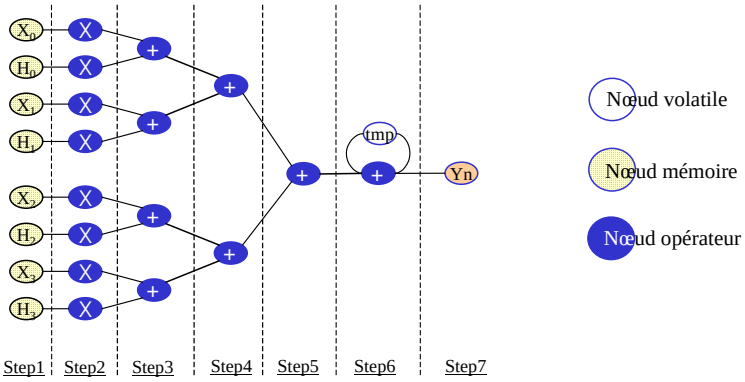

FIG. 3.13 – *Décomposition du DFG d'un filtre FIR en pas de calcul. Un étage de calcul réunit tous les accès mémoire ou les opérations registre-registre réalisés dans un même temps de cycle.*

La phase d'assignation a principalement pour but de mettre à jour la table de correspondance et de générer le flot d'instructions permettant de spécifier la configuration matérielle des chemins de données. Cet outil est également en mesure de fournir une description de la configuration de l'architecture dans un format texte. Cette description peut alors être parsée par l'outil *DARTDesigner* afin de fournir une représentation graphique de l'architecture (Fig. 3.14). Celle-ci peut être analysée par l'utilisateur à des fins de deboggage ou d'optimisation.

3.3.5 Contraintes d'écritures sur le code source et limitations temporaires de l'outil

Compte tenu de l'état actuel d'avancement de gDART, son utilisation nécessite un investissement de la part de l'utilisateur. En particulier, ce dernier doit respecter les contraintes suivantes lorsqu'il décrit une application devant être exécutée sur DART.

Spécification des déroulages de boucle Notre chaîne de développement ne disposant pour le moment pas d'outils permettant d'explorer divers compromis entre parallélisme de tâches et parallélisme d'instructions, ce dernier doit être fixé, pour chaque configuration, par l'utilisateur. De ce fait, chaque boucle, devant être traduite en configuration matérielle, doit être accompagnée d'une directive précisant le parallélisme d'instructions

```
Procédure AssignationPasAPas()
        ListeFuturs[0]=NoeudDeb ;
        Tant Que ListeFuturs[0] !=NoeudFin Faire
                ListeCourant=ListeFuturs ;
                i=0 ;
                Tant Que ListeCourant[i] !=NULL Faire
                        LastNoeudSuivant=0 ;
                        Assignation(ListeCourant[i]) ;
                        M=ListeCourant[i]->DebSuiv ;
                        Pour tous les noeuds suivant ListeCourant[i] Faire
                                Assignation(M) ;
                                ListeFuturs[DernierNoeudSuivant]=M ;
                                M=M->Suivant ;
                                DernierNoeudSuivant++ ;
                        Fait ;
                        i++ ;
                Fait ;
        Fait ;
```

LIST. 3.6 – *Parcours du DFG dans l'algorithme d'assignation sous contrainte de placement mémoire. La fonction d'assignation est comparable à celle utilisée en mode direct. Il s'agit ici, lorsque le nœud passé en paramètre n'a pas déjà été assigné, de lui assigner un opérateur ou une mémoire, puis d'assigner ses prédécesseurs au plus près. Les nœuds de configuration du DFG sont parcourus en suivant les étages de calcul du DFG.*

devant être exhibé. Cette directive prend la forme suivante : *//Unroll n*, où n représente l'ordre du déroulage. Elle doit être placée sur la même ligne que l'instantiation de la boucle pour être extraite par une routine écrite en *PeRL*, afin de paramétrer la passe SUIF qui réalise le déroulage de boucle.

Utilisation d'instructions de retard Cette limitation, déjà évoquée dans la section 3.2.4, est inhérente à l'absence d'outil permettant d'identifier des retards dans les description C de l'application. Elle se traduit par la nécessité de préciser explicitement que la donnée $x(n-1)$ est équivalente à la donnée x(n) au cycle précédent. Elle s'exprime par l'instruction $RETARD(x(n))$.

Exploitation manuelle des capacités SWP L'outil gDART se base sur l'infrastructure de synthèse comportementale BSS. gDART exploite en particulier la partie frontale de cet environnement pour transformer des codes C ou VHDL en graphes flots de données. Cette partie frontale est pour le moment incapable de distinguer une variable de type *short* d'une variable de type *float*. De ce fait, le DFG manipulé par gDART ne dispose d'aucune information quant à la taille des données manipulées. En conséquence, les capacités SWP, ne peuvent pour le moment être exploitées qu'à la condition que l'utilisateur le spécifie manuellement. Cette limitation temporaire de l'outil impose aujourd'hui à l'utilisateur de DART, dans un contexte SWP, de transformer manuellement les instructions de configuration, en modifiant les champs concernant les traitements SWP.

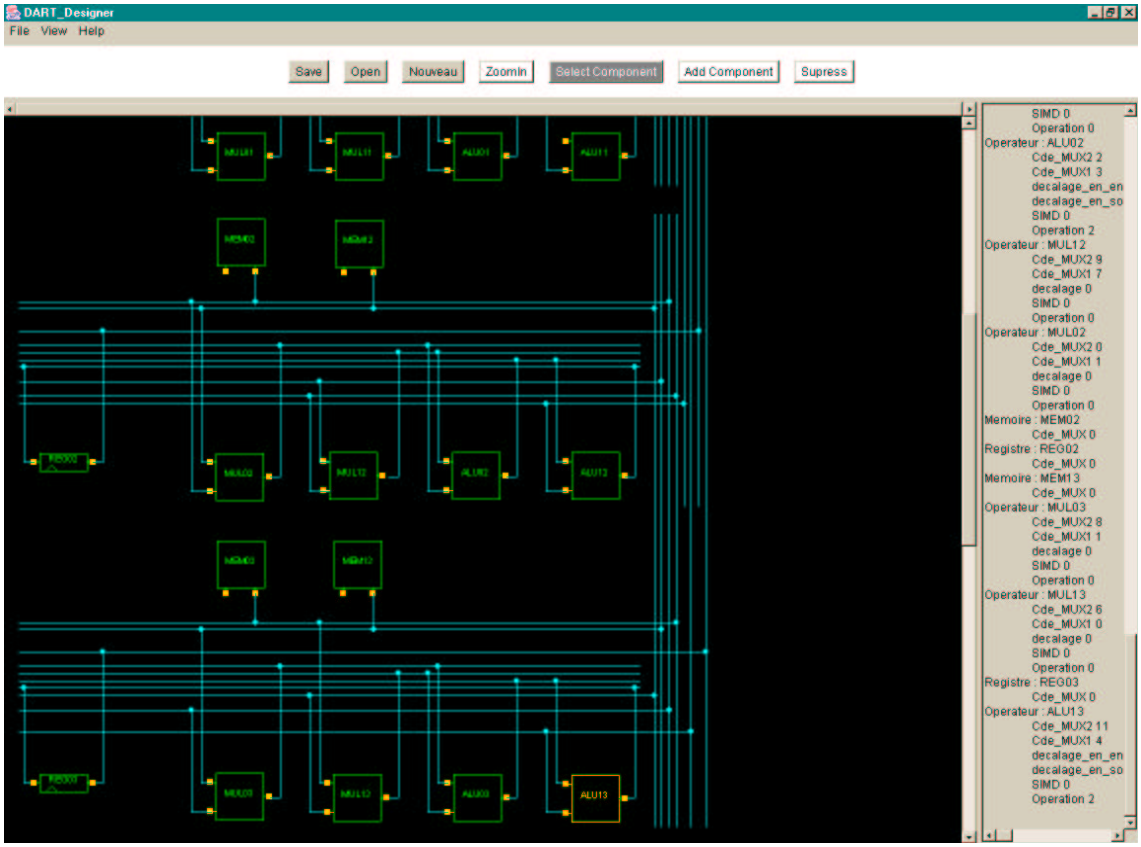

FIG. 3.14 – *Capture d'écran de* DARTDesigner. *La configuration, partiellement représentée, correspond au traitement de la synchronisation de la norme 802.11a. La partie droite de l'outil fait apparaître la configuration des composants sélectionnés par l'utilisateur. Ce dernier peut ainsi vérifier le bon fonctionnement de l'architecture générée.*

3.4 Les outils de compilation

Si les reconfigurations HW sont générées via un outil dédié à DART, et intégrant des passes d'optimisation spécifiques à cette architecture, les configurations SW et les instructions de génération d'adresses sont quant à elles générées via des passes de compilation beaucoup plus traditionnelles. Les outils utilisés pour cela sont respectivement cDART et ACG. Il sont issus de l'environnement de compilation reciblable CALIFE et présentés dans cette section.

3.4.1 Environnement de compilation reciblable CALIFE

CALIFE est une plateforme expérimentale de génération de code reciblable qui exploite les principes de génération de code paramétrée par une description de processeur [124]. Il s'agit ici au niveau du compilateur d'appliquer au code source de l'application des transformations dont les comportements sont paramétrés par les caractéristiques du processeur cible. Compte tenu de la spécificité des architectures ciblées par cette plate-forme, les ASIPs, une même séquence de transformations de code se traduira par des performances très variées selon le processeur visé. Dès lors, l'approche utilisée dans CALIFE consiste à offrir à l'utilisateur la possibilité de contrôler le choix et l'ordre des transformations appliquées sur le code source.

Pour cela, CALIFE s'appuie sur les trois points suivants :

1. un langage de modélisation de processeur ;
2. une bibliothèque d'algorithmes de transformation ;
3. une infrastructure transformant la description du processeur en paramètres de transformation de code.

La bibliothèque est constituée de modules correspondant à la génération de code et aux algorithmes plus ou moins usuels d'optimisation. Parmi ceux-ci sont notamment intégrés les passes classiques de compilation que sont la sélection de code, l'allocation de ressources, l'ordonnancement et l'optimisation de code.

Principales passes de Compilation

La sélection de code Cette étape, autrement appelée sélection d'instructions, consiste à choisir, pour chaque opération dans le programme, l'instruction qui va l'exécuter. Les techniques les plus répandues se basent pour cela sur la reconnaissance de motif. Le programme est représenté par des arbres ou des graphes, symbolisant le flot de données et les opérations effectuées. Les instructions sont représentées suivant le même principe tout en se voyant affecter une fonction de coût. L'algorithme de sélection d'instructions recherche alors la séquence d'instructions qui permettra de reconstituer le programme pour un moindre coût (Fig. 3.15 [125]). Cette méthode, appelée couverture d'arbre, peut également être mise en œuvre par programmation dynamique, i.e. le problème de la sélection de code optimal pour l'arbre représentant le code source est découpé en sous-problèmes optimaux de sélection de codes pour des sous-arbres du code source.

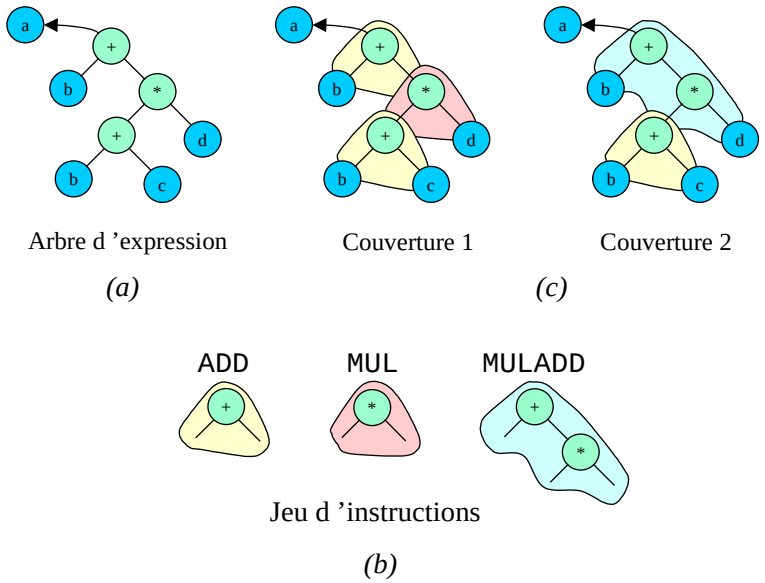

FIG. 3.15 – *Illustration des mécanismes de sélection d'instructions par couverture d'arbre. Le code source (a), représenté par un arbre d'expression, est parcouru en recherchant des motifs (c) correspondant aux instructions supportées par le processeur cible (b). La sélection entre les éventuels candidats à l'implémentation d'un motif vise alors à déterminer l'instruction minimisant le coût global de la sélection.*

L'allocation de ressources L'allocation consiste à assigner aux opérations du code source une unité fonctionnelle du processeur et à assigner aux données des emplacements de mémorisation. Si le problème de l'assignation des opérations aux opérateurs est généralement implicitement réglé durant la phase de sélection de code, il en est tout autrement pour celui de l'allocation des ressources de mémorisation. Celui-ci est typiquement réglé en deux étapes. Dans un premier temps il s'agit d'isoler les variables qui seront stockées en registre de celles qui seront stockées en mémoire, dans une phase qui est généralement appelée allocation de registres. Suit alors la phase d'assignation de registres qui consiste à assigner un registre physique à chacune des variables préalablement sélectionnée. En pratique, ces deux techniques sont bien souvent confondues, notamment dans le cadre d'une résolution du problème de l'allocation de ressources par coloration de graphe d'inférences [126].

L'ordonnancement La passe de sélection de code générant un code purement séquentiel, il est nécessaire de modifier l'ordonnancement de ces instructions afin d'autoriser l'exploitation de l'éventuel parallélisme d'instructions du processeur cible. Pour cela, la phase d'ordonnancement se doit de compacter le code afin de regrouper les opérations pouvant s'exécuter en parallèle. Le code vertical, composé d'une suite d'instructions simples, est transformé en un code horizontal utilisant des instructions complexes, i.e. composées de plusieurs instructions simples exécutées en parallèle. Les algorithmes de ce type sont généralement basés sur une décomposition du code source en blocs de traitement (sans instructions de contrôle). Chacun des blocs est alors ordonnancé de manière indépendante et les instructions sont compactées dans ces blocs en tenant compte des dépendances de données, des ressources disponibles et d'éventuelles contraintes sur le parallélisme d'instructions du processeur [78].

Les optimisations indépendantes de l'architecture Finalement, afin de rendre le code plus rapide ou plus compact, tout en préservant son intégrité, un certain nombre d'optimisations peuvent être réalisées sur la représentation intermédiaire, indépendamment de l'architecture cible. Celles-ci sont issues de l'analyse du flot de données et du flot de contrôle. Elles sont réalisées de manière locale si cette analyse porte sur les instructions au sein d'un bloc, ou globale si cette analyse dépasse les frontière de ce bloc. Les optimisations suivantes sont les plus connues et sont systématiquement intégrées dans les compilateurs [?].

- Élimination des sous-expressions communes : cette optimisation permet d'éviter de recalculer une expression E qui a préalablement été calculée et dont les variables n'ont pas été modifiées depuis la dernière évaluation de E.
- Propagation des constantes : cette phase permet d'évaluer les expressions constantes et de les remplacer par leur valeur dans le code ;
- Propagation des copies : Pour les instructions de type $f{=}g$, f est remplacé par g dans toute la portion de code durant laquelle g n'est pas modifié. Si toutes les expressions contenant f dans le code source ont pu être remplacées par g, l'expression $f{=}g$ peut être éliminée.
- Élimination du code mort : toute expression dont le résultat n'est pas utilisé ultérieurement peut être éliminée.
- Simplification algébrique : les instructions faisant intervenir un élément neutre (e.g. $x{=}x{+}0$ ou $x{=}x{\times}1$) peuvent être aisément détectées puis éliminées. De même, certaines opérations peuvent être remplacées par des opérations moins coûteuses (on parle alors de réduction de force). Les divisions par des constantes pourront par exemple être remplacées par des multiplications, voire des décalages dans le cas des puissances de 2.

– Optimisations de boucle : pour optimiser le temps d'exécution des boucles, les techniques évoquées dans le §3.2.4 pourront également être utilisées, de même que des techniques dépendantes de l'architecture cible telle que le pipeline logiciel.

Construction d'une chaîne de compilation via CALIFE

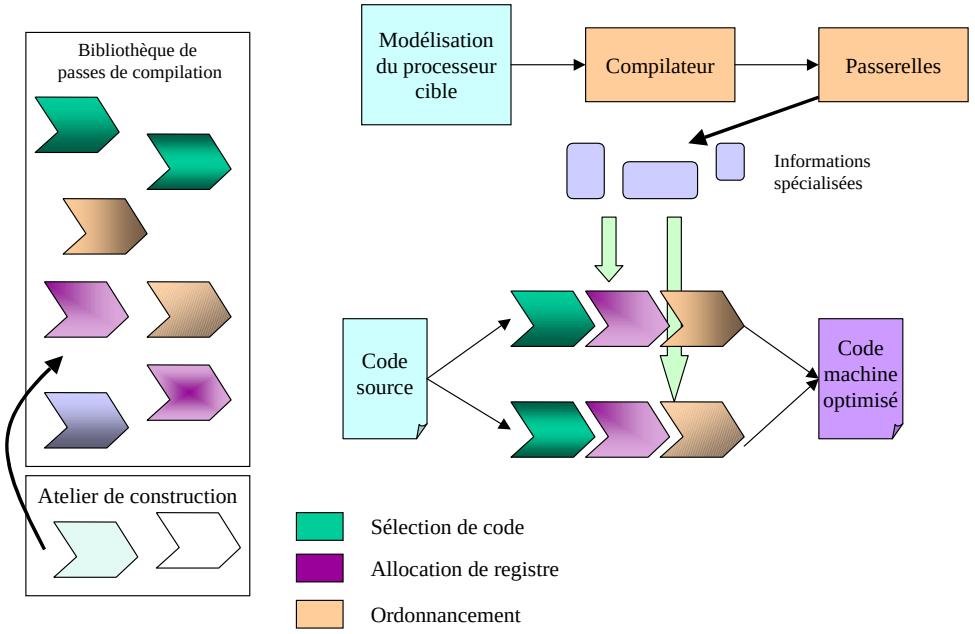

FIG. 3.16 – *Construction d'un compilateur dans CALIFE. L'utilisateur choisit dans une bibliothèque les passes de compilation qu'il souhaite utiliser et les agence. D'une description ARMOR du processeur sont extraites des informations relatives au processeur cible qui viennent paramétrer les passes de compilation souhaitées par l'utilisateur. L'efficacité du compilateur ainsi créé par CALIFE peut alors être vérifiée. Le cas échéant, de nouvelles passes de compilation peuvent être insérées dans la chaîne de compilation afin d'en améliorer la qualité.*

La figure 3.16 [124] illustre le mécanisme de construction d'un compilateur dans l'environnement CALIFE. Pour construire un compilateur visant une architecture particulière, l'utilisateur décrit celle-ci dans le langage ARMOR (cf. §3.4.2), sélectionne les modules de transformation qu'il souhaite intégrer dans la bibliothèque et les agence. Dans CALIFE, alors que la structure globale du compilateur est définie par l'utilisateur, l'adaptation des modules de transformation est réalisée automatiquement par l'outil. Ainsi, il est possible de générer rapidement un compilateur optimisé pour sa cible puisque de nouvelles passes de compilation peuvent aisément être ajoutées ou supprimées du flot de compilation.

Pour la construction d'un tel compilateur, CALIFE exploite une représentation interne du programme, sur laquelle les transformations sont appliquées, ainsi qu'une boite à outils facilitant la construction des modules, paramétrée par une représentation interne du processeur cible. La représentation interne du programme est comparable à celle de SUIF et peut être illustrée par la figure 3.17. La boite à outils offre quant à elle un support pour la conception de modules de transformation et est constituée d'un assemblage de méthodes d'analyse (analyseurs de flots de données, de dépendances, ...) et d'interfaces (structure de données représentant le parallélisme existant, détails sur l'utilisation des ressources, ...).

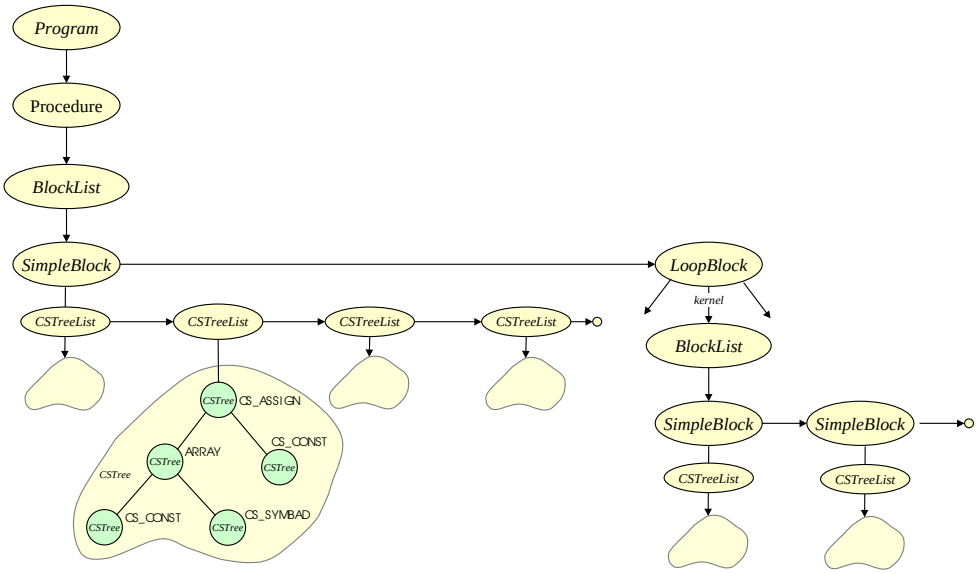

FIG. 3.17 – *Représentation interne de CALIFE. Un programme est représenté sous la forme d'une liste de procédure. Chaque procédure intègre une liste de blocs qui pointent eux-même sur des listes d'instructions. Ces dernières sont représentées sous la forme d'arbres d'expressions.*

L'objectif premier de CALIFE est donc de fournir au concepteur de compilateur des modules de transformation de codes flexibles. Les modules sont donc écrits indépendamment de toute cible et en conséquence, toute information spécifique au processeur passe par une structure (C++) de la boite à outils [124].

3.4.2 Le langage de description ARMOR

Afin de paramétrer les différents outils évoqués précédemment (générateurs de code, estimateurs, ...), les informations sur le processeur cible doivent être suffisamment précises pour spécifier des aspects tels que la sémantique des instructions, l'utilisation des ressources, le parallélisme ou encore le pipeline. Ces différents aspects doivent être décrits dans un langage simple et non ambigu. À ces fins, le langage ARMOR, utilisé dans CALIFE et défini dans [127], décrit le comportement d'un processeur à travers son jeu d'instructions.

Une description ARMOR est une grammaire dont toute dérivation est un comportement possible du jeu d'instructions du processeur. Une description définit des composants comparables aux instructions et les structure en groupes parallèles qui reflètent le parallélisme au niveau instructions de la cible. La description débute par une règle racine (*InstructionSet*) qui présente toutes les alternatives du jeu d'instructions. Des règles *gp* expriment alors la hiérarchie et des règles *restriction* décrivent les contraintes relatives à l'architecture (e.g. parallélisme, registres dédiés, ...). La combinaison de ces règles modélise le jeu d'instructions complet du processeur et son degré de parallélisme. Le comportement de chaque instruction est par ailleurs défini, soit par une règle *df* qui correspond au parcours du flot de données d'une unité fonctionnelle, soit par une règle *ctr* exprimant le flot de contrôle. Les ressources du processeur sont quant à elles définies par les règles : *reg* pour les registres, *regFile* pour un banc de registres, *mem* pour les mémoires et *fu* pour les unités fonctionnelles. Les différents

modes d'adressage sont finalement définis par les règles *address* qui permettent d'exprimer aisément les modes d'adressage auto-modifiés (e.g. post-incrément).

Cette description ARMOR étant vouée à paramétrer les différents modules de transformation de code précédemment évoqués, elle doit ensuite être compilée afin de générer une représentation interne correspondant à des tables de symboles (pour les mémoires, les unités fonctionnelles, ...) et des listes d'instructions.

3.4.3 Compilateur des configurations SW : cDART

Le modèle architectural ciblé par la reconfiguration logicielle correspond à un sous-ensemble d'un DPR de DART. Il peut être modélisé par la figure 3.18. La reconfiguration SW peut être considéré comme un mode de fonctionnement dégradé de l'architecture, dans lequel la flexibilité du réseau d'interconnexions n'est que très partiellement exploitée. Ainsi, les instructions de configuration SW devant être générées par cDART ne concernent qu'une portion limitée de l'architecture. Celles-ci n'ont en effet qu'à spécifier la fonctionnalité des opérateurs et à préciser les sources des opérandes.

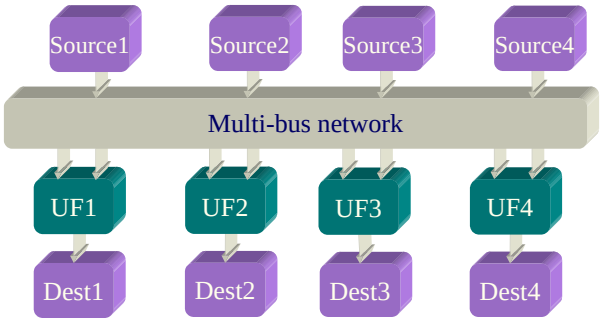

FIG. 3.18 – *Sous-ensemble de DART supportant la reconfiguration logicielle. Il est constitué de 4 unité fonctionnelles se partageant 4 sources d'opérandes. Les sorties de ces unités sont directement envoyées dans une destination propre à chaque opérateur.*

Les informations nécessaires à la définition de cDART via CALIFE concernent donc principalement la fonctionnalité des unités fonctionnelles et le degré de parallélisme de l'architecture. Ces informations sont réunies par la portion de code ARMOR décrivant le jeu d'instructions de DART en fonctionnement logiciel (listing 3.7).

Dans le cadre d'une reconfiguration logicielle, les opérateurs travaillent en arithmétique simple précision sur des données codées exclusivement sur 16 bits. Les traitements SWP n'ont donc pas à être considérés par cDART. Il est en revanche nécessaire d'orienter les données entre les mémoires et les unités fonctionnelles. Les opérateurs disposant d'une totale flexibilité quant aux accès mémoires, chaque unité travaille sur l'un des 4 registres représentant les 4 mémoires locales du DPR. Si chaque unité peut accéder librement à ces 4 mémoires en lecture, elles ne peuvent en revanche modifier le contenu que de l'une d'entre elle. Par ailleurs, la sémantique des instructions doit également être spécifiée et les opérations doivent être assignées aux opérateurs susceptibles de les exécuter. Le listing 3.8, illustre cette modélisation pour une partie des opérations de la première unité arithmétique et logique du DPR.

```
InstructionSet = [ Arith1 || Arith2 || Arith3 || Arith4]

gp Arith1 = Multiplication1 | Multiplication1Shift

gp Arith2 = Operation1 | ShiftgOperation1 | ShiftdOperation1
          | Operation1Shiftg | Operation1Shiftd
          | ShiftgOperation1Shiftg| ShiftgOperation1Shiftd
          | ShiftdOperation1Shiftg| ShiftdOperation1Shiftd

gp Arith3 = Multiplication2 | Multiplication2Shift

gp Arith4 = Operation2 | ShiftgOperation2 | ShiftdOperation2
          | Operation2Shiftg | Operation2Shiftd
          | ShiftgOperation2Shiftg| ShiftgOperation2Shiftd
          | ShiftdOperation2Shiftg| ShiftdOperation2Shiftd

gp Operation1 = Addition1 | Addition1Saturation | Soustraction1
          | Soustraction1Saturation | And1 | Or1

gp ShiftgOperation1 = ShiftgAddition1 | ShiftgSoustraction1
          |ShiftgAnd1 | ShiftgOr1
```

LIST. 3.7 – *Déclaration du jeu d'instructions de DART, en mode SW, dans le langage ARMOR. La règle racine InstructionSet permet de spécifier que 4 instructions arithmétiques peuvent être exécutées concurremment. Ces instructions arithmétiques sont ensuite détaillées afin de décrire l'ensemble des opérations supportées par chacun des opérateurs.*

```
fu mp1 <cycle=1> <wdith in = 16 out = 16 >
fu ual1 <cycle=1> <wdith in = 16 out = 16 >
fu mp2 <cycle=1> <wdith in = 16 out = 16 >
fu ual2<cycle=1> <wdith in = 16 out = 16 >

mode UFSource = M1/M2/M3/M4
mode decALUG = const(1)/const(2)/const(4)
mode decALUD = const(1)/const(2)/const(4)

df Addition1 is  M2 = Add1(UFSource,UFSource)
op Add1(x,y) is ADD(x,y) <ress =ual1>
...
df ShiftgAddition1 is  M2 = ShiftgAdd1(UFSource,UFSource,decALUG)
op ShiftgAdd1(x,y,z) is ADD(LSL(x,z),y) <ress =ual1>
```

LIST. 3.8 – *Spécification des opérateurs et de la sémantique des instructions SW dans le langage ARMOR. La première partie de cette description permet de déclarer les opérateurs de l'architecture cible, en précisant notamment la taille de ses opérandes. Les sources des opérandes ainsi que les différents décalages autorisés sont alors spécifiés avant d'associer à chaque instruction une séquence d'opérations qui sera exécutée sur l'opérateur désigné par la commande <ress = >.*

3.4.4 Compilateur pour la génération d'adresses : ACG

De la même façon que les instructions de configuration SW, les instructions de génération d'adresses sont obtenues via des passes classiques de compilation issues de CALIFE. Elles sont intégrées dans l'outil ACG. Cet outil se doit de générer les programmes devant être exécutés sur les générateurs d'adresses pendant les phases de reconfiguration matérielle et logicielle. Il doit par ailleurs assurer les transferts de données entre la mémoire du *cluster* et les mémoires internes des DPRs en générant les instructions relatives au contrôleur mémoire du *cluster*.

Afin de simplifier les problèmes de synchronisation entre les générations d'adresses et les traitements arithmétiques, lors des traitements irréguliers, la partie compilation de ACG a été intégrée au sein de cDART. Dès lors, les instructions de génération d'adresses et les codes irréguliers ne sont plus distingués. L'architecture de DART modélisée pour CALIFE intègre donc également les générateurs d'adresses et le contrôleur mémoire du *cluster* (Fig. 3.19). La description sous-jacente à cDART doit donc être agrémentée des informations utiles à cette génération d'adresses. Le code figurant dans le listing 3.9 illustre les modifications apportées à la description ARMOR de DART pour modéliser les ressources de mémorisation.

```
Mem MemCluster (16384 * 16) <access Rd=1 Wr=24 > <nbAccess Rd=1 Wr=1>

Mem Mem1 (256 * 16) <access Rd=1 Wr=1 > <nbAccess Rd=1 Wr=1>
Mem Mem2 (256 * 16) <access Rd=1 Wr=1 > <nbAccess Rd=1 Wr=1>
...
fu AG1 <cycle=1> <nbUse=2> <width in = 8 out = 8 >
fu AG2 <cycle=1> <nbUse=2> <width in = 8 out = 8 >
...
RegFile AG1_reg(8 * 8) <access Rd=1 Wr=1><nbAccess Rd=1 Wr=1>
RegFile AG2_reg(8 * 8) <access Rd=1 Wr=1><nbAccess Rd=1 Wr=1>
```

LIST. 3.9 – *Spécification ARMOR des ressources de mémorisation de DART.*

Outres ces ressources de calcul et de mémorisation, il est également nécessaire de mettre à jour le jeu d'instructions de l'architecture cible. Les instructions de lecture et d'écriture mémoire doivent ainsi être précisées, de même que les différents modes d'adressage supportés. Le listing 3.10 décrit la méthode utilisée pour spécifier les modes d'adressage, au niveau d'un générateur d'adresses.

3.4.5 Conclusions et perspectives

Compte tenu de la simplicité des modèles de programmation correspondant aux générations d'adresses et aux configurations SW, la mise en œuvre de techniques classiques de compilation suffit à la génération de codes exécutables par cDART et ACG. Malgré l'adéquation des générateurs d'adresses et du sous-ensemble des DPRs supportant les opérations *Read-modify-Write*, avec les cibles architecturales de CALIFE, certains problèmes de développement retardent l'obtention de compilateurs utilisables pour DART.

