Optimisation par synthèse architecturale des méthodes de partitionnement temporel pour les circuits reconfigurables

Ting Liu

Nancy-Université
Université Henri Poincaré

U.F.R. Sciences et Techniques, Mathématiques, Informatiques, Automatique
École Doctorale Informatique Automatique Electronique Electrotechnique Mathématiques
Département de formation Doctorale Électronique Électrotechnique

Optimisation par synthèse architecturale des méthodes de partitionnement temporel pour les circuits reconfigurables

THÈSE

présentée et soutenue publiquement le 13 Mai 2008

pour l'obtention du

Doctorat de l'Université Henri Poincaré – Nancy 1

(spécialité Instrumentation et Micro-Électronique)

par

Ting LIU

Composition du jury

Rapporteurs :	M.Elbey Bourennane	Professeur, Université de Bourgogne
	M.Jean Philippe Diguet	Chargé de Recherche CNRS (HDR), Lab-STICC, Université de Bretagne Sud
Examinateurs :	M.Fabrice Monteiro	Professeur, Université de Metz
	M.Serge Weber	Professeur, Université Henri Poincaré, Nancy I
	M.Camel Tanougast	Maître de Conférences, Université Henri Poincaré, Nancy I

Laboratoire Instrumentation Electronique Nancy

Laboratoire d'Instrumentation Électronique de Nancy
Faculté des Sciences - 54506 Vandoeuvre-lès-Nancy

Remerciements

J'adresse mes sincères remerciements à Monsieur Elbey BOURENNANE, Professeur à l'Université de Bourgogne ainsi qu'à Monsieur Jean Philippe DIGUET, Chargé de recherche CNRS au Lab-STICC à l'Université de Bretagne Sud, qui m'ont fait l'honneur de juger cette thèse en qualité de rapporteurs.

J'adresse aussi mes remerciements à Monsieur Fabrice MONTEIRO, Professeur à l'Université de Metz d'avoir accepté d'examiner ce travail et de participer au jury.

Je tiens à remercier Monsieur Mustapha NADI, Professeur à l'Université Henri Poincaré de Nancy et ancien directeur du Laboratoire LIEN, de m'avoir accueilli au cours de sa direction au sein du Laboratoire LIEN.

Je tiens à remercier Monsieur Serge WEBER, Professeur à l'Université Henri Poincaré de Nancy, de m'avoir accueilli au sein de l'équipe " Architecture " qu'il dirige et d'avoir accepté d'être mon directeur de thèse.

Je souhaite remercier particulièrement Monsieur Camel TANOUGAST de m'avoir encadré durant cette thèse sur un sujet très intéressant, mais aussi pour ses qualités scientifiques, sa disponibilité et son soutien.

Toute ma sympathie et mes remerciements vont également à l'ensemble des chercheurs et techniciens du laboratoire.

Ting LIU
Février 2008

Je dédie cette thèse à mon père :
Gaulle.

Table des matières

Résumé

Résumé :

Les travaux de recherche présentés se situent dans le contexte des méthodologies d'aide à l'implémentation d'algorithmes graphe flot de données sur architectures reconfigurables dynamiquement de type RSoC (Reconfigurable System on Chip) à base de technologie FPGA.

La stratégie visée consiste à mettre en œuvre une approche de conception basée simultanément sur la reconfiguration dynamique (RD) et la synthèse architecturale (SA) en vue d'atteindre la meilleure Adéquation Algorithme Architecture (A^3) pour l'implémentation d'applications sous contraintes multiples.

La méthodologie consiste à identifier et extraire les parties d'une application décrite sous forme d'un graphe flot de données afin de les implanter soit par parties successivement reconfigurées (partitionnement temporel), soit par la synthèse architecturale (optimisation par unités fonctionnelles partagées) ou bien en combinant les deux méthodes.

Le manuscrit est organisé de la manière suivante :

– Après avoir présenté et détaillé les deux approches d'implémentation que sont la RD et la SA ainsi que leurs outils associés, nous montrons et nous expliquons la nécessite d'une exploration architecturale optimale dédiée au thème A3 par une approche reconfiguration dynamique - synthèse architecturale. Ceci mettra en évidence l'intérêt et les bénéfices attendus par la mise en œuvre de la complémentarité dans une approche combinée.

– Ensuite, nous proposons et détaillons cette nouvelle approche méthodologique combinant la reconfiguration dynamique et la synthèse architecturale. Plus précisément, nous montrons comment optimiser par une approche synthèse architecturale basée sur " la réutilisation d'unités fonctionnelles ", des méthodologies de partitionnement temporel.

La formalisation de la méthode proposée ainsi que le critère d'identification des sous-ensembles de l'application s'exécutant par partition ou par synthèse architecturale sont détaillés et explicités à partir d'un algorithme décrit sous forme de graphe flot de donnée lors de son implémentation.

Le flot de conception de notre approche ainsi que les phases d'analyse sont définis et détaillés. Elles reposent sur plusieurs critères importants tels que : le taux d'utilisation des ressources, l'évaluation des temps d'exécution et de reconfiguration pour une technologie considérée, la granularité des opérateurs ainsi que leur degré de factorisation.

L'originalité de notre approche par rapport aux travaux existants, repose sur une estimation du degré possible d'exploitation d'unités fonctionnelles partagées par synthèse architecturale sur des partitions temporelles préalablement déterminées.

Pour développer notre solution dans un but d'optimisation et de juste compromis entre les deux approches RD et SA, nous avons défini un paramètre permettant une évaluation du degré inter-partition de mise en œuvre d'unités fonctionnelles partagées.

Ce paramètre est calculé d'une part grâce à un algorithme permettant l'identification de l'ensemble des unités fonctionnelles contenues dans les partitions.

D'autre part, à partir des types d'opérations et du nombre d'opérateurs mettant en œuvre ces unités fonctionnelles de traitement entre les partitions.

Le résultat obtenu est une estimation du degré de factorisation possible des unités fonctionnelles contenues entre deux partitions successives. Ce paramètre calculé, définissant le taux de " Ressemblance Mutuel " ou inversement d'" Exclusion Mutuelle ", permet la mise en œuvre d'une analyse d'optimisation par synthèse architecturale inter-partition.

Nous obtenons alors une méthode permettant de réduire les inconvénients du partitionnement temporel par introduction d'une identification des parties s'exécutant par synthèse architecturale en vue d'une minimisation des surcoûts de la reconfiguration dynamique.

– En vue de valider la stratégie méthodologique proposée, nous présentons les résultats de l'application de notre approche sur deux applications temps réel : un algorithme de chiffrement AES et un algorithme de détection de contours d'images. Une analyse comparative en terme de résultats d'implémentation illustre l'intérêt et la capacité d'optimisation de notre méthode pour l'implémentation en reconfiguration dynamique d'applications complexes sur RSoC.

– Enfin, nous discutons et dressons le bilan de nos travaux en présentant les apports de la méthode proposée et en dégageant les perspectives de ce travail ainsi que les améliorations à apporter.

Mots clés : Synthèse Architecturale, FPGA, Partitionnement Temporel, Réutilisation de l'unité fonctionnelle, AES, Sobel, Méthodologie de la conception numérique.

Abstract :

The research work presented in the context of methodologies is to assist the implementation of data flow graph algorithms on dynamically reconfigurable RSoC (Reconfigurable System on Chip)-based FPGA architectures.

The main strategy consists in implementing a design approach based on simultaneously both the dynamic reconfiguration (DR) and synthesis architecture (SA) in order to achieve a best Adequacy Algorithm Architecture (A^3) for the implementation of applications under the multiple constraints.

The methodology consists in identifying and extracting the parts of an application which is described in form of data flow graph in order to implement either by successively partial reconfiguration (temporal partitioning), or by the architectural synthesis (shared on functional units) or by combining the two approaches.

The manuscript is organized as follows :

– After the presentation and detail of these two implementation approaches, that are, DR and AS together with their associated tools, we show and explain the necessity of an optimal architectural exploration dedicated to the theme A^3 by an approach of dynamic reconfiguration - architectural synthesis. This will highlight the value and benefits expected by the implementation of the complementarity in a combined approach.

– Secondly, we propose and describe this new approach that combines dynamic reconfiguration and architectural synthesis. More precisely, we show how to optimize through an architectural synthesis approach based on reuse of functional units, temporal partitioning methodologies. The formalization of the method proposed as a criterion for identification of the subsets of the application by running partition or architectural synthesis are detailed and clarified from an algorithm described in the form of data flow graph at its implementation. The design processes of our approach as well as the analysis phases are defined and detailed. They are based on several main criteria such as : the utilization rate of resources, the evaluation of the execution time and the reconfiguration time for associated technology, the granularity of operators and factorization level. The originality of our approach compared to the existing work, based on an estimation of possible degree of exploitation of functional units shared by architectural synthesis on partitions previously determined time. To develop our solution with a view of optimizing and suitable compromise between the two approaches RD and SA, we propose a parameter in order to evaluate the degree of the inter-partition implementation based on functional units shared. This parameter is calculated through an algorithm which allows identifying all functional units contained in the scores application. On the other hand, from the types of operations and the number of operators implementing, how these functional units are treated between partitions. The obtained result is a degree estimation of the functional units which are contained

between two successive partitions. The calculation of this parameter defines a rate of “Mutual Similarity” or vice versa “Mutual Exclusion”, allows implementing an optimization analysis by synthesis architectural with inter-partition. We propose this method to reduce the disadvantages of temporal partitioning through the introduction of an identification of the parties which run for architectural synthesis with a view of minimizing of incremental costs during the dynamic reconfiguration.

– In order to validate the proposed methodological strategy, we present the results of the implementation of our approach on two real-time applications : an encryption algorithm AES and the algorithm for detecting contours of images. A comparative analysis with the respecting of the implementation results illustrates the interest and the optimization ability of our method, which is also for dynamic reconfiguration implementation of the complex applications on RSoC.

– Finally, we discuss and assess our work through presenting the contributions of the proposed methodology and identifying prospects of this work as well as opportunities for improvement.

Table des figures

*

Glossaire

A^3	Adéquation Algorithme Architecture
AES	Advanced Encryption Standard
AG	Algorithme Génétique
ALAP	As Later As Possible
ALU	Arithmetic logic unit
APL	A Programming Language
ASAP	As Soon As Possible
ASIC	Application Specific Integrated Circuit
BRAM	Block RAM
CD	Cellule dédiée
CL	Cellule logique
CLB	Configurable Logic Block
CSoC	Configurable System on Chip
DAGARD	Découpage Automatique de GFD sur Architecture Reconfigurable Dynamiquement
DC	Design Compiler
DSP	Digital Signal Processor
DR	Dynamic Reconfiguration
ECB	Electronic code book
FIFO	First In First Out
FPE	Processeurs élémentaires
FPGA	Field Programmable Gate Array
GAUT	Générateur Automatique d'Unité de Traitement
GC	Graphe de Contrôle
GFC	Graphe flot de contrôle
GFD	Graphe flot de Données
GFDC	Graphe flot de données/contrôle
IP	Intellectual Property

IPS	Inter Partition Synthèse
LASTI	Laboratoire d'Analyse des Systèmes de Traitement de l'Information
LESTER	Laboratoire d'Electronique des Systèmes TEmps Réel
LIEN	Laboratoire d'Instrumentation Électronique de Nancy
LIFO	Last In First Out
LUT	Look up Table
NoC	Net on Chip
PSLIC	Field Programmable System Level Integrated Circuit
PT	Partition Temporel
RAM	Random access memory
RD	Reconfiguration Dynamique
RISC	Reduced instruction set computer
ROM	Read-only memory
RSoC	Reconfigurable System on Chip
RTL	Register transfer level
SA	Synthèse Architecturale
SFG	Signal Flow Graph ou graphe flot de signaux
SoC	System on Chip
TDSI	Traitement du Signal et de l'Image
TEM	Taux d'exlusion mutuelle
TP	Temporal Partition
TRM	Taux de Ressemblance Mutuel
UCOM	Unité de Communication
UCR	Unité de Calcul Reconfigurable
UF	Unité Fonctionnelle
UM	Unité de mémorisation
UT	L'unité de traitement
VHDL	VHSIC Hardware Description Language

Chapitre 1

Introduction générale

1.1 Historique et contexte général : “Le Calcul Reconfigurable”

En 1960, Geral Estrin, informaticien de l'Université de Californie à Los Angeles, propose une idée concernant une “ architecture fixe et variable ” [ET63]. Cette dernière se composerait d'un processeur standard associé à un tableau de matériel “ reconfigurable ” commandé par le processeur. Ce matériel reconfigurable peut être configuré pour exécuter rapidement une tâche spécifique. Une fois cette tâche effectuée, le matériel peut être figé de nouveau pour exécuter une nouvelle tâche. Cette nouvelle vision structurelle matérielle aboutit sur un ordinateur hybride combinant la flexibilité du logiciel avec les performances en terme de vitesse de calcul des technologies matérielles rapides [Est00, Est02].

A cette époque, bien que cette approche fut validée, les technologies n'ont pas permis de mettre en œuvre l'idée proposée par Géral Estrin. Ce ne fut qu'en 1985, que cette approche a connu une renaissance grâce au développement du premier circuit de matrice logique programmable (circuit FPGA : Field Programmable Gate Array) par la société Xilinx. Depuis, ce type de circuit n'a pas cessé d'évoluer en terme de complexité, d'intégration et de performances pour devenir aujourd'hui des circuits logiques totalement ou partiellement reconfigurables [Tho04]. Plus précisément, ces circuits peuvent être reprogrammés à volonté entièrement ou bien par parties, tandis que les autres parties du circuit continuent de fonctionner.