Deux principales difficultés doivent en particulier être surmontées[1] :

[1] Celles-ci sont en cours d'étude à l'IRISA.

```
df Load1 is M1 = Mem1[AG1_use]
df Store1 is Mem1[AG1_use] = M1

Address AG1_use = immediat1 | simple1 | INCR1 | DECR1
                | ADD_VAL1 |SOUS_VAL1

Address immediat1 is imm(byte)

Address simple1 is AG1_reg

Address INCR1 is AG1_reg%iINCRAG1(AG1_reg%i,const(1))
op INCRAG1(x,y) is ADD(x,y) <ress =AG1>

Address ADD_VAL1 is AG1_regAddAG1(AG1_reg, CalculSource1)
op AddAG1(x,y) is ADD(x,y) <ress =AG1>

Address DECR1 is AG1_reg%iDECRAG1(AG1_reg%i,const(1))
op DECRAG1(x,y) is SUB(x,y) <ress =AG1>

Address SOUS_VAL1 is AG1_regSousAG1(AG1_reg, CalculSource1)
op SousAG1(x,y) is SUB(x,y) <ress =AG1>
```

LIST. 3.10 – *Spécification ARMOR des modes d'adressage des générateurs d'adresses.*

La gestion de boucles matérielles Bien que modélisables dans le langage ARMOR, les boucles matérielles ne sont exploitées par aucune passe de compilation de CALIFE. En conséquence, une boucle ne peut être implémentée qu'en exploitant un registre général en tant que compteur, et en monopolisant une unité fonctionnelle pour mettre à jour ce compteur. Il est donc nécessaire d'insérer dans le code assembleur des instructions visant à gérer l'évolution de cette boucle. Les DPRs intégrant un module de gestion matérielle de boucle, l'insertion de ces instructions se traduit donc par un code qui n'est pas conforme au comportement attendu du DPR.

L'allocation mémoire L'allocation mémoire est également un aspect qui n'est que partiellement géré sous CALIFE. En particulier, la dispersion des données entre les différentes mémoires de DART est problématique, cet environnement ayant principalement été validé sur des architecture manipulant une mémoire de données unique. La seconde difficulté relative au placement mémoire concerne le support d'un *mapping* mémoire spécifié par l'utilisateur. Dans l'état actuel de l'outil, CALIFE décide en effet totalement du placement des données en mémoire. Un placement des données par l'utilisateur doit cependant être possible afin de prendre en compte l'environnement extérieur au *cluster*.

Dans l'état actuel d'avancement de CALIFE, cDART et ACG ne peuvent donc pas être intégrés à la chaîne de développement de DART. Afin de valider l'architecture, il est cependant nécessaire de générer les codes binaires spécifiant les configurations de DART. Si la génération de configuration SW est peu problématique (gDART peut être utilisé à cet effet), il est également indispensable de générer les instructions devant être exécutées sur les AGs. Afin d'accélérer le temps nécessaire à la génération de ces instructions, un parser a donc été réalisé par le biais des outils *Lex* et *Yacc*. Cet outil autorise l'utilisateur à décrire le comportement des générateurs d'adresses par l'intermédiaire d'un code assembleur. L'outil *parse* alors le fichier assembleur et génère les instructions binaires pour les différents générateurs d'adresses.

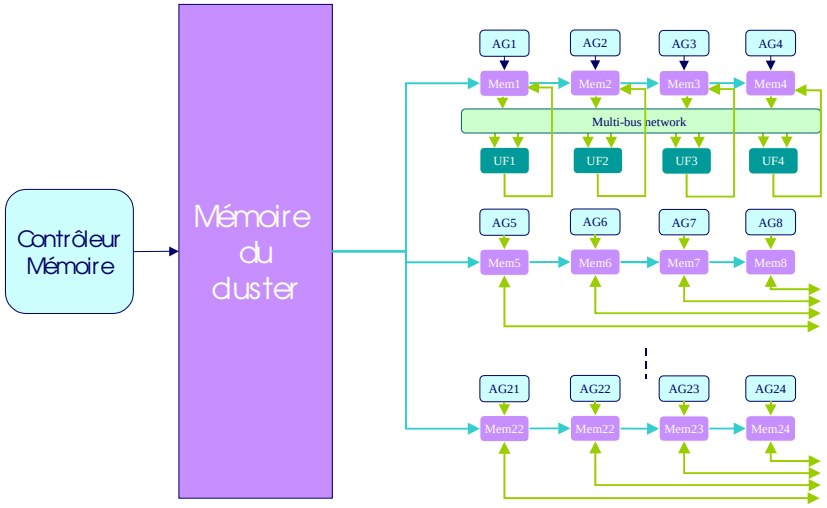

FIG. 3.19 – *Modèle de l'architecture supportant la reconfiguration logicielle et les transferts de données. Outre les ressources de calcul, deux niveaux de hiérarchie mémoire doivent également être modélisés en ARMOR.*

3.5 Simulateur de DART : SCDART

Outre les modules précédemment décrits, qui visent à implémenter une application sur DART, un dernier outil complète notre chaîne de développement, SCDART. Celui-ci permet de simuler le comportement de l'architecture lorsqu'elle est configurée par les codes binaires générés par gDART, ACG et cDART. DART ayant été conçue de manière à minimiser la consommation d'énergie, SCDART doit être en mesure d'estimer rapidement l'énergie consommée lors de l'exécution d'une application.

3.5.1 Introduction

Les outils voués à simuler le comportement d'une architecture exploitent typiquement une modélisation fonctionnelle ou comportementale de leur cible.

Modélisation fonctionnelle

La modélisation fonctionnelle d'une architecture aboutit à la définition d'un simulateur de jeu d'instructions (ISS : Instruction Set Simulator). Celui-ci permet une validation rapide de la fonctionnalité de l'application. L'avantage majeur de cette approche porte sur l'effort limité qu'elle requiert pour modéliser l'architecture. Dans certains cas, la modélisation du jeu d'instructions intègre des informations telles que la hiérarchie mémoire ou la structure du pipeline d'exécution afin d'améliorer la qualité de l'évaluation de la description sur des critères de performances.

La définition d'estimateurs de consommation pertinent dans de tels outils n'est pas triviale, loin s'en faut. Diverses méthodes ont été exploitées à ce jour. Elles se distinguent par le niveau de précision de l'estimation, ainsi que par la complexité de la phase de caractérisation. Celle-ci va de la mesure de la consommation moyenne des instructions et de leurs interactions, jusqu'à la prise en compte des problèmes de rupture de pipeline, ou de défaut de cache [128].

Quelle que soit la méthode employée, il est en particulier nécessaire d'estimer le coût des effets inter-instructions. Celles-ci correspondent aux mouvements de bits inhérent au passage d'une instruction à une autre. En effet, chaque modification d'instruction se traduit par des commutations de bits sur les commandes des différents éléments du chemin de données. Ces commutations de bits se traduisent à leur tour par la modification du comportement des différentes unités et donc par l'exécution de traitements non utiles à l'exécution du programme. Les aléas ainsi générés se traduisent dès lors par une consommation significative d'énergie qu'il est capital d'estimer. Pour cela, il est nécessaire de mesurer cette consommation inter-instructions pour toutes les paires d'instructions possibles. La simplicité du modèle architectural se traduit donc ici par une phase de caractérisation particulièrement complexe.

Modélisation architecturale

La modélisation architecturale de la cible permet la définition d'un simulateur précis au cycle près et au bit près. Dans de tels simulateurs, la description est réalisée au niveau transfert de registre. Ainsi chaque ressource de l'architecture est modélisée et l'utilisateur est en mesure de connaître la valeur de chaque signal binaire transitant à travers ces ressources. Compte-tenu de la précision du modèle, concevoir un tel simulateur est une tâche dont la complexité est proche de celle de la synthèse logique.

La définition d'estimateurs au sein de tels simulateurs est en revanche une tâche relativement aisée. En effet, l'architecture étant modélisée par un assemblage de composants élémentaires, ils peuvent aisément être caractérisés en consommation. Dès lors, estimer l'énergie consommée pendant l'exécution d'une application revient à additionner les consommations induites par chacun des composants de base de l'architecture.

3.5.2 Méthodologie de modélisation

Compte-tenu de la flexibilité de DART, le nombre d'arrangements possible d'opérations (et donc de configurations) est considérable. Si DART devait être modélisé par le biais de son jeu d'instructions, il faudrait en effet un nombre prohibitif d'instructions ($> 3{,}5.10^{110}$ instructions, cf Fig. 3.20). Cette constatation suffit à justifier une modélisation plus fine de notre architecture pour SCDART. Les sections qui suivent décrivent les concepts sous-jacents à ce simulateur.

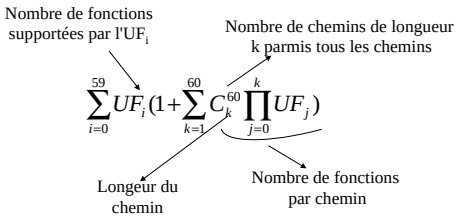

FIG. 3.20 – *Nombre de configurations possibles des DPRs de DART.*

Présentation générale

SCDART est principalement voué à la validation de la description de l'application. L'utilisateur doit ainsi être en mesure de contrôler l'exécution du programme et de vérifier le comportement de l'architecture à tous les stades de l'algorithme. Ceci se traduit par la nécessité de pouvoir définir des points d'arrêts dans l'exécution du programme, et d'assurer

une visualisation aisée des données transitant au sein de l'architecture (e.g., configuration d'opérateurs, contenu des mémoires, ...).

Le paragraphe 3.5.1 a par ailleurs mis en évidence l'intérêt d'une modélisation fine de l'architecture. SCDART est ainsi un simulateur précis au bit près et au cycle près, basé sur une description au niveau transfert de registre de DART (RTL : Register-Transfer Level). Un tel grain de modélisation nécessite la possibilité d'émuler logiciellement des concepts matériel tels que ceux décrits ci-dessous.

La notion de concurrence Pour décrire le comportement d'une architecture, il est indispensable de pouvoir expliciter la concurrence inhérente à toute mise en œuvre matérielle.

La notion de structure Les notions de structure et de hiérarchie caractérisant les architectures doivent également être spécifiées. Il est ainsi nécessaire de pouvoir décrire l'interface des différents composants constituant l'architecture, de même que les structures véhiculant les informations entre ces éléments.

La notion de synchronisation Il est finalement indispensable de décrire les mécanismes qui autorisent les synchronisations des différents constituants de l'architecture. Ces synchronisations peuvent se faire sous la forme de données de contrôle (e.g. clock, enable) ou par le biais d'une base de temps.

Il est ainsi nécessaire de combiner dans SCDART une modélisation matérielle de l'architecture et une gestion logicielle de l'exécution de l'application simulée. Ce contexte est précisément celui ayant conduit à la définition du langage SystemC [115, 129]. Ce langage de modélisation consiste en une librairie C++ autorisant 3 principaux types de constructions.

Les modules SystemC sont équivalents à ceux du Verilog ou aux "entités" du VHDL. Ils sont caractérisés par leurs entrées/sorties et par les fonctions qu'ils mettent en œuvre. Ils explicitent la structure du composant et permettent la mise en place d'une hiérarchie d'éléments.

Les signaux assurent la connectique entre les différents modules constituant l'architecture. Ils transmettent des données typées.

Les canaux servent à la mise en œuvre de protocoles de communication entre les modules. En supportant plusieurs niveaux d'abstraction, ils autorisent la mise en place d'une politique de raffinement de l'architecture.

Par l'assemblage de ces différentes briques, la modélisation de l'architecture est ainsi possible. Afin de décrire les fonctionnalités associées à chacun de ces blocs, 3 méthodes sont exploitables [130].

- *SC_METHOD* : cette fonction décrite en C++ est appelée à chaque fois que l'un des signaux de sa liste de sensibilité évolue. Ce type de description, parfaitement adapté à la modélisation de processus combinatoire, s'exécute entièrement à chaque invocation.
- *SC_THREAD* : cette fonction décrite en C++ s'exécute indéfiniment. Des points de synchronisation doivent être spécifiés dans la description afin de ne pas monopoliser l'exécution. Lorsqu'un tel point de synchronisation est rencontré, le processus rend la main à l'ordonnanceur qui ne rappellera la fonction qu'à la prochaine évolution des signaux de sa liste de sensibilité. Ce type de fonction est particulièrement adapté à la modélisation de processus de tests ou à la gestion de domaines multi-horloges.

- *SC_CTHREAD* : cette fonction peut en première approximation être assimilée à un *SC_THREAD* sensible sur un seul signal. Ce signal est alors considéré comme une horloge. Cette fonction s'exécute indéfiniment et peut être interrompue par des primitives de synchronisation. Dans de telles fonctions, le processus échantillonne ses entrées lors du front actif de l'horloge et écrit ses résultats au front actif d'horloge suivant. Les *SC_CTHREAD* visent à modéliser des systèmes purement synchrones.

Afin de modéliser finement l'architecture, différents types de données peuvent être utilisés pour modéliser les connexions entre les différents modules constituant l'architecture. Les principaux types utilisables sont les suivants :

- les données natives du C++ (bool, char, int, long, ...) ;
- *SC_INT<NB>, SC_UINT<NB>* : ces deux types de données permettent de représenter respectivement des nombres signés (codage en complément à deux) et non signés, en caractérisant le nombre de bits nécessaires à leur codage (*NB*) ;
- *SC_BIT, SC_BV<NB>* : ces types permettent de représenter des bits ou des vecteurs de bits pouvant prendre les valeurs '0' ou '1' ;
- *SC_LOGIC, SC_LV<NB>* : ces types permettent de représenter des bits ou des vecteurs de bits pouvant prendre les valeurs '0', '1', 'X' ou 'Z', afin de modéliser le comportement de connexions partagées entre plusieurs maîtres ;
- *SC_FIXED<LENGTH, ROUND_MOD, SAT_MOD>* : ce type permet de manipuler d'une manière très flexible des données codées en virgule fixe. La déclaration d'une variable de ce type s'accompagne de la spécification de la méthode d'arrondi ou de saturation utilisée.

Par le biais de cette bibliothèque SystemC, il est possible de combiner dans SCDART une modélisation précise de l'architecture et une gestion abstraite du déroulement de l'exécution. SCDART est ainsi bâti sur une description RTL de DART. Cette description est encapsulée dans un module, et exploitée par le biais de fonctions gérant le déroulement de l'exécution. Ces fonctions sont chargées de réaliser des points d'arrêt, d'afficher la valeur de certains points d'observation de l'architecture, ... En outre, en décrivant DART au niveau RT, nous sommes en mesure de rapidement synthétiser les différents blocs décrits, soit par une traduction directe des codes SystemC en VHDL, soit par l'utilisation d'outils de synthèse supportant ce langage en point d'entrée (*SystemC Compiler* [131], *CoCentric* [132], ...).

Validation de l'architecture

Un des enjeux majeurs de la modélisation au niveau RTL est la validation de l'architecture. L'usage de bus globaux pour transférer les données entre les DPRs implique la nécessité de partager un même bus entre plusieurs ressources susceptibles de s'en rendre maître à un instant donné. Si deux composants se rendent simultanément maître du bus, un conflit intervient. Cette éventualité est totalement inadmissible compte-tenu des conséquences désastreuses de ce type de comportement. Il est donc indispensable de vérifier logiciellement que de telles conditions environnementales ne peuvent jamais se produire. Ceci est réalisé pour DART par le biais de son simulateur qui se doit de manipuler des données résolues, dans lesquelles les valeurs "Z..Z" ou "X..X" prennent un sens physique, représentant respectivement la mise en haute impédance et le conflit.

Un autre aspect clé de l'architecture devant être validé est la synchronisation des ressources. Le contrôle étant distribué dans l'architecture, il est en effet indispensable de vérifier que les signaux de synchronisation et de contrôle sont distribués en temps et en heure dans l'architecture. L'interface de chaque module constituant le modèle doit ainsi respecter scrupuleusement

celle des composants matériels constituant l'architecture. Le langage *SystemC* fournit toutes les primitives nécessaires à cette modélisation.

3.5.3 Estimation de consommation

Caractérisation des éléments de l'architecture

Afin d'estimer la quantité d'énergie consommée lors de l'exécution d'une application, SCDART se base sur une caractérisation en consommation des différents éléments constituants le circuit. Ces caractérisations sont issues de l'outil *Design Power* de Synopsys [133]. Elles se basent sur la netlist générée par la synthèse logique, et sur des jeux de vecteurs de tests générés aléatoirement. De ce fait, les estimations réalisées peuvent être considérées comme pessimistes puisqu'elles occultent la possibilité de manipuler des données fortement corrélées.

La méthode d'estimation diffère en fonction du type de composant considéré. Dans les paragraphes qui suivent nous donnons la méthode employée pour chacun des éléments constituants l'architecture de DART.

Les opérateurs Les opérateurs intégrés dans DART supportant plusieurs opérations, leur caractérisation en consommation doit tenir compte de leur structure. Deux approches sont alors possibles. La première consiste à modéliser les unités fonctionnelles sous la forme d'un assemblage de composants élémentaires et de caractériser en consommation chacun de ces blocs. Une autre solution consiste à décrire l'unité d'une manière comportementale et à caractériser chacune des opérations supportées par l'opérateur. Par soucis de concision dans la description, cette seconde solution a été retenue pour DART. Cette caractérisation s'est traduite par les résultats résumés dans les tableaux 3.2 et 3.3, caractérisant respectivement les multiplieurs/additionneurs et les UALs.

Opération	Données 8 bits	Données 16 bits	SWP 8 bits
Addition (pJ)	10.2	33.9	18
Multiplication (pJ)	14.9	46.6	26.7
+ décalage (pJ)	3	3	3.4

TAB. 3.2 – *Caractérisation énergétique des multiplieurs/additionneurs de DART.*

Les composants d'interconnexions Les composants d'interconnexions, tels que les multiplexeurs ou les tri-states ne consomment pas seulement lorsqu'ils sont excités par un signal de commande. Dès lors qu'un signal les traverse, ces composants consomment de l'énergie. Ceci s'admet aisément lorsque l'on considère les modèles au niveau porte ou transistor de ces éléments. La figure 3.21 représente ces modèles pour un multiplexeurs 2 vers 1 réalisé à base d'une porte AOI. En conséquence, la caractérisation de ces composants d'interconnexions s'est traduite par l'extraction de deux paramètres : la consommation induite par la traversée de la porte, et celle induite par sa configuration. Les résultats de cette étude sont donnés pour les différents composants d'interconnexions dans le tableau 3.4. Il est à noter dans ce tableau qu'en première approximation, nous ne tenons pas compte de la taille des données traversant les multiplexeurs, bien que lors de traitements 8 bits, les 8 bits de poids faibles constituant le mot de 16 bits sont systématiquement à '0'.

Opération	Données 8 bits	Données 16 bits	Données 32 bits	Données 40 bits	SWP 8 bits	SWP 16 bits
Addition	8	12.4	21.8	25.3	13.5	22.1
Add + Saturation	10.9	14.5	25	28.6	18.1	25.7
Soustraction	8	12.2	21.8	25.2	13.5	21.9
Sous + Saturation	10.6	14.4	24.6	28.1	17.8	25.2
Minimum	10.2	14.4	24.2	-	16.8	25.2
Maximum	10.1	14.3	24.1	-	16.8	25.2
Valeur absolue	8.8	10.4	19.8	27.1	16.9	20.1
+ Décalage entrée	7.8	7.8	7.8	7.8	7.8	7.8
+ Décalage sortie	8.7	8.7	8.7	8.7	8.7	8.7
Et	9.5	9.5	9.5	9.5	9.5	9.5
Ou	9.8	9.8	9.8	9.8	9.8	9.8
Xor	12.3	12.3	12.3	12.3	12.3	12.3
NOP	8.9	8.9	8.9	8.9	8.9	8.9

TAB. 3.3 – *Caractérisation énergétique des UALs de DART (en picoJoules).*

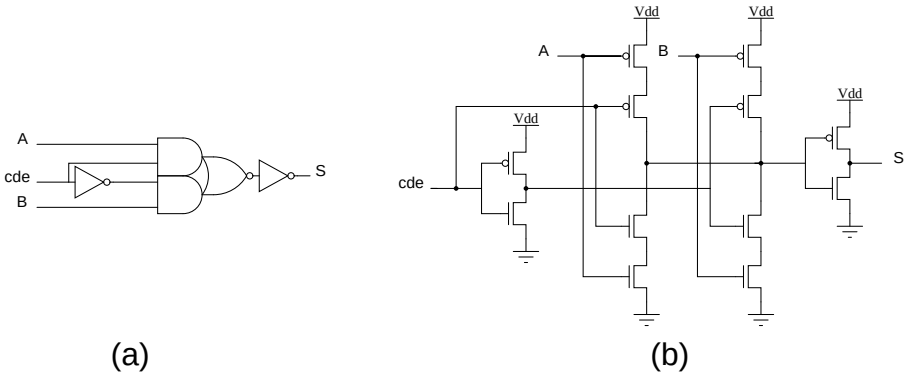

FIG. 3.21 – *Modèles portes (a) et transistors (b) d'un multiplexeur 2 vers 1 à base de porte AOI (And-Or-Invert).*

Opérateur	Traversée	Configuration
MUX Multiplieur (18 :1)	3.6	1.4
MUX UAL (18 :1)	4.7	1.3
MUX Bus globaux (11 :1)	3.5	4.1
MUX registres (18 :1)	3.4	1.5
MUX Mémoire (10 :1)	4.3	1.4
Tri-state (32 bits)	0.45	0.35

TAB. 3.4 – *Caractérisation énergétique des composants d'interconnexions de DART (en pico-Joules).*

Les registres Le modèle au niveau porte d'une bascule D est représenté sur la figure 3.22. Bien que la plupart des outils d'estimation de la consommation au niveau logique considèrent qu'une bascule ne consomme que lors des transitions 0→1 de leur sortie [134], l'examen de cette structure conduit cependant à la constatation que l'activité de l'entrée D n'est pas la seule responsable de la consommation de la bascule. Celle-ci consomme de l'énergie, y compris

lorsque ses sorties ne commutent pas, de part la bufferisation de l'horloge notamment. La caractérisation des registres de DART doit donc extraire deux paramètres : la consommation intrinsèque et la consommation dynamique. Par consommation intrinsèque nous entendons ici la consommation d'énergie induite par l'attaque de l'horloge (cf. tableau 3.5) et le maintien des données au sein du buffer.

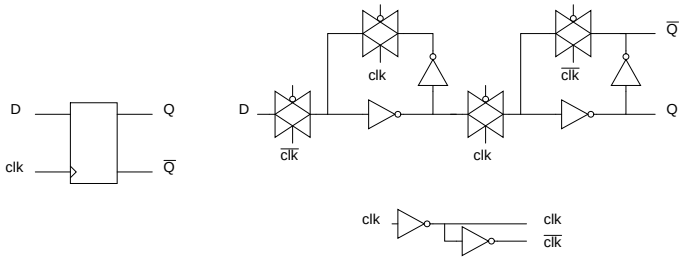

FIG. 3.22 – *Modèle au niveau porte d'une bascule D.*

Opérateur	Consommation intrinsèque	Consommation dynamique
Registre 40 bits (ALU)	1.3	5.7
Registre 32 bits (Mult)	1.1	4.2
Registre 16 bits (DPR)	0.4	2.2
Registre 8 bits (AG)	0.3	1.1

TAB. 3.5 – *Caractérisation énergétique des registres de DART (en picoJoules).*

Les mémoires Les valeurs de consommation des mémoires de DART sont issues d'estimations réalisées dans le cadre de la thèse de David Saillé [135]. Les résultats de cette caractérisation sont données dans le tableau 3.6 et concernent les mémoires de données des DPRs, la mémoire de données du *cluster* ainsi que la mémoire de configurations du contrôleur de *cluster*.

Format mémoire	Lecture	Écriture
256×16 bits	18	23
16k×16 bits	133	171
2k×52 bits	100	164

TAB. 3.6 – *Caractérisation énergétique des mémoires de DART (en picoJoules).*

Le modèle exploité pour caractériser cette architecture est dérivé des travaux de Wilton et Jouppi [136] qui ont développé l'outil CACTI. Ce dernier évalue le temps d'accès à un cache paramétré par sa taille, la taille des blocs et son associativité, après évaluation des capacités internes. À partir de leur estimation des capacité internes, l'évaluation de l'énergie dissipée au nœud considéré est effectuée en appliquant la formule $E = \frac{1}{2}N.C.V^2$. Dans cette équation, N représente le nombre de transitions du nœud de capacité C. David Saillé a défini un modèle pour chacun des éléments composants les mémoires (Fig. 3.23). Pour obtenir la consommation globale de la mémoire, les énergies élémentaires doivent être sommées en tenant compte du nombre d'accès.

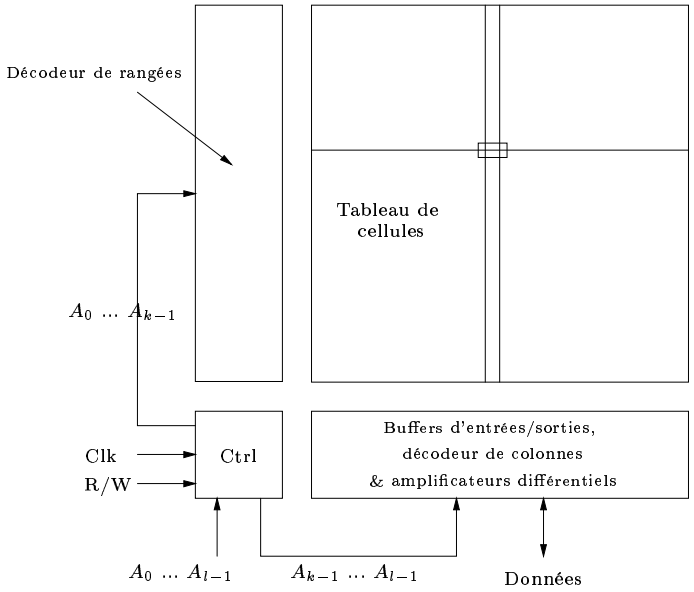

FIG. 3.23 – *Architecture d'une mémoire : Ces éléments sont les décodeurs de rangées et de colonnes, les wordlines, les bitlines, les éléments servant à la détection (sense amplifiers) et enfin les buffers de sorties.*

Mise en œuvre de l'estimation de la consommation

Sur la base de ces caractérisations en consommation des différents modules constituant notre architecture, SCDART doit alors estimer la consommation d'énergie induite par l'exécution d'une application. Cette estimation est basée sur l'équation 3.1. Celle-ci illustre le fait que la consommation d'énergie d'un circuit est la somme des énergies consommées par chacun des éléments le constituant.

$$Energie = \sum_{\forall\ operateurs} \sum_{\forall\ Typeacces} Nb_{acces} * Conso_{moy} \tag{3.1}$$

L'intégration de cette méthode d'estimation est relativement aisée compte-tenu du modèle d'architecture choisi et des fondements même du langage SystemC. En effet, la spécification d'un élément du circuit est réalisée en systemC en créant un module. Ce module contient alors des méthodes qui décrivent le comportement du circuit. Celles-ci sont totalement décorellées, ce qui autorise la description de comportements radicalement différents. Ainsi, dans SCDART, en sus des méthodes décrivant le comportement physique de l'élément considéré, sont intégrées des méthodes dédiées à l'estimation de la consommation. Ces méthodes varient légèrement selon le type de l'élément modélisé.

Les opérateurs et les mémoires En première approximation, nous avons considéré que les opérateurs ne consomment de l'énergie qu'à la condition que leur sortie commute. De ce fait, l'estimation de la consommation induite par les opérateurs est mise en œuvre par le biais d'une méthode unique. Cette méthode n'est sensible qu'à la sortie de l'opérateur. Au sein de la méthode, l'examen des entrées du composant permet de déterminer sa consommation en sélectionnant parmi les paramètres issus de la caractérisation celui qui correspond à l'opération

traitée. Cette méthode est décrite pour le multiplieur/additionneur de DART dans le listing 3.11.

```
SC_CTOR (MULT){
        SC_METHOD (ModelMult)
        Sensitive «In1«In2«dec«op«SWP;

        SC_METHOD (EstimCconso)
        Sensitive «Out;
}
void MULT : :EstimCconso() {
        if SWP.read()==16 then
                if Op.read()==0 then
                        Consommation+=ConsoMultUF1;
                else
                        Consommation+=ConsoAddUF1;
        else
                if Op.read==0 then
                        Consommation+=ConsoMultUF1SWP;
                else
                        Consommation+=ConsoAddUF1SWP;
        if dec!=0
                Consommation+=ConsoDecUF1;
}
```

LIST. 3.11 – *Méthode d'estimation de la consommation des multiplieurs/additionneurs.*

Cette méthode d'estimation peut également être appliquée aux mémoires. La méthode sera cependant cette fois sensible aux évolutions du bus d'adresses du composant et non de sa sortie. Au sein de la méthode, nous différencions par ailleurs les opérations de lecture et d'écriture.

Les registres et les composants d'interconnexions Afin de distinguer les consommations intrinsèque et dynamique des registres, deux méthodes sont utilisées. La première modélise la consommation induite par l'attaque de l'horloge. À chaque cycle, cette méthode est appelée et la variable de consommation est incrémentée conditionnellement à la valeur de la garde du registre. Une seconde méthode modélise quant à elle la consommation dynamique du registre. Elle est sensible à la valeur de la sortie.

De même, deux méthodes sont utilisées pour estimer la consommation issue des ressources d'interconnexions. Par l'exploitation de ces deux méthodes, la consommation induite par la configuration de ces composants peut ainsi être distinguée de celle due à leur traversée. La première méthode est sensible aux modifications intervenant sur la commande des composants d'interconnexions. La seconde traduit la consommation du composant lors de sa traversée.

Potentiel d'optimisation de l'estimateur de la consommation

Afin d'améliorer la précision de l'estimation de la consommation, il est relativement aisé d'introduire de nouveaux mécanismes dans SCDART. En particulier, la flexibilité de la description nous autorise à raffiner l'estimation en prenant en compte la corrélation des signaux transitant dans le circuit. En particulier, Mathieu Denoual a développé dans sa thèse un outil

d'estimation de consommation au niveau architectural, *PowerCheck*, prenant en compte ces paramètres [134]. La corrélation entre les données est alors exprimée de manière probabiliste, suivant un modèle DBT (Dual Bit Type). Ce modèle, proposé par P. Landman [137, 138], s'appuie sur une analyse statistique des signaux pour démontrer que l'activité des bits n'est pas la même pour tous les éléments du vecteur de codage. Typiquement, l'activité des bits de poids faible est assimilable à un bruit blanc uniforme. Au contraire, les bits de poids fort ont une activité très limitée. Ainsi, ce modèle définit trois zones (Fig 3.24). Pour les bits de poids fort est définie une corrélation qui rend compte de la probabilité de transition d'un bit d'un vecteur à l'autre. Une seconde zone modélise les bits de poids faible et correspond à une corrélation nulle (bruit blanc). Entre ces deux zones, la corrélation suit une variation linéaire depuis 0 vers la valeur de corrélation des poids forts. Ces différentes régions sont délimitées par des points de rupture qui repèrent le rang des bits séparant ces régions.

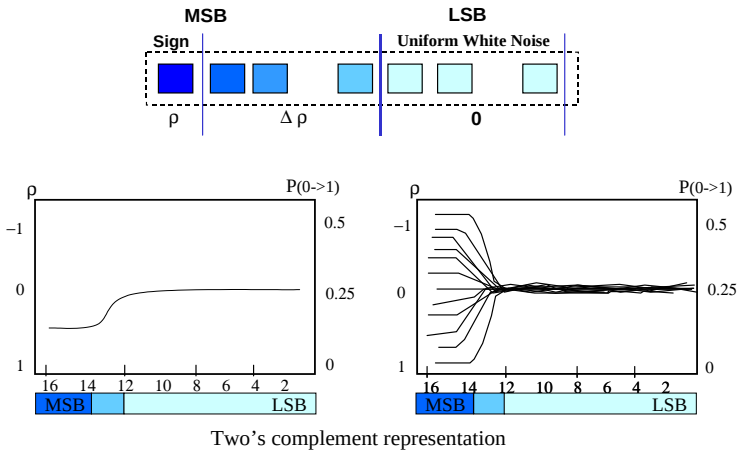

FIG. 3.24 – *Modèle DBT d'un signal.*

Par la spécification de ces valeurs de corrélation et de ces points de rupture, il est ainsi possible de modéliser les différents comportements des signaux transitant dans l'architecture. Ainsi, la caractérisation des différents modules constituants DART, peut être réalisée de manière nettement plus précise. À titre d'exemple, cette méthode permet de distinguer des accès séquentiels aux mémoires des accès aléatoires qui génèrent beaucoup plus d'activité sur les lignes d'adresses et donc consomment beaucoup plus d'énergie.

Dans une évolution future de SCDART, il est ainsi possible de mettre en œuvre ces méthodes d'estimations de la consommation. La caractérisation des opérateurs, en fonction de l'activité des signaux d'entrées sera alors issue de l'outil *PowerCheck*. Pour exploiter ces données, les méthodes d'estimation intégrées dans SCDART devront également être modifiées. Compte-tenu de la flexibilité de la description, l'effort de développement à fournir est cependant très limité. Dans le même ordre d'idée, la précision de l'estimation peut être accrue en caractérisant la consommation statique de DART [139]. Compte-tenu de l'impact croissant de cette consommation statique dans les circuits profondément sub-micronique, cette perspective s'avère extrêmement intéressante, et une fois encore relativement aisée à mettre en œuvre.

3.6 Conclusions et perspectives

Dans ce chapitre, la chaîne de développement associée à DART a été présentée. Par l'extraction de modèles d'exécution simples, nous avons pu construire cette chaîne de développement en exploitant des techniques ayant fait leurs preuves dans le cadre de la synthèse de circuits dédiés et de la compilation pour processeurs programmables. Notre chaîne de développement se base ainsi sur un module dont la tâche principale est d'identifier dans le code source les traitements devant être exécutés suivant chacun des modèles d'exécution de DART. À chaque modèle d'exécution (reconfiguration HW, SW, génération d'adresses) est alors associé un module de génération de code.

Les configurations SW et les générations d'adresses représentant un comportement comparable à celui des processeurs programmables, des techniques de compilation sont utilisées pour générer les instructions devant être manipulées. Par le biais de l'outil CALIFE, les outils cDART et ACG peuvent être conçus avec un moindre effort. Les configurations HW reproduisant quant à elles un comportement d'architecture matérielle, des techniques de synthèse de haut niveau sont exploitées pour générer les différentes structures de chemin de données. Par le développement d'un simulateur précis au bit près et au cycle près, les codes binaires générés par ces trois outils peuvent finalement être validés, et la qualité de la solution peut être quantifiée.

Bien que certains retards de développement interdisent aujourd'hui une transformation totalement automatique des programmes source en codes binaires exécutables, nous avons pu démontrer la viabilité de cette approche et l'intérêt de l'effort de hiérarchisation produit au début de cette étude. Certaines perspectives à cette étude permettraient d'accroître la valeur ajoutée de cette chaîne de développement.

Allocation des ressources L'exploitation de cette chaîne de développement impose à l'utilisateur de déterminer le nombre de ressources disponibles pour implémenter une application. Afin de limiter l'effort de développement, une perspective intéressante à ce travail serait donc d'automatiser l'allocation des DPRs. À cet effet, une passe SUIF pourrait être développée afin de paramétrer gDART en fonction de la complexité de l'application.

Exploration du parallélisme Privilégier un niveau de parallélisme au détriment des autres a un impact très significatif sur l'efficacité de l'implémentation (cf. chapitre 4). En particulier, l'exploitation du parallélisme de tâches peut favoriser le partage des données lors des traitements par blocs. À l'inverse, certains traitements flots de données bénéficient d'un parallélisme massif au niveau opération qu'il est bon d'exploiter massivement. Privilégier l'un ou l'autre de ces degrés de parallélisme impose une écriture consciencieuse du code source. En conséquence, l'introduction d'un module permettant, à partir d'une description unique de l'application, d'évaluer les différents niveaux de parallélisme inhérents à l'application permettrait de réduire significativement le temps de développement.

Support SMT Bien que l'architecture de DART soit parfaitement adaptée à l'exécution concurrente de plusieurs tâches, rien dans la chaîne de développement ne facilite ce type d'exploitation de l'architecture. Introduire un support logiciel aux traitements SMT (Simultaneous Multi-Threading) permettrait donc d'exploiter au maximum les capacités de DART. La mise en œuvre de ces techniques n'est cependant pas triviale. Il faut dans un premier temps être en mesure de recevoir plusieurs descriptions C en point

d'entrée, puis d'extraire de chacune de ces spécifications, les traitements susceptibles d'être exécutés concurremment. La mise en œuvre de ces techniques SMT nécessite donc l'introduction préalable des deux traitements évoqués précédemment. Il faut en effet être capable d'explorer le parallélisme intrinsèque de chacune des tâches devant être implémentée et d'allouer un certain nombre de DPRs à chaque tâche en fonction de leurs complexités respectives. Les synchronisations entre les différentes tâches sont également des difficultés qui doivent être impérativement surmontées.

Chapitre 4

Validation de l'architecture

Sommaire

Dans ce chapitre nous validons les concepts évoqués précédemment et démontrons le potentiel de DART dans le cadre des applications embarquées de prochaines générations. Pour cela, nous présentons les résultats issus d'implémentations de fonctions clés des télécommunications 3G. Nous analysons en particulier le comportement de DART lors du traitement d'un récepteur WCDMA. Cette analyse est décrite en deux temps. Les résultats d'implémentations des parties calculatoires de ce récepteur sont tout d'abord examinés et critiqués. Par la suite, l'implémentation du récepteur complet est étudiée et les performances de DART sont comparées à celles d'architectures représentatives de l'offre commerciale en matière d'architectures reconfigurables. En particulier, nous examinerons les architectures reconfigurables au niveau logique et système par le biais du FPGA Xilinx VirtexE et du DSP TMS320C64x de Texas Instruments. Ces comparaisons visent à mettre en avant le potentiel de la reconfiguration au niveau fonctionnel dans un tel domaine applicatif, et à étudier les atouts et les limitations des différents paradigmes de reconfiguration.

L'évaluation de DART dans le cadre des traitements multimédia fait l'objet de la seconde partie de ce chapitre. Nous résumerons dans cette partie les principaux résultats issus d'implémentation de ce type de traitement. La transformée en cosinus discrète et l'estimation de mouvement seront en particulier étudiées. Par l'étude de la synchronisation d'un système OFDM (Orthogonal Frequency Division Multiplexing), l'adéquation de DART avec les télécommunications de quatrième génération sera finalement évaluée.

4.1 Description d'un système WCDMA

Dans le cadre de l'UMTS, toute communication met en scène deux protagonistes : Le terminal mobile et la station de base. Le sens des communications permet alors de distinguer les voies montantes et descendantes. Les premières se réfèrent aux communications amorcées par le terminal mobile en direction de la station de base (voie montante) et les secondes évoquent les communications depuis la station de base vers le terminal mobile (voie descendante). Dans le cadre de cette étude, nous nous concentrerons exclusivement sur les traitements réalisés au niveau du terminal mobile.

4.1.1 Émetteur WCDMA

Le synoptique de l'émetteur devant être implémenté au niveau du terminal mobile est donné sur la figure 4.1. Les données à transmettre sont regroupées au sein de différents canaux physiques en fonction de la nature des informations véhiculées [140]. Les principaux canaux physiques de la norme pouvant être transmis par le terminal mobile véhiculent des données (DPDCH) ou des informations de contrôle (DPCCH). Ces dernières spécifient les informations de gestion de puissance, le facteur d'étalement utilisé pour les données transmises, ainsi qu'une séquence de bits permettant d'estimer le canal à la réception (bits pilotes). Ces canaux sont organisés en trames de $10ms$ composées chacune de 15 *slots*.

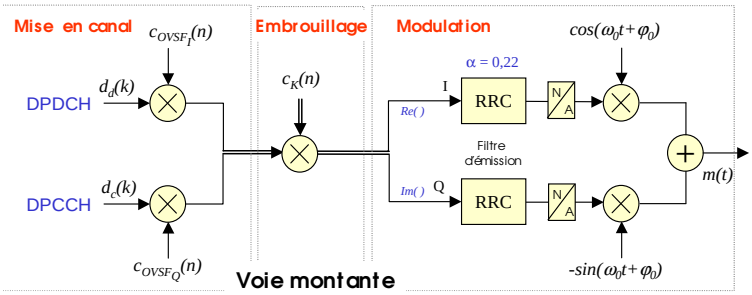

FIG. 4.1 – *Synoptique d'un émetteur de terminal mobile. La mise en canal multiplie les signaux à émettre par un code OVSF afin d'identifier et de séparer les différent canaux. L'embrouillage permet ensuite d'identifier et de séparer les utilisateurs par le biais d'un code pseudo-aléatoire. Une modulation de phase à quatre états adapte finalement le signal au support de transmission.*

Les données à transmettre *d(k)* possèdent un débit *Ds* variant de 15Ksymboles/s à 960 Ksymboles/s pour la voie montante. Lors de la mise en canal, ces données sont multipliées par un code OVSF (Orthogonal Variable Spreading Factor) orthogonal de longueur variable [141]. L'élément correspondant à un symbole du code est dénommé *chip* et sa durée *Tc* est

inférieure à la durée du symbole à transmettre Ts. Le débit du code Dc est fixé à 3.84Mbits/s. Le facteur d'étalement est alors défini comme le rapport entre la durée d'un symbole et la durée d'un *chip*. Ces codes OVSF permettent d'identifier et de séparer les différents canaux. Ces codes étant orthogonaux, l'intercorrélation entre deux codes est nulle dans la mesure où ils sont synchronisés.

Le signal résultant de la mise en canal est ensuite embrouillé, par le biais d'une multiplication par un code pseudo-aléatoire (code PN) [141], afin d'identifier et de séparer les utilisateurs se partageant le support de transmission. Dans la cadre de l'UMTS, ces codes PN sont des codes de Kasami [141]. Si l'intercorrélation entre deux codes PN n'est pas nulle, ces derniers disposent en revanche d'excellentes propriétés d'autocorrélation. Leur fonction d'autocorrélation est en effet proche d'une impulsion de Dirac. Ainsi, ces codes PN sont utilisés pour synchroniser les codes générés en interne. Le couplage des codes OVSF et PN permet de séparer deux canaux dont les facteurs d'étalement peuvent être différents.

Une modulation de phase à quatre état (QPSK : Quadrature Phase Shift Keying) est finalement appliquée au signal embrouillé. Pour éliminer l'interférence entre symboles, au niveau de la réception, un filtre de demi-Nyquist, mis en œuvre sous la forme d'un filtre en cosinus surélevé d'un facteur de retombée de 0.22, est appliqué sur les voies réelles et imaginaires. Le signal est ensuite converti dans une forme analogique et mélangé avec les porteuses en phase et en quadrature.

4.1.2 Récepteur WCDMA

Lors 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 plusieurs 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. Le déplacement du terminal mobile peut par ailleurs conduire à une variation de la puissance du signal reçu. Ce phénomène, dénommé *fading*, produit des évanouissements du signal lorsque la combinaison des multi-trajets est destructrice. Dans ces conditions, la réponse impulsionnelle du canal est modélisée par l'équation 4.1. Dans cette équation, le terme α_k représente l'amplitude du $k^{ième}$ trajet. θ_k représente la phase de ce trajet et τ_k le retard qui lui est associé.

$$h(t) = \sum_k \alpha_k e^{j\theta_k} \delta(t - \tau_k) \tag{4.1}$$

Le synoptique d'un récepteur mono-utilisateur WCDMA est représenté sur la figure 4.2. Dans un tel récepteur, le signal capté par l'antenne est tout d'abord translaté en bande de base, puis converti en une forme numérique. L'exploitation d'une modulation de phase à quatre états se traduit par la représentation des signaux sous une forme complexe. Afin d'exploiter au mieux la dynamique du convertisseur analogique-numérique, une gestion automatique de gain est mise en œuvre (CAG : Contrôle Automatique de Gain) et permet de maîtriser les conséquences du phénomène de *fading*. Pour faciliter la synchronisation entre l'émetteur et le récepteur, la numérisation du signal opère à une fréquence multiple de celle d'un *chip* ($3.84MHz$). Le facteur de sur-échantillonage N_{se} ne doit cependant pas être trop élevé afin de limiter la complexité des calculs effectués par la suite. Dans le cadre de notre expérimentation, celui-ci a été fixé à 4.

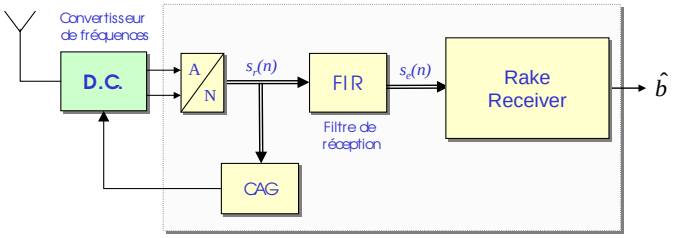

FIG. 4.2 – *Synoptique d'un récepteur WCDMA de terminal mobile.*

Une fois numérisé, le signal complexe attaque un filtre de demi-Nyquist, dit de réception. Celui-ci permet d'éliminer l'interférence entre symboles. Les parties réelles et imaginaires de ce filtre sont traitées indépendamment par des filtres FIR à coefficients réels composés de 64 cellules. La complexité de ces deux filtres est de $1966.10^6 MAC/s$. Soit M le nombre d'utilisateurs, L le nombre de répliques du signal transmis disponibles en entrée du récepteur et w, le bruit introduit par le canal et présent en sortie du filtre de demi-Nyquist, le signal $s_e(n)$ en sortie du filtre de réception satisfait l'expression suivante :

$$s_e(n) = \sum_{m=1}^{M} \sum_{l=1}^{L} s_m(n - \tau_l) + w(n) \tag{4.2}$$

La composante $s_m(n-\tau_l)$ correspond au signal reçu de l'utilisateur m, via le trajet l affecté du retard τ_l. Si les signaux arrivent avec un retard supérieur à la durée d'un *chip*, alors le système peut les séparer. Pour un signal issu d'un multi-trajet, les autres signaux sont perçus comme des interférences et éliminés par le gain de traitement. Un bénéfice supplémentaire est ainsi obtenu en combinant les signaux issus des différents multi-trajets après traitement de ceux-ci au sein d'un récepteur en râteau (*rake receiver*). Soit $c(k)$, le $k^{ième}$ symbole transmis, $\alpha_l(k)$ l'amplitude complexe du canal pour le trajet l et $n_l(k)$ le bruit additif gaussien présent en sortie de cette branche, le signal y_l représentant le trajet l s'exprime alors de la manière suivante :

$$y_l(k) = \alpha_l(k).c(k) + n_l(k) \tag{4.3}$$

Suite au filtrage, le symbole est estimé, en tenant compte de la diversité des trajets caractérisant les communications mobiles, à partir du critère de maximum de vraissemblance défini par l'équation 4.4 [142].

$$y(k) = \sum_{l=1}^{L} \alpha_l^*(k) y_l(k) \tag{4.4}$$

La 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. La détermination du terme $\alpha_l^*(k)$ est issue de l'estimation de la phase et de l'amplitude complexe du canal. Cette estimation est réalisée grâce à la connaissance a priori de bits pilotes (cf. section 4.2.4). Ce récepteur en râteau est modélisé sur la figure 4.3. Chaque trajet l est traité par un module appelé *finger* dont le synoptique est donné par la figure 4.4. Chaque *finger* réalise le décodage des données, l'estimation de l'amplitude complexe du canal et la synchronisation des codes générés en interne.

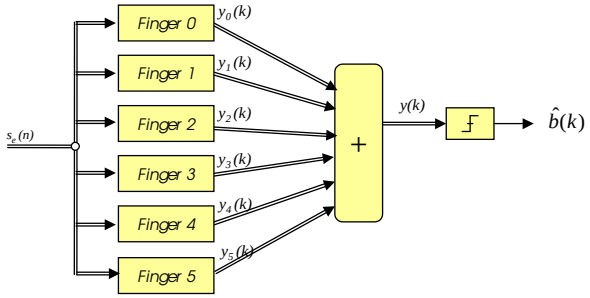

FIG. 4.3 – *Synoptique d'un rake receiver comportant six fingers. Chaque trajet est traité par un module particulier (finger). La décision, portant sur la valeur du symbole émis, est prise en combinant linéairement les symboles décodés au sein des différents fingers.*

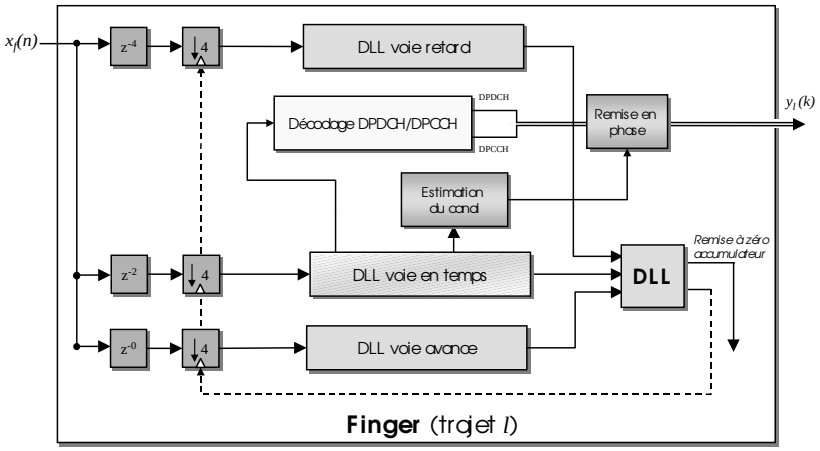

FIG. 4.4 – *Synoptique d'un finger d'un rake receicer. Ce module se charge de la synchronisation (DLL sur les voies en retard, en temps et en avance), de l'estimation du canal et du décodage des données.*

4.2 Implantation d'un récepteur WCDMA sur DART

Dans cette section, nous présentons les résultats d'implémentations des parties calculatoires de ce récepteur WCDMA. Celles-ci incluent le filtrage, le décodage des données, l'estimation du canal, la synchronisation et la combinaison des multi-trajets. Ces applications sont ici implémentées indépendamment les unes des autres. Elles ne communiquent que par le biais des mémoires et sont exécutées sur un flux continu de données. Les reconfigurations intervenant entre chaque application seront examinées dans la section suivante.

4.2.1 Le filtre de réception

Le filtre de réception traite des données complexes dont les parties réelles et imaginaires sont codées sur 8 bits. Les coefficients du filtre sont réels et codés sur 8 bits également. Les filtrages sur les parties réelles et imaginaires des données étant parfaitement symétriques, nous avons implémenté le filtre de réception en exploitant les possibilités SWP de DART. Ceci implique un placement consciencieux des données en mémoire. Nous considérerons que

les 8 bits de poids fort des données stockées en mémoire représentent leur partie réelle et les 8 bits de poids faible, leur partie imaginaire. Ce format de données est respecté tout au long de la chaîne de réception (Fig. 4.5).

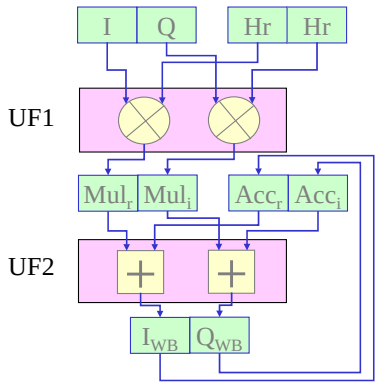

FIG. 4.5 – *Traitement SWP du filtre de réception. Chaque opérateur traite en parallèle les parties haute et basse des données qu'il reçoit sur ses entrées. Celles-ci représentent respectivement les parties réelle et imaginaire du signal. En sortie des opérateurs, l'organisation des données est maintenue.*

Exploitation du parallélisme d'opérations

Les applications de filtrage disposant d'un degré de parallélisme d'opérations important, elles sont typiquement implémentées en exploitant cette propriété. Cette mise en œuvre "naturelle " des 1966 millions de multiplications-accumulations requis par cette application exploite les 6 DPRs de DART. L'ordre du parallélisme d'opérations disponible est alors de 24. Les 64 opérations SWP 8 bits de ce filtre n'étant pas un multiple de 24, la régularité de l'application ne peut être parfaitement reproduite au niveau de l'architecture et le taux d'utilisation des ressources n'est pas optimal (88%). L'usage du SWP permet cependant de disposer d'une puissance de calcul suffisante pour implémenter en temps réel ce filtre de réception (Fig. 4.5). La puissance de calcul développée par un *cluster* de DART dans ce cas de figure est de 5.5 GOPS.

La faible régularité de l'architecture générée limite également l'efficacité du SCMD. Celui-ci ne permet de réduire que de 36% le volume de configuration requis pour traiter cette application. 12 instructions sont nécessaires à la spécification du chemin de données de DART pour le filtre de réception. Outre ces instructions de configuration, une instruction de *reset* doit par ailleurs être envoyée tous les 6 cycles afin d'initialiser le registre d'accumulation entre chaque échantillon.

Le traitement en temps réel de cette application se traduit par un taux moyen d'utilisation d'un *cluster* de DART de 70%. La consommation de puissance lors de l'exécution avoisine les 143 mW. L'efficacité énergétique de DART sur ce filtrage est donc de 27.5 MOPS/mW. Il est à noter que l'usage intensif des mémoires inhérent à cette application se traduit par une dispersion de près de 28% de l'énergie consommée dans les accès mémoires et 17% dans la génération d'adresses. Malgré le très grand nombre d'instructions de *reset* devant être distribuées vers les DPRs (1 instruction tous les 6 cycles), la méthode de reconfiguration adoptée dans DART permet de minimiser le coût lié à cette opération. Grâce à la minimisation du nombre de reconfigurations et du volume de données devant être lues et décodées pour

gérer l'architecture, la puissance dissipée dans le contrôleur du *cluster* (0.9 mW) est en effet inférieure à 1% de la puissance consommée par le circuit.

Exploitation du parallélisme de tâches

L'identité entre les équations 4.5 et 4.6, décrivant le comportement du filtre de réception, permet cependant d'envisager l'utilisation d'une autre forme de parallélisme. Il est en effet possible d'implémenter le filtre de réception en exploitant un parallélisme de tâches. Celui-ci se traduit par le filtrage concurrent de plusieurs échantillons, ici au nombre de 6. Cette solution est rendue possible sur DART par la structure des *clusters*, et en particulier par le réseau segmenté interconnectant les différents DPRs.

$$Y(n) = \sum_{i=0}^{N-1} X(n+i)H(N-i) \forall n \in [0, Nech_{slot}] \tag{4.5}$$

$$\forall p \in [0, \frac{Nech_{slot}}{6}], \begin{cases} Y(6p) = \sum_{i=0}^{N-1} X(6p+i)H(N-i) \\ Y(6p-1) = \sum_{i=0}^{N-1} X(6p-1+i)H(N-i) \\ Y(6p-2) = \sum_{i=0}^{N-1} X(6p-2+i)H(N-i) \\ Y(6p-3) = \sum_{i=0}^{N-1} X(6p-3+i)H(N-i) \\ Y(6p-4) = \sum_{i=0}^{N-1} X(6p-4+i)H(N-i) \\ Y(6p-5) = \sum_{i=0}^{N-1} X(6p-5+i)H(N-i) \end{cases} \tag{4.6}$$

Les 6 filtres ainsi mis en œuvre travaillent en mode SWP avec les mêmes coefficients, sur des données retardées de 0 à 5 échantillons. Les 4 UFs intégrées au sein des DPRs permettent de mettre en œuvre chacun des filtres avec un parallélisme d'opérations de degré 4^1. Les filtres travaillant sur les mêmes coefficients, ces derniers sont partagés entre les différents DPRs via les bus segmentés. Dans le même ordre d'idée, les retards entre les différentes données manipulées sur les DPRs sont implémentés par le biais des registres internes aux DPRs. Grâce à ces chaînes de retard, les 6 filtres sont approvisionnés en données suite à un unique accès mémoire. Les registres internes aux DPRs et le réseau segmenté permettent donc d'exploiter le parallélisme de tâches inhérent à cette application et de réduire d'un facteur 6 le nombre d'accès aux mémoires de données. L'architecture générée pour cette application peut être modélisée par la figure 4.6.

La figure 4.7 représente la distribution de la consommation entre les différents éléments de DART. Comme précédemment, le coût énergétique de la distribution du contrôle dans l'architecture reste marginal et inférieur à 1%. Il ressort de cette figure que le partage des données entre les DPRs permet de limiter le coût des accès aux mémoires locales. Bien que faisant un usage intensif des données, seul 9% de l'énergie consommée lors du filtrage est issue des mémoires locales. Dans le même temps, le coût énergétique de la génération d'adresses est minimisé pour atteindre 5% de la consommation globale du composant. Du fait de la bonne localité temporelle des données, la part de la consommation dissipée dans la mémoire du *cluster* est elle aussi maîtrisée (4% de la consommation globale). En absorbant 81% de l'énergie consommée par DART, les opérateurs constituent donc la principale source de consommation. Par opérateurs nous entendons ici l'association des unités fonctionnelles, des registres et de la logique de multiplexage de DART. La distribution de la consommation entre ces différents éléments est représentées sur la figure 4.8. La minimisation des gaspillages d'énergie se traduit au final par une efficacité énergétique atteignant 40.1MOPS/mW.

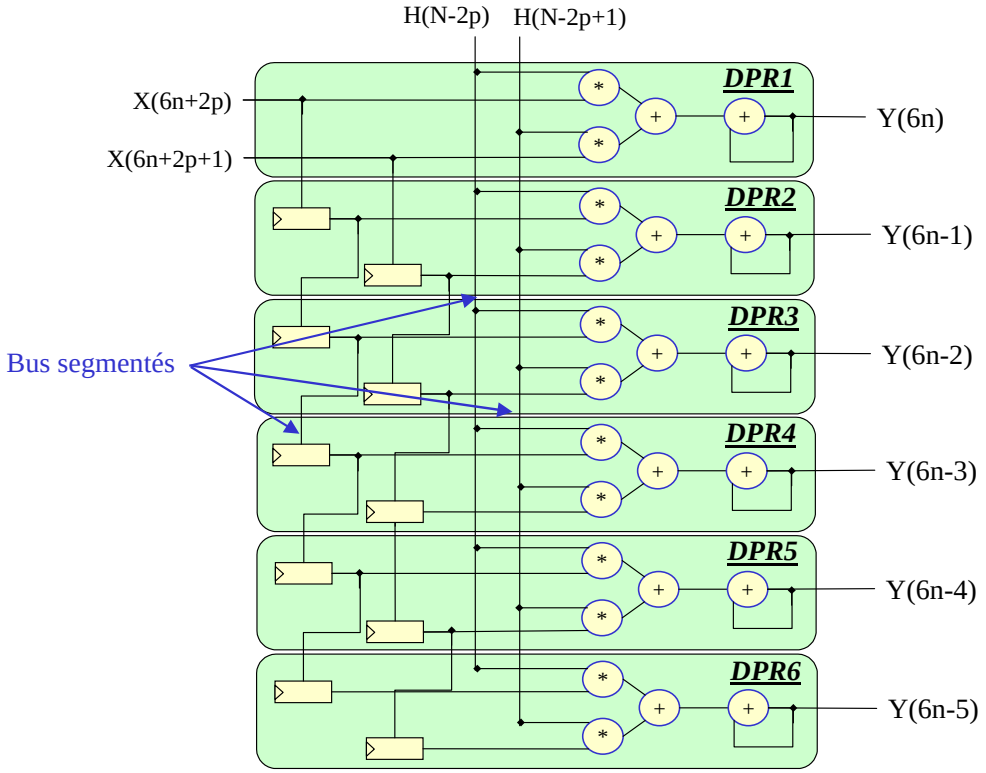

FIG. 4.6 – *Implémentation du filtrage exploitant le parallélisme de tâches. Chaque DPR est chargé de filtrer un échantillon par les mêmes coefficients. Ceux-ci sont partagés entre les DPRs grâce au réseau segmenté. Les échantillons à filtrer sont retardés d'un DPR à l'autre par le biais de chaînes de retard construites en interconnectant les registres internes aux DPRs.*

Outre la réduction de la consommation, cette solution s'accompagne d'une augmentation du taux d'utilisation des ressources. En effet, en exploitant un parallélisme d'opérations d'ordre 4, les unités fonctionnelles des DPRs sont utilisées à 100 %. Le gain en performance est dès lors non négligeable puisque la puissance de calcul délivrée par un *cluster* de DART atteint 6.2GOPS. Le taux d'utilisation du *cluster* par le filtre de réception est alors de 63%.

4.2.2 Le décodage des données et la combinaison des multi-trajets

Un autre traitement clé d'un récepteur WCDMA est le décodage des données (*complex despreading*). Celui-ci vise à retrouver, dans les échantillons filtrés, l'information émise par la station de base. Ce décodage des données est réalisé au niveau de chacun des *fingers* sur les données filtrées et synchronisées (cf §4.2.3). Ce désembrouillage est réalisé par la multiplication complexe du signal d'entrée $s_e(n)$ et des conjugués des codes de Kasami et OVSF utilisés à l'émission. Afin de limiter le nombre de multiplications entre le signal et le code, le produit des codes OVSF et de Kasami est réalisé par le biais d'opérations logiques[2], avant de désembrouiller le signal. Ainsi, le décodage des données au niveau du terminal mobile peut être représenté par la figure 4.9. La cadence des échantillons correspondant à la fréquence

[1] Ce degré de parallélisme est exhibé, sous SUIF, par un déroulage partiel d'ordre 2.

[2] Ce problème concernant la synthèse de cette opération sur le FPGA embarqué, nous ne l'examinerons pas dans ce chapitre.

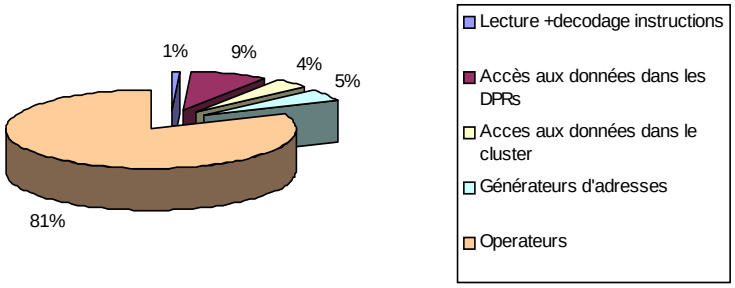

FIG. 4.7 – *Distribution de la consommation de DART lors d'un filtrage. Les 97 mW consommés lors de l'exécution du filtrage sont consommés à 81% par les unités fonctionnelles. Les accès aux mémoires (locales et globale) sont quant à eux responsables de 18% de l'énergie consommmée. Les lectures et décodages d'instructions n'ont quant à eux pas d'influences significatives sur la consommation (1% de l'énergie totale consommée).*

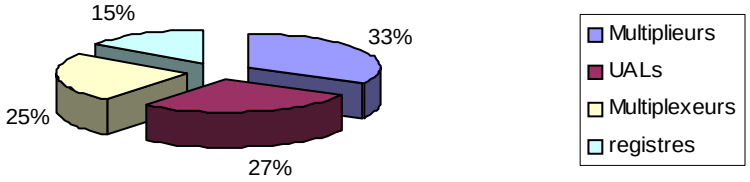

FIG. 4.8 – *Répartition de la consommation entre les différents éléments constituant les opérateurs de DART : Du fait de la complexité des multiplexeurs en entrée des unités fonctionnelles, qui doivent réaliser les extensions de signe, les alignement de données, ..., ces modules consomment une large part de l'énergie absorbée par les opérateurs (25%).*

chip ($F_c = 3.84MHz$), la complexité associée au décodage des données est, pour chaque *finger*, de 30.72 MOPS.