Pour mettre en œuvre l'idée de Géral Estrin, les premiers FPGA devaient être couplés à un processeur. Cette association entraîne comme principal inconvénient l'augmentation de la complexité de programmation du système, étant données les structures matérielles différentes des deux composants (lecture d'instructions en mémoire pour le processeur, écriture de données de configuration pour le FPGA). Pour atténuer ce principal inconvénient, la tendance actuelle est d'associer directement sur la même puce ces deux structures en combinant les blocs de logique FPGAs traditionnels avec les microprocesseurs tout en incorporant des périphériques d'intercom-

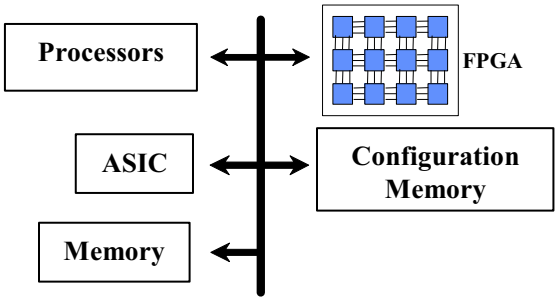

FIG. 1.1 – Illustration de la structure architecturale d'un RSoC.

munications (contrôleur, NoC, etc.) afin de former un système de calcul " complet " fonctionnel. On parle alors de Systèmes sur Puce (re)configurable (CSoC ou RSoC : (Re)Configurable System On Chip) (Figure 1.1).

L' association directe d'une structure micro-programmée et d'une matrice logique configurable peut être effectuée de deux manières :

- Soit par une réalisation technologique intégrant physiquement ces deux composants. Comme exemples non exhaustifs de telles technologies hybrides, on trouve les dispositifs :

  - Virtex-II PRO et Virtex-4 de Xilinx, qui incluent un ou plusieurs processeurs PowerPC incorporés dans le tissu de la logique du FPGA [Xil07b, Xil07a]. La figure 1.2 présente la vue d'ensemble du Virtex II Pro. Le cœur de processeur IBM PowerPC 405 est un cœur de processeur RISC 32 Bits, PowerPC 405D5 en technologie 0,13 μm dérivé du cœur de processeur IBM PowerPC 405D4. Il peut atteindre 300 MHz tout en maintenant une faible consommation. Le Virtex II Pro X peut accueillir jusqu'à quatre de ces coeurs de processeur (XC2VP125).
  - Le circuit Atmel 94K FPSLIC (Field Programmable System Level Integrated Circuit). qui emploie un processeur AVR en combinaision avec l'architecture programmable de la logique d'Atmel (Figure 1.3) [Atm]. Ce circuit possède sur la même puce un microcontrôleur RISC avec une architecture Harvard (mémoire d'instruction et mémoire de données séparées), développant une puissance de 30 MIPS à 12 MHz (nommé AVR), une mémoire de 36 Ko et un FPGA de la famille AT40K [Atm98].
- Soit en important directement dans la matrice FPGA un ou plusieurs noyaux de processeurs. On parle alors d'IP (Intellect Property) dédié FPGA. Cette approche alternative est la mise en application de noyaux de processeurs dans la logique de FPGA. Parmi eux, on trouve les processeurs MicroBlaze et PicoBlaze de chez Xilinx [Xil]. En effet, la haute densité du FPGA Virtex II (jusqu'à 8 millions de portes logiques équivalentes et 1108

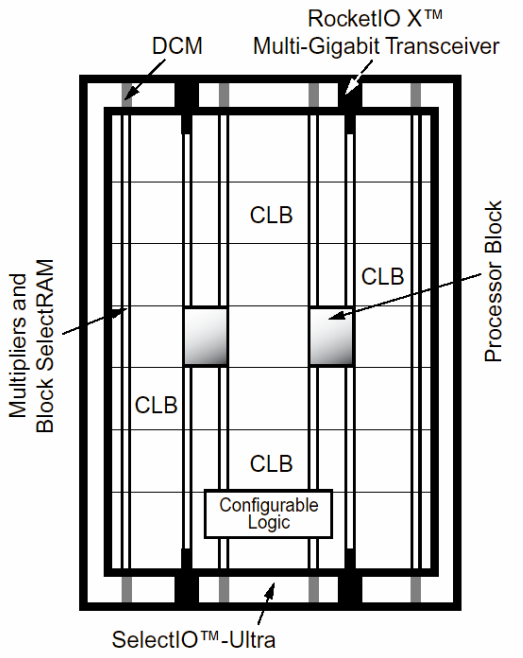

FIG. 1.2 – Vue globale de l’architecture Xilinx VirtexII-Pro et Virtex 4.

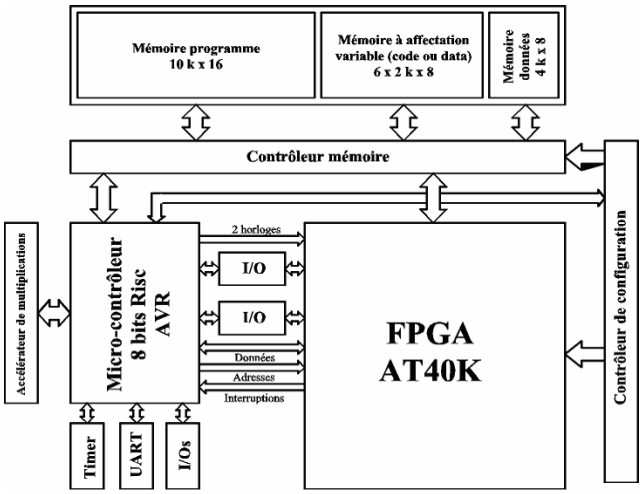

FIG. 1.3 – L’organisation simplifiée ATMEL FPSLIC 94K.

ports d'entrée/sortie) permet l'implantation du processeur MicroBlaze 32-bit@125 MHz.. Les processeurs Nios et de Nios II de chez Altera [Alt], les processeurs LatticeMico32 et LatticeMico8 de chez Lattice [Sem]. Cette dernière approche permet une plus grande flexibilité de conception tout en ayant une meilleure gestion des ressources au cours du temps. Cependant, elle s'obtient au détriment des performances de fonctionnement étant donné que les performances d'implémentation sont tributaires des ressources de placement et de routage comme n'importe quelle fonction de traitement importée.

Ces deux approches peuvent être également combinées formant ainsi des structures complexes faisant intervenir des blocs sur puce et des IPs fonctionnelles sur la zone de la puce reprogrammable.

1.2 Motivation

Aujourd'hui, si l'intérêt de ces architectures reconfigurables sur puce (RSoC *: Reconfigurable System on Chip*) est reconnu (meilleure utilisation des ressources, adaptation par rapport à l'environnement, flexibilité et réutilisation) et a permis de répondre aux critères d'intégration et d'embarquabilité, les outils qui leur sont associés restent limités et inadaptés [TCW00]. En effet, la principale difficulté réside dans l'absence de méthodes et d'outils efficaces pour les programmer à un niveau système, dans la mesure où ils découlent de la chaîne de développement classique de type ASIC. Or, il est évident qu'aujourd'hui la maîtrise des architectures reconfigurables est stratégique dans la perspective des applications à venir. La puissance de calcul doit être associée à une grande flexibilité dans le but, par exemple, de faire coexister différents standards dans un même système. Par conséquent, il est nécessaire de mettre en œuvre des outils basés sur de nouvelles approches de conception prenant en compte leurs spécificités pour obtenir des développements optimisés. Il s'agit alors de définir des méthodes d'implémentations basées sur l'Adéquation Algorithme - Architecture.

Dès les débuts de la reconfiguration dynamique, le L.I.E.N [1] s'est impliqué dans cette technologie prometteuse [Gue97]. Les premiers travaux ont montré d'une part que nous sommes tributaires de la technologie disponible et d'autre part qu'il n'existait peu ou pas d'outils dédiés à l'exploitation de la reconfiguration dynamique. Une méthodologie mettant en œuvre cette dernière faisait donc défaut, malgré plusieurs études traitant des problèmes de gestion de la reconfiguration dynamique. Or, cette problématique apparaissait comme une nécessité dans le but de gérer et d'automatiser le calcul reconfigurable. C'est ce qui nous a poussé à réfléchir à l'élaboration de méthodes d'implantation en reconfiguration dynamique. Pour cela, plusieurs interrogations devaient être considérées. Combien de partitions ? Comment choisir les partitions pour réaliser la décomposition en reconfiguration dynamique des algorithmes à implanter ? Il

[1] Laboratoire d'Instrumentation Électronique de Nancy, Nancy, http ://www.lien.uhp-nancy.fr

s'agissait donc de déterminer une méthodologie de partitionnement en reconfiguration dynamique d'une application. Ces questions ont été le sujet de thèse traité par Mr. Camel Tanougast [Tan01]. Cette thèse a permis de développer une première méthodologie de partitionnement restreinte à certaines conditions particulières (application sous contrainte de temps, permettant une minimisation de la surface logique nécessaire à l'implantation d'algorithme en reconfiguration dynamique, applicable à une application décrite sous forme de graphe flot de données). Néanmoins, après une quantification de l'apport de la reconfiguration dynamique sur la réduction de surface que l'on peut atteindre avec une application et une technologie donnée, le partitionnement induit un certain nombre de modifications dans les caractéristiques de l'algorithme à implanter. Ces points et l'automatisation de la méthodologie développée ont été le sujet de thèse traité par Mr. Philippe Brunet [Bru04]. Cette thèse a permis d'ajouter une prise en compte plus globale des contraintes apportées par la reconfiguration dynamique et étendre les possibilités de sorte à intégrer des graphes flot de données non-réguliers en intégrant et en automatisant une étude exploratoire de l'ensemble des solutions qu'apporte une solution par reconfiguration dynamique. L'ensemble des travaux menés au laboratoire LIEN a permis le développement de l'outil de partitionnement temporel intitulé D.A.G.A.R.D (*Découpage Automatique de GFD sur Architecture Reconfigurable Dynamiquement*) pour l'implémentation d'algorithmes sur plateformes reconfigurables dynamiquement.

L'objectif de l'Adéquation Algorithme - Architecture (A^3) est de trouver la meilleure " correspondance " entre un algorithme et une architecture pour réaliser son implémentation optimisée, en satisfaisant diverses contraintes (temps réel, surface, consommation, etc.). Actuellement, deux approches ayant pour objectif de trouver cette meilleure adéquation entre un algorithme et une architecture, permettent une optimisation des ressources d'implémentation sur un SoC reconfigurable. Il s'agit du partitionnement temporel par reconfiguration dynamique (RD) et de la synthèse architecturale (SA). La Figure 1.4 est un exemple illustrant les principes de ces deux approches méthodologiques d'implémentation sur technologies reconfigurables au cour d'une conception FPGA.

La reconfiguration dynamique consiste en l'exécution fractionnée et successive d'un algorithme par reconfiguration successive de la partie matérielle configurable. Cette approche permet de minimiser la surface logique nécessaire pour l'implémentation d'un algorithme tout en augmentant l'efficacité silicium par traitement [Ta03, Ba03]. Ceci est très important pour le développement d'applications embarquées car elle permet d'éviter un surdimensionnement des ressources nécessaires pour une application et optimise donc son coût global. La mise en œuvre de la reconfiguration dynamique consiste alors à identifier les différentes parties à exécuter séquentiellement (partitions temporelles) selon un ordonnancement et des dépendances de données. Des travaux récents montrent également son intérêt pour la réduction de consommation d'énergie au cours de l'exécution d'une application dans la mesure où aux instants précis de l'exécution

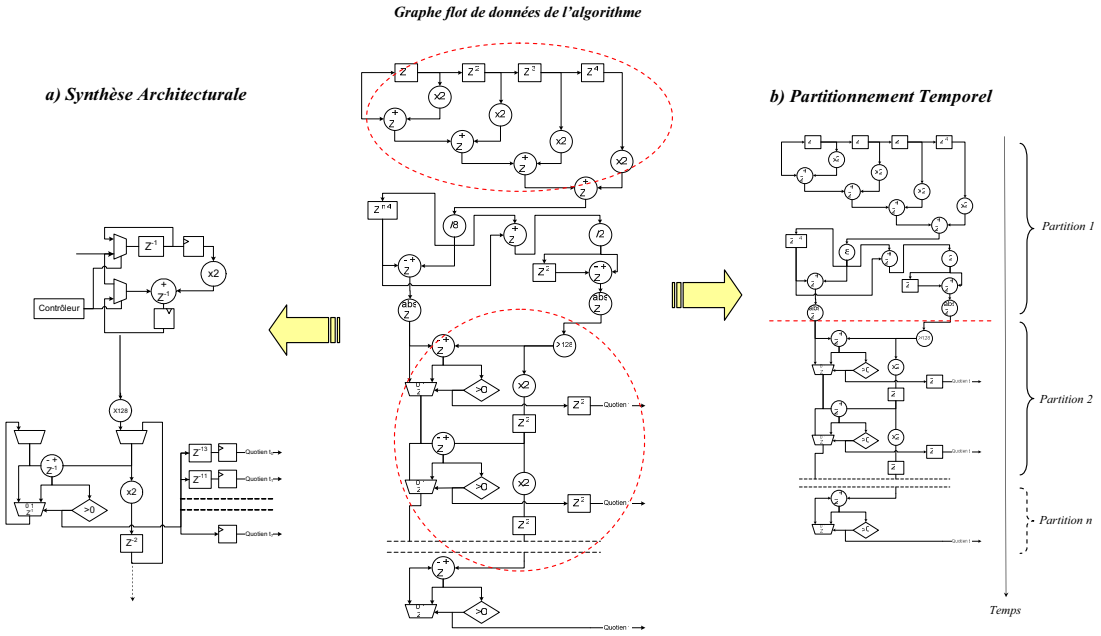

FIG. 1.4 – Illustration du partitionnement en reconfiguration dynamique et de la synthèse architecturale.

seules les ressources minimales nécessaires à l'application sont exploitées [KPB07].

La synthèse architecturale est basée sur la réutilisation d'opérateurs à différents instants pour exécuter l'ensemble de l'algorithme [Ta00, SJ93]. Cette réutilisation est mise en œuvre par l'addition supplémentaire de ressources de contrôle et la mutualisation des opérations dans l'unité de traitement. La synthèse architecturale se compose de deux tâches principales que sont l'ordonnancement et l'allocation. L'ordonnancement analyse les dépendances entre les opérations pour extraire leur parallélisme et affecter les opérations aux transitions. L'allocation fait correspondre les éléments d'une description comportementale à des ressources matérielles.

Selon les applications et les contraintes d'implémentations visées, l'une ou l'autre de ces approches paraît la mieux adaptée en permettant d'atteindre une meilleur A^3 vis à vis de l'autre. C'est dans le but de prendre en compte les avantages simultanés des différentes approches de conception que sont la synthèse architecturale et la reconfiguration dynamique (le partitionnement temporel), en y apportant des modifications avant même l'implantation dans le but d'aboutir à un outil d'automatisation, que ce travail de thèse a été réalisé.

1.3 Problématique de l'étude

Il existe plusieurs travaux qui se focalisent sur l'optimisation de l'implémentation d'une application. Ces méthodes d'exploration de conception SoC reposent soit sur le partitionnement temporel réalisé à l'aide de la reconfiguration dynamique, soit sur la synthèse architecturale basée aussi bien sur la synthèse comportementale que sur la synthèse au niveau transfert registre. Ces approches sont établies en fonction d'éventuelles contraintes et en prenant en compte la cible technologique reconfigurable [Wil97, KV98, Car03, ZN00a, Vem01, HZC02, La00]. Cependant, dans ce contexte, elles ciblent un ensemble réduit d'architectures pour les mêmes performances. Ces travaux vont se poursuivre dans la mesure où les applications sont de plus en plus complexes et nécessitent une consommation en terme de ressources de plus en plus importante. Dans ce cadre, la recherche du meilleur compromis entre les performances et les ressources matérielles exploitées nécessite l'élaboration de méthodologies et d'outils associés permettant d'identifier correctement ce compromis en vue de converger rapidement vers une solution auto-adaptable efficace entre la reconfiguration dynamique et la synthèse architecturale pour une application développée sur un RSoC. Dans ce cadre, la problématique posée pour ce travail de thèse est la suivante :

"L'exploitation potentielle de la reconfiguration et de la synthèse architecturale permet-elle de définir une nouvelle méthodologie d'implémentation efficace sur la partie FPGA d'un RSoC ? "

Pour tenter de répondre à cette problématique, nous apportons une réponse sur l'insertion de la synthèse architecturale dans une approche d'implémentation sur des structures reconfigurables dynamiquement basée sur le partitionnement temporel.

1.4 Contributions

On entend par implémentation efficace, l'adéquation algorithme - architecture permettant la meilleure utilisation des ressources tout en maintenant un minimum de flexibilité et adaptabilité par rapport à l'environnement pour l'implantation d'une application donnée. Ce manuscrit présente les travaux pour le développement d'une méthodologie d'aide à la conception sur la partie FPGA d'un RSoC basée simultanément sur la reconfiguration dynamique et la synthèse architecturale en vue d'atteindre la meilleure adéquation architecture algorithme pour des applications données tout en considérant des contraintes de traitement et ou architecturales. Plus précisément, la méthodologie permet d'identifier et d'extraire les parties d'une application qui s'exécutent par partitionnement temporel reconfiguré et/ou par synthèse architecturale. L'originalité de notre approche est basée sur une estimation du degré d'exploitation de la synthèse architecturale sur des partitions temporelles préalablement déterminées. Cette analyse repose

sur plusieurs autres critères importants tels que : le taux d'utilisation des ressources, le temps d'exécution, l'évaluation de la reconfiguration et la latence ou propagation, la hiérarchie de l'architecture et la granularité des opérateurs.

1.5 Plan de mémoire

Cette thèse est structurée de la manière suivante. Le **Chapitre 2** constitue un bref état de l'art de la reconfiguration dynamique et de la synthèse architecturale. Ce chapitre débutera par une **définition de la reconfiguration dynamique et précisera la notion de synthèse architecturale adoptée**. Nous passerons ensuite en revue les approches existantes ainsi que leurs outils associés. L'approche méthodologique adoptée et présentée dans le chapitre suivant étant en continuité directe avec les travaux précédents développés au laboratoire LIEN, nous détaillerons l'approche adoptée pour le partitionnement temporel de graphe flot de données régulier répondant à une contrainte de temps d'exécution de l'application [Tan01, Bru04]. Dans ce chapitre, nous expliquerons également la nécessite d'une exploiration architecturale optimale de ces deux approches dédiées à l'Adéquation Algorithme-Architecture. Nous mettrons en évidence la complémentarité de ces approches et l'intérêt de leur association en terme d'optimisation.

Le **Chapitre 3** propose et détaillera une approche méthodologique combinant synthèse architecturale et partitionnement temporel. Plus précisément, nous présenterons une méthodologie optimisée de partitionnement temporel à base de synthèse architecturale basée sur des unités fonctionnelles partagées. Le chapitre présentera la modélisation que nous faisons de l'application à implanter ainsi que le critère d'identification des sous-ensembles d'une application s'exécutant par partitions et pouvant être optimisés par synthèse architecturale inter-partition lors d'une implémentation. Le flot conception de notre approche ainsi que les phases d'analyse seront définies et détaillées.

En vue de valider l'approche proposée, Le **chapitre 4** présentera les résultats de l'application de notre approche pour des applications temps réels (Détection de Contours d'images, Algorithmes de cryptage AES). Une analyse comparative en terme de résultats d'implémentation illustrera l'intérêt et la capacité d'optimisation de notre méthode pour l'implémentation d'applications complexes sur RSoC.

Enfin, Le **chapitre 5** tiendra lieu de conclusion finale de ce manuscrit. Nous discuterons et dresserons le bilan de nos études en présentant les apports de la méthode proposée et en dégageant les perspectives de ce travail ainsi que les améliorations à apporter.

Chapitre 2

Approches Méthodologiques d'Implémentation : État de l'Art et Analyse Comparative

2.1 Introduction

Ce chapitre est consacré à la problématique générale de l'exploration architecturale dédiée à la conception à base de technologie reconfigurable tel que les **RSoC**. Nous passerons en revue les différentes approches d'implémentation (synthèse architecturale et reconfiguration dynamique par partitionnement temporel) ainsi que leurs outils associés (outils GAUT et DAGARD) tout en détaillant les différentes étapes de leurs flots de conception dans le cas d'une approche Adéquation Algorithme Architecture. Ensuite, une analyse comparative des approches présentées sera alors discutée. Enfin, une présentation de méthodologies mixtes basées sur une approche simultanée Synthèse architecturale - Reconfiguration dynamique clôtura ce chapitre.

2.2 Approches méthodologiques d'implémentation

2.2.1 Synthèse Architecturale

2.2.1.1 Principe général

L'objectif de la synthèse architecturale est de transformer une description comportementale en description structurelle. Il s'agit de générer de manière automatique une description structurelle au niveau transfert de registre (RTL) à partir d'une description comportementale au niveau algorithme. La description comportementale décrit la fonctionnalité qui doit être réalisée par le système synthétisé. La description du niveau de transfert de registre (RTL) caractérise souvent

la définition du système en termes de registres, multiplexeurs, opérateurs, etc. Le résultat est typiquement une description de chemin de données et d'opérateurs au niveau RTL associé à un contrôleur tout en satisfaisant un certain nombre de contraintes [Kuc98, DGL92a, Gov00, Sil94, Ell00, Aic94, Sug00, Din96, Lin97a, TF69, Zim79, APM86, SN91].

De façon concrète, la synthèse architecturale également appelée la synthèse comportementale correspond à un ensemble de tâches de raffinement ou de transformation pour implémenter une spécification comportementale avec des contraintes d'implémentation tout en optimisant les fonctions décrivant l'application. En général, les entrées d'une synthèse architecturale correspondent :

- à une spécification comportementale d'un système à mettre en œuvre,
- aux contraintes de conception (souvent définies en termes d'ordre de coût, de temps d'exécution ou de performance, de délai, de limitation de consommation de puissance ou de ressources en terme de surface, de mémoire, etc.) qui dépendent du type de l'application à synthétiser, d'une fonction optimisation et d'une bibliothèque de modules (représentant des composants disponibles sous forme de niveau de transfert de registre) permettant de mettre en œuvre les opérations de la description selon une solution répondant aux objectifs respectant les contraintes spécifiées.

Une sortie du processus de synthèse architecturale est une description structurelle d'implémentation au niveau du transfert de registre (Netlist) associée à un contrôleur [Kis96, Hei96]. Le résultat de la synthèse architecturale est alors composé d'un chemin de données représentant la partie opérative du système et constituée d'unités fonctionnelles (additionneurs, multiplieurs, ALUs et unités logiques), d'unités de mémorisation (RAMs, ROMs, Registres ou Flip-flop, etc.) et d'unités de communication interconnectées (bus et multiplexeurs, etc.). Ce chemin de données décrit les dépendances des éléments constituant la partie opérative. Le contrôleur, correspondant à la partie commande du système, est une machine d'états finis qui déterminent à chaque cycle horloge système, les opérations de la partie opérative qui doivent être exécutées à partir des ses unités fonctionnelles. Physiquement, dans la matrice reconfigurable, tous les éléments sont représentés par un ensemble de composants réels mise en œuvre par les ressources de la matrice. La figure 2.1. illustre le résultat final d'une conception par synthèse architecturale.

Ce processus de synthèse est l'application de techniques de conception. Au sein d'un flot de synthèse correspondant à une séquence d'étapes, le processus crée le système en faisant correspondre les variables et les opérations d'une description comportementale du système à des unités fonctionnelles, de mémorisation et de communication décrit au niveau RTL.

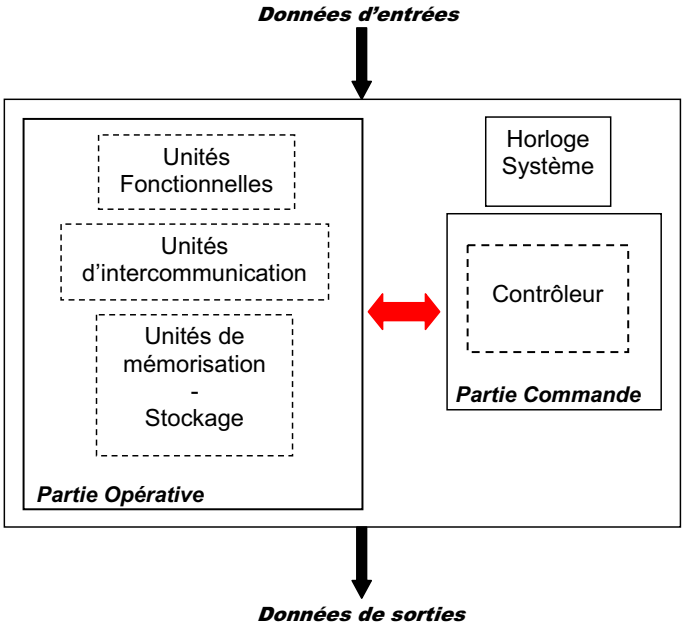

FIG. 2.1 – Illustration d'un système conçu par synthèse architecturale.

2.2.1.2 Flot de conception par synthèse architecturale

De façon concrète, la synthèse architecturale consiste à convertir une spécification abstraite en une spécification structurelle. Traditionnellement, cette dernière est divisée en une synthèse de chemin de données mis en œuvre par des unités fonctionnelles et/ou de stockage et d'un chemin de contrôle mis en œuvre par une unité de commande coordonnant et organisant la gestion des données entre les éléments constituant le chemin de données. Pour aboutir à une description structurelle optimale en fonction de contraintes, le processus typique de la synthèse architecturale est composé principalement de trois étapes pouvant être interdépendantes : l'***Analyse-Traduction***, l'***Ordonnancement*** et l'***Allocation-Assignation-Optimisation***. La figure 2.2 illustre le flot de conception traditionnel du processus de synthèse architecturale.