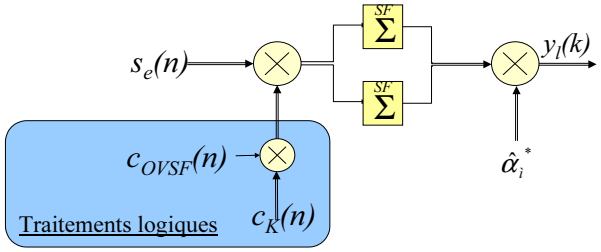

FIG. 4.9 – *Synoptique de la partie décodage des données au niveau d'un terminal mobile. Le produit des codes OVSF et de Kasami est réalisé par le biais d'opérations logiques. Le produit de ce résultat et du signal d'entrée est accumulé sur SF cycles. La multiplication traitant des données complexes, 8 opérations sont nécessaires à ce premier traitement. Le résultat de cette opération est finalement adapté par le biais d'une nouvelle multiplication complexe, afin de tenir compte des distorsions de l'onde radio-électrique survenues lors du trajet entre la station de base et le terminal mobile.*

Les données étant codées sur 8 bits, l'application peut être implémentée sur DART en exploitant le parallélisme de données. Contrairement au cas du filtrage évoqué précédemment, les traitements appliqués sur les voies réelles et imaginaires ne sont cependant cette fois pas

totalement symétriques. En particulier, la multiplication de 2 nombres complexes, représentée par l'équation 4.7, fait apparaître des motifs de calcul différents pour le traitement de la partie réelle et imaginaire du produit. Le traitement de la partie réelle nécessite 2 multiplications et une soustraction, lorsque le traitement de la partie imaginaire remplace la soustraction par une addition. Un opérateur SWP appliquant nécessairement la même opération sur les parties haute et basse des entrées qu'il manipule, le traitement des opérations d'addition et de soustraction nécessite, a priori, deux opérateurs.

$$C = A * B \equiv \begin{cases} C_{Re} = A_{Re} * B_{re} - A_{Im} * B_{Im} \\ C_{Im} = A_{Re} * B_{Im} + A_{Im} * B_{Re} \end{cases} \tag{4.7}$$

Afin d'optimiser l'utilisation des ressources, ce problème a été contourné dans DART en remplaçant la soustraction par l'addition d'une valeur inversée (Eq. 4.8). L'introduction d'une symétrie parfaite entre les voies réelles et imaginaires implique donc une duplication des codes en mémoire puisque le code C_q doit être stocké de la même façon que sa valeur inversée $-C_q$. Il est donc indispensable de placer en mémoire les couples de coefficients $\{C_p, C_p\}$ et $\{C_q, -C_q\}$ pour pouvoir implémenter le décodage des données en mode SWP. L'architecture générée pour décoder les données d'un trajet, ainsi que l'organisation des données manipulées, est représentée sur la figure 4.10.

$$I_{NB}(k) = \sum_{n=0}^{SF-1} I_{WB}(k+n) * C_p(k) + Q_{WB}(k+n) * (-C_q(n)) \tag{4.8}$$
$$Q_{NB}(k) = \sum_{n=0}^{SF-1} I_{WB}(k+n) * C_q(k) + Q_{WB}(k+n) * C_p(n)$$

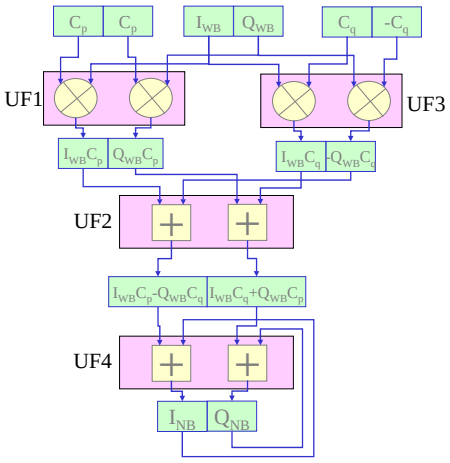

FIG. 4.10 – *Implémentation du complex despreading en mode SWP.*

Le parallélisme de tâches inhérent au *rake receiver* a été exploité afin d'implémenter au mieux le décodage des données. Ainsi, chaque *finger* est implémenté sur un seul DPR. Cette réduction du degré de parallélisme d'opérations exploité, par rapport à une implémentation sur 6 DPRs de chaque *finger*, autorise la définition d'une architecture dont la régularité est optimale. L'emploi du SCMD permet de spécifier concurremment les configurations des 6

DPRs et ainsi de réduire le coût associé à la phase de reconfiguration. Dans ce cas de figure, seules 3 instructions permettent de configurer le *cluster* en vue de l'implémentation des 6 *fingers* du *rake receiver*[3]. Outre ces 3 instructions de configuration, 1 instruction de *reset* devra par ailleurs permettre de réaliser la mise à zéro des accumulateurs tous les 256 cycles.

L'implémentation en temps réel de ce décodage des données ne consomme que 6mW. La distribution de la consommation entre les différents éléments de DART est représentée sur la figure 4.11. Du fait de l'absence de partage de données entre les différents DPRs, la proportion d'énergie consommée dans les accès mémoires est supérieure à celle observée lors du filtrage. Par ailleurs, les données en sortie du filtre n'étant lues qu'une fois, leur localité temporelle est minime. En conséquence, les accès à la mémoire du *cluster* prennent une part non négligeable dans la consommation globale du circuit. Chaque accès à une donnée dans le DPR nécessite en effet un transfert préalable entre la mémoire globale et la mémoire locale. Bien que réduite, la proportion de l'énergie consommée dans les opérateurs reste prédominante (59%), garantissant une très bonne efficacité énergétique de DART sur ce décodage des données (30.9 MOPS/mW). Plus encore que pour le filtrage, du fait du nombre limité d'instructions de *reset*, le contrôle de l'architecture se traduit par une pénalité énergétique négligeable.

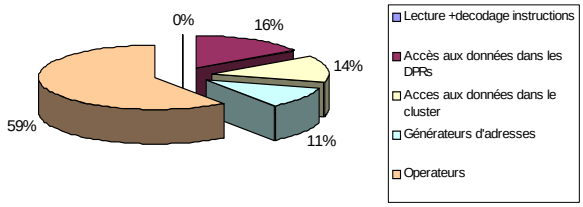

FIG. 4.11 – *Distribution de la consommation lors du décodage des données. L'absence de partage des données et la très faible localité temporelle des données ne permettent pas d'optimiser la consommation de la hiérarchie mémoire. La minimisation du coût énergétique du contrôle permet toutefois de localiser l'essentiel (59%) de la dissipation d'énergie dans les opérateurs et de garantir une excellente efficacité énergétique (30.9MOPS/mW).*

Suite au décodage des données issues des différents trajets, les symboles doivent être combinés afin de réaliser une décision optimale. Chaque symbole décodé est ainsi multiplié par l'estimation de l'amplitude complexe du canal (cf §4.2.4), et additionné aux symboles correspondant aux autres trajets. La complexité de cette opération est extrêmement faible puisque les 6 multiplications et l'addition finale ne sont réalisées qu'à la cadence des *slots* ($F_{slot} = 1.5kHz$). La consommation induite par cette combinaison des multi-trajets est donc extrêmement faible. Elle se limite à $23\mu W$.

4.2.3 La synchronisation

Les opérations de corrélation utilisées pour décoder les données issues des différents trajets requièrent une synchronisation précise entre les codes générés au sein du récepteur et le code contenu dans le signal transmis. Cette synchronisation est réalisée en deux temps. En premier lieu, une estimation du retard associé à chaque trajet est réalisée avec une précision de $\pm T_c/2$. Cette estimation grossière est mise en œuvre en recherchant de manière séquentielle, sur des fenêtres de largeur T_c, des maximums dans la fonction d'autocorrélation entre le signal reçu

[3]Les DPRs travaillant de manière autonome, il est inutile de configurer le réseau segmenté entre les DPRs.

et le code généré en interne. Les L multi-trajets dont les puissances sont les plus élevées sont ensuite affectés aux *fingers* du *rake receiver* afin de réaliser le décodage des données. Cette synchronisation grossière n'est réalisée qu'entre chaque période de cohérence du canal. Sa complexité est donc très limitée. Du fait de son faible impact sur les performances du système cette synchronisation grossière ne sera pas examinée dans cette étude.

Suite à cette synchronisation grossière, une synchronisation fine est appliquée, pour chaque trajet, avant de décoder les données dans les différents *fingers*. Celle-ci a pour but de déterminer l'instant initial d'échantilonnage, avec une précision inférieure au temps chip T_c, et est mise en œuvre par le biais d'une boucle d'asservissement de retard (DLL : Delay Locked Loop). La structure de cette DLL est représentée sur la figure 4.12. Pour cette synchronisation, le signal est divisé en trois lignes. La première est dite en avance (*early*) et correspond au signal d'entrée sous-échantillonné d'un facteur 4. La seconde est la ligne dite en temps (*on-time*). Elle correspond au signal d'entrée retardé de 2 échantillons et sous-échantillonné d'un facteur 4. La dernière ligne est dite en retard (*late*) et correspond au signal d'entrée retardé de 4 échantillons et sous-échantillonné comme les deux précédentes lignes. Sur chacune de ces voies, la valeur de la fonction d'autocorrélation correspondant au signal traité est calculée en multipliant le signal par le code de Kasami puis le code OVSF, et en accumulant le résultat de cette multiplication sur 256 cycles. Les échantillons en sortie de l'accumulateur évoluent au rythme symbole ($T_s = SF.T_c$).

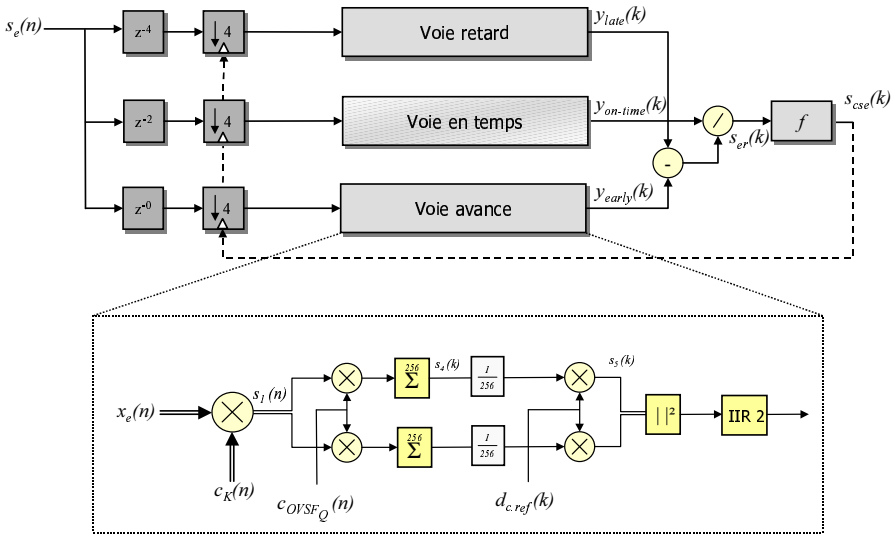

FIG. 4.12 – *Synoptique de la boucle d'asservissement de retard. Chaque voie applique le même traitement sur des séquences de données retardées et sous-échantillonnées, afin de déterminer l'instant initial d'échantillonnage (S_{cse}) du flot de données à décoder, avec une précision inférieure à $\pm T_c/2$. Cette commande des sous-échantillonneurs est proportionnel au signal d'erreur s_{er} et prend des valeurs entières comprises entre -2 et 2.*

Les modules des signaux complexes résultant de ce calcul sont déterminés et filtrés, par un filtre IIR du premier ordre, afin d'isoler leur composante basse-fréquence. Un signal d'erreur répondant à l'équation 4.9 est alors calculé. Ce signal d'erreur est ensuite comparé à un seuil afin de déterminer si les codes sont en phase ou s'il est nécessaire d'avancer ou de retarder le signal devant être décodé dans le *finger*. Sachant que ce signal de commande spécifie la

position de l'échantillon à traiter, il doit être représenté par un entier compris entre -2 et 2. Ce calcul de synchronisation est effectué une fois par *slot* ($F_{slot} = 1.5kHz$).

$$s_{er}(k) = \frac{y_{early} - y_{late}}{y_{on-time}} \tag{4.9}$$

Afin d'implémenter cette synchronisation, nous avons isolé les étapes de calcul selon la fréquence des signaux manipulés. Ainsi, deux reconfigurations de DART sont nécessaires à la synchronisation. La première calcule la corrélation entre le signal d'entrée et les codes générés en interne, pour les 6 bits pilotes, à la fréquence *chip*($3.84MHz$). La seconde configuration traite l'extraction des bits de contrôle et l'isolation de la composante basse fréquence du module. Elle manipule des données cadencées à la fréquence symbole ($F_{symbole} = 15kHz$). Les deux configurations traitent des données 8 bits.

Les traitements à la fréquence chip

La première configuration, constituée d'une multiplication complexe suivie d'une accumulation, est identique à celle générée pour le décodage des données. De même que précédemment, le produit des codes de Kasami et des codes OVSF est réalisé par le biais d'opérations logiques et le parallélisme de tâches est privilégié au détriment du parallélisme d'opérations. Ainsi, les voies *early*, *on-time* et *late* de chaque *finger* sont implémentées sur trois DPRs différents et 2 *fingers* sont traités en parallèle. En pratique, les DPRs 0, 1 et 2 implémentent successivement les *fingers* 0, 2 et 4, et les DPRs 3,4 et 5 les *fingers* 1, 3 et 5. En conséquence, les ensembles constitués par les 3 premiers et les 3 derniers DPRs disposent de la même configuration. Ainsi, 9 instructions suffisent à la spécification des configurations des 6 DPRs. À ces 9 instructions de configuration s'ajoute une instruction de *reset* devant être envoyée tout les 256 cycles. Du fait des décalages temporels entre les voies, ces instructions de *reset* ne peuvent cependant pas cette fois être envoyées simultanément à tous les DPRs. Les accumulateurs des 3 voies sont ainsi remis à zéros via des instructions envoyées à des instants disjoints. Le parallélisme de données est par ailleurs exploité par le biais des capacités SWP de DART.

Outre la réduction du volume de configuration, cette option est intéressante du point de vue de la minimisation du nombre d'accès mémoire. En effet, les 3 voies d'un *finger* étant implémentées en parallèle, les codes peuvent être partagés entre les voies sans nécessiter de duplication des données. Le code lu pour la corrélation de la voie *early* peut ainsi être utilisé dans les voies *on-time* et *late* par le biais d'un seul accès mémoire, grâce au réseau segmenté. Dans le même ordre d'idée, le retard entre les différentes voies étant de courte durée, ils peuvent être implémentés par le biais des registres internes aux DPRs et du réseau segmenté. Le signal d'entrée peut ainsi être partagé entre les 3 voies afin de réduire (d'un facteur 3) le nombre d'accès mémoire.

La figure 4.13 représente la distribution de la consommation entre les différents éléments d'un *cluster* de DART sur cette application. Le pourcentage de consommation pouvant être considéré comme indispensable au bon fonctionnement d'une application, issu des opérateurs, atteint sur cette application 63%. La part de la consommation induite par le contrôle de l'application reste marginale et inférieure à 1 %. L'optimisation du partage des données entre les DPRs, de même que la faible localité temporelle des données explique la part importante de la consommation issue des accès à la mémoire du *cluster*. Celle-ci s'élève à 20% de l'énergie totale consommée. Les 331 MOPS requis par cette application se traduisent par une consommation moyenne de 10.9 mW, soit une efficacité énergétique de 30.5MOPS/mW.

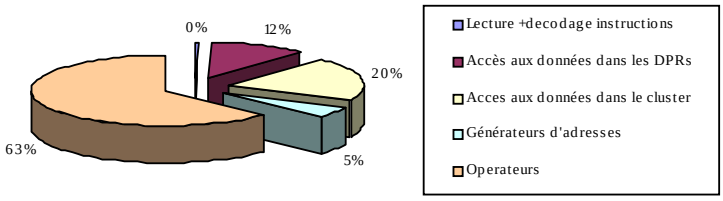

FIG. 4.13 – *Distribution de la consommation de DART lors du traitement de la synchronisation à la fréquence chip. La minimisation des gaspillages d'énergie permet de localiser l'essentiel de la consommation dans les opérateurs (63%). Le partage des données entre les DPRs, ainsi que la faible localité temporelle des données expliquent la part non négligeable de la consommation issue des accès à la mémoire du cluster (20%). L'efficacité énergétique de DART sur cette application s'élève à 30.5 MOPS/mW.*

Les traitements à la fréquence symbole

L'essentiel de la complexité de la tâches de synchronisation est localisée dans les traitements opérant à la fréquence *chip* (331 MOPS). La complexité des traitements cadencés à la fréquence symbole se limite quant à elle à 7.2 MOPS. La difficulté inhérente à l'implémentation de ce traitement tient dans le degré de parallélisme de données qui évolue entre les différentes étapes du graphe flot de données. Le DFG représenté sur la figure 4.14 distingue la multiplication par les bits de référence et la mise au carré du calcul de module, qui peuvent être implémentées en mode SWP, de la fin du calcul du modulo et du filtre IIR dont les parallélismes de données sont insuffisants pour exploiter les capacités SWP de DART.

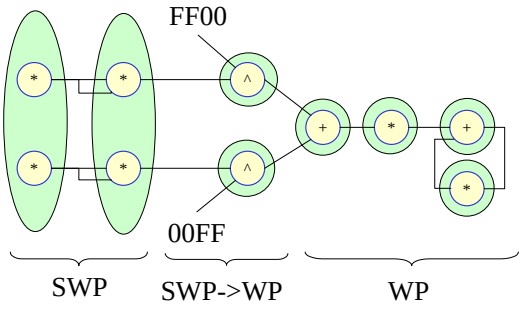

FIG. 4.14 – *DFG du traitement de la synchronisation à la fréquence symbole. La multiplication des signaux d'entrées par le code de référence, de même que la mise au carré de ce produit, nécessaire au calcul du modulo, peuvent être traités en mode SWP. La fin du calcul du modulo et le filtrage IIR sont quand à eux réalisés au niveau mot (WP :Word Processing). Le passage SWP → WP, nécessite une phase de conversion qui monopolise deux opérateurs traitant ces opérations logiques. Sur ce schéma, les opérations apparaissent en clair. Les opérateurs qui les implémentent sont représentés en arrière-plan.*

Entre ces deux phases de calcul, il est ainsi indispensable d'insérer une étape visant à transformer un mot de 32 bits contenant deux données 16 bits en deux mots pouvant être manipulés de façon indépendante. Cette étape est réalisée par le biais d'un masquage par les valeurs $FFFF0000_h$ et $0000FFFF_h$, respectivement pour les bits de poids fort et les bits de poids faible.

Les 10 opérations de ce traitement peuvent être implémentées sur 2 DPRs de DART. Ainsi,

les trois voies d'un *finger* peuvent être traitées concurremment afin de réduire le nombre d'accès aux mémoires. Dans ce cas de figure, il est en effet possible de partager les données de références entre les différentes voies du *finger*, de même que les coefficients du filtre IIR. Les données en entrées des trois voies de la synchronisation sont obtenues par un accès unique à la mémoire de données, qui sert de point d'entrée à une chaîne de retard.

La spécification de la configuration de DART correspondant à ce traitement nécessite 9 cycles. Compte-tenu du faible nombre de bits pilotes, 6 cycles suffisent à la synchronisation d'un *finger*. Le traitement à la fréquence symbole du *rake receiver* ne nécessite donc que 36 cycles d'exécution. Le ratio entre temps de configuration et temps de traitement (36/9=4) est par conséquent très sensiblement réduit par rapport aux applications évoquées précédemment. Dès lors, la part de la consommation induite par la distribution des informations de contrôle n'est plus négligeable. Elle atteint sur cette application 9% de la consommation totale du *cluster* (Fig. 4.15). Du fait de la faible complexité du traitement de la synchronisation à la fréquence symbole, la consommation de puissance inhérente à ce traitement ne représente que 0.1% des 10.9mW consommés lors de la synchronisation.

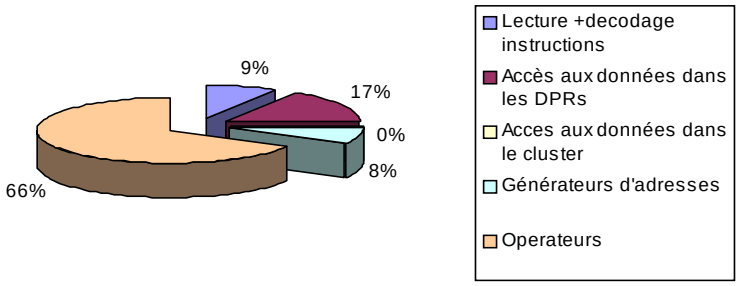

FIG. 4.15 – *Distribution de la consommation de DART lors du traitement de la synchronisation à la fréquence symbole. Le faible temps d'utilisation de la configuration explique la part non négligeable d'énergie consommée dans le contrôleur de cluster. Le partage des données permet quant à lui de limiter le coût énergétique des accès aux données et de la génération d'adresses (25%). Les opérateurs consomment une fois encore l'essentiel de l'énergie absorbée par le circuit (66%).*

4.2.4 L'estimation du canal

L'amplitude complexe du canal est estimée, pour chaque trajet, à chaque nouveau *slot* ($F_{slot} = 1.5kHz$). En d'autres termes, le canal est supposé invariant sur la durée d'un *slot*. Cette estimation consiste à corréler les bits pilotes reçus avec une séquence déterministe reconstituée au niveau du récepteur. Cette estimation se base donc sur la connaissance à priori de la valeur des bits pilotes, au nombre de 6 dans chaque *slot*.

Le début du traitement est identique à celui appliqué à la voie *on-time* de la synchronisation. Ce traitement réalise la multiplication complexe avec le code de Kasami puis avec le code OVSF et accumule le résultat sur 256 échantillons. Afin d'estimer l'amplitude complexe du canal, il est ainsi possible de réutiliser les résultats préalablement calculés et stockés en mémoire. La suite du traitement vise à éliminer les données de contrôle présentes au sein du signal s_4. Pour cela, la corrélation de ce signal avec les bits de référence est calculée puis les

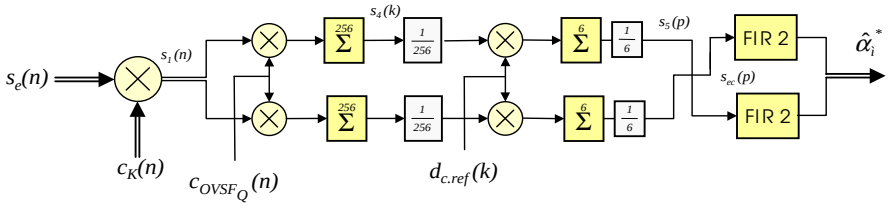

FIG. 4.16 – *Synoptique du traitement d'estimation de canal. La première partie du traitement corrèle le signal d'entrée avec le code OVSF et le code de Kasami. Les données de contrôle présentes au sein du signal sont ensuite éliminées par la corrélation de ce signal avec les bits de référence. Afin d'estimer le conjugué de l'amplitude complexe du canal, ses parties réelle et imaginaire sont échangées avant d'attaquer un filtre moyenneur.*

voies réelle et imaginaire de ce signal sont échangées afin d'estimer la valeur du conjugué de l'amplitude complexe du canal. Cette valeur estimée est finalement moyennée, avec la valeur estimée pour le *slot* précédent, par le biais d'un filtre FIR composé de deux cellules dont les coefficients sont égaux à 0.5. La complexité de ce traitement est sans commune mesure par rapport à ceux étudiés précédemment (0.045MOPS).

La simplicité des coefficients du filtre FIR autorise son traitement en une seule opération. L'identité 4.10 permet en effet d'implémenter ce filtrage sous la forme d'une addition suivie d'un décalage de 1 bit à droite (i.e. division par deux du résultat de l'addition). La multiplication par les bits de référence ainsi que les accumulations seront quant à elles implémentées en parallèle en exploitant les capacités SWP de DART. Trois opérateurs suffisent pour implémenter cet algorithme de manière optimale (en terme de performance). Dès lors, la mise en œuvre de l'estimation des canaux des 6 trajets d'un *rake receiver*, maximisant le taux d'utilisation des ressources, nécessite 5 DPRs. L'inconvénient majeur associé à cette implémentation est que le traitement des différents trajets est distribué entre les différents DPRs. De ce fait, la configuration générée est très irrégulière. Le SCMD étant dans ce cas peu exploité, 12 instructions sont nécessaires à la configuration des 5 DPRs et de leurs interconnexions.

$$y[n] = \frac{1}{2}a[n] + \frac{1}{2}b[n] = (a[n] + b[n]) >> 2 \tag{4.10}$$

Afin de réduire le volume de configuration, une implémentation sur 6 DPRs est possible. Dans ce cas de figure, chaque DPR dispose de la même configuration. L'exploitation du SCMD permet dans ce cas de réduire à 3 le nombre d'instructions de configuration. En contrepartie, le taux d'utilisation des ressources s'amenuise, pour atteindre 75%. Le faible parallélisme d'opérations de cette application conduit donc à une utilisation des ressources non optimale. Ceci ne se traduit en revanche par aucune perte d'efficacité. Cette solution à en effet l'avantage de réduire le temps de configuration (de 12 à 3 cycles), sans introduire de pénalité du point de vue énergétique. En effet, l'emploi d'horloges gardées permet d'isoler les unités fonctionnelles non utilisées de chaque DPR du reste du composant, éliminant ainsi tout risque de gaspillage d'énergie. L'utilisation de 6 DPRs, par rapport à la solution initiale basée sur l'exploitation de 5 DPRs, permet d'augmenter l'efficacité énergétique associée à cette application (de 28.9 MOPS/mW à 33.7 MOPS/mW) en minimisant le coût de la distribution des informations de contrôle.

4.2.5 Synthèse

Dans cette section, le comportement de DART sur les différents algorithmes mis en œuvre dans un récepteur WCDMA a été analysé. Ce récepteur a été implémenté en considérant un traitement par *slot*, i.e. les différentes phases de traitement opèrent sur l'ensemble des données contenues dans un *slot*. Cette solution permet de minimiser le nombre de reconfigurations (Fig. 4.17), et donc le surcoût associé à cette opération, tout en limitant la latence séparant la réception du signal de son décodage à 0.6 ms.

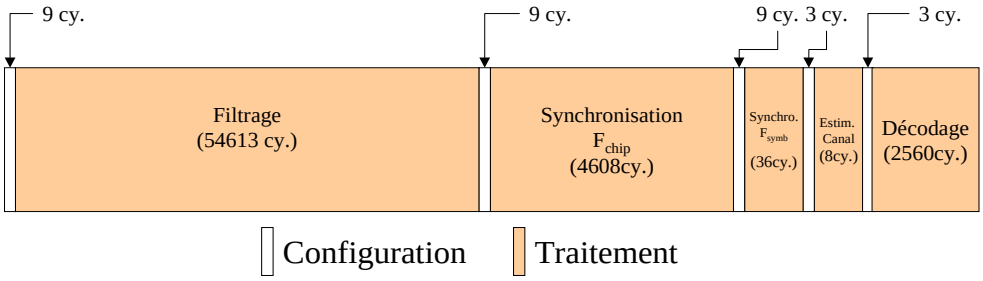

FIG. 4.17 – *Reconfigurations de DART lors de la réception d'un slot.*

Du fait de la flexibilité de DART, les différents motifs de calcul exploités dans ce récepteur, bien que variés, sont implémentés très efficacement. La puissance de calcul effective de DART sur ce récepteur WCDMA varie de 5.5 à 6.2 GOPS, ce qui conduit à un taux d'utilisation du *cluster* par le récepteur WCDMA de 72.6%. De même, la hiérarchie mémoire démontre son efficacité sur une large gamme d'applications. L'introduction de cette hiérarchie, combinée à l'exploitation d'un réseau segmenté pour interconnecter les DPRs, permet en effet de partager un grand nombre de données et ainsi d'économiser de précieux accès mémoires. De ces résultats d'implémentations, il est également important de retenir l'intérêt de l'intégration de chaînes de retard au sein de l'architecture. Bien que peu influentes sur la surface du circuit, celles-ci permettent en effet de réduire très sensiblement la quantité d'accès mémoire, en particulier lorsqu'un parallélisme de tâches est exploité (e.g. filtrage, synchronisation, ...).

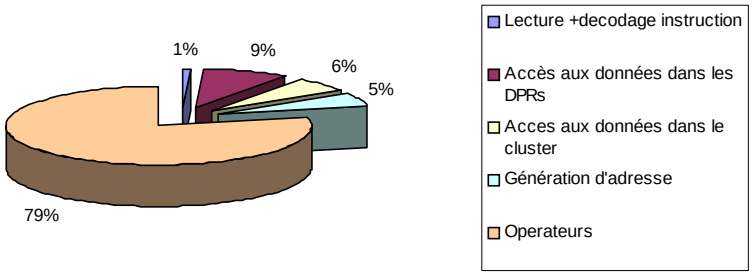

FIG. 4.18 – *Distribution de la consommation de DART lors du traitement du récepteur WCDMA. Le coût de la distribution du contrôle dans DART peut être considérée comme globalement négligeable (1%). En maîtrisant par ailleurs le coût des accès aux données dans les DPRs et dans le cluster (15%), ainsi que celui de la génération d'adresses (5%), l'essentiel de la consommation d'énergie de DART est localisée dans les opérateurs (79%). L'efficacité énergétique de DART sur le récepteur WCDMA atteint ainsi 38.8 MOPS/mW.*

Bien que la part de consommation induite par le contrôleur du *cluster* soit extrêmement

faible, la lecture et le décodage des instructions consomment 0.9mW (Fig. 4.18). Il est à noter que cette consommation est principalement issue des instructions de *reset*, et non des instructions de configuration. Une évolution souhaitable de DART serait donc d'introduire des mécanismes permettant de limiter le coût énergétique de ces instructions. Une solution simple permettant de répondre à ce besoin consiste à exploiter un *buffer* stockant les dernières instructions lues par le contrôleur du *cluster*. Cette solution peu coûteuse permettrait de réduire d'un facteur 5 le coût énergétique de ces instructions de *reset*.

La consommation moyenne de DART lors de ce traitement est de 114.8 mW. La figure 4.18 résume la distribution de la consommation entre les différents éléments de l'architecture. L'optimisation de la hiérarchie mémoire, ainsi que la minimisation du coût associé au contrôle de l'architecture assurent une efficacité énergétique moyenne de 38.8 MOPS/mW. Cette efficacité énergétique, très supérieure à celle proposée par les architectures concurrentes de DART (cf. section 4.3), s'explique notamment par la minimisation des gaspillages d'énergie qui se traduit par une exploitation quasi-optimale des opérateurs (cf. §1.3.1). Bien qu'optimisés en consommation, notamment pour les traitements SWP, ceux-ci absorbent en effet 79% de l'énergie consommée par le composant.

4.3 Positionnement de DART

Dans cette section nous comparons les performances et l'efficacité énergétique de DART à celles d'architectures représentatives des solutions reconfigurables au niveau logique et système. Les premières sont représentées par le FPGA Xilinx Virtex 200E et les secondes par le processeur de traitement du signal TMS320C64x de la société Texas Instruments.

4.3.1 Les architectures concurrentes

La famille de DSP TMS320C64x proposée par Texas instruments se base sur une architecture VLIW théoriquement capable d'exploiter un parallélisme d'instructions de degré 8 [79]. Cette architecture dispose en effet de 2 multiplieurs et de 6 UALs, réparties au sein de 2 *clusters*. 2 des UALs sont principalement exploitées à des fins de génération d'adresses. Ces 8 unités de calcul se contrôlent, à chaque cycle, par le biais d'instructions de 256 bits. Outre le parallélisme d'instructions, ce DSP est également en mesure d'exploiter un parallélisme de données par le biais de techniques SWP. Les unités arithmétiques 32 bits sont susceptibles de traiter concurremment 2 opérations 16 bits ou 4 opérations 8 bits. Pour les implémentations qui suivent, les multiplieurs traiteront 4 multiplications 8 bits, renvoyant 4 résultats codés sur 16 bits. Les additionneurs manipulant les résultats de ces multiplications traiteront ensuite 2 opérations 16 bits. Les UALs manipulant des données stockées en mémoire exécuteront quant à elles 4 opérations 8 bits. Bien que le dernier né de cette famille de composant (TMS320C6414 [143]) soit théoriquement capable de délivrer 5760 MIPS, la puissance de calcul effective délivrée par ce circuit ne permet pas d'implémenter en temps-réel un récepteur WCDMA. Ce circuit est fondu sur une technologie $0.12\mu m$ et alimenté par une tension de 1.4 Volt.

Le composant Xc200E de la famille VirtexE de Xilinx a été retenu pour représenter les architectures reconfigurables au niveau logique. Ce choix est motivé par notre volonté d'étudier l'impact de la reconfiguration sur les performances du système. Pour ce faire, il est indispensable de mettre en œuvre des reconfigurations au cours du traitement du récepteur WCDMA. L'emploi de FPGAs suffisamment puissants pour implémenter dans sa globalité un récepteur

WCDMA ne nécessitant pas de telles opérations, nous avons choisi un composant dont la puissance de calcul se limite strictement à celle nécessaire au traitement du filtre de réception. Le taux d'occupation du FPGA sur cette application est de 93%. Les traitements de synchronisation, d'estimation de canal et de décodage de données seront quant à eux traités suite à une reconfiguration du composant. Deux configurations se succéderont donc sur le FPGA : Le filtrage puis le *rake receiver*. Le volume de configuration nécessaire pour ce circuit est de 1.442.016 bits. Comme DART, se composant est distribué sur une technologie $0.18\mu m$.