2.2.1.2.1 Analyse - Traduction en graphe flot de données/contrôle

La sous-étape ***Analyse-Traduction*** consiste, à partir d'une spécification comportementale d'un système, à concevoir en utilisant un langage procédural (tel que le langage C) ou fonctionnel (APL [Ive62], Erlang [Arm03, Arm07], Haskell [CF02], Lisp [Vei98], ML [RMH90], etc.) ou bien un langage de description matériel (e.g. VHDL [MT93], Verilog [CDR96], AHDL, SystemC, etc.), une succession d'analyses et de transformation de la description abstraite dans le but d'aboutir à une description sous forme de graphe de flot de données (GFD) et de graphe de flot de contrôle (GFC) de la spécification initiale. La structure des graphes représente les entrées, les opérations

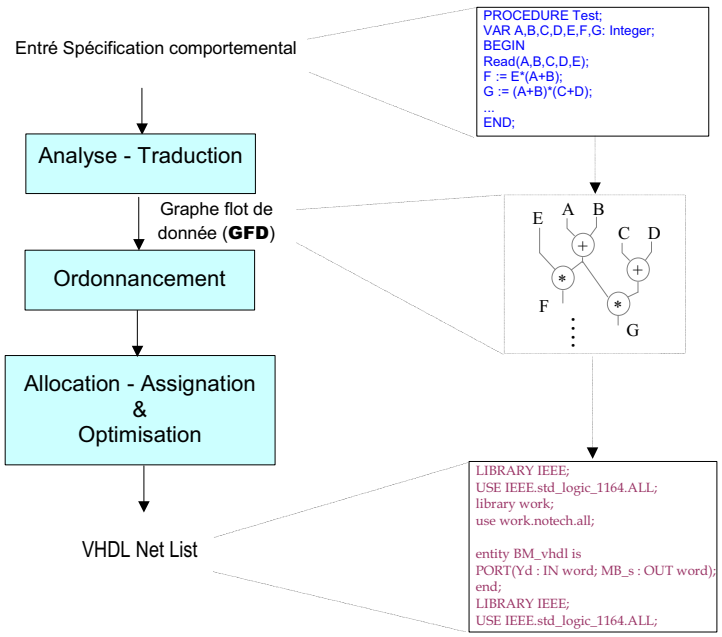

FIG. 2.2 – Flot de conception du processus de synthèse architecturale.

et les transferts de données de la spécification comportementale du système à travers ses noeuds, ses bords et sommets et ses arcs [Ste97].

Les principales sous-étapes de l'***Analyse-Traduction***, qui reposent également sur les spécifications de condition en termes de temps d'exécution, de consommation d'énergie, de surface occupée, etc., sont les suivantes : l'***élimination*** de construction abstraite et de code mort, la ***détection de Sub-expression commun***, l'***extraction du parallélisme*** permettant une réorganisation des tâches et la ***transformation*** du programme telle que le déroulement de boucle et la ***traduction***.

– L'***élimination*** de construction abstraite consiste à supprimer certaines structures de description. En effet, une description abstraite intègre typiquement des structures de haut niveaux de langage facilitant la programmation mais qui ne peuvent être interprétées ou traduites sous forme de structure d'implémentation au niveau transfert de registre pour une technologie matérielle tel que le FPGA. L'élimination du code mort consiste à supprimer dans la description les fonctions et les variables qui ne sont jamais utilisées et dont les opérations qui les produisent peuvent être supprimées [IRI01].

– la ***détection de Sub-expression commune*** consiste en une optimisation simple par suppression de sous-expressions communes de la description comportementale [Ste97, Coc70, BP97]. L'objectif est de rechercher et d'analyser des expressions identiques pouvant être

remplacées par une variable simple de calcul afin d'aboutir à une optimisation en terme de vitesse ou de ressources dans la description finale structurelle du système. Si l'on considère, par exemple, les expressions suivantes contenant une sous-expression commune :

$$a = b * c + g;$$
$$d = b * c * d;$$

Il est intéressant de traduire le code de la manière suivante à travers une variable intermédiaire *tmp* :

$$tmp = b * c;$$
$$a = tmp + g;$$
$$d = tmp * d;$$

Cette simple transformation de code introduisant un calcul intermédiaire permet d'aboutir, dans la description structurelle, soit à une exécution plus rapide de la description initiale, soit à une optimisation en terme de ressources si on évalue le coût de mise en œuvre par un élément de mémorisation de la temporisation symbolisé par la variable *tmp* inférieur au coût d'un multiplieur.

- L'***Extraction de parallélisme*** consiste à identifier les opérations pouvant s'exécuter en parallèle ou simultanément tout en veillant à leur synchronisation à partir d'une description séquentielle d'un système [PBAS97, BP06]. Ce parallélisme est important pour une mise en œuvre optimisée de la spécification sur des systèmes parallèles ou distribués. Plusieurs approches ont été proposées pour extraire le parallélisme depuis un programme séquentiel [AK01, Bel04, Fea92a, Fea92b, Fea94, WL94, WL97]. D'une manière générale, cette extraction s'effectue souvent pour un compromis entre la performance, la flexibilité et les contraintes de temps et de ressources.

- La ***Transformation*** consiste, tout en conservant la sémantique initiale de description, à apporter des modifications permettant une interprétation physique par des unités fonctionnelles dans la description structurelle. Parmi ces transformations, on trouve le déroulement de boucle [SGN02b, SGN03]. Ce dernier est un procédé qui déroule les itérations d'une boucle de traitement dans le but de faciliter une optimisation globale du code [Muc97, SGN02a].

- La ***Traduction*** est la transformation de la description abstraite modifiée et optimisée par les sous-étapes précédantes sous la forme de graphes flot de données et de contrôle.

La figure 2.3. suivante illustre un exemple simple de l'étape d'***Analyse-Traduction*** du flot de conception par synthèse architecturale.

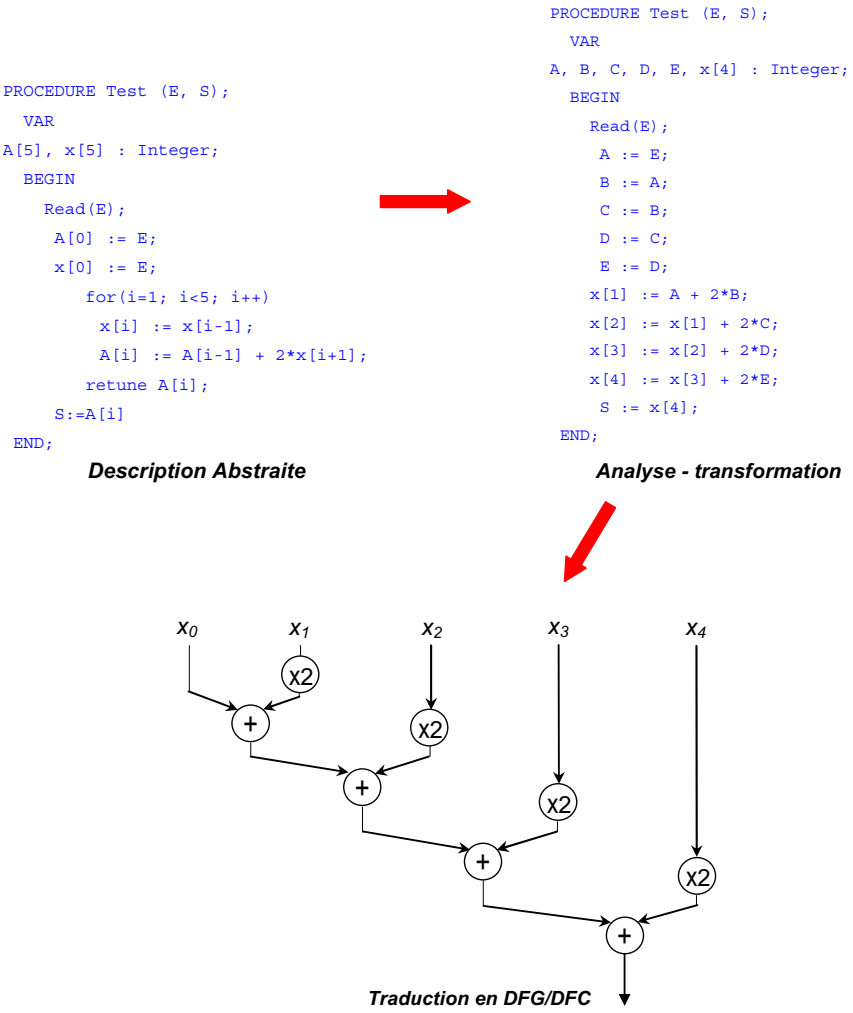

FIG. 2.3 – Illustration de l'étape ***Analyse-Traduction***.

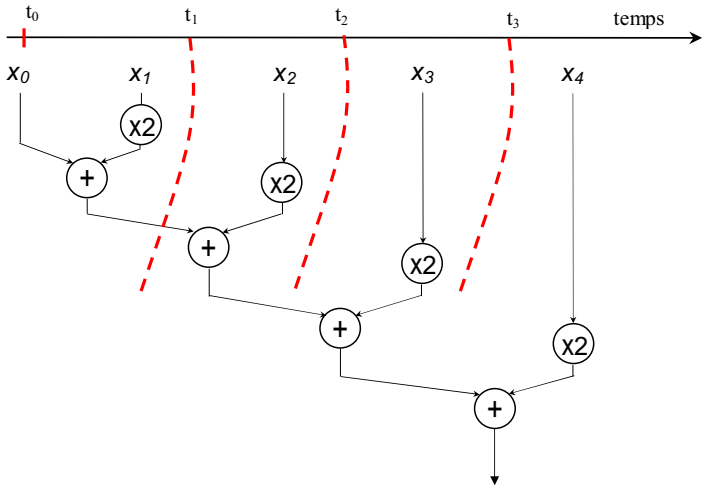

FIG. 2.4 – Illustration de l'étape d'***Ordonnancement***.

2.2.1.2.2 Ordonnancement

L'étape d'Ordonnancement s'effectue sur le graphe interne de données et de contrôle issue de l'étape précédente [PK87, MMC90, CW91, LIN97b, Fis90, KASO90]. Son rôle est de définir les instants d'exécution de chaque opération du graphe en respectant les dépendances de données entre les noeuds du graphe, et en s'assurant la possibilité d'allouer les ressources nécessaires pour exécuter toutes les opérations ordonnées à chaque instant (figure 2.4). Il ordonne ainsi les instants de départ de chaque opération tout en intégrant les étapes de contrôle nécessaires. En pratique, ces cycles de contrôle correspondront aux états d'une machine d'état effectuant la gestion et l'organisation des opérations pouvant s'exécutant en parallèle.

Dans le domaine de la synthèse architecturale, les algorithmes d'ordonnancement peuvent être classés en deux catégories : les algorithmes d'ordonnancement de flot de données et les algorithmes d'ordonnancement de flot de contrôle [Nar98, Sug00, Gov95]. Ces algorithmes prennent compte des boucles conditionnelles, des branchements indépendants, des dépendances de données et des contraintes sur les ressources matérielles, sur les temps d'exécution, sur la consommation, etc. Cependant, d'une manière générale, l'***Ordonnancement*** s'effectue en compromis entre une vitesse maximale d'exécution et une surface minimale de silicium. Un ordonnancement sous contraintes de ressources cherche à minimiser la durée d'exécution tout en fixant la quantité de ressources matérielles. Un ordonnancement sous contrainte de temps cherche à minimiser les ressources tout en bornant la durée d'exécution. Lorsque l'***Ordonnancement*** des tâches est réalisé, l'étape d'***Allocation-Assignation*** optimisée des opérations aux ressources disponibles (telles que l'additionneur, les registres, les multiplicateurs, etc.) peut être réalisée.

2.2.1.2.3 Allocation - Assignation - Optimisation

L'allocation, l'assignation et optimisation des ressources selon des contraintes sont des étapes importantes de la synthèse architecturale, une fois que l'ordonnancement affecte les tâches et détermine le nombre d'opérateurs au cours du temps. L'étape d'ordonnancement ayant réparti les opérations de la description comportementale en étapes de contrôle et de calcul, la description finale du circuit au niveau RTL est composée d'une partie contrôle et d'une partie opérative. Le contrôleur correspond à une machine d'états finis ou à une table de transition qui définit le séquencement des opérations. La structure de la partie opérative ou du chemin de données est définie par les étapes d'allocation, d'assignation et d'optimisation.

- L'allocation et l'assignation des ressources fait correspondre des opérateurs physiques aux opérations, des registres aux variables et des connexions aux transferts de données de la description initiale. L'étape d'allocation détermine le nombre et le type des ressources nécessaires pour l'exécution de la description comportementale [Ell00, Ber97, Sug00, DGL92b, Kis96]. Cette étape fixe le nombre et les types d'unités fonctionnelles (additionneurs, multiplieurs, ALUs, etc.), les unités de stockage (registres, bancs de registres, mémoires) et les unités d'interconnexion (multiplexeurs, bus, etc.) à partir d'une bibliothèque de composants RTL. Cette dernière peut contenir plusieurs types d'unités fonctionnelles, chacune avec des caractéristiques différentes (fonctionnalité, taille, possibilité de pipeline) permettant de fournir plusieurs solutions pouvant assurer l'exécution de la description comportementale tout en satisfaisant les contraintes de conception existantes. L'étape d'assignation des ressources affecte ou assigne respectivement les opérations, variables, et les transferts de données dans le graphe de flot, aux unités fonctionnelles, de stockage, et d'interconnexion. D'une manière générale l'Allocation- Assignation consiste en 3 étapes interdépendantes :

  - L'allocation/assignation d'unités fonctionnelles : à partir de la bibliothèque prédéfinie d'UFs disponibles avec leurs caractéristiques, cette tâche détermine quels sont les opérateurs ou modules fonctionnels nécessaires et fait correspondre les opérations à l'ensemble des unités fonctionnelles (UFs) sélectionnées pour l'exécution de l'algorithme initial [Kis96]. Les unités fonctionnelles peuvent exécuter aussi bien des opérations standard (opérations arithmétiques ou booléennes) que des opérations complexes définies par le concepteur par l'intermédiaire de procédures et de fonctions.