4.3.2 Performances des architectures sur le filtre de réception et le Rake receiver

Dans cette partie, nous présentons les résultats d'implémentations du filtre de réception et du *rake receiver* obtenus sur le DSP et le FPGA [13]. Nous considérons ici des traitements sur des flots continus de données, i.e. aucune phase de reconfiguration n'est nécessaire. L'analyse de ces résultats ainsi que l'étude de l'implémentation du récepteur complet font l'objet du paragraphe 4.3.3.

Le TMS320C64x

Le traitement du filtre de réception sur le DSP nécessite 512000 cycles d'exécution pour chaque symbole, soit 50 cycles par échantillon complexe filtré [145]. Si sont excluent les opérations de génération d'adresses et de gestion de boucle, comme il l'a été fait pour DART, la puissance de calcul effective délivrée par ce processeur est alors de 1843 MOPS. Malgré les 8 unités de calcul et leurs capacités SWP, seules 2.56 opérations de calcul sont ainsi exécutées en parallèle à chaque cycle. Le partage des unités fonctionnelles entre les opérations de calcul et les opérations de génération d'adresses ou de gestion de boucle explique la difficulté d'exploiter un parallélisme massif. En particulier, si le traitement de 8 multiplications 8 bits en parallèle ne pose aucun problème sur les 2 multiplieurs du DSP, l'exploitation des 8 produits est délicate. Ces résultats étant codés sur 16 bits, 4 UALs sont en effet nécessaires au traitement de 8 MACs par cycle (le parallélisme de données est réduit à 2 pour des données 16 bits). La gestion de boucle et les générations d'adresses nécessitant 3 UALs, le nombre de ressources est ici insuffisant. En pratique, un parallélisme de données de 2.5 a été observé. Le traitement du *rake receiver* se traduit par un taux d'utilisation du DSP de 14.3%. Le traitement du filtre en temps réel est quant à lui impossible. Il nécessiterait un accroissement de la fréquence de fonctionnement d'un facteur de 1.3.

Pour évaluer la qualité du code généré par le compilateur C, le nombre d'instructions exécutées par cycle a été déterminé. Les techniques de pipeline logiciel, utilisées par le compilateur C, exploitent au sein des boucles le parallélisme proposé par le processeur et conduisent à un code assembleur de bonne qualité. L'ILP moyen utilisé pendant cette application étant de 5.5, les performances de ce processeur sont limitées par l'architecture et non par la qualité des outils de développement.

Les consommations de puissance associées à ces implémentations ont été estimées d'après les résultats présentés dans [146]. Pour ces applications, l'estimation a été faite en considérant une activité haute 75% du temps (cœur de boucle) et basse les 25% restant. L'activité haute correspond au traitement parallèle de 8 instructions, et l'activité basse au traitement parallèle de 2 instructions (lors des phases de prologue et d'épilogue du pipeline logiciel). Un facteur d'échelle a par ailleurs été appliqué pour le calcul du filtrage, afin de tenir compte du supposé accroissement de la fréquence de fonctionnement nécessaire au respect des contraintes temps-réel de cette application. Afin de rester cohérent avec les estimations faites sur DART et

sur le Xc200E, nous ne tiendrons pas compte de la consommation des E/S de ce circuit. La taille limitée du code autorisant son stockage dans les mémoires internes du DSP, cette consommation sera par ailleurs très limitée sur notre application. Nous avons ainsi estimé la consommation du DSP à 1.48 Watt lors du filtrage, et à 1.06W lors du traitement du *rake receiver*. L'efficacité énergétique de cette architecture est alors de 2.6MOPS/mW pour le filtrage et de 1.8 MOPS/mW pour le *rake receiver*.

Le Xc200E

Les 2 filtres FIR de 64 cellules ont été implantés dans le FPGA par le biais de l'outil *SYNPlify Pro* de Synplicity et l'outil de placement routage de Xilinx. L'architecture générée occupe 72% des ressources du circuit, soit 1720 *slices* et 992 bascules. Cette solution conduit à un temps d'exécution du filtre de 38.6ns par échantillon. L'implémentation en temps réel du *rake receiver* nécessite quant à elle 851 *slices* et 20kbits de mémoire SRAM embarquée. Elle conduit à un taux d'utilisation des ressources du circuit de 27.6%. Les consommations associées à ces implémentations ont été estimées par l'outil *XPower* de Xilinx. Celles-ci s'élèvent à 670mW pour le filtrage et 180mW pour le *rake receiver*. Ces chiffres traduisent une efficacité énergétique de 5.9MOPS/mW pour le filtrage et de 3 MOPS/mW sur le *rake receiver*.

Il est à noter que la consommation estimée par *Xpower* ne tient pas compte du coût de la reconfiguration. La transmission et le stockage des 1.4Mbits de configuration du composant dépendant de l'environnement du circuit, i.e. localisation de la mémoire de configuration, architecture du contrôleur (processeur, RISC, contrôleur DMA, ...), nous ne tenterons pas d'estimer son coût dans cette étude. L'incertitude des résultats de l'estimation de la consommation sur ce composant ne pénalise cependant pas la tâche d'analyse des résultats. La part de la consommation due aux phases de reconfigurations est en effet relativement limitée dans ce type de composant, dès lors que les configurations sont maintenues sur des durées suffisamment importantes.

4.3.3 Comparaison des performances sur le récepteur complet

Densité de calcul

Si le FPGA se montre potentiellement capable de surpasser ses concurrents lors du filtrage ou du *rake receiver*, la densité de calcul proposée par ce dernier limite très sérieusement ses possibilités d'intégration dans des systèmes sur silicium. Les deux circuits examinés dans cette étude n'étant pas distribués sous forme de cœurs, les informations de surface sont délicates à obtenir. La densité de calcul de ces composants peut cependant être grossièrement estimée de manière simple. Nous avons pour cela considéré le ratio entre le nombre d'opérateurs disponibles sur l'architecture et le nombre d'opérations qu'ils exécutent par seconde. Tous les aspects relatifs aux mémoires ou à la gestion de l'architecture et des entrées/sorties ont été évincé de cette étude. Nous définissons ainsi ce critère par l'équation 4.11.

$$Densité = \frac{NbOpération.s^{-1}}{NbOp_{1bit}} \tag{4.11}$$

Dans cette équation, le paramètre $NbOpération.s^{-1}$ correspond au nombre d'opérations arithmétiques pouvant être traitées par seconde. Les calculs d'adresses n'étant pas considérés comme des opérations utiles à l'obtention d'un résultat, ils n'interviennent par ailleurs pas dans ce calcul. Le paramètre $NbOp_{1bit}$ correspond quant à lui au nombre d'opérateurs

arithmétiques, ramené à une largeur de 1 bit. Ce critère permet de prendre en compte le sur-dimensionnement de certains opérateurs. Nous avons considéré 256 opérateurs 8 bits pour le FPGA (pour un taux d'utilisation du composant de 93%) et 8 opérateurs 32 bits pour le DSP. Ce paramètre vaut pour DART la somme des 24 unités 6 bits des générateurs d'adresses, des 12 multiplieurs 16 bits et des 12 ALUs 32 bits.

Application	C64	Xc200E	DART
Filtre	12	1.9	8.6
rake receiver	14.2	0.2	7.8

TAB. 4.1 – *Densité de calcul du C64, du Xc200E et de DART sur le filtre de réception et le rake receiver. Cette densité de calcul s'exprime en Million d'Opérations Par Opérateur binaire.*

Ce tableau fait clairement apparaître la supériorité du DSP sur ses concurrents. Ce chiffre est cependant à pondérer par la technologie utilisée. Le C64 est en effet fabriqué avec une technologie $0.12\mu m$ alors que DART et le Xc200E ciblent une technologie $0.18\mu m$. Le DSP bénéficie ainsi d'une fréquence d'horloge plus élevée. La différence de technologie explique donc les différences de densité de calcul entre DART et le C64. Pour le FPGA en revanche, l'explication de son inefficacité est le sur-dimensionnement de l'architecture. Dès lors que le FPGA implémente plusieurs applications de complexité variées, il doit être dimensionné au pire cas. Les applications moins complexes se traduisent donc par une sous-utilisation des ressources. Par ailleurs, la difficulté de concevoir des opérateurs arithmétiques avec des cellules logiques se traduit par des performances dégradées des opérateurs. Pour maintenir un niveau de performance suffisant il est donc nécessaire d'exploiter un parallélisme massif et d'introduire un grand nombre d'opérateurs dans l'architecture. Il est cependant à noter que les techniques de reconfiguration partielle permettent de limiter le problème de sur-dimensionnement de l'architecture pour certaines opérations. L'introduction d'opérateurs dédiés dans l'architecture permet également d'accroître cette densité de calcul, grâce à l'accroissement de la fréquence de fonctionnement.

Coût de la configuration

Le coût de la configuration a un impact déterminant sur les performances et l'efficacité énergétique du système. Celui-ci est principalement issu du volume d'informations nécessaire au traitement d'une application. Nous distinguons ici les notions de configuration, qui permettent de spécifier la structure matérielle de l'architecture (synthèse des opérateurs, interconnexions, ...), et de contrôle qui permettent de gérer l'évolution des données au travers de l'architecture. La figure 4.19 résume le volume d'information nécessaire à ces deux opérations (sur une échelle logarithmique) pour les applications de filtrage et de décodage des données, pour le C64, le Xc200E et DART.

Cette figure fait clairement apparaître les divergences conceptuelles entre la reconfiguration au niveau logique et la reconfiguration au niveau système. Dans le premier cas de figure, un volume de configuration très élevé est distribué au sein de l'architecture. Le coût de la reconfiguration est ici très important mais une fois spécifiée, celle-ci est transparente lors de l'exécution. À l'inverse, les architectures reconfigurables au niveau système éliminent le surcoût lié à la spécification du chemin de données, celui-ci étant matériellement figé. En revanche le coût du contrôle de l'architecture, en cours de traitement, est conséquent. Il correspond, pour le C64, à la lecture et au décodage d'une instruction de 256 bits à chaque cycle. Les architectures reconfigurables au niveau fonctionnel, et en particulier DART, permettent

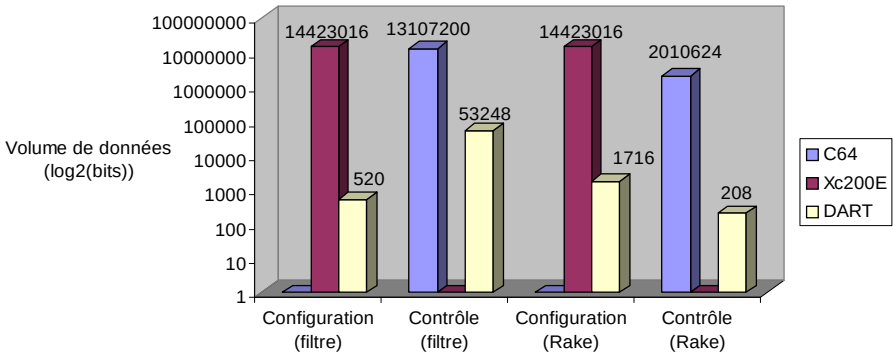

FIG. 4.19 – *Volume de données nécessaire à la configuration et au contrôle du Xc200E, du TMS320C64x et de DART sur le filtre de réception et le rake receiver. Pour traiter un slot, le FPGA nécessite un grand nombre de données pour spécifier la structure matérielle de l'architecture. À l'inverse, le DSP systématise l'envoi d'informations de contrôle en cours d'exécution, sans nécessiter de configuration matérielle préalable de l'architecture. DART optimise quant à lui ces deux paramètres.*

de limiter les volumes de configuration associés à ces deux opérations. La configuration matérielle préalable au traitement est en effet limitée à quelques centaines de bits (de 3 à 19 instructions 52 bits) lorsque le contrôle de l'architecture en cours de traitement est réduit à la spécification d'instructions de *reset*.

Le coût de la reconfiguration pour un FPGA est donc directement lié à la régularité de l'application. Ce type d'architecture ne pourra en effet limiter le surcoût de la reconfiguration que dans la mesure ou la durée de vie des configurations est suffisamment importante. À l'inverse, ce surcoût est constant pour un processeur programmable et la régularité de l'application n'a aucune influence. Le surcoût du contrôle de l'application est ici à prendre en compte à chaque cycle. Le mode de reconfiguration adopté pour DART permet d'éliminer cette contrainte. Il n'existe pas de surcoût systématique lié au contrôle de l'architecture. À la manière des FPGAs, l'efficacité de la reconfiguration au niveau fonctionnel dépend également de la régularité de l'application. Cependant, celle-ci n'a pas à être estimée en millions de cycles d'exécution. Dès lors que la configuration est maintenue sur quelques dizaines de cycles, cette approche est satisfaisante. Cette analyse valide donc le critère de rémanence évoqué dans le chapitre 1.3.3.

Performances sur le récepteur complet

La figure 4.20 résume les performances du DSP, du FPGA et de DART sur les applications de filtrage et le *rake receiver*. Elle démontre clairement le potentiel des architectures reconfigurables au niveau fonctionnel et logique, capables d'exploiter une large part du parallélisme de l'application.

Les résultats présentés jusqu'à lors considèrent cependant le traitement d'un algorithme, ou éventuellement d'une somme d'algorithme, qui sont implémentés en une seule phase sur le FPGA, i.e. sans nécessiter de reconfigurer celui-ci. Le traitement d'un récepteur WCDMA complet ne peut cependant être supporté par le Xc200E, celui-ci ne disposant pas d'une puissance de calcul suffisante. Il est dès lors nécessaire de reconfigurer régulièrement le circuit afin d'implémenter successivement le filtre de réception puis le *rake receiver* (synchronisation + estimation de canal + décodage des données). La figure 4.21 représente le temps d'exécution

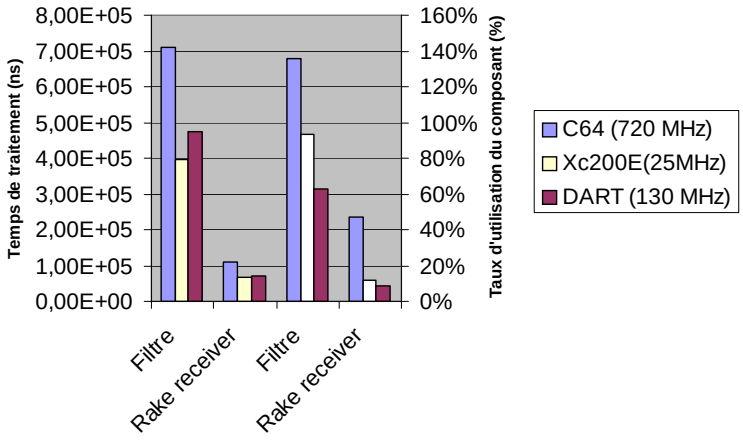

FIG. 4.20 – *Performance du Xc200E, du TMS320C64x et de DART sur le filtre de réception et le rake receiver. Les deux premiers histogrammes représentent le temps de traitement pour un slot du filtre de réception et du rake receiver. Les deux derniers donnent les taux d'occupation des différentes architectures sur ces mêmes applications.*

du récepteur complet sur les 3 architectures examinées jusqu'à lors en fonction de la fréquence des reconfigurations, évaluée en nombre de symboles manipulés entre deux reconfigurations.

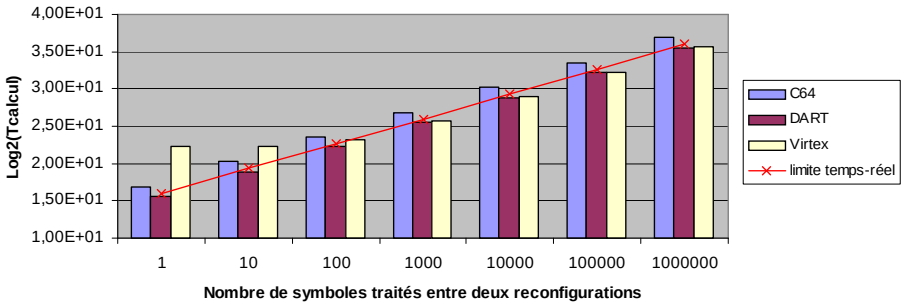

FIG. 4.21 – *Performance d'un FPGA Virtex, d'un C64 et de DART sur le récepteur complet. Les performances du DSP n'autorisent pas le traitement en temps réel du récepteur WCDMA. Le surcoût lié à la reconfiguration du FPGA impose quant à lui de maintenir les configurations pendant de longues périodes. La limite temps-réel n'est en effet franchie qu'à partir du moment où 150 symboles (100ms) peuvent être traités entre chaque reconfiguration.*

Le transfert des 1.4Mbits nécessaires à la reconfiguration du FPGA est une étape très consommatrice en temps d'exécution. Configurer le circuit nécessite en effet 2.7ms, dans le mode de configuration le plus rapide (SelectMAP avec $F_{config} = 66MHz$). Le Xc200E n'est par conséquent performant que dans la mesure où il manipule de très longs flots de données. Les résultats présentés sur la figure 4.21 montrent en effet que ce composant n'est en mesure de supporter des contraintes temps-réel que lorsque le nombre de symboles traités entre deux reconfigurations excède 150, soit une trame. Dans ces conditions, il est nécessaire de filtrer tous les échantillons d'une trame (4*256*10*15=153600 échantillons), de stocker les résultats en mémoire, puis de reconfigurer le composant pour décoder les données stockées en mémoires.

Un inconvénient notable relatif à l'implémentation du récepteur WCDMA sur ce composant

tient donc dans la latence importante séparant la réception des données de leur décodage. Celle-ci dépasse en effet les 100 ms, occasionnant de ce fait un décalage temporel entre le signal émis et le signal reçu pouvant être perçu comme une gêne par l'utilisateur du terminal mobile, lorsque le signal transmis est un signal de parole par exemple. Un autre inconvénient majeur lié à cette solution tient dans le volume de données temporaires. En effet, la nécessité de stocker les échantillons filtrés avant de les décoder se traduit par l'utilisation de nombreuses cellules mémoire. Le stockage d'une trame complète nécessite 1.2Mbits de mémoire. Les capacités de stockage du Xc200E étant limitées à 114kbits, il est ici nécessaire d'utiliser des mémoires externes qui occasionneront un surcoût énergétique et financier très élevé.

4.3.4 Conclusions

Dans cette section, nous avons comparé DART à deux architectures représentatives de solutions reconfigurables au niveau logique et système, dans le cadre d'une application conséquente, un récepteur WCDMA. Les architectures retenues pour évaluer l'efficacité des solutions reconfigurables au niveau logique et système l'ont été en raison de leur disponibilité, de leur notoriété et de leurs performances.

Les diverses expériences menées ont permis de mettre évidence le potentiel de DART dans le cadre des télécommunications de troisième génération. En particulier, nous avons démontré les capacités de cette architecture à exploiter les divers degrés de parallélisme inhérents aux applications ciblées. La flexibilité de cette architecture autorise une exploitation quasi-optimale des ressources de calcul, conduisant à une bonne densité de calcul. Sur les applications étudiées, la puissance de calcul observée atteint 6.2GOPS.

L'exploitation de cette puissance de calcul est notamment rendue possible par la simplicité du mode de reconfiguration adopté pour DART. En limitant le volume de configurations et en distribuant les configurations sous la forme de flots d'instructions, nous sommes en particulier capables de minimiser le coût de la spécification du chemin de données, tout en autorisant un contrôle simple de celui-ci en cours d'exécution. Nous combinons ainsi le potentiel d'optimisation des FPGAs tout en assurant une flexibilité proche de celle des processeurs programmables.

Cette efficacité de la gestion de l'architecture, combinée à une hiérarchie mémoire efficace sont les principaux garants de l'efficacité énergétique de cette architecture. En minimisant le coût énergétique de ces deux composantes, DART est en particulier en mesure de traiter, en moyenne sur le récepteur WCDMA, près de 39 MOPS pour chaque mW consommé lorsque dans le même temps, les architectures concurrentes n'en traitent que quelques unités (2 MOPS/mW pour le C64 et 5 MOPS/mW pour le Xc200E).

Un dernier critère d'évaluation, évincé jusqu'alors, est l'effort de développement nécessaire à l'implémentation d'une application. La flexibilité des architectures alliée à la maturité des outils de développement expliquent la supériorité des architectures programmables dans ce domaine. Une simple description C de l'application suffit en effet à l'obtention d'un résultat de bonne qualité. Les études menées sur le code généré pour le C64 ont par exemple démontré une bonne exploitation des ressources de calcul du processeur par son compilateur. À l'inverse, l'exploitation du FPGA nécessite une phase de développement relativement complexe. En premier lieu, l'utilisateur doit être capable de déterminer l'architecture optimale pour exécuter l'application. Cette phase d'exploration est suivie d'une phase de synthèse relativement fastidieuse, notamment dans le cas où le taux d'utilisation du composant est élevé, e.g. 72%

pour le filtre de réception. Les phases de placement-routage, dans ce cas particulier, sont en effet extrêmement critiques.

DART se présente de ce point de vue comme une solution intermédiaire. Par le biais de la chaîne exposée dans le chapitre précédent, une simple spécification C suffit à l'obtention d'une solution fonctionnelle. Cette chaîne de développement n'offrant pas d'outil pour explorer les différentes options d'implémentation, les phases d'optimisations demandent en revanche un effort non négligeable de la part de l'utilisateur. Celui-ci est en effet en charge le déterminer le type de parallélisme qu'il souhaite privilégier (données, opérations, tâches). La flexibilité de l'architecture et la simplicité des modèles de programmation laisse cependant supposer que la maturité des outils de développement pour DART pourrait être atteinte rapidement. À titre indicatif, dans l'état actuel d'avancement de la chaîne de développement de DART, le temps de développement peut être évalué pour ce récepteur WCDMA à une semaine pour le DSP, deux semaines pour DART et deux mois pour le FPGA.

4.4 Comportement de DART sur des traitements multimédia

En sus de l'étude du récepteur WCDMA, nous avons complété l'évaluation de DART dans le cadre des télécommunications 3G, par l'implémentation de divers types d'algorithmes, principalement issus des applications multimédia. Nous avons extrait de ces implémentations trois applications qui sont présentées dans cette section. Les deux premières concernent le traitement de l'image et de la vidéo. Ces domaines applicatifs sont illustrés par une transformée en cosinus discrète et par l'estimation de mouvement. La troisième étudie la viabilité de DART sur des systèmes OFDM (base des télécommunication de quatrième génération) par l'implémentation d'une des opérations les plus critiques d'un récepteur 802.11a : la synchronisation.

4.4.1 Implémentation de la transformée en cosinus discrète

La transformée en cosinus discrète (DCT : Discrete Cosine Transform) est au cœur de la majorité des applications de traitement d'image ou de vidéo. Elle permet de représenter une image dans un espace transformé afin d'augmenter la corrélation des données et par voie de conséquence, d'améliorer l'efficacité des algorithmes de compression usuels. Cette transformation peut être modélisée par l'équation 4.12. Dans cette équation, $f(x, y)$ représente le point image de coordonnée (x, y) et $F(u, v)$, le coefficient de coordonnées (u, v) dans l'espace transformé. Les coefficients $c(u)$ et $c(v)$ sont des coefficients de normalisation décrits par l'équation 4.13.

$$F(u,v) = \frac{4c(u).c(v)}{N^2} \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x,y).cos(\frac{(2x+1)u\pi}{2N}).cos(\frac{(2y+1)v\pi}{2N}) \tag{4.12}$$

$$\begin{aligned} c(i) &= \frac{1}{\sqrt{2}} && \text{si i=0} \\ &= 1 && \text{sinon} \end{aligned} \tag{4.13}$$

L'indépendance des axes horizontaux et verticaux des images permet la séparation des variables dans cette équation. Ainsi, la DCT en 2 dimensions peut être obtenue par 2 DCT, l'une verticale et l'autre horizontale. Dans ce cas, le nombre d'opérations nécessaire est de

2N fois celui d'une DCT monodimensionnelle, soit $2N.N^2$, lorsque la complexité initiale est de N^4. Cette DCT à 2 dimensions peut ainsi être décrite par le listing 4.12.

```
for (j=0;j<N;j++){ // Pour toute colonne
        for (y=0;y<N;y++){
                value[y]=0
                for (x=0;x<N;x++)
                        value[y]+=Coe[y+x*8].bloc[j+x*N];
                bloc[j+y*N]=value[y];
        }
}
for (i=0;j<N^2;i+=N){ // Pour toute ligne
        for (y=0;y<N;y++){
                value[y]=0
                for (x=0;x<N;x++)
                        value[y]+=Coe[y+x*N].bloc[i+x];
                bloc[y+x]=value[y];
        }
}
```

LIST. 4.12 – *Algorithme de DCT 2-D par découpage ligne-colonne. Le motif de calcul inhérent au traitement des lignes et des colonnes est identique. Ces deux traitements ne se distinguent que par les données manipulées.*

L'implémentation de cette application se traduit par la mise en œuvre de 2 cœurs de boucle dont les motifs de calculs sont identiques, et correspondent à une Multiplication-Accumulation. La dimension du bloc à transformer est typiquement de 8x8 pixels (N=8 dans le listing 4.12). Bien que les données manipulées soient généralement codées sur 8 bits, l'exploitation des capacités SWP de DART sur le traitement d'un bloc de l'image à transformer est délicate. Les traitements sur les lignes et les colonnes ne gérant pas l'accès aux données de la même façon, la mise en place de traitements SWP nécessite en effet une réorganisation des données entre les traitements sur les lignes et les traitement sur les colonnes. Les gains attendus de l'utilisation de techniques SWP sont ainsi absorbés par la phase de réorganisation des données. Il est cependant possible d'exploiter ces capacités si l'on exploite un parallélisme de tâches. En effet, en traitant concurremment deux blocs de l'image, il est possible d'arranger les données de manière à stocker les échantillons d'un bloc sur la partie haute des données et les échantillons d'un bloc suivant sur la partie basse. Ces capacités SWP ne pourront donc être utilisées que dans la mesure où le contrôleur du système dans lequel s'intègre DART travaille à cette réorganisation des données. Nous donnerons donc par la suite les résultats avec et sans exploitation des capacités SWP de DART.

Comme pour la plupart des traitements par bloc, l'exploitation d'un parallélisme de tâches facilite le partage des données et se traduit par une minimisation du nombre d'accès mémoire. Nous avons donc exploité ce parallélisme pour l'implémentation sur DART. Celui-ci s'est traduit par l'allocation d'un DPR à la transformation d'un bloc de l'image. Six blocs sont donc transformés en parallèle sur les six DPRs d'un *cluster*. La régularité de la configuration permet d'exploiter efficacement le SCMD pour réduire à 7 le nombre d'instructions de configuration. Pour une image contenant LxH blocs de NxN échantillons nous obtenons donc les performances résumées dans le tableau 4.2.

Format des données	**Cycles de traitement**	**Cycles par bloc (8x8 pixels)**	**Efficacité énergétique**
SWP 8 bits	H.L/12.2.(N^3/2+1)	42.8	30 MOPS/mW
WP 8 bits	H.L/6.2.(N^3/2+1)	85.6	21.1 MOPS/mW
WP 16 bits			15 MOPS/mW

TAB. 4.2 – *Performance et efficacité énergétique de DART sur le traitement de la DCT 2D.*

La flexibilité de DART autorise ainsi le traitement d'un bloc de 8x8 pixels en moins de 86 cycles, sans exploiter les capacités SWP de DART. Dans le cas contraire, le temps de traitement d'un bloc est réduit à 42 cycles. Si l'on considère des images de 1024x768 pixels codés sur 16 bits, DART est ainsi en mesure de les transformer en 8ms, soit une cadence de 123 images encodées par seconde. Pour un format d'image plus adapté à la téléphonie mobile, de 176x144 pixels 8 bits, la cadence de traitement atteint 7664 images par seconde. En mode SWP, 15328 images peuvent être transformées à chaque seconde.

Le tableau 4.2 fait par ailleurs apparaître de fortes différences en matière d'efficacité énergétiques, selon le format des données manipulées. Il est en particulier intéressant de noter l'efficacité des optimisations en consommation des opérateurs dans le cadre des traitements 8 bits. Par une optimisation poussée des opérateurs, nous avons en effet pu réduire de 29% la consommation d'énergie lors des traitements 8 bits. La minimisation des accès mémoires et du contrôle de l'application lors d'un fonctionnement SWP permet quant à elle de diminuer de 30% la consommation par rapport à une solution dans laquelle les données de 8 bits seraient traitées une à une. Ce gain en terme d'efficacité justifie alors l'exploitation du contrôleur du système pour adapter le placement des données en vue d'une implémentation SWP de la DCT sur DART.

4.4.2 Implémentation de l'estimation de mouvement

Des applications multimédia, le codage vidéo est sans aucun doute la plus demandeuse en terme de puissance de calcul. Les différents formats susceptibles d'être utilisés au sein de terminaux multimédia sont des déclinaisons des normes MPEGx et H.26x [147]. Tous les codeurs de ce type se basent sur l'exploitation de la localité temporelle entre images consécutives pour réduire le volume de données nécessaire au codage de la séquence. Cette localité temporelle est extraite par le biais d'un algorithme extrêmement gourmand en puissance de calcul : l'estimation de mouvement.

Les images traitées par l'algorithme d'estimation de mouvement sont préalablement découpées en MacroBlocs de 16x16 pixels (MB). Le principe de cet algorithme peut être représenté par la figure 4.22. Il consiste, pour chaque MB constituant l'image courante, à retrouver dans une image de référence le MB qui lui ressemble le plus. Afin de limiter la complexité de ce traitement, tous les algorithmes considèrent que les mouvements des pixels constituant un MB sont uniformes. Le déplacement peut donc être modélisé, pour chaque MB, par un seul vecteur. Par ailleurs, le mouvement d'un MacroBloc entre deux images successives étant de faible amplitude, les algorithmes limitent l'espace de recherche du MB le plus ressemblant à un sous-ensemble de l'image de référence, centré sur la position du MB courant.

La recherche du MB le plus ressemblant se base sur un calcul de distorsion. Plusieurs métriques permettent d'évaluer cette distorsion. Le plus utilisé est la somme des valeurs

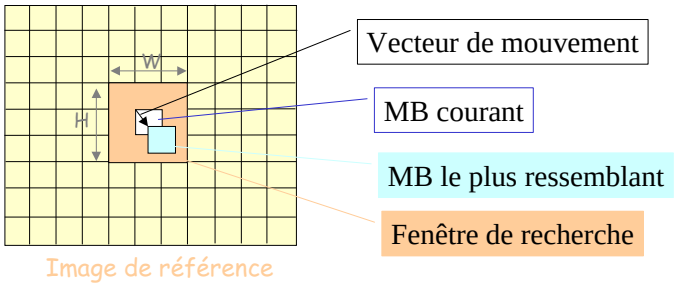

FIG. 4.22 – *Modélisation de l'algorithme d'estimation de mouvement. Pour chaque MB de l'image courante, une fenêtre de référence centrée sur la position du MB courant est défini. À l'intérieur de cette fenêtre, le MB le plus ressemblant est recherché afin de calculer un vecteur de mouvement qualifiant le déplacement du MB entre les deux images.*

absolues des différences (SAD : Sum of Absolute Difference), modélisé par l'équation 4.14. Afin de réduire la complexité du traitement, un certain nombre de méthodes ont été développées. Celles-ci ont principalement pour but de réduire le nombre moyen de calculs de distorsion, en adoptant une démarche de recherche structurée. Ces algorithmes rapides parcourent la fenêtre de recherche en privilégiant les déplacements de faible amplitude (recherche en spirale) ou une direction particulière (recherche discrète). Dès lors qu'un calcul de distorsion satisfait un critère de vraisemblance, le MB est considéré comme le plus ressemblant et la recherche est arrêtée. Le caractère non déterministe de ces algorithmes interdit cependant leur mise en œuvre sur DART. Ne pouvant prédire le nombre de calculs de distorsion devant être réalisés pour un MB donné, il est en effet nécessaire d'effectuer régulièrement des tests d'échappement. Du fait de l'inefficacité de DART lors des traitements d'instructions conditionnelles, nous considérerons par la suite une recherche exhaustive, i.e. le calcul de distorsion est calculé systématiquement pour tous les MB de la fenêtre de recherche.

$$SAD = \sum_{j=0}^{15} \sum_{i=0}^{15} |MB_{cur}(i,j) - MB_{ref}(i,j)| \tag{4.14}$$

Cet algorithme d'estimation de mouvement a été implémenté sur DART en considérant des vidéos au format QCIF (Quarter Common Intermediate Format). Les images constituant le flux vidéo contiennent 176x144 pixels codés sur 8 bits. Elles sont décomposées en macroblocs de 16x16 pixels. Le déplacement maximum autorisé est de 15 pixels, ce qui conduit à une fenêtre de recherche de 46x46 pixels. Le traitement d'un calcul de distorsion nécessite 256 soustractions, 256 valeurs absolues et 255 additions. L'UAL étant la seule unité susceptible de traiter les opérations de soustraction et de valeur absolue, les multiplieurs seront utilisés pour le traitement de l'accumulation. Une sous-utilisation des ressources est donc ici inévitable, les opérations pouvant être traitées sur le multiplieur/additionneur étant deux fois moins nombreuses que celles traitées sur l'UAL. Pour cette implémentation, le parallélisme d'instructions a été privilégié au détriment du parallélisme de tâches. En conséquence, les 6 DPRs de DART travaillent à l'implémentation d'un traitement de SAD. La spécification de la configuration de DART pour ce calcul de distorsion nécessite 12 instructions.