  - L'allocation/assignation des unités de stockage : cette tâche fait correspondre les valeurs de la description comportementale (constantes, variables, structures de données de type tableaux, etc.) à des éléments de stockage ou de mémoire (ROMs, RAMs, registres, bancs de registres, etc.).

- L'allocation/assignation des unités d'interconnexion : cette tâche fait correspondre les transferts de données à un ensemble d'unités d'interconnexion. Elle détermine les ressources (multiplexeurs, bus, etc.) nécessaires pour la communication entre les unités (unités de stockage et unités fonctionnelles) du chemin de données. Le but ici est également de définir le réseau d'interconnexion de façon à minimiser la surface du circuit, et ceci en utilisant un nombre minimal de connexions (multiplexeurs et bus). Ce nombre influe non seulement sur la taille du chemin de données mais également sur la taille du contrôleur.

- L'étape d'***Optimisation*** consiste principalement, tout en respectant les contraintes de conception et en étant interdépendante de l'Allocation et de l'Assignation, à minimiser les ressources en partageant les unités fonctionnelles mettant en œuvre des opérations par des méthodes de mutualisation ou de factorisation. Il s'agit d'une étape de mise en œuvre d'unités fonctionnelles partagées.

La figure 2.5. illustre sur un exemple, les étapes d'***Allocation-Assignement*** et ***Optimisation*** par mise en œuvre d'unités fonctionnelles partagées. Dans cet exemple, deux solutions possibles des étapes ***Allocation-Optimisation*** et d'***Assignation*** sont présentées.

2.2.1.3 Un Outil de Synthèse architecturale : GAUT

GAUT (Générateur Automatique d'Unité de Traitement) est un environnement de synthèse d'architecture matérielle, dédié aux algorithmes de Traitement du Signal et de l'Image (TDSI) sous contrainte de cadence d'échantillonnage. Gaut est aussi un outil universitaire qui résulte de travaux de recherche commencés au LASTI dans les années 1990 et poursuivis au LESTER depuis 1994 ([EM93] et [JPM95]). Le développement informatique représente actuellement un volume de l'ordre de 200000 lignes de code C et JAVA. À partir de la spécification d'un algorithme de TDSI en VHDL (spécifié en niveau comportementale), il synthétise une description structurelle VHDL de niveau RTL optimisée en surface et destinée aux outils de synthèse logique du marché tels que ISE (Intergrated Software Environment) Foundation de Xilinx (Profil FPGA), Quartus d'Altera (Profil FPGA) ou DC (Design Compiler) de Synopsys (Profil ASIC) [Syn07].

Le modèle cible des architectures synthétisées par GAUT est un coeur générique de processeur dédié au traitement du signal DSP. Ce modèle est composé de trois unités fonctionnelles travaillant en parallèle : unité de traitement (UT), unité de mémorisation (UM), et unité de communication (UCOM). Une horloge commune assure le fonctionnement synchrone entre les différentes unités qui fonctionnent en mode pipeline et qui s'échangent les données au travers

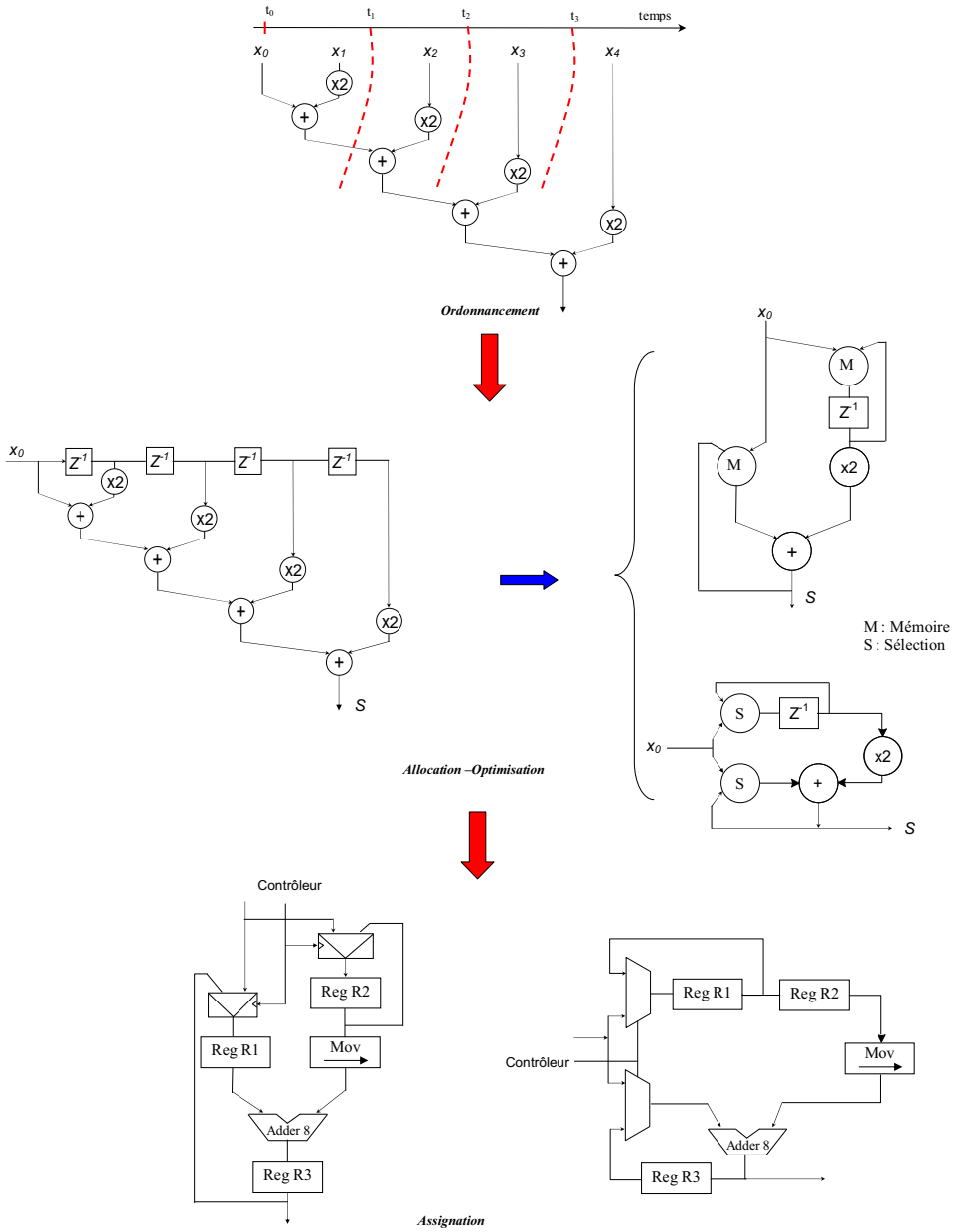

FIG. 2.5 – Illustration des étapes interdépendantes d'***Allocation-Assignation-Optimisation***.

d'un réseau multi-bus parallèles.

Unité de Traitement

L'unité de traitement (UT) est en charge de la partie calcul de l'algorithme. Le modèle sur lequel elle est basée s'articule autour de cellules (arithmétiques ou logiques) élémentaires. Chaque cellule est composée d'un opérateur, d'un ensemble de registres, de multiplexeurs, de démultiplexeurs et de buffers trois états. Les registres permettent le stockage temporaire des données et la synchronisation des transferts de données au sein de l'UT. Les multiplexeurs, démultiplexeurs et buffers trois états ont en charge l'aiguillage des données. Leur présence optionnelle dans une cellule résulte du partage des registres et des opérateurs, réalisé durant la phase d'optimisation du matériel. Les cellules communiquent au travers d'un réseau de bus parallèles. Les bus dédiés assurent les échanges entre les cellules, alors que les bus généraux permettent les accès aux mémoires contenues dans l'unité de mémorisation. Une machine d'états finis, qui résulte de l'ordonnancement des opérations, et un décodeur d'instructions se chargent de générer les signaux de contrôle à destination des registres et buffers trois états.

Unité de mémorisation

L'unité de mémorisation (UM) répond au problème de stockage des données. Elle est constituée de bancs mémoires (RAM / ROM), de registres et de générateurs d'adresses. Un générateur est une machine d'états finis dont chaque état correspond à la lecture de la valeur d'une constante ou à la lecture/écriture de la valeur d'une variable en RAM [GCM03a, GCM03b].

Unité de Communication

L'unité de communication UCOM est composée d'une partie opérative (chemin de données) et d'un contrôleur [ABM98]. Le chemin de données inclut des modules de stockage des données d'entrées-sorties (FIFO, LIFO, registres), des bus (internes et externes) et des éléments d'interconnexion (multiplexeurs, démultiplexeurs et buffers trois états). Les bus externes représentent les liens physiques entre le coeur d'IP et les composants du système. Les bus internes représentent les liens physiques avec l'unité de traitement. Le nombre est défini par la synthèse et peut être supérieur ou égal au nombre maximum d'échanges simultanés de données entre l'UCOM et l'UT. Le contrôleur réalise d'une part le protocole de communication système, et d'autre part la synchronisation entre l'unité de communication et l'unité de traitement.

La conception de l'UCOM s'effectue à partir des contraintes de l'unité de traitement synthétisée sous contrainte de cadence, et des contraintes du système dans un flot de conception matériel / logiciel [Bag97]. Dans certains cas, les contraintes imposées par l'unité de traitement et le système sont trop éloignées pour être adaptées par une interface, rendant ainsi la synthèse

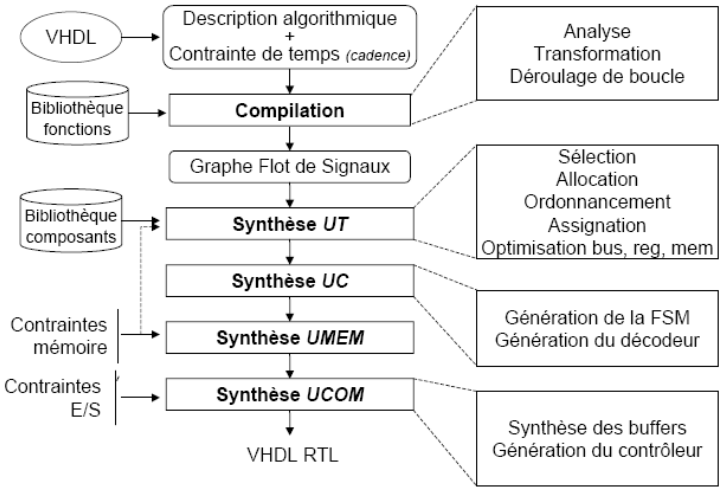

FIG. 2.6 – Flot de conception de l'environnement GAUT.

de l'UCOM impossible. Actuellement, la stratégie de synthèse accepte en entrée une spécification des contraintes temporelles sur les entrées-sorties [Cou03].

Le flot de conception mis en œuvre dans GAUT est représenté par la figure 2.6. La description algorithmique de la fonction à réaliser est décrite à l'aide du langage VHDL par un unique processus (couple entité / architecture). La description initiale est accompagnée d'une contrainte temporelle. Cette contrainte, appelée " cadence ", correspond à la plage temporelle sur laquelle s'étale l'arrivée de l'ensemble des données nécessaires pour une itération de l'algorithme. La phase de compilation effectue une analyse syntaxique, un contrôle sémantique et une parallélisation du code. Cette dernière étape réalise la suppression des dépendances de contrôle (déroulage des boucles, mise en ligne des appels de procédures, parallélisation des branchements conditionnels) et la suppression des fausses dépendances de données. En effet, une distinction entre les vraies dépendances et les fausses dépendances que sont l'anti-dépendance et la dépendance de sortie est faite dans l'outil GAUT. En raison de l'assignation unique, la parallélisation du code se termine par un renommage des variables [Bom04, LES06].

La compilation produit une représentation de l'algorithme sous la forme d'un graphe flot de signaux (SFG) spécifiés dans le format GC [INR93]. Le format GC est une représentation hiérarchisée de style flot des données ou bloc-diagramme d'un programme synchrone. Il est utilisé par les outils de programmation synchrone. Il permet l'interfaçage du coeur de synthèse de GAUT avec ces derniers. Cette modélisation permet l'expression du parallélisme de l'algorithme. Après l'étape de synthèse de l'unité de traitement, les unités de contrôle, de mémorisation et de communication sont générées. Afin de garantir la compatibilité avec le plus grand nombre d'outils de synthèse RTL du commerce, le VHDL RTL produit par GAUT est strictement conforme à

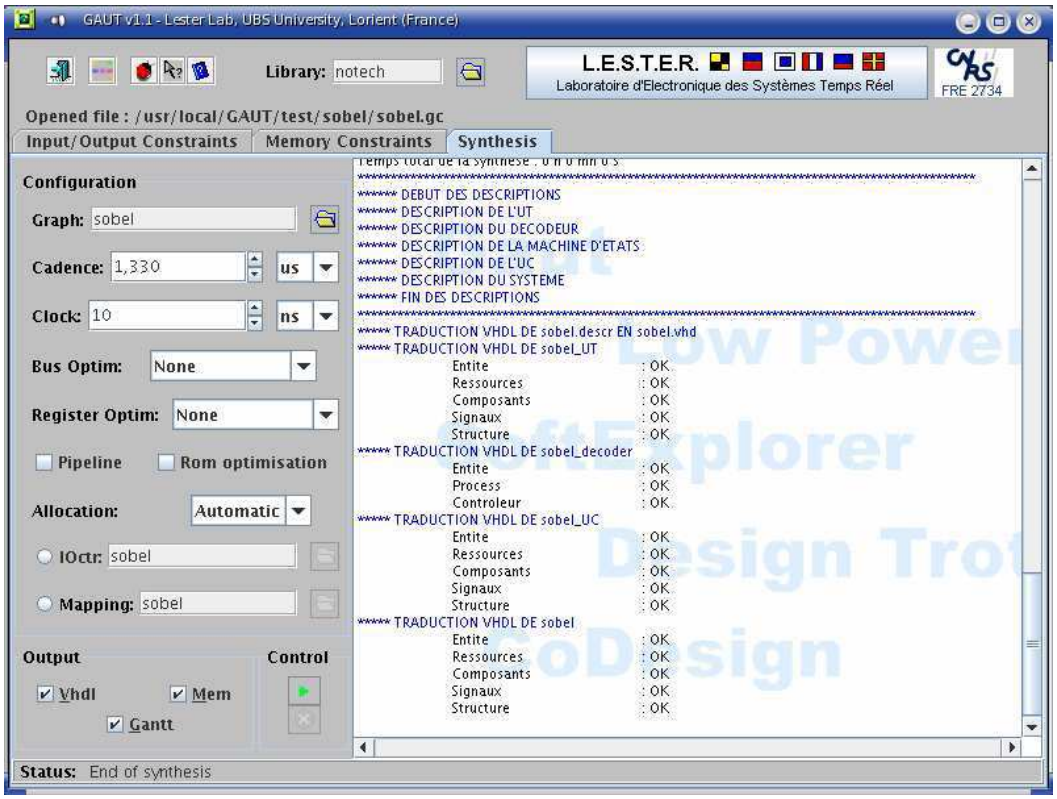

FIG. 2.7 – L'interface de l'environnement de GAUT.

la norme IEEE P1076 (Standard for VHDL Register Transfer Level Syntesis). La figure 2.7 présente l'environnement graphique de l'outil GAUT.

2.2.2 Reconfiguration Dynamique par Partitionnement Temporel

2.2.2.1 Principe Général

Le calcul reconfigurable dynamiquement par partitionnement consiste à une exécution par parties successives d'un traitement sur un même circuit. L'objectif est d'implémenter par permutation et repliement temporel différentes parties d'un algorithme sur une même structure matérielle par reconfiguration d'une matrice FPGA, selon un partitionnement et un ordonnancement respectant d'éventuelles contraintes matérielles ou de traitement [GL99, LD94], afin d'augmenter la densité fonctionnelle [Wir97]. Ainsi, il est possible de minimiser les ressources logiques en terme de surface nécessaire pour implémenter une application tout en augmentant le rendement effectif du silicium [Ta03, Ma06]. Cette approche est intéressante pour le développement d'applications embarquées basées sur des circuits de type RSoC car elle permet de conserver une flexibilité sans pénaliser la surface globale du SoC en évitant un surdimensionnement des ressources et une optimisation significative totale du coût. Cependant, cette approche nécessite une identification des différentes parties de l'algorithme permettant une décomposition temporelle optimale (partitions temporelles). En effet, en plus d'un partitionnement spatial, un **partitionnement temporel** doit être mis en œuvre. Chaque partition déduite de ce partitionnement spatio-temporel correspond à une étape de traitement et donc à une configuration du circuit. Chaque partition correspondra alors à un " floorplan " temporel possible de la matrice FPGA du système embarqué reconfigurable dynamiquement. La figure 2.8 illustre la décomposition par partitionnement temporel d'une application décrite sous forme d'un graphe flot de données.

2.2.2.2 Considérations pour la mise en œuvre de la Reconfiguration Dynamique

La mise en œuvre du partitionnement temporel par l'exploitation de la reconfiguration dynamique doit prendre en considération un certain nombre de contraintes. Ces dernières sont d'une part de nature matérielle (technologies existantes) et d'autre part liées aux caractéristiques et à la diversité des algorithmes de traitement à mettre en œuvre.

– Au niveau technologique : Les systèmes exploitant la reconfiguration dynamique des FPGAs nécessitent une durée allouée aux reconfigurations. Le respect de contrainte de temps réel suppose un minimum de solutions au niveau gestion et exécution matériel pour réduire la durée des calculs (traitement à accélérer) et minimiser l'impact du délai nécessaire à la reconfiguration [JIA07]. Au niveau technologique, pour obtenir des durées de reconfigura-

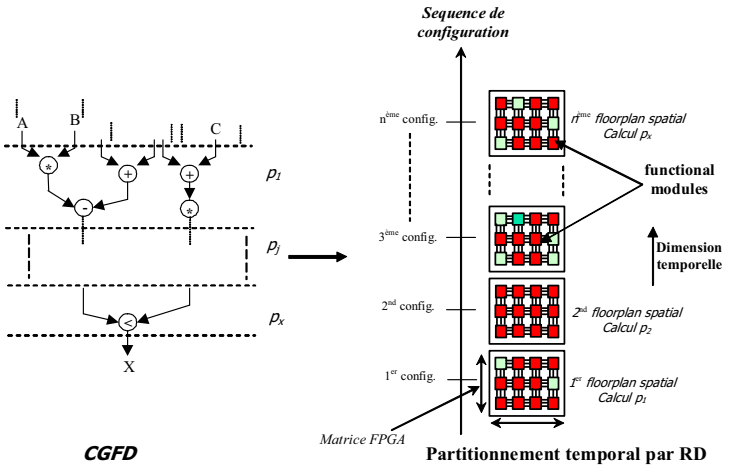

FIG. 2.8 – Illustration de la RD par partitionnement temporel [Tan01].

tion plus courtes, les fabricants de FPGA ont rendu possible la reconfiguration partielle de leur circuit. Cela permet de ne reconfigurer que les différences entre deux étapes successives, et par ce biais de réduire le volume des données et leur durée de chargement. De plus, les chemins critiques variant d'une partition à une autre, il en résulte une cadence de travail maximale propre à chaque partition. C'est pourquoi, pour bénéficier de la meilleure accélération des traitements, il faut disposer d'une variété de fréquences, afin d'adapter la période d'horloge de calcul au chemin critique de l'étape à implanter. Or, l'un des problèmes majeurs dans les systèmes reconfigurables, est la génération de signaux d'horloges qui doivent s'adapter de manière très rapide afin de ne pas compromettre l'exploitation de la reconfiguration (un temps de stabilisation d'une synthèse de fréquence inférieure au temps de reconfiguration de la partition considérée). Dans le cas contraire, la rapidité de traitement sera tributaire du temps de stabilisation du réseau de distribution de signaux d'horloge. Actuellement, outre un réseau fiable de distribution d'horloge, une solution pour éviter ces contraintes consiste à utiliser des méthodes de synthèse de fréquence interne en exploitant des multiplieurs ou diviseurs de fréquence à partir d'une fréquence de référence.

– Au niveau application : Dans beaucoup de domaines d'applications (traitement du signal et des images, etc.), les FPGAs sont capables d'atteindre des fréquences de travail nettement supérieures à la fréquence d'arrivée des données. Il peut exister alors un rapport élevé entre la fréquence de travail et la fréquence d'acquisition des données à traiter. L'accélération des traitements est alors possible et peut donc permettre de libérer un temps précieux pour reconfigurer le circuit tout en respectant des contraintes temporelles. Pour exploiter pleinement cette opportunité, il est intéressant de prévoir un mécanisme de désynchroni-

sation entre la fréquence du système d'acquisition et la fréquence de travail du système de calcul [H.G97, RBK98, DD98, LKD06]. Cette étape de désynchronisation peut être assurée par un tampon d'entrée grâce à des mémoires adressées en alternance à des cadences différentes : l'une à la cadence d'entrée des données et l'autre à celle des traitements. Ce tampon d'entrée présente un autre intérêt pour un système reconfigurable dynamiquement (**RD**) car il permet de traiter des données par paquets, plutôt que de manière individuelle. En procédant ainsi, le temps perdu pour chaque reconfiguration est amorti sur le volume de données. Pour certaines applications, la nature des traitements et des données dicte la taille des données à choisir. Par exemple, dans le domaine d'application du traitement d'images, on choisit souvent une taille de bloc de données multiple entier du nombre exact de pixels de l'image (taille de l'image, trame, ligne, etc.) et identique à chaque étape de traitement [BB05]. En pratique, on portera une grande attention à ce choix, car il détermine le nombre de reconfiguration possible de l'implantation RD des opérateurs. Il est important de remarquer que cette désynchronisation entre acquisition et traitement a l'inconvénient d'introduire un délai de latence supplémentaire afin de constituer le paquet de données à traiter. Il s'agira alors, pour certaines applications, de savoir si ce temps de latence est préjudiciable ou non.

2.2.2.3 Flot de conception pour l'exploitation de la RD par partitionnement temporel

Le processus typique d'un partitionnement temporel d'une application par reconfiguration dynamique est composé d'une succession d'étapes qui extrait les partitions successives d'une description à travers les étapes interdépendantes suivantes : ***Analyse des contraintes***, ***Ordonnancement - Allocation***, ***Assignement*** des ressources FPGA et ***Génération de Bitstreams***. La figure 2.9 illustre le flot de conception traditionnel du processus de partitionnement temporel par reconfiguration dynamique. Ce processus d'exploitation de la reconfiguration dynamique s'apparente à celui de la synthèse architecturale. En effet, comme pour la synthèse architecturale, le processus de conception RD est basé sur une représentation d'un algorithme sous forme de graphe flot de données/contrôle (GFDC). Le graphe flot de données (GFD) renseigne sur les dépendances de données de l'algorithme. Il modélise les chemins de données de l'algorithme. Les noeuds du graphe représentent les opérations de calcul et les arcs du graphe représentent les dépendances de données entre les différents noeuds. Le graphe de contrôle (GC) représente le séquencement des opérations. Il modélise la partie contrôleur de l'algorithme. À chaque arc sont associées les conditions d'enchaînement des opérations.

– L'étape d'**Analyse consiste**, à partir d'une description GFDC du traitement, à mettre

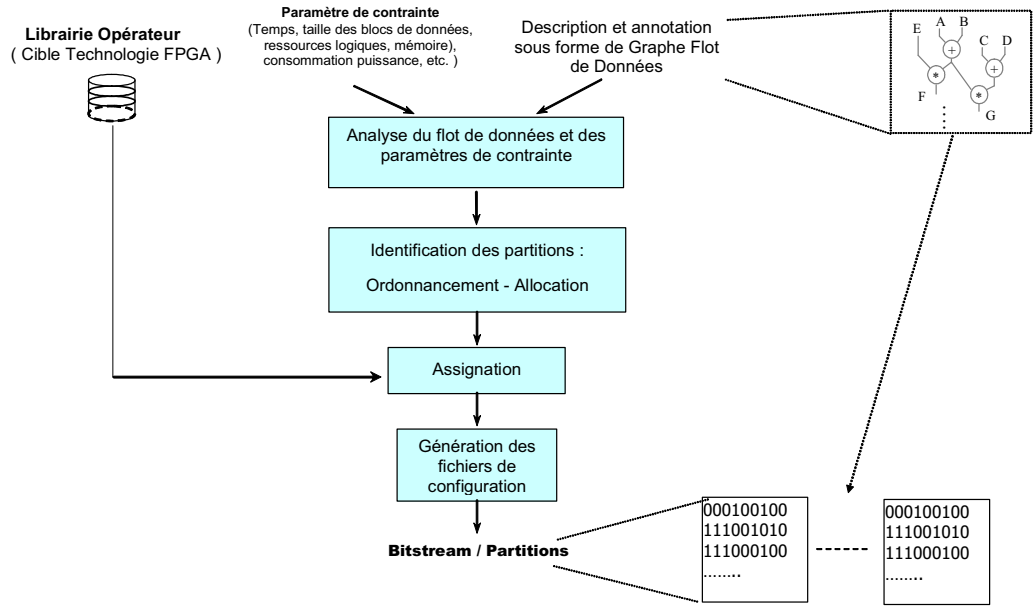

FIG. 2.9 – Flot de conception du processus de partitionnement temporel par RD.

en œuvres, des contraintes de traitement ou de conception (temps réel, bande passante, consommation puissance, surface logique, etc.), des caractéristiques de la technologie reconfigurable utilisée, d'effectuer une exploration des partitionnements possibles répondant aux contraintes de conception, technologique et de traitement. C'est dans cette étape que la méthode proposée est réalisée par une estimation du nombre de partitions temporelles répondant aux contraintes temporelles d'une application [Tan01, Bru04].

– Contrairement au flot de conception par synthèse architecturale, l'étape d' ***- Allocation*** correspond à l'affectation temporelle et spatiale des opérations dans des partitions (ordonner les opérations par placement sur la matrice reconfigurable à des instants précis correspondant à l'exécution d'une partition du traitement). Par contre, l'étape d'***Assignation*** effectue les même tâches que celles existantes dans le flot de conception de la synthèse architecturale (voir § 2.2.1.2) sans cependant intégrer la sous-étape d'***Optimisation*** existant dans le flot par synthèse architecturale et qui correspond à une recherche de mutualisation des unités fonctionnelles (optimisation par unités fonctionnelles partagées).