Le mode de fonctionnement de DART lors de l'estimation de mouvement peut être modélisé par le diagramme de Gantt représenté sur la figure 4.23. Pour chaque MB de l'image courante, le traitement débute par le transfert du macrobloc courant et du MacroBloc de référence, depuis la mémoire du *cluster*, jusqu'aux mémoires des DPRs. La phase de reconfiguration

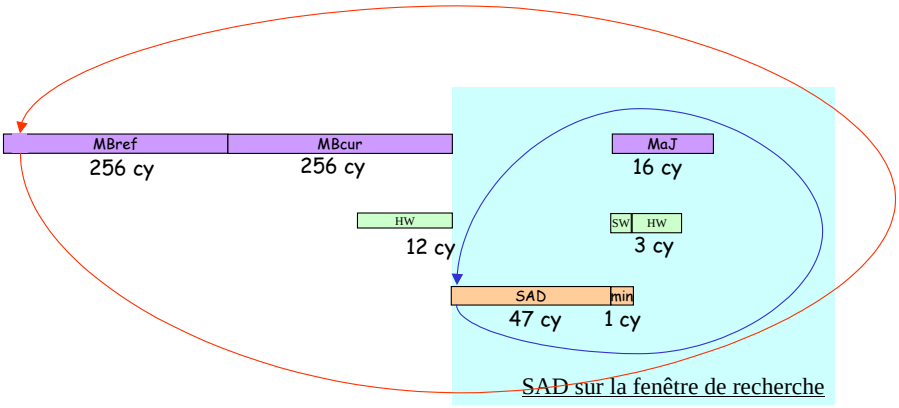

FIG. 4.23 – *Diagramme de Gantt de l'estimation de mouvement. Avant de procéder à la recherche du MB le plus ressembalnt, les MBs courant et de référence doivent être transférés depuis la mémoire du cluster jusqu'aux mémoires locales. Pour chaque MB de la fenêtre de recherche, le calcul de distorsion est effectué puis le MB de référence est mis à jour.*

débute au cours de ces transferts, ce qui la rend transparente du point de vue du temps d'exécution. Une fois les données présentes au sein des mémoires locales, le traitement de la SAD est effectué pour chaque MB de l'image de référence. Ce calcul de distorsion nécessite 47 cycles d'exécution. Le résultat du calcul de distorsion est alors comparé à la plus petite valeur obtenue jusqu'à lors par le biais d'une fonction $min(A, B)$, spécifiée sous la forme d'une reconfiguration logicielle. Concuremment à ce traitement, le contrôleur du *cluster* doit être en mesure de mémoriser l'adresse du MB minimisant la distorsion (cf. §4.4.3). Une fois la valeur minimale stockée en mémoire, il est nécessaire de reconfigurer le DPR ayant servi à ce calcul de manière à se replacer dans une configuration de calcul de distorsion. Puisque la reconfiguration logicielle ne concerne que le DPR stockant la valeur minimale de SAD, seules 3 instructions de reconfiguration suffisent à retrouver la configuration initiale. Parallèlement au calcul de la plus petite valeur de distorsion, les données relatives au MB de référence sont mises à jour, par le transfert de 16 nouvelles données. Ces 16 données correspondent (selon le déplacement dans la fenêtre de recherche) à 1 colonne ou à 1 rangée du MB de référence.

Le nombre d'itérations de cette boucle, visant à déterminer le MB pour lequel la distorsion est minimale, varie selon la position du MB courant dans l'image. Les MB placés sur les angles et les côtés de l'image ont en effet un nombre réduit de déplacements possibles, respectivement d'un facteur 4 et 2, par rapport aux MB situés au cœur de l'image. Ainsi, 256 et 496 itérations de cette boucle sont respectivement nécessaires au traitement des MB d'angles et de côtés. 961 calculs de SAD doivent en revanche être traités pour les MB situés au cœur de l'image. Au final, l'estimation du mouvement entre deux images au format QCIF nécessite 7.35 millions d'opérations arithmétiques. Le traitement d'une SAD sur DART, nécessitant 63 cycles (dont 16 pour les accès mémoire), 37.9 ms sont nécessaires au traitement de cette estimation. 26 images peuvent donc être traitées à chaque seconde. Le taux de rafraîchissement associé aux vidéos de format QCIF étant de 15 images par seconde, le traitement temps-réel de cette application est amplement assuré par DART avec un taux d'occupation des ressources de 58%. La puissance de calcul effective délivrée sur cette application atteint 1.57GOPS.

Cette application se distingue de celles évoquées jusqu'à lors par le coût des transferts de données. Malgré le recouvrement des phases de reconfiguration et des phases de transfert de données, ces derniers sont responsables de 26% des cycles d'exécution. Ces transferts de

données ont par ailleurs une influence non négligeable sur la consommation d'énergie. Bien que l'efficacité énergétique de DART sur cette application reste excellente (20.5MOPS/mW), la figure 4.24 permet de constater que 34% de la consommation d'énergie est issue des mémoires de DART. Malgré l'insertion de reconfigurations logicielles en cours de traitement, le coût énergétique de la distribution du contrôle dans l'architecture reste négligeable.

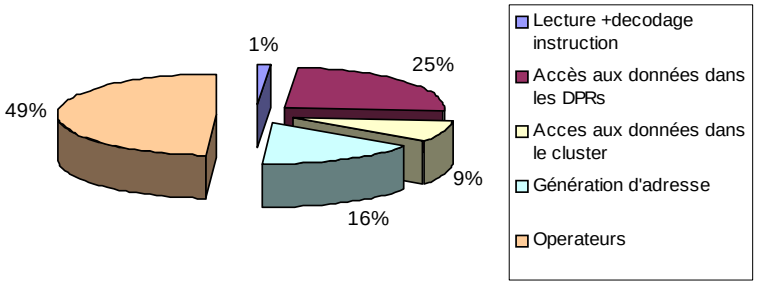

FIG. 4.24 – *Distribution de la consommation dans DART lors de l'estimation de mouvement. L'utilisation intensive des mémoires inhérente aux traitements vidéos se traduit par une part importante d'énergie dissipée dans ces composants. Bien que des reconfigurations logicielles interviennent en cours de traitement, la consommation du contrôleur de cluster reste maîtrisée.*

4.4.3 Implémentation de la synchronisation d'un récepteur OFDM 802.11a

Compte-tenu de son efficacité spectrale et de sa robustesse vis à vis du phénomène de *fading*, la technologie OFDM est largement utilisée dans les standards de communications sans fils, tels que l'IEEE 802.11 ou l'HIPERLAN/2 [148]. Cette technologie se retrouve par ailleurs dans le domaine de la télévision numérique (DAB : Digital Audio Broadcasting, DVB : Digital Audio Broadcasting) et est préconisée dans le cadre de la téléphonie mobile de quatrième génération (T4G). Afin d'évaluer DART dans le cadre des T4G, nous considérons dans cette section l'implémentation d'une partie déterminante de la norme : la synchronisation.

Les techniques OFDM [149] sont extrêmement sensibles aux erreurs de synchronisation et de ce fait leurs performances sont fortement dépendantes de la synchronisation temporelle et fréquentielle entre l'émetteur et le récepteur. La synchronisation temporelle consiste à déterminer l'instant précis auquel débute chaque symbole OFDM. Si cet instant ne peut être connu, le récepteur ne peut éliminer le préfixe du symbole et séparer correctement les symboles complexes, issus des différentes porteuses, avant de calculer la FFT de leurs échantillons.

Pour faciliter cette synchronisation, les *bursts* transmis par l'émetteur sont découpés en un préambule et un train de données utiles. Le format d'un *burst* pour la norme 802.11a est représenté sur la figure 4.25. Chaque préambule débute par une séquence de 10 données identiques, constituées chacune de 16 échantillons complexes qui doivent être détectés par le récepteur pour déterminer le début du symbole OFDM. La suite du préambule contient des informations exploitées dans le cadre de l'estimation du canal [150].

Le principe de la synchronisation consiste à calculer concurremment l'inter-corrélation de deux blocs de 16×L échantillons, retardés de (9-L)×16 cycles. En pratique, 6 inter-corrélations sont simultanément évaluées. Les valeurs de ces fonctions étant maximales lorsque le signal d'entrée correspond à une suite de symboles courts (16 échantillons) identiques, le traitement

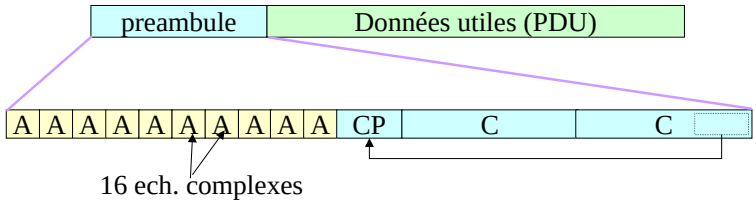

FIG. 4.25 – *Structure d'un burst physique dans la norme 802.11a.*

de synchronisation consiste à comparer les valeurs des inter-corrélations à un seuil. Dès lors qu'un seuil est franchi, le préambule est détecté et l'adresse de début du paquet doit être transmise au contrôleur du système. Afin de réaliser une synchronisation précise, les valeurs des fonctions d'inter-corrélation $S_1..S_6$ doivent être mises à jour à chaque nouvel échantillon, soit toutes les 50ns. La méthode de mise à jour de ces fonctions est modélisée, pour la fonction S_k, sur la figure 4.26. Elle nécessite, pour chaque fonction, 2 multiplications, une soustraction et une addition. Ces opérations manipulant des données complexes, 16 opérations sont donc nécessaires à la mise à jour de chacune des fonctions d'inter-corrélation.

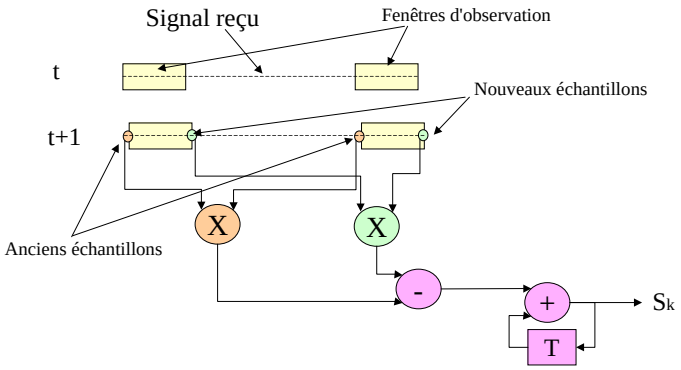

FIG. 4.26 – *Principe de la mise à jour des fonctions d'inter-corrélation. L'arrivée d'un nouvel échantillon se traduit par le déplacement de la fenêtre d'observation. La mise à jour de la fonction S_k consiste à lui rajouter le produit des échantillons entrant dans cette fenêtre d'observation et à en soustraire le produit des échantillons sortant. Toutes ces opérations manipulent des données complexes dont les parties réelle et imaginaire sont codées sur 12 bits.*

La valeur du seuil à partir duquel la suite de symboles identiques est détectée dépend de la puissance du signal reçu. Parallèlement aux calculs de corrélation, il est donc indispensable de mettre à jour la valeur de la puissance du signal reçu. La méthode adoptée pour ce calcul est similaire à la mise à jour des fonctions de corrélation et consiste à ajouter, à la puissance du signal, le module des nouveaux échantillons et à soustraire celui des anciens. Cette mise à jour nécessite 8 opérations pour chaque ensemble de données et pour chaque nouvel échantillon. En exploitant un parallélisme de tâches, les calculs de mise à jour des fonctions de corrélation et de puissance du signal peuvent donc être traités en parallèle, respectivement sur les 4 premiers et les 2 derniers DPRs. La puissance de calcul délivrée par DART est alors de 2.88 GOPS et suffit à l'implémentation des parties calculatoires de cet algorithme.

Outre ces deux traitements, hautement calculatoires, une dernière étape doit être implémentée pour détecter le début d'un paquet. Celle-ci consiste à comparer le module de la corrélation au produit du seuil de corrélation et de la puissance du signal. Bien que la complexité de cette

partie décisionnelle de l'algorithme soit nettement inférieure à celle inhérente aux calculs de corrélation ou de puissance, sa mise en œuvre n'est pas triviale sur DART. Deux obstacles doivent en effet être surmontés pour implémenter cet algorithme de synchronisation.

Dépassement des capacités de calcul d'un cluster

La première difficulté est relative à la quantité de ressources disponibles sur l'architecture. Compte-tenu des très fortes contraintes temps-réel inhérentes à cet algorithme de synchronisation, la reconfiguration de DART en vue de calculer le module des fonctions de corrélation et de le comparer au seuil est complexe. Il est nécessaire d'effectuer ce traitement en parallèle avec les parties calculatoires de l'algorithme. Les 6 DPRs étant d'ores et déjà alloués au calcul des corrélations et des puissances, la mise en œuvre de ce traitement est inefficace et nécessite l'utilisation d'un second *cluster*. L'ajout d'un septième DPR au sein de l'architecture est donc indispensable pour traiter efficacement cette application.

Bien que DART atteigne ses limites sur cet algorithme, ce cas de figure ne remet pas en cause les concepts étudiés jusqu'à lors. Dans le cadre de cette thèse, nous avons en effet illustré les concepts exploités dans DART en considérant une architecture de *cluster* à 6 DPRs, parfaitement adaptée aux télécommunications de troisième génération. Lors de la définition de ces concepts, un modèle plus général à cependant été adopté et l'accroissement du nombre de DPRs n'influe pas sur la véracité des commentaires énoncés jusqu'à lors.

En particulier, par l'emploi des mécanismes de reconfiguration HW et SW, ainsi que par l'utilisation de techniques SCMD, le coût de la reconfiguration peut être maîtrisé. À titre d'exemple, le passage de 6 à 8 DPRs ne se traduit que par l'ajout de 2 bits au sein des instructions de configuration, ces derniers permettant d'identifier les 2 DPRs supplémentaires. Du point de vue architectural, l'augmentation du nombre de DPRs doit en outre s'accompagner d'une augmentation du nombre de bus constituant le réseau segmenté. Conjointement à l'augmentation du parallélisme d'opérations, il est en effet indispensable d'accroître les capacités de communication au sein de l'architecture. L'élargissement du réseau segmenté s'accompagne également d'une augmentation du nombre de points de configuration au sein du DPR. En pratique, un multiplexeur supplémentaire devra être inséré dans le DPR afin d'autoriser l'exploitation des 2 nouveaux bus globaux. Dans le même ordre d'idée, les multiplexeurs en entrée des unités fonctionnelles devront être élargis pour supporter ces nouvelles entrées.

Si les évolutions architecturales nécessaires à la modification du chemin de données sont nombreuses, elles sont également simples à mettre en œuvre et ne se traduisent pas par des pénalités énergétiques notables. Les différentes analyses faites dans le cadre d'une architecture à 6 DPRs restent donc valides. Du point de vue logiciel, si la modification des outils est une fois encore nécessaire, rien ne laisse présumer d'une quelconque difficulté devant être surmontée lors de la mise à jour des outils de développement.

Afin d'augmenter la puissance de calcul de DART, il est également possible d'intégrer un plus grand nombre de ressources de calcul dans les DPRs. Le coût de cette évolution est cependant plus élevé que celui évoqué précédemment. Dans ce cas, il est en effet nécessaire d'augmenter le nombre de bus au sein des DPRs et par voie de conséquence d'élargir chacun des multiplexeurs de l'architecture. Une augmentation conséquente du nombre de points de configuration, pour chaque DPR, est donc dans ce cas indispensable.

Limitation du contrôleur du cluster

Outre cette première limitation, relative au nombre de ressources de calcul intégrées au sein de l'architecture, une seconde difficulté doit être surmontée afin d'implémenter cet algorithme de synchronisation. En effet, suite à la détection d'un maximum dans les fonctions de corrélation, le contrôleur de *cluster* doit être en mesure d'arrêter le traitement, et de renvoyer l'adresse de début du paquet au contrôleur du système. Ce dernier se chargera alors de distribuer les données utiles aux modules en charge de les décoder. Cet algorithme est donc caractérisé par un comportement non déterministe qui ne peut être supporté par DART.

Afin de réduire l'espace des solutions architecturales envisageables dans le cadre des applications embarquées de prochaines générations, nous avons en effet considéré, au début de cette étude, des applications totalement déterministes. Dans de telles applications, le comportement de l'architecture peut être entièrement prédit au moment de la compilation et de ce fait, les mouvements de données, de même que les reconfigurations, sont statiquement programmés au sein des ressources de l'architecture. De ce fait, nous n'avons introduit aucun mécanisme permettant de modifier, en cours d'exécution, le comportement de DART. Bien que couramment admise par les architectes, cette limitation est extrêmement contraignante pour les ingénieurs système développant les programmes devant être exécutés. En effet, si la très grande majorité des applications embarquées de prochaines générations peuvent être implémentées sous une forme déterministe, d'autres (e.g. synchronisation, estimation de mouvement, ...) font nécessairement appels à des structures conditionnelles lors de l'exécution.

L'architecture du contrôleur initialement envisagée pour DART affiche donc ses limites sur certaines classes d'applications. En effet, aux tâches de distribution de configuration, initialement dévolues à ce composant, doivent s'ajouter des capacités de décision. Une évolution de DART semble donc s'imposer de manière à élargir le spectre des applications exécutables sur l'architecture. Selon le niveau de flexibilité souhaité, il est possible d'introduire au sein du contrôleur la capacité d'interagir avec l'environnement du *cluster* ou d'interagir avec les ressources de calcul internes au *cluster*. La premier cas se réfère aux propriétés d'observabilité du *cluster* et le second aux propriétés de réactivité.

Observabilité Afin de supporter le traitement de la synchronisation, les modifications devant être apportées au contrôleur du *cluster* sont minimes. Il s'agit en effet d'offrir au contrôleur la possibilité d'interagir avec l'environnement du *cluster* lorsqu'une condition est détectée. En d'autres termes, il s'agit de rendre les drapeaux des unités fonctionnelles (A>B, A=B, A<B) accessibles au contrôleur du *cluster*[4]. Dès lors que le contrôleur du *cluster* est en mesure de détecter le dépassement d'un seuil ou l'obtention d'une condition particulière, ce dernier doit également pouvoir spécifier à son environnement extérieur l'état d'avancement de l'exécution. Cet état d'avancement peut par exemple correspondre à l'adresse du début d'un paquet pour l'OFDM, ou à l'adresse du MB minimisant un calcul de distorsion pour l'estimation de mouvement.

Réactivité La possibilité d'adapter le comportement de l'architecture aux résultats de précédents calculs est en revanche beaucoup plus complexe à mettre en œuvre. Si la détection d'événements particuliers peut être réalisée de la même façon que précédemment, la modification du comportement de l'architecture nécessite quant à elle des évolutions significatives de l'architecture. En effet, dans l'état actuel de DART, le contrôleur du *cluster* n'a accès qu'à la structure du chemin de données des DPRs. La modification du

[4]Ces drapeaux sont d'ores et déjà positionnés par les unités fonctionnelles intégrées dans les DPRs.

comportement de l'architecture nécessite en outre une modification des programmes des générateurs d'adresses assurant la distribution des données. Le caractère distribué du contrôle de l'architecture est donc ici un inconvénient majeur à la mise en œuvre d'une architecture réactive. Deux options permettent cependant de contourner ce problème. La première consiste à rendre observables les drapeaux des unités de calcul par toutes les ressources de contrôle de l'architecture. En cas de modification du comportement du *cluster*, toutes les ressources de contrôle (contrôleur de *cluster*, AG, contrôleur mémoire) doivent réagir simultanément. La seconde option consiste à considérer un organe de décision unique : le contrôleur de *cluster*. Lorsque un événement est observé, une phase de reconfiguration, pendant laquelle les programmes des générateurs d'adresses sont modifiés, intervient. Cette seconde solution permet de réduire le coût de la réactivité du système mais se paie par des performances dégradées lorsque les traitements conditionnels sont nombreux. Cette "autoreconfiguration" de l'architecture est par exemple mise en œuvre dans l'architecture SYMPTOME développée au CEA [151].

Si l'objet de cette section n'est pas de préjuger des futures évolutions de l'architecture, il est important de noter que l'observabilité d'un système suffit à l'implémentation de l'ensemble des algorithmes inhérents aux télécommunications mobiles de troisième et de quatrième génération.

4.5 Conclusions

Dans ce chapitre, nous avons présenté les résultats d'implémentation de fonctions clés des télécommunications de troisième génération. Pour toutes les applications présentées, nous nous sommes attachés à mettre en avant les spécificités architecturales exploitées dans DART pour optimiser les performances et l'efficacité énergétique du système. Nous avons par ailleurs comparé les performances de DART à celles d'architectures plus conventionnelles, exploitant une reconfiguration au niveau logique et système. En balayant un large spectre d'application nous avons ainsi pu valider l'ensemble des concepts architecturaux ayant abouti à la définition de DART. Finalement, par l'étude d'une application critique des télécommunications de quatrième génération nous nous sommes confrontés aux principales limitations de l'architecture et avons évoqué un certain nombre d'évolutions possibles de DART.

De part le flexibilité de DART, nous avons en particulier démontré sa capacité à s'adapter à tout type de motif de calcul et à exploiter les faibles largeur de données par le biais de techniques SWP. Que ce soit dans le cadre d'un récepteur WCDMA ou d'applications multimédia, nous avons pu vérifier que la flexibilité du réseau interne aux DPRs est indispensable à l'exploitation d'un parallélisme d'opérations important, et que le dimensionnement du réseau segmenté permet d'assurer un niveau de flexibilité suffisant pour garantir un fonctionnement concurrent des DPRs du *cluster* efficace. La flexibilité des interconnexions permet par ailleurs de réduire très sensiblement la quantité d'accès mémoires dans bon nombre d'applications. À titre de rappel, ce partage de données entre les différentes ressources de calcul, rendu possible par le réseau segmenté, autorise une réduction d'un facteur 6 du nombre d'accès mémoire lors du filtrage réalisé au niveau du récepteur. 46 mW ont ainsi pu être économisés, soit 32% de la consommation globale issue du traitement de cette application. Dans le même ordre d'idée, l'insertion de registres au sein des DPRs est un atout majeur dans le cadre du partage des données. La construction de chaînes de retard, à partir de ces registres, autorise en effet une réduction sensible du coût de l'accès aux variables lors des traitements flots de données.

Lors de toutes nos expérimentations, nous avons en outre insisté sur la consommation énergétique de DART. Cette contrainte, particulièrement sévère dans le cadre des applications embarquées, est amplement respectée sur DART. Par un développement consciencieux de l'architecture, nous avons en effet pu exploiter le potentiel lié à la reconfiguration de niveau fonctionnel, pour atteindre une efficacité énergétique sans commune mesure par rapport aux architectures concurrentes. Sur les applications présentées dans ce chapitre, et sur la technologie considérée ($umc0.18\mu m$ alimentée en 1.8V), celle-ci varie en effet de 40 MOPS/mW lors des traitements SWP, à 20 MOPS/mW lors des traitements 16 bits, tout en assurant une puissance de calcul avoisinant les 6.2GOPS en mode SWP. Les contraintes inhérentes aux T3G en terme de faible consommation ($\sim 24MOPS/mW$) sont donc respectées. Sur le récepteur WCDMA, nous avons pu constater que DART consomme 25 fois moins qu'une architecture reconfigurable au niveau système et 9 fois moins qu'un FPGA, tout en assurant un niveau de performance suffisant à l'implémentation des télécommunications de troisième génération.

Cette efficacité énergétique est en partie due à une gestion efficace de la reconfiguration. La minimisation du nombre de points de configuration, liée à l'exploitation d'une reconfiguration au niveau fonctionnel de DART, a en effet permis la mise en place de stratégies de reconfiguration évoluées. En particulier, par la distinction des reconfigurations matérielles, intervenant lors des calculs intensifs, et des reconfigurations logicielles, mises en œuvre sur les traitements irréguliers, DART est en mesure de proposer une flexibilité comparable à celle des processeurs programmables, tout en assurant un potentiel d'optimisation proche de celui des solutions matérielles câblées. Par l'introduction du concept de SCMD, nous avons finalement réduit le volume de données moyen nécessaire aux reconfigurations des DPRs, et par voie de conséquence, réduit la latence et la consommation de ces opérations sans surcoût matériel notable.

Malgré l'efficacité de DART sur l'ensemble des applications constituant les télécommunications de troisième génération quelques évolutions de l'architecture semblent cependant souhaitables. En particulier, l'implémentation d'un des algorithmes clés des télécommunications de quatrième génération, la synchronisation pour l'OFDM, à mis en exergue un problème relatif à la quantité de ressources disponibles sur l'architecture. Ainsi, la distribution de DART sous forme de plateforme semble être une perspective intéressante à ce travail. Par la caractérisation du domaine applicatif, et compte-tenu de la structure très hiérarchique de DART, un dimensionnement adéquat de l'architecture (nombre d'unités fonctionnelles, de DPR, taille des mémoires, ...) permettrait en effet d'optimiser l'utilisation des ressources. Cette application a par ailleurs mis en avant la difficulté de ne supporter que des applications entièrement déterministe. Cette contrainte, bien que caractérisant la très grande majorité des applications embarquées, limite en effet le spectre des applications implémentables sur DART. Rendre l'architecture observable, voire réactive, semble donc être une perspective intéressante dans le cadre de l'élargissement des domaines d'utilisation de DART.

Conclusions et perspectives

Sommaire

Synthèse des travaux

Les travaux présentés dans ce mémoire portent sur la définition d'une architecture répondant à l'ensemble des contraintes inhérentes aux applications mobiles de prochaines générations. Ces contraintes de flexibilité, de performance et de consommation d'énergie ont été illustrées dans cette étude par le biais des télécommunications de troisième génération. Par l'analyse de ce domaine applicatif ainsi que par l'exploration de l'état de l'art, nous avons abouti à la définition d'une architecture reconfigurable dynamiquement au niveau fonctionnel : DART.

Les contraintes de haute performance et de faible consommation se traduisent nécessairement par la nécessité d'optimiser au mieux l'architecture, pour la rendre en adéquation avec le traitement à réaliser. Pour répondre aux besoins en flexibilité, le volume de données de configuration doit dans le même temps être réduit autant que possible. Ces deux contraintes se sont traduites par la mise en œuvre d'une reconfiguration fonctionnelle de DART. Bien que cette granularité de reconfiguration permette de maîtriser la quantité d'informations nécessaires a la spécification d'une structure de chemin de données, la distribution de ces informations à un coût non négligeable, aussi bien en terme de performance que de consommation. Pour le réduire, nous avons introduit un nouveau concept, le SCMD. Celui-ci est une évolution du SIMD et vise à reconfigurer simultanément tous les sous-ensembles du circuit qui implémentent une fonction identique.

Afin de minimiser le coût de la gestion de l'architecture, nous avons par ailleurs adapté la stratégie de distribution du contrôle au type de calcul implémenté. Ainsi, les traitements irréguliers, qui traduisent un modèle d'exécution temporel, sont distingués des traitements réguliers implémentés spatialement. Les modifications systématiques des traitements irréguliers sont spécifiées sous la forme de reconfigurations SW. Celles-ci permettent de reconfigurer l'architecture à chaque cycle mais disposent d'un potentiel d'optimisation limité. À l'inverse, l'architecture peut être optimisée, en exploitant toute la flexibilité de DART, pour les traitements critiques réguliers par le biais d'une reconfiguration HW. Le potentiel de DART est

dans ce cas totalement exploité au prix d'une phase de reconfiguration nécessitant quelques cycles. Une fois les configurations spécifiées, les informations de contrôle sont maintenues au sein de l'architecture par le biais de cellules de mémorisation. Ces deux modes de reconfiguration traduisent donc la règle des 80/20 qui affirme que 80% du temps d'exécution d'un programme est consommé par 20% du code, et que seul les 20% du temps d'exécution restant sont consommés par le reste des instructions du code source. Par le biais de ces deux modes de reconfiguration, nous sommes ainsi en mesure d'optimiser l'architecture pour les 20% de codes réguliers, tout en assurant une flexibilité suffisante à l'implémentation des 80% de codes irréguliers.

Les contraintes de performances associées à notre domaine applicatif nous ont conduit à exploiter tous les niveaux de parallélisme susceptibles d'être présents dans l'application à traiter. Afin de supporter un parallélisme de données, nous avons en particulier développé des opérateurs capables de supporter des traitements SWP. Le support d'un parallélisme d'opérations s'est quant à lui traduit par l'intégration de nombreuses unités fonctionnelles. Afin d'assurer un très haut niveau de flexibilité dans l'interconnexion des opérateurs, sans pour autant induire un coût de communication trop important, ces ressources de calcul ont été regroupées au sein de DPR. À l'intérieur de ces DPRs, une totale flexibilité est assurée par le biais d'un réseau totalement connecté. Les DPRs communiquent entre eux par le biais d'un réseau segmenté constituant un second niveau de hiérarchie d'interconnexion. L'association de ce réseau segmenté et de la décomposition en DPR facilite en outre l'exploitation d'un parallélisme de tâches. Lorsque plusieurs tâches doivent être implémentées concurremment, les DPRs peuvent en effet travailler indépendamment les uns des autres. Dans le cas contraire, par le biais du réseau segmenté, les DPRs peuvent communiquer de manière très flexible afin d'exploiter un parallélisme d'opérations massif. Le dernier niveau de parallélisme supporté par DART est le parallélisme d'applications. Il est assuré dans DART grâce à une découpe en *cluster*. Ces derniers intégrant leurs propres ressources de contrôle et de mémorisation, ils disposent d'une autonomie suffisante à l'implémentation concurrente de plusieurs applications.

Afin d'assurer une bande passante suffisante entre les ressources de stockage et de calcul de DART, une hiérarchie mémoire à 3 niveaux a été définie. Le premier niveau correspond à la mémoire de masse. Elle est accessible par tous les *clusters* de l'architecture. Le second niveau de la hiérarchie correspond aux mémoires intégrées au sein des *clusters*. Elles permettent de leur assurer une autonomie suffisante à l'implémentation d'applications complexes. Finalement, le dernier niveau de la hiérarchie est constitué de multiples bancs mémoires, de petite taille, distribués au sein des DPRs. Les unités fonctionnelles n'accèdent qu'aux données stockées dans ce dernier niveau de hiérarchie. Afin de maîtriser le surcoût lié à la gestion de ces mémoires, chacune d'elle dispose de son propre contrôleur qui séquence les accès à ces mémoires de manière totalement déterministe.

L'association de ces concepts nous a permis de concevoir une architecture répondant aux contraintes de performance, de faible consommation et de flexibilité inhérentes aux applications mobiles de prochaine génération. En considérant une version de DART à 4 *clusters*, disposant chacun de 24 unités fonctionnelles reparties dans 6 DPRs, la puissance de calcul atteint en effet, 12.4 milliard d'opérations arithmétiques par seconde. L'exploitation de techniques SWP permet par ailleurs de doubler ce niveau de performance sur les applications manipulant des données de petite taille et disposant d'un parallélisme de données. Dans le même temps, l'efficacité énergétique moyenne atteint 20 MOPS/mW lors des traitements arithmétiques 16 bits. Sur un récepteur WCDMA, une efficacité énergétique de 38 MOPS/mW a par

ailleurs pu être observée, notamment grâce à l'exploitation des capacités SWP de DART, à la minimisation du coût de distribution du contrôle dans l'architecture et à l'optimisation de la hiérarchie mémoire.