– La dernière étape, ***Génération Bitstream***, consiste à générer les fichiers de configuration de la cible reconfigurable des partitions à l'aide des outils de développement de la technologie reconfigurable utilisée. Cette phase est réalisée à partir de description synthétisable des partitions (description schématique, langage de description matériel synthétisable tel

que VHDL, Verilog, SystemC, etc.), par les outils associés à la technologie reconfigurable considérée en mettant en œuvre des étapes successives de synthèse logique, de placement et de routage sur les ressources disponibles dans la cible technologique.

La figure 2.10. illustre sur un exemple, les étapes d'***Ordonnancement-Allocation***, ***Assignement-Optimisation*** par mise en œuvre d'unités fonctionnelles partagées. Dans cet exemple, deux solutions possibles des sous étapes précédentes sont présentées.

Deux types de technique de partitionnement sont généralement utilisées : celles regroupant les méthodes constructives et celles basées sur des améliorations itératives ou heuristiques [Bou04]. Pour les méthodes constructives, aucune partition initiale n'est requise. Ces méthodes partitionnent le graphe en démarrant par un ou plusieurs noeuds. La création des partitions se fait alors en ajoutant progressivement des noeuds tout en respectant des fonctions de coût. Avec ces méthodes, le partitionnement se fait rapidement mais les partitions générées ne sont pas toujours optimales. Les méthodes de partitionnement par amélioration itérative débutent par une partition initiale. Le résultat est amélioré en déplaçant des noeuds entre partitions selon des approches pouvant être heuristiques.

2.2.2.4 Méthodes de Partitionnement temporel pour systèmes reconfigurables

2.2.2.4.1 Méthodologie de partitionnement temporel par approche itérative

[ea03] propose une méthode itérative de partitionnement temporel de tâches sur une unité reconfigurable dynamiquement (FPGA). L'approche proposée opère sur une représentation graphe flots de données mixtes (flots de contrôle/flots de données). Le partitionnement temporel proposé s'effectue uniquement sur le graphe flot de données. Le partitionnement porte donc sur l'ensemble des tâches dans les chemins de données. Les tâches sont regroupées pour former des contextes. Un seul contexte est actif à la fois sur l'unité reconfigurable. Plusieurs tâches peuvent s'exécuter en parallèle au sein d'un contexte. Le partitionnement est ici considéré comme un processus d'allocation et ordonnancement de tâches plus ou moins prioritaire dans l'unité reconfigurable tel qu'un ou plusieurs critères soient optimisés. Le critère retenu par la méthode proposée consiste à profiter au mieux des ressources disponibles dans l'unité reconfigurable FPGA pour minimiser le temps global d'exécution de l'application. L'approche effectue un ordonnancement statique et non préemptif des tâches du GFD [ea03]. Le partitionnement proposé ici est basé sur un algorithme génétique (AG). Cet algorithme se base sur une procédure itérative durant laquelle un ensemble de générations (ou populations) sont créées par itération. L'entière population évolue simultanément de façon à ce que la probabilité de convergence vers un minimum local soit réduite [ea03]. La procédure d'évolution est détaillée dans les étapes suivantes :

– *Le Codage* : Les temps d'exécution de chaque tâche sont estimés en terme de points de com-

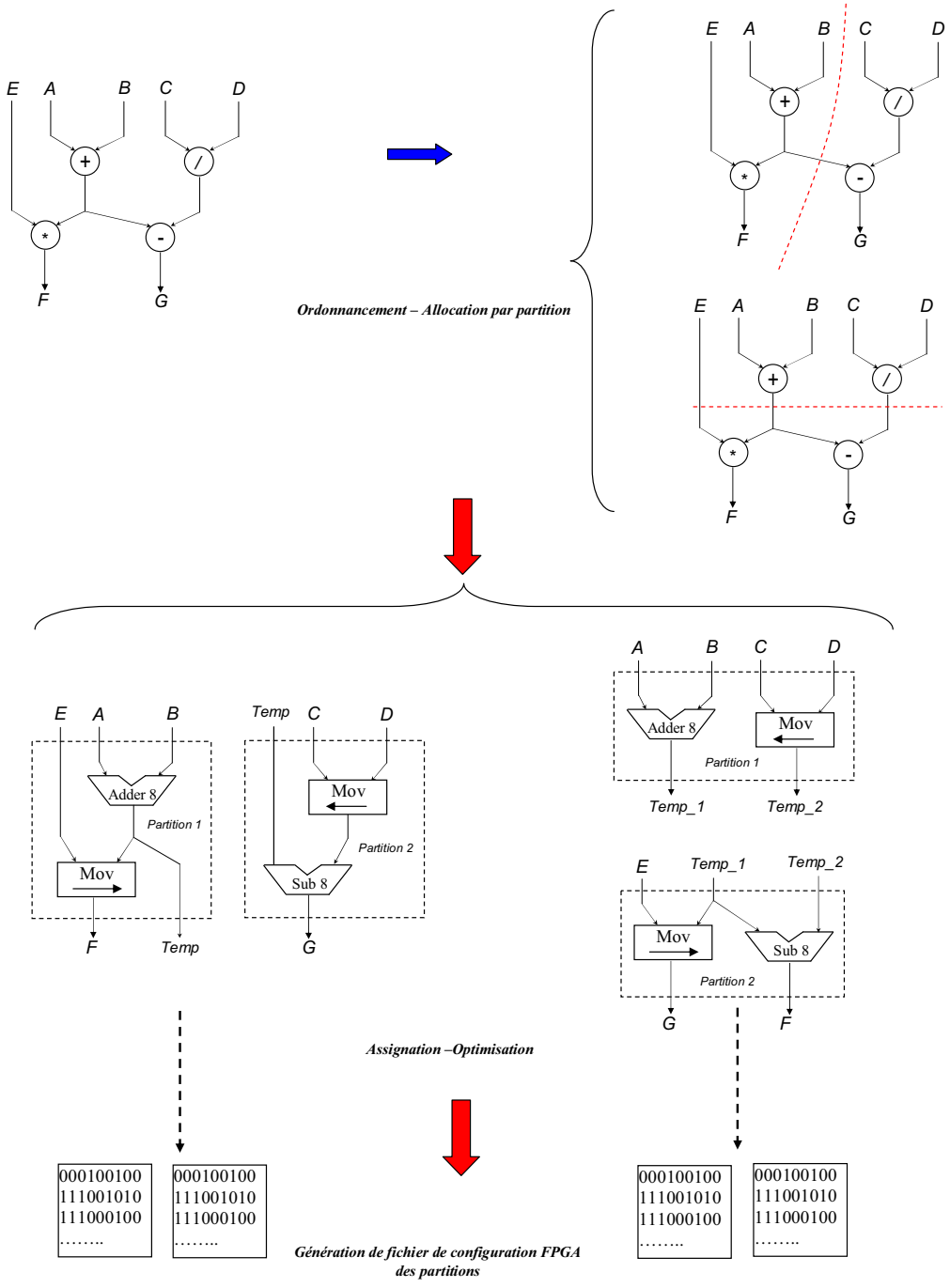

FIG. 2.10 – Illustration des étapes interdépendantes du flot de conception par partitionnement temporel.

promis temps/surface. Le nombre de ces points est variable en fonction de l'exploitation du parallélisme disponible dans chacune des tâches. Le codage de toute solution fournit une information sur la correspondance entre chaque tâche et l'un de ces points d'implantation. Si on considère N tâches, la méthode de codage code un chromosome C avec un vecteur de gènes de longueur N. Chaque gène $C(i)$ est un entier représentant un pourcentage. La valeur maximale 100% qu'un gène $C(i)$ peut prendre est associée à l'implantation la plus coûteuse en terme de cellule logique (CL) dans l'unité reconfigurable de la tâche T_i. Le point d'implantation sélectionné est le point le plus proche de $C(i)$ sur l'axe de la surface. Toutes les solutions délivrées par ce codage sont ainsi viables. Les tâches allouées doivent être groupées au sein de contextes pour arriver à évaluer les performances de l'individu.

– *L'Evaluation* : La performance de chaque chromosome (ou solution) délivré par l'AG est évaluée pour le classer à l'intérieur de la population courante. L'approche consiste à évaluer une solution en terme de son temps global d'exécution y compris les temps de reconfiguration du FPGA suite aux changements de contexte et les temps de communication entre les différentes tâches.
Le groupement des tâches dans des contextes s'effectue à partir d'une évaluation des temps d'exécution des tâches. La priorité d'une tâche est la longueur du plus long chemin partant de cette tâche et arrivant à une feuille du GFD évaluée en terme de temps d'exécution et de communication. La priorité P_i de la tâche T_i est calculée en considérant le temps d'exécution $T_{exec}(i)$ de la tâche T_i, les priorités de ses successeurs j et les temps de communication entre l'unité reconfigurable et la mémoire d'interfaces (mémoire de stockage des données intermédiaires entre les configuration) $T_{com}(e_j)$ des données des tâches T_i vers les tâches T_j lorsque la reconfiguration est terminé [ea03].

$$P_i = T_{exec}(t_i) + SUP_{(t_j \in Succ(t_i))}(P_j + T_{com}(t_i, t_j)) \tag{2.1}$$

Le temps de communication prend en compte le temps de reconfiguration dépendant de la quantité de cellules logiques utilisées pour réaliser un contexte sur FPGA.

– *Le partitionnement* : la méthode considère la taille d'un contexte, limitée à la taille maximale du FPGA en terme de cellule logique (CL) $S_{max(CL)}$ (en pratique, 80 à 85 % du nombre total de CL) et en terme de cellules dédiées (CD) $S_{max(CD)}$. Initialement, toutes les tâches sont triées par ordre de priorité décroissante établi par l'étape précédente. Ensuite, on considère la tâche à allouer dans un contexte $T_i(t_i, S_i(CL); S_i(CD))$ possédant la plus haute priorité, où t_i représente le temps d'exécution et $S_i(CL)$ (respectivement $S_i(CD)$) le nombre de CL (respectivement CD) correspondant au point d'implantation pointé par

$C(i)$ dans le chromosome. Les ressources disponibles $Ress(C_{curr})$ dans un contexte courant Ccurr (initialement à $S_{max} = S_{max(CL)} + S_{max(CD)}$) sont décrémentées de S_i , la taille de T_i. $(S_i(CL) + S_i(CD))$. Cette procédure de construction de contexte est itérée avec les tâches $T_j(t_j, S_j(CL); S_j(CD))$ tant que :

$S_j(CL) \cdot Ress(CL)(C_{curr})\&\&S_j(CD) \cdot Ress(CD)(C_{curr})$.
Sinon, un nouveau contexte est créé et le processus est ainsi répété jusqu'à ce que toutes les tâches soient attribuées à des contextes.

Une fois les contextes définis, l'algorithme met à jour les temps de communication inter-contextes (entre différents contextes), et mettant à jour de nouveau l'ordre de priorité des tâches. Un processus d'itération du partitionnement est alors de nouveau mis en œuvre. Plusieurs opérateurs génétiques sont appliqués aux $N_{parents}$ individus sélectionnés pour générer les $N_{enfants}$ solutions représentant la nouvelle génération. Avec la création de la nouvelle génération, le renouvellement de la population est effectué selon le principe élitiste. Lorsqu'un nombre de générations N_{gen} s'est écoulé sans aucune amélioration du meilleur individu, l'AG arrête son exécution et affiche la meilleure solution rencontrée.

Enfin, on obtient un partitionnement de contextes ordonnancés où les communications requises pour transférer les données et les opérations de reconfiguration sont également définies.

2.2.2.4.2 Méthodologie de partitionnement temporel par approche constructive

Cette méthode de partitionnement [Cha98] consiste à décomposer une application en modules vi élémentaires (tâches ou opérations) puis à les grouper pour les intégrer dans des processeurs élémentaires (PE) en satisfaisant certaines contraintes. Ces PE correspondent à des processeurs tel que des DSP, des FPGA, etc. Leur nombre est inférieur au nombre de modules vi. Les contraintes considérées sont relatives à la surface de calcul disponibles, au nombre d'entrées/sorties qui ne doit pas dépasser le nombre de broches disponibles et à la puissance consommée par la totalité des modules dans le PE qui ne doit pas dépasser une certaine limite.