Lors du développement de DART, l'accent a été mis sur le respect des contraintes de faible consommation mais aussi sur la faisabilité d'une chaîne de conception. L'outil est basé sur l'utilisation conjointe d'un front-end permettant la transformation et l'optimisation de code C/C++ (Suif de Stanford), d'un compilateur reciblable (outil CALIFE de l'IRISA) et d'un environnement de synthèse de haut niveau (outil BSS de l'ENSSAT). À partir d'une application décrite en C/C++, l'utilisation de Suif permet de générer un graphe CDFG et de réaliser un certain nombre de transformations automatiques. Pour ce problème ont été développées des passes spécifiques de déroulage de boucles et d'extraction de traitements réguliers. La séparation entre les phases de configurations HW, de configurations SW et de génération des adresses, permet de traiter par la compilation et la synthèse architecturale l'ensemble d'une application. En sortie des outils de génération de configurations, un simulateur précis au bit-près et au cycle-près permet de valider l'application et d'évaluer son efficacité énergétique.

Bien que respectant l'ensemble des contraintes identifiées au début de cette étude, DART peut être optimisée de bien des façons. Dans la section suivante, nous discutons tout d'abord des évolutions possibles ou souhaitables de DART et de sa chaîne de développement. Les perspectives d'utilisation de cette architecture sont ensuite évoquées.

Perspectives

Optimisations architecturales

Dans le cadre de cette étude, nous nous sommes principalement concentré sur l'optimisation architecturale et logique de DART. Des gains importants en consommation et en performance peuvent cependant être attendus d'une optimisation physique de l'architecture. La phase de placement-routage de DART doit donc être considérée comme l'une des principales priorité dans le cadre de sa valorisation. Des études sont menées à ce titre au laboratoire pour arriver au plus vite à cette fin.

Dans ce mémoire, l'exploitation de techniques de gestion dynamique de la puissance (DVS) pour DART a été évoquée. La caractérisation de la consommation de DART suivant différentes tensions d'alimentation n'a cependant été que partiellement réalisée. Les études menées pour caractériser les opérateurs en consommation, suivant différentes tensions d'alimentation, ont permis de constater une réduction moyenne de 30 et 80 % de la puissance consommée lorsque la tension d'alimentation est respectivement réduite à 1.6 et 1.2 Volt. Ces réductions des tensions d'alimentation se traduisent dans le même temps par une augmentation de 20 à 480% de la latence des opérateurs. La mise en œuvre des techniques DVS nécessite cependant une caractérisation plus poussée de la consommation de DART. L'étude du comportement d'un *cluster* complet suivant différentes tensions d'alimentation doit donc être réalisée, afin de confirmer les tendances observées sur les opérateurs.

Le chapitre 4 a par ailleurs mis en évidence certaines évolutions possibles de l'architecture. Afin d'élargir le spectre des applications implémentables sur DART, il apparaît en particulier souhaitable d'introduire un peu plus de souplesse dans le contrôle de l'architecture.

Rendre l'architecture observable, voire réactive, permettrait en effet d'accroître l'attractivité de DART. La valorisation de DART passe également par une évaluation de son coût. Dans le cadre de cette étude, l'accent a été mis sur les respect des contraintes de performances, de flexibilité et de faible consommation. Celles-ci étant respectées, il apparaît désormais intéressant d'optimiser l'architecture en vue d'en réduire la surface.

Évolutions de la chaîne de développement

Dans ce mémoire, nous avons démontré la faisabilité d'une chaîne de développement pour DART et présenté les principaux modules permettant de transformer une description de haut niveau en codes exécutables. L'effort de développement devant être fourni par l'utilisateur souhaitant l'utiliser est cependant, à ce jour, relativement important. Une perspective importante à ce travail consiste donc à automatiser totalement cette phase de développement. Pour cela, les verrous relatifs à l'utilisation de CALIFE doivent être levés et les outils de compilation finalisés. Une fois les programmes des générateurs d'adresses et les configurations SW générées, une étape visant à automatiser la synchronisation des différents codes exécutables devra finalement être développée.

Dans cette étude, nous avons par ailleurs supposé que le développement des applications devant être exécutées sur le cœur de FPGA est réalisé par le biais des outils proposés par les fournisseurs de l'IP, où par le biais d'outils génériques. Nous avons en outre considéré que ces traitements sont spécifiés par l'utilisateur. L'automatisation de notre chaîne de développement passe donc également par le rapprochement des outils propres au FPGA avec la chaîne de développement de DART. Ceci implique la définition de nouvelles passes de transformation sous SUIF, destinées à extraire du code source les traitements de grains fins devant être exécutés sur le FPGA. Une fois encore, le problème de la synchronisation entre ces traitements et ceux exécutés sur les DPRs devra être étudié.

Une perspective à plus long terme a par ailleurs été évoquée dans le chapitre 3. Celle-ci concerne la possibilité d'exploiter les modèles d'exécution de type SMT supportés par DART. La mise en œuvre de ces techniques n'est pas triviale. Il faut dans un premier temps être en mesure de recevoir plusieurs descriptions C en points d'entrée, puis d'extraire de chacune d'elles, les traitements susceptibles d'être exécutés concurremment. Par la suite, le parallélisme intrinsèque de chacune des tâches devant être implémentée doit être extrait et des ressources doivent être allouées à chaque tâche, en fonction de leur complexité. Les synchronisations entre les différentes tâches sont également des difficultés qui doivent être impérativement surmontées. Dans ce mémoire, nous avons par ailleurs considéré la compilation de codes devant être exécutés sur un seul *cluster*. Cette chaîne doit ainsi évoluer de manière à supporter la prise en compte de plusieurs *clusters*.

Exploitation de DART

Suite à l'étude de l'architecture et des outils de développement de DART, il est également indispensable de mener une réflexion sur les différentes voies permettant d'exploiter cette architecture. Si à l'heure actuelle, la plupart des travaux ont porté sur la conception et l'exploitation d'un *cluster* de DART, il est ainsi nécessaire d'approfondir l'étude du contrôleur système, et du système d'exploitation, chargé de gérer un ensemble de *clusters*. Ceci est réalisé

dans le cadre de la thèse d'Imène Benkermi. L'étude de l'exploitation d'un *cluster* de DART comme une IP doit également être poursuivie. Celle-ci est réalisée dans le cadre d'un projet avec ST microelectronics qui vise notamment à coupler un *cluster* de DART avec un processeur VLIW (ST200) ou RISC (ARM).

Distribuer DART sous la forme d'une plateforme matérielle ciblant les applications mobiles de prochaines génération est un enjeu déterminant dans le cadre de l'exploitation de cette architecture. À ce titre, une perspective à ce travail consiste donc à proposer des outils permettant d'explorer les différentes options architecturales (nombre de *clusters*, nombre de DPRs, nombre et type d'unités fonctionnelles, ...) en fonction du domaine applicatif ciblé. Les outils de développement doivent par ailleurs évoluer afin de supporter différents paramètres architecturaux.

La souplesse de DART permet finalement d'envisager l'exploitation de cette architecture sous la forme d'une plateforme d'expérimentation autour de la reconfiguration. Compte-tenu de l'immaturité des recherches menées dans le cadre de la reconfiguration dynamique, il apparaît en particulier intéressant de disposer d'une plateforme permettant d'évaluer rapidement l'intérêt de différentes options d'implémentation. DART peut ainsi être utilisée afin d'évaluer l'impact de la reconfiguration sur les performances ou l'efficacité énergétique d'une architecture ou encore pour évaluer, sur une application donnée, l'intérêt de la reconfiguration au niveau fonctionnel par rapport à une reconfiguration au niveau logique. DART supportant de nombreux degrés de parallélisme (données, opérations, tâches, applications), cette architecture peut finalement être exploitée à des fins d'exploration de parallélisme.

Annexe A

Architecture du multiplieur de DART

Le multiplieur est une unité critique dans le DPR, tant du point de vue des performances que de la consommation. Cette unité est centrée sur un module supportant les multiplications de nombres signés codés sur 16 bits dans une arithmétique double précision (i.e. la sortie est codées sur 32 bits, sans perte de précision). Pour optimiser cet opérateur, un codage de Booth modifié a été associé à une structure de Wallace.

A.1 Le codage de Booth

La multiplication de deux nombres, illustrée par la figure A.1, peut se décomposer en deux étapes :

1. Génération des produits partiels
2. Addition des produits partiels

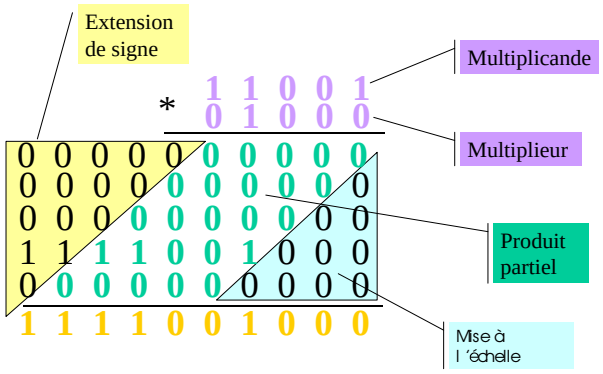

FIG. A.1 – *Exemple de multiplication de deux nombres signés. Cette opération peut grossièrement se décomposer en deux étapes : génération des produits partiels et addition des produits partiels. Chaque produit partiel est obtenu par une simple opération logique (et). Leur addition nécessite une extension de signe.*

Sachant que la génération des produits partiels est triviale (elle correspond à une simple opération *et* logique), l'optimisation de l'opérateur passe par la réduction du nombre des produits partiels à additionner [152]. À cette fin, Booth a introduit en 1951 un codage permettant de réduire le nombre de produits partiels et donc d'additions. Ce codage est une méthode de

réécriture permettant de faire apparaître un maximum de zéros dans l'écriture en base 2 du multiplieur. La méthode de réécriture est basée sur l'identité suivante :

$$\sum_{j=i}^{i+k} 2^j = 2^{i+k+1} - 2^i \tag{A.1}$$

Une chaîne telle que 0011111100 est alors remplacée par la chaîne 0010000010. Le nombre de '0' dans la chaîne est ainsi accru et par voie de conséquence le nombre de produits partiels non nuls décroît. Ainsi, au lieu de traiter la multiplication en examinant le multiplieur digit par digit, une fenêtre de 2 bits est observée.

Le résultat du recodage ne contient cependant pas systématiquement plus de '0' qu'à l'origine (e.g. 10101010 devient, après un codage de Booth : 11111110). Aussi, MacSorley a proposé en 1961 une modification du codage de Booth en élargissant la fenêtre d'observation à 3 bits. Cette modification autorise une réduction systématique du nombre de produits partiels d'un facteur 2. Le principe de cette modification est le suivant. Soit un nombre B (le multiplieur) codé en complément à 2 :

$$B = -2^{n-1}.b_{n-1} + \sum_{i=0}^{n-2} b_i.2^i \tag{A.2}$$

Le codage de Booth modifié transforme cette expression en :

$$B = \sum_{i,i=2p} (-2.b_{i+1} + b_i + b_{i-1}).2^{n-1-i} \tag{A.3}$$

soit :

$$B = \sum_{i,i=2p} \alpha_i.2^{n-1-i} \; avec \; \alpha_i \in \{-2,-1,0,1,2\} \tag{A.4}$$

Ainsi, à chaque pas i de calcul (i=0,2,4,...,n) trois bits du multiplieur sont examinés (eq. A.3) afin de déterminer le coefficient α_i qui définira le décalage et l'éventuel complément à effectuer (eq. A.4). Dès lors, la multiplication de A par B génère des produits partiels appartenant à l'ensemble {-2A,-A,0,A,2A}. Le nombre de pas de calcul correspondant à la moitié du nombre de bits nécessaires au codage de B, le nombre d'additions de produits partiels est réduit de moitié.

A.2 Structure de Wallace

Outre le codage de Booth modifié, qui réduit le nombre de produits partiels dans un multiplieur, il est possible d'accélérer la multiplication en introduisant une structure en arbre de Wallace dans l'opérateur. Cette structure arborescente est basée sur la cellule de full-adder qui est assimilée à un compteur en base 2, et appelée arbre de Wallace à 3 entrées. À partir de 3 entrées binaires (x,y,z), cette cellule génère 2 sorties (c,s) respectant l'équation suivante :

$$x + y + z = 2.c + s \tag{A.5}$$

L'association de telles cellules permet de construire des arbres plus complexes, permettant de compter un plus grand nombre de termes binaires. Ceci est illustré par la figure A.2. L'intérêt de cette structure est alors d'autoriser le traitement concurrent de tous les produits

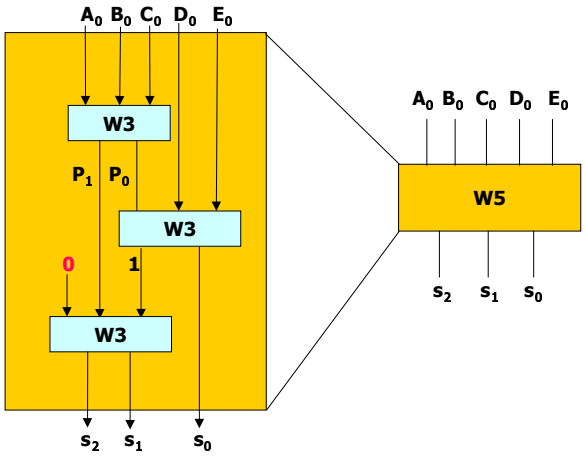

FIG. A.2 – *Arbre de Wallace à 5 entrées.Construite à partir de 3 cellules de Wallace à 3 entrées (Full-Adder), cette cellule permet de compter 5 digits. Le résultat de cette opération est alors* $S_2.2^2 + S_1.2^1 + S_0.2^0 = (A_0 + B_0 + C_0 + D_0 + E_0).2^0$

partiels. Par rapport à une solution plus classique où les produits partiels sont additionnés les uns après les autres (on parle alors de multiplieur tableau) cette solution offre ainsi l'avantage de combiner tous les produits partiels et de réduire à l'unité le nombre d'additions. Ceci est illustré par la figure A.3 qui représente l'arbre de Wallace nécessaire à l'addition de sept produits partiels codés sur 4 bits. L'introduction d'une structure de Wallace dans un multiplieur permet donc de limiter le temps nécessaire à l'addition des produits partiels lorsque la taille des entrées est importante. En effet ce temps, initialement proportionnel à la taille des opérandes, est désormais proportionnel au logarithme de la taille des entrées.

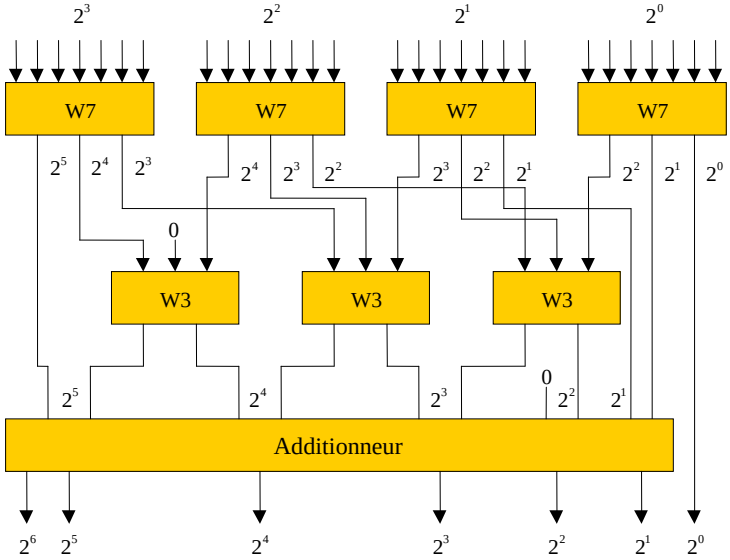

FIG. A.3 – *Addition de 7 données codées sur 4 bits suivant une structure de Wallace. Les deux premiers étages permettent de combiner les sept produits partiels. Le troisième réalise l'addition proprement dite.*

Annexe B

Architecture de l'unité arithmétique et logique de DART

Le chemin critique de l'unité arithmétique et logique de DART est principalement influencé par l'efficacité de l'additionneur. Celui-ci a donc été conçu avec une attention toute particulière.

Dans sa version la plus simple (additionneur séquentiel), un additionneur n bits est constitué de n cellules Full-Adder, calculant un bit de somme S_i et une retenue sortante C_{i+1} en fonction des entrées A_i, B_i et d'une retenue entrante C_i. Ces Full-Adders sont cascadés à la manière de la figure B.1. Les faibles performances de cet additionneur s'expliquent par l'absence de traitements concurrents. En effet, le calcul du $n^{ième}$ bit de somme ne peut être effectué que lorsque la $n\text{-}1^{ième}$ retenue est connue et donc que les n-1 premiers bits de sommes ont été calculés. Le chemin critique de cet opérateur est donc constitué de la chaîne de propagation de la retenue à travers les Full-Adders.

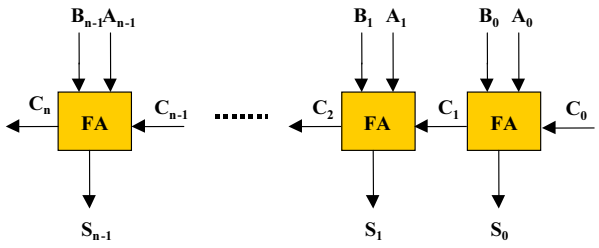

FIG. B.1 – *Structure d'un additionneur séquentiel n bits. Cette architecture est composée de n full-adders cascadés. Le calcul du $n^{ième}$ bit de sortie nécessite la connaissance de la $n-1^{ième}$ retenue et donc le traitement des n-1 premiers bits de somme.*

B.1 Principe de la propagation et de la génération : application à l'additionneur CLA

L'amélioration des performances d'un additionneur passe donc par la réduction de ce chemin critique. Les architectures visant cette accélération se basent généralement sur le principe de propagation-génération de la retenue. Celui-ci est né d'une constatation simple : lors de

l'addition de deux nombres A et B, codés en complément à deux, si A_i est égal à B_i alors il n'est pas nécessaire de connaître la retenue C_i pour calculer C_{i+1}. En effet :
- Si A_i=B_i=0 alors C_{i+1}=0
- Si A_i=B_i=1 alors C_{i+1}=1

Par conséquent, lorsque A_i=B_i il est possible d'additionner les bits supérieurs au rang i, sans attendre l'arrivée de C_{i-1}. Dans ce cas, la retenue est générée par A_i et B_i. Dans le cas contraire, si $A_i \neq B_i$, la retenue sortante est égale à la retenue entrante, soit $C_{i+1} = C_i$. C'est le principe de propagation. De nombreuses solutions se basent sur ces deux principes pour accélérer le traitement de l'opération d'addition. Toutes se basent sur les calculs suivants :

$$P_i = A_i \oplus B_i \text{ (aptitude du rang i à } \textit{propager} \text{ une retenue)} \tag{B.1}$$

$$G_i = A_i.B_i \text{ (aptitude du rang i à } \textit{générer} \text{ une retenue)} \tag{B.2}$$

En fonction de ces éléments de propagation et de génération, il est alors possible de déterminer les valeurs des retenues C_i en parallèle pour toutes les cellules Full-Adders. Pour cela l'équation B.3 doit être appliquée. Elle détermine pour chaque étage de rang i, si l'étage de rang $i-1$ a généré une retenue ou s'il a propagé une retenue générée plus tôt dans la chaîne de Full-Adders :

$$C_i = G_i + \sum_{j=0}^{i-1}(G_j . \prod_{k=0}^{i-j-1} P_k), \text{ avec } G_0 = C_0 \tag{B.3}$$

La formule permettant de déterminer C_i se complique à mesure que le rang i augmente. En conséquence, les additionneurs à retenue anticipée (CLA : Carry Look Ahead) limitent la taille des blocs de génération des couples (P_i, G_i) à 4 bits. Ces derniers ont alors une architecture comparable à celle représentée sur la figure B.2.

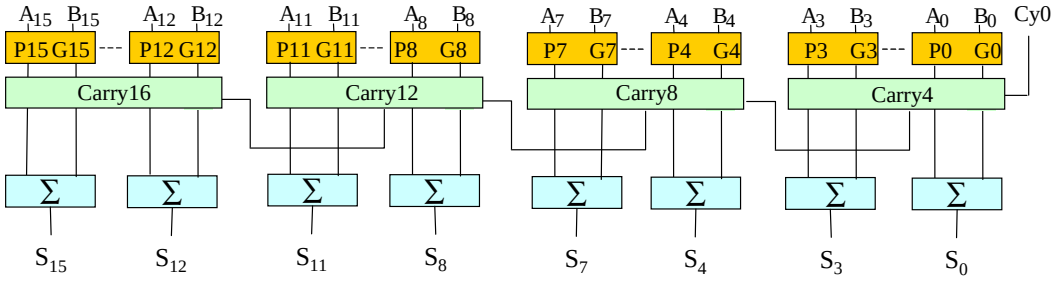

FIG. B.2 – *Structure d'un additionneur CLA 16 bits. Le premier étage calcule les fonctions de propagation et de génération pour chaque rang i. Le second détermine quant à lui les valeurs des retenues. Le dernier étage, constitué de portes XOR, calcule finalement les bits de somme.*

B.2 Additionneur de Sklansky

Bien qu'également basés sur les concepts de propagation et de génération évoqués précédemment, d'autres additionneurs exploitent une structure arborescente pour calculer les retenues C_i en fonction des couples (P_i, G_i). Dès lors, le temps de calcul n'est plus constant, mais proportionnel au logarithme de la taille des entrées. On parle alors d'additionneur à structure arborescente.

Pour expliquer la méthode exploitée dans ces additionneurs, considérons l'opérateur Δ tel que :

$$(\gamma_1, \pi_1)\Delta(\gamma_2, \pi_2) = (\gamma_2 + \gamma_1.\pi_2, \pi_1.\pi_2) \tag{B.4}$$

Cet opérateur est associatif, non commutatif et idempotent. Si γ_i et π_i sont définis comme suit :

$$(\gamma_i, \pi_i) = \begin{cases} (G_0, P_0) & \text{si i=0,} \\ (P_i, G_i)\Delta(\gamma_{i-1}, \pi_{i-1}) & \text{sinon.} \end{cases} \tag{B.5}$$

alors il est possible d'écrire :

$$\gamma_i = C_{i+1} \tag{B.6}$$

$$\pi_i = \prod_{j=0}^{i} P_j \tag{B.7}$$

Par le biais de cet opérateur, il est ainsi possible de calculer les termes C_i en évaluant (γ_0, π_O) puis (γ_1, π_1), ... L'intérêt de cette approche tient dans l'associativité de l'opérateur Δ qui permet de traiter le problème sous une forme arborescente. En effet, calculer $(P_0, G_0)\Delta(P_1, G_1)\Delta \ldots \Delta(P_{2p-1}, G_{2p-1})$ revient à traiter concurremment :

$$K1 = (P_0, G_0)\Delta(P_1, G_1)\Delta \ldots \Delta(P_{p-1}, G_{p-1}) \tag{B.8}$$

$$K2 = (P_p, G_p)\Delta(P_{p+1}, G_{p+1})\Delta \ldots \Delta(P_{2p-1}, G_{2p-1}) \tag{B.9}$$

Le résultat cherché est alors K1ΔK2. Cette approche a été utilisée pour concevoir différents types d'additionneur. Outre l'additionneur séquentiel, le tableau B.1 fait en particulier apparaître différentes variantes d'additionneurs à structure arborescente : *Kogge and Stone* [153], *Brent and Kung* [154], *Han and Carlson* [155] et *Sklansky* [156].

Architecture	**Surface(Full-Adder)**	**Latence (Full-Adder)**
Séquentiel	$n-1$	$n-1$
Brent and Kung	$2n - \log_2(n)$	$2.\log_2(n)$
Sklansky	$\frac{n}{2}.\log_2(n)$	$\log_2(n)$
Kogg and Stone	$n.(\log_2(n) - 1)$	$\log_2(n)$
Han and Carlson	$\frac{n}{2}.\log_2(n)$	$\log_2(n) + 1$

TAB. B.1 – *Performances de différentes architectures d'additionneurs à structure arborescente. L'efficacité de ces additionneurs est sans commune mesure par rapport aux additionneurs séquentiels. La surface et la latence des additionneurs sont données en nombre de Full-Adders, et dépendent de la largeur n des données.*

Compte-tenu de l'efficacité, en terme de latence et de surface, de l'additionneur de Sklantsky, celui-ci a été retenu pour constituer le cœur de l'unité arithmétique et logique de DART. L'architecture de cet additionneur est représentée sur la figure B.3 dans le cadre du traitement de données 16 bits.

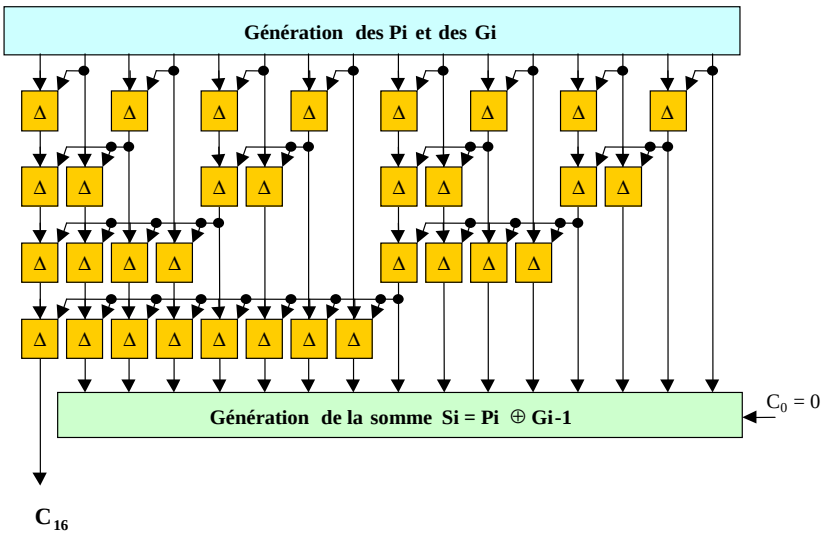

FIG. B.3 – *Structure d'un additionneur Sklantsky 16 bits. Le premier étage calcule les fonctions de propagation et de génération pour chaque rang i. Le second détermine quant à lui les valeurs des retenues, en adoptant une structure arborescente d'éléments* Δ*. Le dernier étage, constitué de portes XOR, calcule finalement les bits de somme.*

Glossaire

AAA ..... *Adéquation Algorithme Architecture*
ACS ..... *Add Compare Select*
AG ..... *Address Generator*
AMR ..... *Adaptative Multi-Rate*
ASIC ..... *Application Specific Integrated Circuit*
ASIP ..... *Application Specific Instruction set Processor*
BSS ..... *Breizh Synthesis System*
CAG ..... *Contrôle Automatique de Gain*
CAO ..... *Conception Assistée par Ordinateur*
CDFG ..... *Control Data Flow Graph*
CDMA ..... *Code Division Multiple Access*
CLA ..... *Carry Look Ahead (adder)*
CLB ..... *Configurable Logic Block*
CMOS ..... *Complementary Metal-Oxyde Semiconductor*
DAB ..... *Digital Audio Broadcasting*
DBT ..... *Dual Bit Type*
DECT ..... *Digital Enhanced Cordless Telecommunications*
DFG ..... *Data Flow Graph*
DLL ..... *Delay Locked Loop*
DMA ..... *Direct Memory Access*
DPR ..... *DataPath Reconfigurable*
DVB ..... *Digital Video Broadcasting*
DVS ..... *Dynamic Voltage Scaling*
EAB ..... *Embedded Array Block*
EFR ..... *Enhanced Full-Rate*
ESB ..... *Embedded System Block*
FIFO ..... *First In First Out*
FIR ..... *Finite Impulse Response (filter)*
FPGA ..... *Field Programmable Gate Array*
GALS ..... *Globaly Asynchronous Localy Synchronous*
GSM ..... *Global System for Mobile communications*

IIR ..... *Infinite Impulse Response (filter)*
ILP ..... *Instruction Level Parallelism*
IP ..... *Intellectuel Property*
IPC ..... *Instruction Per Cycle*
IS-95 ..... *Interim Standard-95*
ISS ..... *Instruction Set Simulator*
LAB ..... *Logic Array Block*
LE ..... *Logic Element*
MAC ..... *Multiply-ACcumulate*
MB ..... *MacroBloc*
MIPS ..... *Millions of Instructions Per Second*
MOPS ..... *Millions of Operations Per Second*
NRE ..... *Non-Recuring Engineering*
OVSF ..... *Orthogonal Variable Spreading Factor*
PAM ..... *Programmable Active Memory*
PDA ..... *Personal Data Assistant*
QCIF ..... *Quarter Common Intermediate Format*
QPSK ..... *Quadrature Phase Shift Keying*
RISC ..... *Reduced Instruction Set Computer*
RTL ..... *Register Transfer Level*
RTOS ..... *Real-Time Operating System*
SCMD ..... *Single Configuration Multiple Data*
SF ..... *Spreading Factor*
SIMD ..... *Single Instruction Multiple Data*
SMT ..... *Simultaneous Multi Threading*
SoC ..... *System on Chip*
SUIF ..... *Stanford University Intermediate Format*
SWP ..... *Sub-Word Processings*
T3G ..... *Télécommunications de troisième génération*
T4G ..... *Télécommunications de quatrième génération*
TLP ..... *Tread Level Parallelism*
UAL ..... *Unité Arithmétique et Logique*
UF ..... *Unité Fonctionnelle*
UMTS ..... *Universal Mobile Telecommunication System*
VLIW ..... *Very Long Instruction Word*
WCDMA ..... *Wide-Band Code Division Multiple Access*

Bibliographie

[1] E. Gaudry. Estimation de la complexité algorithmique d'une chaîne de traitement WCDMA en télécommunication mobile : application au processeur Lx. Master's thesis, DEA STIR, ENSSAT-Université de Rennes 1, June 2001.

[2] ITRS. International technology roadmap for semiconductor : 2002 update. http :public.itrs.net, 2002.

[3] R. Hartenstein. A Decade of Reconfigurable Computing : A Visionary retrospective. In *Design Automation and Test in Europe (DATE 01)*, Munich, Germany, March 2001.

[4] A. Dehon and J. Wawrzynek. Reconfigurable Processing : What, Why and implication for design Automation. In *Design Automation Conference (DAC 99)*, pages 610–615, New Orleans, USA, June 1999.

[5] J. M. Rabaey. Reconfigurable Processing : the Solution to Low-Power Programmable DSP. In *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, April 1997.

[6] H. Zhang, M. Wan, V. George, and J. Rabaey. Interconnect Architecture Exploration for Low-Energy Reconfigurable Single-Chip DSPs. In *International Workshop on VLSI*, April 1999.

[7] A. Dehon. *Reconfigurable Architectures for General-Purpose Computing.* PhD thesis, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, October 1996.

[8] J. L. Hennessy and D. A. Patterson. *Architecture des ordinateurs : une approche quantitative (deuxième édition).* International Thomson Publishing, 1996.

[9] R. Petersen and B.L. Hutchings. An assessment of the suitability of FPGA-based systems for use in digital signal processing. In *International Workshop on Field Programmable Logic and Applications (FPL 95)*, Oxford, England, August 1995.

[10] R. David, D. Lavenier, and S. Pillement. Architectures reconfigurables : du niveau porte au niveau système. *Techniques et Sciences Informatiques*, deuxième soumission.

[11] A. Varma and C.S. Raghavendra. *Interconnection Networks for multiprocessors and multicomputers : Theory and Practice.* IEEE Computer Society Press, 1994.

[12] S. Pillement. *Méthodologies d'évaluation et de prototypage des systèmes numériques intégrés.* PhD thesis, Université de Montpellier II, December 1998.

[13] S. Rubini and D. Lavenier. Les architectures reconfigurables. *Calculateurs Parallèles*, 9(1) :9–27, 1997.

[14] Y. Solihin, K. Cameron, Y. Luo, D. Lavenier, and M. Gokhale. Mutable functional units and their applications on microprocessors. In *International Conference on Computer Design (ICCD 01)*, Austin, Texas, USA, 2001.

[15] Xilinx. *MicroBlaze Harware Reference Guide*, October 2001.

[16] Altera. *Nios Soft Core Embedded Processor*, June 2000.

[17] C. Tavernier. *Circuits logiques programmables.* Microcontrôleur et environnement. Dunod, 1996.

[18] L. Dutrieux and D. Demigny. *Architecture des FPGA et CPLD, méthode de conception, le langage VHDL.* Eyrolles, 1997.

[19] Xilinx Inc. *VIRTEX2 1.5V Series Field Programmable Gate Arrays*, July 2001.

[20] Altera. *APEX20k Programmable Logic Device Family*, August 2001.

[21] Y. Chou, P. Pillai, H. Schmit, and J. Shen. PipeRench Implementation of the Instruction Path Coprocessor. In *International Symposium on Microarchitecture (MICRO-33)*, pages 147–158, Monterey, USA, December 2000.

[22] S. Goldstein, H. Schmit, M. Moe, M. Budiu, and S. Cadambi. PipeRench : A Coprocessor for Streaming Media Acceleration. In *International Symposium on Computer Architecture (ISCA 99)*, Atlanta, USA, May 1999.

[23] A. Marshall, J. Vuillemin, T. Stansfield, I. Kostarnov, and B. Hutchings. A Reconfigurable Arithmetic Array for Multimedia Applications. In *International Symposium on Field Programmable Gate Arrays (FPGA 99)*, pages 135–144, February 1999.

[24] D. Cherepacha and D. Lewis. A Datapath Oriented Architecture for FPGAs. In *International Symposium on Field Programmable Gate Arrays (FPGA 94)*, Monterey, USA, February 1994.

[25] J. Becker, T. Pionteck, and M. Glesner. DReAM : A Dynamically Reconfigurable Architecture for Future Mobile Communication Applications. In *international Workshop on Field Programmable Logic and Applications (FPL 00)*, pages 312–321, Villach, Austria, August 2000. Lecture Notes in Computer Science 1896.

[26] Altera. *Flex 8000 Programmable Logic Device Family*, June 1999.

[27] J. Vuillemin, P. Bertin, A. Smith, and H. Silvermen. Programmable Active Memories : Reconfigurable Systems Come of Age. *IEEE Transaction of VLSI Systems*, 4(1), April 1996.

[28] J. M. Arnold. The splash 2 software environment. *The Journal of Supercomputing*, 9(3) :277–290, 1993.

[29] B. Pottier. *ArMen : une machine parallèle intégrant un réseau de circuits logiques programmables.* PhD thesis, Université de Rennes 1, June 1991.

[30] J. Hauser. *Augmenting a microprocessor with reconfigurable hardware.* PhD thesis, University of California, Berkeley, 2000.

[31] C. Rupp, M. Landguth, T. Graverick, E. Gomersall, and H. Holt. The NAPA Adaptative Processing Architecture. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 98)*, pages 28–37, April 1998.

[32] M. Borgatti, F. Lertora, B. Forêt, and L. Calí. A Reconfigurable System featuring Dynamically Extensible Embedded Microprocessor, FPGA and Customisable I/O. In *Custom Integrated Circuits Conference (CICC)*, Orlando, USA, May 2002.

[33] M2000. *Flexeos family technical manual.* www.m2000.fr.

[34] R. Maestre, M. Fernandez, F.J. Kurdahi, N. Bagherzadeh, and H. Singh. Configuration Management in Multi-Context Reconfigurable Systems for Simultaneous Performance and Power Optimization. In *International Symposium on System Synthesis*, pages 107–113, September 2000.

[35] ATMEL. 5K -50K Gates Coprocessor FPGA with FreeRAM. Technical report 0896c-rev 04/02, ATMEL, April 2002.

[36] Didier Demigny, Michel Paindavoine, and Serge Weber. Architecture à reconfiguration dynamique pour le traitement temps réel des images. *Technique et Science de l'Information Numéro Spécial Architectures Reconfigurables*, 18(10) :1087–1112, December 1999.

[37] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee. CHIMAERA : a high-performance architecture with a tightly-coupled reconfigurable functional unit. *ACM SIGARCH Computer Architecture News*, 28(2) :225–235, May 2000.

[38] D. Rizzo and O. Colavin. A video compression case study on a reconfigurable vliw architecture. In *Design Automation and Test in Europe (DATE 02)*, Paris, France, March 2002.

[39] J. Rabaey. A low-energy heterogeneous reconfigurable DSP IC. In *Design Automation Conference (DAC 00)*, Los Angeles, USA, June 2000.

[40] D. C. Cronquist, P. Franklin, C. Fisher, M. Figueroa, and C. Ebeling. Architecture Design of Reconfigurable Pipelined Datapath. In *Advance Research in VLSI (ARVLSI 99)*, pages 23–40, Atlanta, USA, March 1999.

[41] X. Tang, M. Aalsma, and R. Jou. A compiler directed aproach to hiding configuration latency in chameleon processors. In *International Workshop on Field Programmable Logic and Applications (FPL 00)*, Villach, Autriche, April 2000. Lecture Notes in Computer Science 1896.

[42] M. Lee, H. Signh, G. Lu, N. Bagherzadeh, and F. Kurdahi. Design and Implementation of the MorphoSys Reconfigurable Computing Processor. *Journal of VLSI and Signal Processing-Systems for Signal, Image and Video Applications*, 24(2) :147–164, March 2000.

[43] G. Smit P. Heysters. Mapping of DSP Algorithms on the MONTIUM Architecture. In *International Reconfigurable Architecture Workshop (RAW 03)*, Nice, France, April 2003.

[44] G. Smit, P. Havinga, P. Heysters, and M. Rosien. Dynamic Reconfiguration in Mobile Systems. In *International Conference on Field Programmable Logic and Applications (FPL 02)*, pages 171–181, Montpellier, France, September 2002. Lecture Notes in Computer Sciences 2438.

[45] V. Baumgarte, F. May, A. Nükel, M. Vorbach, and M. Weinhardt. PACT XPP - A self-Reconfigurable Data Processing Architecture. In *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA 01)*, Las Vegas, USA, June 2001.

[46] G. Sassatelli, L. Torres, J. Galy, G. Cambon, and C. Diou. The systolic ring : A dynamically reconfigurable architecture for embedded systems. In *International Workshop on Field Programmable Logic and Applications (FPL 01)*, pages 409–419. Lecture Notes in Computer Science 2147, 2001.

[47] A. Abnous and J. Rabaey. *Ultra Low-Power Specific Multimedia processors*, volume November. VLSI Signal Processing IX, ieee press edition, 1996.

[48] D. C. Cronquist, P. Franklin, S. G. Berg, and C. Ebeling. Specifying and Compiling Applications for RaPiD. In *Symposium on Field-Programmable Custom Computing Machines (FCCM 98)*, pages 116–125, Los Alamitos, USA, April 1998.

[49] PACT. The XPP White Paper : A Technical Perspective. Release 2.1, PACT, March 2002.

[50] M. Wan, H. Zhang, M. Benes, and J. Rabaey. A Low-Power Reconfigurable Data-Flow Driven DSP System. In *Workshop on Signal Processing Systems (SIPS 99)*, Taipei, Taiwan, October 1999.

[51] D. Bursky. Upgraded DSP Core Tackles future Communication needs. *Electronic design*, 48(8) :66 – 68, Avril 2000.

[52] Jennifer Eyre and Jeff Bier. Carmel enables Customizable DSP. *Microprocessor report*, 12(17), decembre 1998.

[53] P. Kievits, E. Lambers, C. Morman, and R. Woudsma. R.E.A.L. DSP Technology for Telecom Baseband Processing. Technical report, Philips Semiconductors, ASIC Service Group, 1998.

[54] E. van der Horst, W. Kloosterhuis, and J. van der Heyden. A C Compiler for the Embedded R.E.A.L. DSP. In *International Conference on Signal Processing (ICSPAT 98)*, Toronto, Canada, September 1998.

[55] R. David. Analyse qualitative des nouvelles architectures embarquées et outils associés : étude du cas des processeurs ARM et DSP. Mémoire DEA, Univeristé de Rennes1/ INSA/ SUPELEC, July 2000.

[56] S. Pillement, R. David, and O. Sentieys. Architectures reconfigurables : Opportunités pour la faible consommation. In *papier invité aux journées Faible Tension Faible Consommation (FTFC)*, Paris, France, May 2003.

[57] B. Brodersen. Wireless Systems-on-a-Chip Design. In *Invited paper in International Symposium on Quality Electronic Design (ISQED 02)*, San Jose, California, USA, March 2002.

[58] K.K.W. Poon. Power Estimation for Filed Programmable Gate Arrays. Master's thesis, University of British Columbia, 2002.

[59] L. Shang, A.S. Kaviani, and K. Bathala. Dynamic Power Consumption in Virtex-II FPGA family. In *International Symposium on Field Programmable Gate Arrays (FPGA 02)*, pages 157–164, Monterey, USA, February 2002.

[60] E. Kusse. Analysis and Circuit Design for Low Power Programmable Logic Modules. Master's thesis, University of California, Berkeley, 1997.

[61] A.D. Garcia. *Etude sur l'estimation et l'optimisation de la consommation de puissance des circuits logiques programmables du type FPGA*. PhD thesis, Ecole Nationale Supérieure des Télécommunications, 2000.

[62] V. George. *Low Energy Field-Programmable Gate Array*. PhD thesis, University of California, Berkeley, 2000.

[63] H. Zhang and J. Rabaey. Low-Swing Interconnect Interface Circuits. In *International Symposium on Low Power Electronics and Design (ISLPED 98)*, pages 161–166, New York, USA, August 1998.

[64] G. Stitt, B. Grattan, J. Villarreal, and F. Vahid. Using On-Chip Configurable Logic to Reduce System Software Energy. In *Symposium on Field-Programmable Custom Computing Machines (FCCM 02)*, Napa, California, September 2002.

[65] J. Villarreal, D. Suresh, G. Stitt, F. Vahid, and W. Najjar. Improving Software performance with Configurable Logic. *Design Automation for Embedded Systems*, 7(4) :325–339, November 2002.

[66] J. Rabaey. Silicon Platforms for the Next-Generation Wireless Systems. In *Tutorial in Symposium on High-Performance Chips (HOT Chips 01)*, August 2001.

[67] Lucent Technologies. Dsp 16xx. Technical report, Lucent Technologies.

[68] Analog Devices. ADSP-2100 Family DSP microcomputers. Datasheet rev b., Analog Devices, 1996.

[69] M. Hiraki, R. S. Bajawa, H. Kojima, D.J. Gomy, K. Nitta, A. Shridhar, K. Sasaki, and K. Seki. Stage-Skip : A Low-Power Processor Architecture Using a Decoded Instruction Buffer. In *International Symposium on Low Power Electronics and Design (ISLPED 96)*, Monterey, USA, August 1996.

[70] J. Fridman. Sub-Word Parallelism in Digital Signal Processing. *IEEE Signal Processing Magazine*, 17(2) :27–35, March 2000.

[71] P. Faraboshi, G. Desoli, and J. A. Fisher. The Latest Word in Digital and Media Processing. *IEEE Signal Processing Magazine*, pages 59 – 85, March 1998.

[72] V. S. Krishnan. *Speculative Multithreading Architectures.* PhD thesis, University of Illinois, 1998.

[73] Altera. Stratix FPGA Family. Datasheet ver. 3.0, December 2002.

[74] D. Menard, P. Quemerais, and O. Sentieys. Influence of Fixed-point DSP architecture on Computation Accuracy. In *European Signal Processing Conference (ESIPCO 02)*, Toulouse, France, September 2002.

[75] J.B. Groe and L.E. Larson. *CDMA Mobile Radio Design.* Harteck House Publishers, 2000.

[76] P. Faraboshi, G. Brown, J. Fisher, and G. Desoli. Lx : A technology Platform for Customizable VLIW Embedded Processing. In *International Symposium on Computer Architecture (ISCA 00)*, Vancouver, Canada, June 2000.

[77] T. Ojanpera and R. Prasard. *Wideband CDMA For Third Generation Mobile Communication.* Hartek Publishers, 1998.

[78] P. Faraboshi, J.A. Fisher, and C. Young. Instruction scheduling for instruction level parallel processors. *Proceedings of the IEEE*, 89(11) :1638–1659, November 2001.

[79] Texas Instruments. *TMS320C64x Technical Overview.* Texas Instruments, February 2000.

[80] D. Demigny, N. Boudouani, N. Abel, and L. Kessal. La rémanance des architectures reconfigurables : un critère significatif de classification des architectures. In *Journées Francophones sur l'Adéquation Algorithme Architecture (JFAAA 02)*, Monastir, Tunisie, December 2002.

[81] Xilinx Inc. *Xilinx 6200 Preliminary Data Sheet.* San Jose, CA, 1996.

[82] R. David, S. Pillement, O. Sentieys, and D. Chillet. Architectures enfouies reconfigurables dynamiquement. In $7^{ème}$ *SYMPposium en Architectures nouvelles de machines (SYMPA'7)*, pages 23–32, Paris, France, Avril 2001.

[83] R. David. Vers la définition d'une architecture enfouie reconfigurable dynamiquement. Rapport Interne LASTI, mars 2001.

[84] O. Wolf and J. Bier. TigerSHARK Sinks Teeth into VLIW. *Microprocessor report*, 12(16), December 1998.

[85] J.P. Wittenburg, P. Pirsh, and G. Meyer. A Multithreaded Architecture Approach to Parallel DSPs for High-Performance Image Processing Applications. In *Workshop on Signal Processing Systems (SIPS 99)*, Taipei, Taiwan, October 1999.

[86] J. Rabaey and M. Pedram. *Low Power Design Methodologies*. Kluwer Academic, 1996.

[87] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *International Symposium on Operating Systems Principles (SOSP 01)*, pages 89–102, Banff, Alberta, Canada, 2001.

[88] J. Pouwelse, K. Langendoen, and H. Sips. Dynamic Voltage Scaling on a Low-Power Microprocessor. In *International conference on Mobile computing and networking (MobiCom 01)*, pages 251–259, Rome, Italy, June 2001.

[89] I. Benkermi. Systèmes temps réel et ordonnancements. Rapport interne, LASTI, December 2002.

[90] G. Snider, B. Schackleford, and R. J. Carter. Attacking the Semantic Gap Between Application Programming Langage and Configurable Hardware. In *International Symposium on Field Programmable Gate Arrays (FPGA 01)*, pages 115–124, Monterey, USA, February 2001.

[91] M. Coppola, S. Curaba, M. Grammatikakis, G. Maruccia, and F. Papariello. On-Chip Communication Network : User Manual V1.0. Technical report, ST Microelectronics, April 2003.

[92] D. Flynn. AMBA : Enabling Reusable On-Chip Design. *IEEE Micro*, 17(4) :20–27, August 1997.

[93] H. Tanaka. Data structure of huffman codes and its application to efficient encoding and decoding. *IEEE Transaction on Information and Theory*, 33 :154–156, January 1987.

[94] M. Lenoble. Description d'un circuit programmable enfoui dans un ASIC. projet de fin d'étude ENSSAT, April 2003.

[95] L. Lagadec. *Abstraction, modélisation et outils de CAO pour les circuits intégrés reconfigurables*. PhD thesis, Université de Rennes 1, 2000.

[96] J.M.P. Cardoso and H.C. Neto. Compilation for FPGA-based Reconfigurable Hardware. *IEEE Design and Test of Computers*, 20(2) :65–75, March 2003.

[97] T. Saidi. Conception de fonctions de base de la reconfiguration dynamique. Master's thesis, ENSSAT, August 2001.

[98] J.J.F. Cavanagh. *Digital Computer Arithmetic, design and implementation*. Mc Graw-Hill Computer Science Series, 1984.

[99] G. Koutroumpezis, K. Tatas, D. Soudris, S. Blionas, K. Masselos, and A. Thanailakis. Architecture Design of a Reconfigurable Multiplier for Flexible Corse-grain Implementations. In *International Conference on Field Programmable Logic and Applications (FPL 02)*, La grande Motte, France, September 2002. Lecture Notes in Computer Science 2438.

[100] H. Meyer. *Analysis and Design of Low Power Digital Multipliers*. PhD thesis, Carnegie Mellon University, Pennsylvania, USA, August 1999.

[101] P. Lapsley, J. Bier, A. Shoham, and E. A. Lee. *DSP Processor Fundamentals, Architectures and Features*. IEEE Press, 1997.

[102] R. David, D. Chillet, S. Pillement, and O. Sentieys. DART : A Dynamically Reconfigurable Architecture dealing with Next Generation Telecommunications Constraints. In *International Reconfigurable Architecture Workshop (RAW 02)*, Fort lauderdale, USA, April 2002.

[103] Z. Huang and S. Malik. Managing Dynamic Reconfiguration Overhead in System-on-a-Chip Design Using Reconfigurable Datapath and Optimized Interconnect Networks. In *Design Automation and Test in Europe (DATE 01)*, Munich, Germany, March 2001.

[104] K. Keutzer, S. Malik, A. R. Newton, J.M. Rabaey, and A.Sangiovani-Vincentelli. System-Level Design : Orthogonalisation of Concerns and Plateform-Based Design. *IEEE Transaction on Computer-Aided Design*, 19(12) :1523–1543, December 2000.

[105] Dick Poutain and David May. *A tutorial Introduction to OCCAM Programming.* INMOS, 1988.

[106] G.S. Stiles and B. Young. *Transputer research and applications, 1.* Salt Lake City, Utah, April 1989.

[107] K. Compton and S. Hauck. Reconfigurable Computing : A Survey of Systems and Software. *ACM Computing Surveys*, 34(2) :171–210, June 2002.

[108] R. Schreiber, S. Aditya, S. Mahlke, V. Kathail, B. Ramakrishna Rau, D. Cronquist, and M. Sivaraman. PICO-NPA : High-Level Synthsis of NonProgrammable Hardware Accelerators. Technical Report HPL-2001-249, Hewlett-Packard Labortories, 2001.

[109] M. Wan. *Design Methodology for Low Power Heterogeneous Digital Signal Processors.* PhD thesis, Berkeley Wireless Design Center, 2001.

[110] S.-F. Li, M. Wan, and J. Rabaey. Configuration Code Generation and Optimization for Low-Energy Reconfigurable DSPs. In *International Workshop on Signal Processing Systems (SiPS 99)*, Taipei, Taiwan, September 1999.

[111] Timothy J. Callahan, John R. Hauser, and John Wawrzynek. The Garp architecture and C compiler. *IEEE Computer*, 33(4) :62–69, April 2000.

[112] R. Wilson and al. SUIF : An Infrastructure for Research on Parallelizing and Optimizing Compilers. Technical report, Computer Systems Laboratory, Stanford University, May 1994.

[113] V. Messé. *Production de compilateurs flexibles pour la conception de processeurs programmables spécialisé.* PhD thesis, Université de Rennes 1, March 1999.

[114] O. Sentieys, J.P. Diguet, and J.L. Philippe. GAUT : A High Level Synthesis Tool Dedicated To Real Time Signal Processing Applications. In *European Design Automation Conference (EURODAC 95)*, Brighton, UK, September 1995.

[115] Open SystemC Initiative. SystemC version 2.0 Uers's Guide. Technical report, 2001.

[116] G. Aigner, A. Diwan, D. L. Heine, M.S. Lam, D. L. Moore, B. R. Murphy, and C. Sapuntzakis. An overview of the SUIF2 compiler Infrastructure. Technical report, Computer Systems Laboratory, Stanford University, 1999.

[117] T. Saidi. Déroulage de boucle sous SUIF. Technical report, LASTI, 2002.

[118] A. Fraboulet. *Optimisation de la mémoire et de la consommation des systèmes multimédia embarqués.* PhD thesis, INSA de Lyon, November 2001.

[119] A. Fraboulet. Optimisation de la mémoire et de la consommation des systèmes multimédia embarqués. In *Troisième colloque CAO*, May 2002.

[120] A. Fraboulet, K. Godary, and A. Mignotte. Loop fusion for memory space optimization. In *International Symposium on Systems Synthesis (ISSS 01)*, Montreal, Canada, October 2001.

[121] A. Fraboulet, G. Huard, and A. Mignotte. Loop alignment for memory accesses optimization. In *Symposium on Systems Synthesis (ISSS 99)*, San Jose, USA, November 1999.

[122] M. Potkonjak and J.M. Rabaey. *VLSI Design Methodologies for Digital Signal Processing*, chapter Exploring the Algorithmic Design Space Using High Level Synthesis, pages 131–167. International Series in Engineering and Computer Science : VLSI,. 1994.

[123] I. Lemke and G. Sander. Visualization of Compiler Graphs. Design report d 3.12.1-1, Universität des Saarlandes, 1993.

[124] F. Djieya and F. Charrot. Approche d'estimation flexible de performance pour DSP. In $7^{ème}$ *SYMPposium en Architectures nouvelles de machines (SYMPA'7)*, Paris, France, December 2001.

[125] D. Menard. Méthodolgies d'implantation d'algorithmes spécifiés en virgule flottante dans les architectures en virgule fixe. Technical report, ENSSAT-LASTI, December 2000.

[126] G.J. Chaintin. Register Allocation ans spilling via graph coloring. In *SIGPLAN*, 1982.

[127] F. Charot, G. Le Fol, and V. Messe. Programmable processor modeling for retargetable compiler design and architecture exploration. Technical Report 1167, IRISA, November 1998.

[128] J. Laurent. *Estimation de la consommation dans la conception système des applications embarquées temps réels*. PhD thesis, Université de Bretagne Sud, December 2002.

[129] T. Grötker and S. Liao and G. Martin and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[130] W. Mueller, J. Ruf, D. Hoffmann, J. Gerlach, T. Kropf, and W. Rosenstiehl. The simulation semantics of systemc. In *Design Automation and Test in Europe (DATE 01)*, Munich, Germany, March 2001.

[131] Coware. Flexible Platform Based Design with the Coware N2C Design System. Technical report, CoWare Inc., October 2000.

[132] Synopsys. CoCentric SystemC Compiler, RTL User and modeling guide. Technical report, Synopsys Inc., August 2001.

[133] Synopsys. Power Products Reference Manual. Technical Report 1999.10, Synopsys, 1999.

[134] M. Denoual. *Estimation de haut niveau de la consommation de systèmes sur Silicium*. PhD thesis, ENSSAT-Université de Rennes 1, October 2001.

[135] D. Saille. *Optimisation de la conception des unités de mémorisation pour les applications de traitement du signal embarquées*. PhD thesis, Université de Rennes1, 2003.

[136] S. Wilton and N. Jouppi. An enhanced access and cycle time model for on-chip caches. Technical report, Digitak-Western Research Laboratory, July 1994.

[137] P. Landman and J. Rabaey. Black-Box Capacitance Models For Architectural Power Analysis. In *Low Power Design Workshop*, April 1994.

[138] P. Landman and J. Rabaey. Activity-sensitive Architectural Power Analysis. *IEEE Transaction on Computer Aided Design*, 15(6), June 1996.

[139] C. Piguet, S. Cserveny, J.-F. Perotto, and J.-M. Masgonty. Techniques de circuit et méthodes de conception pour réduire la consommation statique dans les technologies profondément sub-microniques. In *Journées d'études Faible Tension Faible Consommation (FTFC 03)*, Paris, France, May 2003.

[140] ETSI. Universal Mobile Telecommunication System (UMTS) : Physical channels and mapping transport channels onto physical channels (FDD). 3G TS 25.211 version 3.3.0, ETSI, June 2000.

[141] E. Dinan and B. Jabbari. Spreading Codes for Direct Sequence CDMA and Wideband CDMA Cellular Network. *IEEE Communications Magazine*, 36(9) :48–54, September 1998.

[142] A.F. Molisch. *Wideband Wireless Communications*. Prentice Hall, November 2000.

[143] Texas Instruments. TMS320C6414/15/16 Fixed-Point Digital Signal Processors. Application report SPRS146G, Texas Instruments, March 2003.

[144] D. Menard, M. Guitton, R. David, S. Pillement, and O. Sentieys. Évaluation comparative de plate-formes reconfigurables et programmables pour les télécommunications de $3^{ème}$ génération. In *Colloque GRETSI sur le traitement du signal et des images (GRETSI 03)*, Paris, France, September 2003.

[145] D. Menard, M. Guitton, S. Pillement, and O. Sentieys. Design and Implementation of WCDMA Platforms : Challenges and Trade-offs. In *International Signal Processing Conference (ISPC 03)*, Dallas, USA, October 2003.

[146] Texas Instruments. TMS320C6414/15/16 Power Consumption Summary. Application report, Texas Instruments, spra811a, March 2002.

[147] L. Hanzo and E.-L. Kuan P. Cherriman. Interactive cellular and cordless video telephony : State-of-the-art system design principles and expected performance. *Proccedings of the IEEE*, September 2000.

[148] A. Doufexi, S. Armour, P. Karlsson, A. Nix, and D. Bull. A comparison of HIPERLAN/2 and IEE802.11a. White paper : easy.intranet.paper4.pdf, 2000.

[149] R. Van Nee and R. Prasad. *OFDM for Wireless Multimedia Communications*. Universal Persoanal Communications. Hartek House Publishers, 2000.

[150] M. Huemer, R. Weigle, and L. Reindl. A review of Cyclically Extended Single Carrier Transmission with Frequency Domain Equalization for Broadband Wireless Transmission. *ETT European Transactions on Communications*, to be published.

[151] F. Blanc. *Étude d'un nouveau concept de calculateur reconfigurable : Architecture et Outils*. PhD thesis, CEA, Saclay, France, December 2002.

[152] J-M. Muller. *Arithmétique des ordinateurs, opérateurs et fonctions élémentaires*. Masson, 1989.

[153] P.M. Kogge and H.S. Stone. A Parallel Algorithm for the Efficient Solution of a General Purpose Class of Recurrence Equations. *IEEE transaction on Computers*, 22(8) :783–791, August 1973.

[154] H.T. Kung and R.P Brent. A Regular Layout for Parallel Adders. *IEEE transaction on Computer*, C-31 :260–264, March 1982.

[155] D.A. Carlson and T. Han. Fast Area-Efficient VLSI adders. In *International Symposium on Computer Arithmetic (ISCA 87)*, pages 49–56, May 1987.

[156] J. Sklantski. Conditional Sum Addition Logic. *IRE transaction on Electronic Computers*, 9(2) :226–231, 1960.

Publications personnelles

Contributions à des ouvrages scientiques

[1] **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. Sélection des meilleures contributions de IFIP 12^{th} International Conference on Very Large Scale Integration (VLSI'01). Kluwer Academic Publishers, 2002.

[2] **R. David**, D. Lavenier, and S. Pillement. *Architecture des ordinateurs : Le processeur sous toutes ses formes*, chapter 5 : Les Architectures Reconfigurables. en cours d'édition.

[3] O.Sentieys, **R. David**, and S. Pillement. *Low Power Electronics Design*, chapter Low Power Reconfigurable Processsor. CRC Press, En cours d'édition.

Revues nationales

[4] **R. David**, S. Pillement, and D. Lavenier. Architectures reconfigurables : du niveau porte au niveau système. *Techniques et Sciences Informatiques*, deuxième soumission.

Conférences internationales

[5] **R. David**, D. Chillet, S. Pillement, and O. Sentieys. A Dynamically Reconfigurable Architecture for Low-Power Multimedia Terminals. In *IFIP* 12^{th} *International Conference on Very Large Scale Integration (VLSI'01)*, Montpellier, France, December 2001.

[6] **R. David**, D. Chillet, S. Pillement, and O. Sentieys. DART : A Dynamically Reconfigurable Architecture dealing with Next Generation Telecommunications Constraints. In *9th International Workshop on Reconfigurable Architecture (RAW'02)*, Fort Lauderdale, USA, April 2002.

[7] **R. David**, D. Chillet, S. Pillement, and O. Sentieys. Mapping Future Generation Mobile Telecommunication Applications on a Dynamically Reconfigurable Architecture. In *International Conference on Accoustics, Speech and Signal Processing 2002 (ICASSP 2002)*, Orlando, USA, May 2002.

[8] **R. David**, D. Chillet, S. Pillement, and O. Sentieys. A compilation framework for a dynamically reconfigurable architecture. In 12^{th} *International Conference on Field Programmable Logic and applications (FPL'02)*, La grande Motte, France, September 2002.

[9] **R. David**, D. Chillet, S. Pillement, and O. Sentieys. A high-performance dynamically reconfigurable embedded architecture. In 5^{th} *Sophia Antipolis forum on MicroElectronics (SAME)*, Sophia Antipolis, France, October 2002.

Conférences nationales

[10] **R. David**, S. Pillement, O. Sentieys, and D. Chillet. Architectures enfouies reconfigurables dynamiquement. In $7^{ème}$ *SYMPposium en Architectures nouvelles de machines (SYMPA'7)*, pages 23–32, Paris, France, Avril 2001.

[11] **R. David**, D. Chillet, S. Pillement, and O. Sentieys. Flot de conception pour plateforme reconfigurable. In *Troisième Colloque CAO*, Paris, France, May 2002.

[12] S. Pillement, **R. David**, and O. Sentieys. Architectures reconfigurables : Opportunités pour la faible consommation. In *Papier invité aux* $4^{ème}$ *Journées Francophones Faible Tension Faible Consommation (FTFC 03)*, Paris, France, May 2003.

[13] D. Menard, M. Guitton, **R. David**, S. Pillement, and O. Sentieys. Évaluation comparative de plate-formes reconfigurables et programmables pour les télécommunications de $3^{ème}$ génération. In $19^{ème}$ *colloque GRETSI sur le traitement du signal et des images*, Paris, France, September 2003.

Rapports de recherche internes

[14] **R. David**. Analyse qualitative des nouvelles architectures embarquées et outils associés : Étude du cas des processeurs ARM et DSP. Mémoire de DEA, Université de Rennes1/ INSA/ SUPELEC, juillet 2000.

[15] **R. David**. Vers la définition d'une architecture enfouie reconfigurable dynamiquement. Rapport interne, LASTI, mars 2001.

[16] **R. David**. Modèles de programmation de DART et état d'avancement de la chaîne de développement. Rapport interne, LASTI, juin 2002.

Résumé

Conjointement à l'évolution rapide et constante des technologies de la microélectronique, l'essor des applications embarquées grand public amène des contraintes de conception sans précédents. Aux traditionnelles exigences de performance et de moindre coût, s'ajoutent désormais des contraintes de faible consommation et de flexibilité qui imposent la définition de nouveaux paradigmes architecturaux.

À cette fin, nous étudions ici une architecture enfouie reconfigurable dynamiquement, DART. La définition de cette architecture fait suite à un état de l'art en matière d'architecture reconfigurable et à une analyse du domaine applicatif. Le premier aspect de ce travail porte donc sur l'extraction de mécanismes architecturaux autorisant l'association des différentes contraintes inhérentes aux applications mobiles de prochaines générations.

Suite à la définition des fondements de l'architecture, la mise en œuvre de ces concepts constitue le second aspect de ce travail. Dans un premier temps, l'architecture système de DART est présentée. Par raffinements successifs, nous atteignons progressivement les plus bas niveaux de la hiérarchie constituant l'architecture et détaillons les primitives de calcul, de contrôle et de mémorisation sur lesquelles est bâti le circuit. Les modèles de programmation sous-jacents à cette architecture sont par ailleurs explicités.

Une architecture n'étant rien sans outils permettant de l'exploiter, la chaîne de développement associée à DART fait l'objet du troisième point de cette étude. Nous démontrons en particulier que par l'association d'un *front-end* permettant la transformation et l'optimisation de codes C, d'un compilateur reciblable et d'un outil de synthèse de haut niveau, une chaîne de développement efficace peut être construite. Les différentes optimisations logicielles mises en œuvre dans ces outils sont par ailleurs détaillées, de même que le simulateur permettant leur validation et l'évaluation de la pertinence de la solution proposée.

Le dernier aspect de ce travail concerne la validation des fondements de l'architecture et de leur mise en œuvre. Nous présentons pour cela les résultats issus de l'implémentation de fonctions clés des télécommunications de troisième génération et analysons le comportement de DART sur ces applications. Une évaluation poussée de la consommation d'énergie et des performances nous conduit par ailleurs à comparer DART à des paradigmes architecturaux plus communs que sont les processeurs programmables et les FPGAs.

Pour assurer le suivi de nos recherches, nous effectuons finalement une analyse critique de cette architecture et des outils qui lui sont associés. Nous évoquons également les nombreuses perspectives de ce travail.

Mots clés : traitement du signal, télécommunications, applications mobiles, architecture reconfigurable, FPGA, processeur programmable, arithmétique des ordinateurs, compilation, synthèse de haut niveau, hiérarchie mémoire, estimation de la consommation.