Le processus de cette méthode de partitionnement est constitué de plusieurs étapes :

- La première étape consiste en la construction d'un graphe d'assignation G_a, qui est composé d'un certain nombre de rangées ou de couches correspondant chacune à un PE. Chaque couche contient un ensemble de noeuds qui correspondent aux différentes assignations possibles. La constitution des noeuds dans chaque rangée se fait sous contrainte de surface, de nombre d'entrées/sorties et de puissance.
- La deuxième étape consiste à définir les liaisons entre noeuds. Des poids sont attribués à chaque liaison, ils correspondent au temps requis par le PE pour exécuter les modules défi-

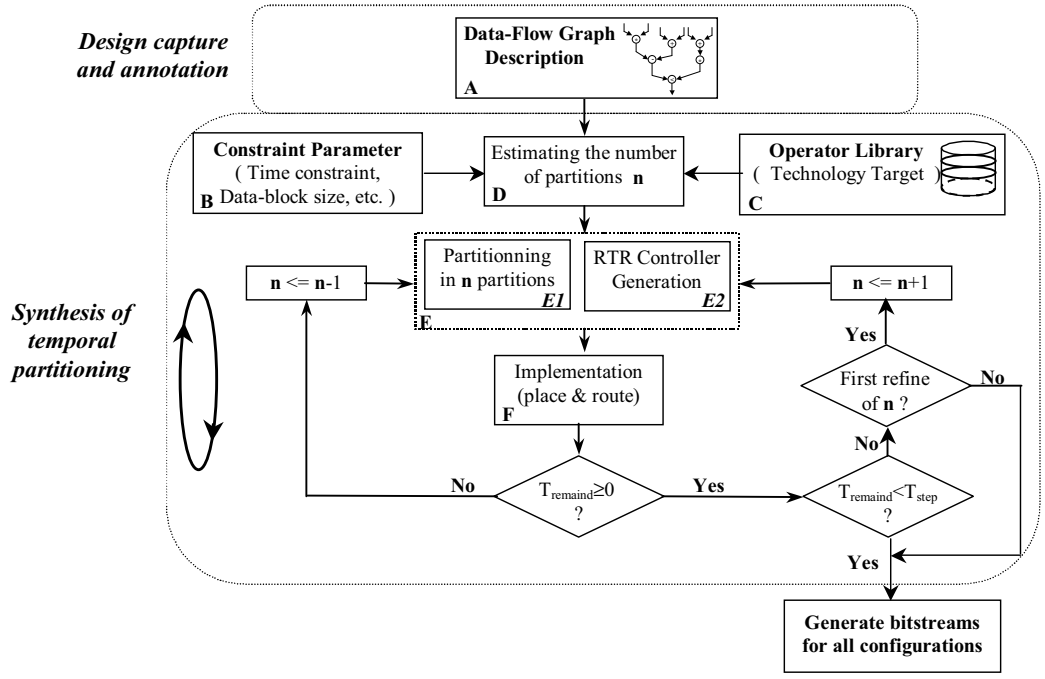

FIG. 2.11 – Partitionnement temporel sous contrainte de temps par estimation et raffinement.

nis par le noeud. Dans le cas d'un FPGA, ce temps correspond au module dont l'exécution est la plus lente.

– La troisième et dernière étape de cette technique de partitionnement consiste à rechercher les assignations optimales. Chaque module sera alors définitivement placé dans un PE. Cette recherche s'effectue en maximisant le débit et en minimisant le temps d'exécution de tous les modules. Elle est basée sur l'algorithme présenté dans [DGL92b].

2.2.2.4.3 Méthodologie de partitionnement temporel constructive par estimation et raffinement

Cette technique de partitionnement constructive par estimation et raffinement du nombre de partition a été développée dans [Tan01] et s'applique aux architectures reconfigurables dynamiquement [Abe06]. La figure 2.11. présente une vue globale de l'approche méthodologique de partitionnement par estimation et raffinement du nombre de partitions sous contrainte de temps.

Cette technique de partitionnement temporel s'effectue en trois phases :

– Dans la première phase, le nombre minimum de partitions est estimé. L'approche consiste d'abord à partitionner l'application en considérant les contraintes de temps et de dépendance de données en C partitions d'une application décrite sous forme d'un graphe flot

de données. Puis, on cherchera la surface minimale suffisante A_T qui permette d'exécuter l'implantation globale de l'application en reconfiguration dynamique, telle que :

$$\forall j \in [1 \dots C], Area(P_j) \leq A_T \tag{2.2}$$

Où $Area(P_j)$ correspond à la taille de tous les opérateurs élémentaires contenus dans la partition j. L'application doit impérativement respecter une contrainte temps réel dont la relation entre la contrainte de temps et la vitesse des opérateurs mise en œuvre s'exprime par l'expression suivante :

$$N \sum_{j=1}^{C} max(tv_i) \leq T - \sum_{j=1}^{C} trec_j \tag{2.3}$$

Où N est le nombre de données à traiter, C le nombre de reconfigurations, T la contrainte de temps, $trec_j$ le temps de reconfiguration de la j^{ieme} étape correspondant à la partition P_j. $Max(tv_i)$ est le temps d'exécution élémentaire de l'étape j, évalué à partir de la relation suivante :

$$max(tv_i) = to_j + tr_j \tag{2.4}$$

Où to_j est le temps d'exécution de l'opérateur le plus lent de l'étape j, et tr_j le temps de propagation le plus long lié au routage dans l'étape j ; si on le considère égal à une fraction du temps d'exécution élémentaire $\alpha_j.te_j$ $(0 < \alpha < 1)$, nous avons alors :

$$max(tv_i) = (\frac{1}{1-\alpha_j}) \cdot to_j = te_j = k \cdot to_j \qquad Avec \qquad k_j = \frac{1}{1-\alpha_j} > 1 \tag{2.5}$$

La contrainte temps réel de l'équation 2.3 s'exprimera alors par la relation suivante :

$$N \sum_{j=1}^{C} k_j \cdot to_j \leq T - \sum_{j=1}^{C} trec_j \tag{2.6}$$

De plus, si l'on considère l'expression suivante de $trec_j$:

$$trec_j = \frac{Area(P_j)}{V_c} = \frac{1}{V_c} \cdot L_j \tag{2.7}$$

Où L_j correspond au nombre de cellules logiques exploitées au cours de l'étape j. V_c est la vitesse de reconfiguration du circuit. Alors la contrainte de temps devient :

$$N \sum_{j=1}^{C} k_j \cdot to_j \leq T - \frac{1}{V_c} \cdot \sum_{j=1}^{C} L_j \tag{2.8}$$

Dans le cas d'une technologie qui ne permette qu'une reconfiguration totale du composant, la condition 2.8 s'exprimera par :

$$N \sum_{j=1}^{C} k_j \cdot to_j \leq T - E \cdot \frac{1}{V_c} \cdot L_c \tag{2.9}$$

L_c étant le nombre total de cellules logiques disponibles sur le circuit.
La détermination du nombre minimal d'étapes C_{min} est obtenue en considérant deux critères. Premièrement, avoir une reconfiguration maximale du circuit à chaque nouvelle étape. Deuxièmement, considérer le temps d'exécution élémentaire le plus long te_{max} pour chaque étape. À partir de ces deux considérations, le nombre d'exécutions/reconfigurations minimal est déduit de la relation suivante :

$$C_{min} = \frac{T}{N \cdot k_m \cdot t_{0max} + \frac{1}{V_c} \cdot L_{At}} \geq \frac{T}{N \cdot k_m \cdot t_{0max} + \frac{1}{V_c} \cdot L_{A_{G(V,E)}}} \tag{2.10}$$

L_{At} est le nombre maximal de cellules logiques par étape, et correspond donc à la surface minimale suffisante en cellules logiques. T_{0max} correspond au temps d'exécution maximal désignant l'opérateur le plus lent, et k_m la valeur moyenne liée à un taux d'occupation constant pour chaque étape. $L_{A_{G(V,E)}}$ est le nombre de cellules logiques correspondant à la surface $A_{G(V,E)}$ du graphe flot de données $G(V, E)$.
Dans le cas d'une utilisation d'une technologie à reconfiguration totale, le temps de reconfiguration maximale correspond au temps de reconfiguration globale du circuit $trec_{max} = TRG$. Le nombre minimal C_{min} d'exécutions/reconfigurations devient alors :

$$C_{min} = \frac{T}{N \cdot k_m \cdot t_{0max} + TRG} \geq \frac{T}{N \cdot k_m \cdot t_{0max} + \frac{1}{V_c} \cdot L_C} \tag{2.11}$$

Cette valeur C_{min} permet d'évaluer la faisabilité de l'implantation.

– Dans la seconde phase, la surface optimale pour chaque partition est calculée de la manière suivante :

$$S_c = \frac{L}{C} \tag{2.12}$$

Où L correspond au nombre de cellules logiques de l'algorithme. C représente le nombre d'étapes d'exécutions/reconfigurations. Dans cette seconde phase, un partitionnement par construction est réalisé. Le graphe est à nouveau parcouru en accumulant la taille des noeuds traversés, jusqu'à s'approcher au plus près de la surface optimale S_c. Cela constitue alors la première partition dont les noeuds sont retirés du graphe G. Ce procédé est réitéré jusqu'à obtention d'un graphe vide. Le processus de partitionnement est alors terminé. Si l'on constate l'existence d'un temps restant après ce premier partitionnement, un raffinement du nombre d'exécutions/reconfigurations est appliqué pour arriver à un

partitionnement plus précis. On entre alors dans la dernière phase.

– La dernière phase concerne l'affinement après implantation. Si le temps de calcul est plus grand que la contrainte de temps, le partitionnement est repris avec un nombre inférieur de partitions. En revanche, si le temps de calcul est plus faible que la contrainte de temps avec un écart significatif, le partitionnement est repris avec un nombre de partitions plus grand, entraînant alors une réduction de la surface optimale de calcul. Cette phase peut s'effectuer de manière heuristique.

2.2.2.5 Outils de partitionnement et d'exploration : D.A.G.A.R.D

Le laboratoire **L.I.E.N** de Nancy a développé un outil prenant en compte l'aspect reconfiguration dynamique des FPGAs et permettant de spécifier les caractéristiques d'une architecture reconfigurable : SoC ou plateforme à base de FPGA. L'objectif est de trouver la meilleure adéquation entre une application et son implantation sur cette plateforme. Cet outil appelé DAGARD (**D**écoupage **A**utomatique de **G**raphes pour **A**rchitectures **R**econfigurable **D**ynamiquement) permet de réaliser une exploration de découpage d'une application complète représentée sous forme d'un graphe flot de données (GFD) en plusieurs sous applications successivement exécutées sur la surface logique visée grâce à la RD.

Contrairement à l'outil Design Trotter [Ma05] qui s'intègre dans la conception conjointe logicielle matérielle sur FPGA et qui permet l'exploration de l'espace de conception vers les architectures reconfigurables en fournissant différents types d'estimation (surface, temps d'exécution, etc.) et d'information visant à guider le concepteur de systèmes embarqués hétérogènes [JPa06, Bos04, MAM03], DAGARD se limite uniquement à l'exploration du choix d'un partitionnement temporel selon différents critères.

La figure 4.5 représente une vue globale de l'environnement DAGARD [Ba03, Bru04]. Le fonctionnement de DAGARD repose sur la méthode de partitionnement initialement développée dans [Ta03] et présentée § 2.2.2.4.3. La figure 2.13 décrit son flot de conception.

DAGARD est basé sur une librairie d'opérateurs caractérisés qui dépend de l'architecture cible. Ces opérateurs sont des opérateurs bas niveau. Ils constituent les noeuds du graphe et les dépendances de données entre ces opérateurs constituent les arcs du graphe. Il est également possible de définir des macro-opérateurs dès lors que l'on est capable de les caractériser en terme de temps d'exécution et de surface logique sur la cible technologique. L'outil permet une recherche par exploration, des découpages optimaux d'une application de GFD réguliers ou non réguliers (traitements pouvant varier en fonction des données ou des résultats précédents) en sous-applications de tailles aussi proches que possible. Il délivre un nombre de partitions n

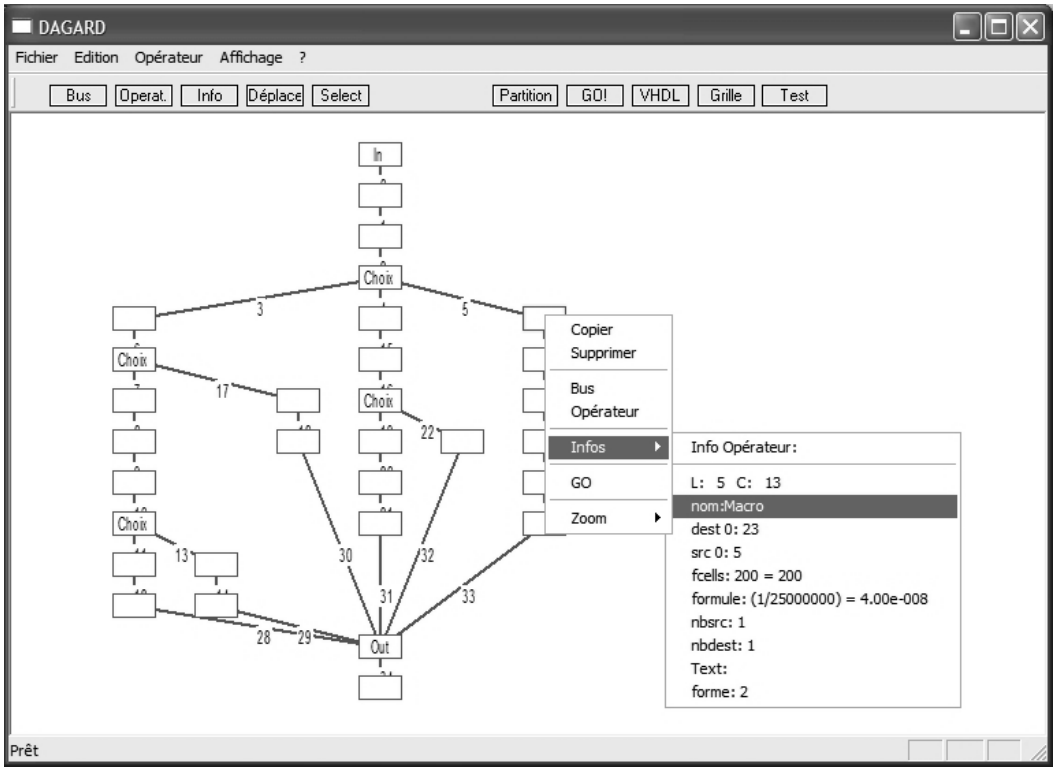

FIG. 2.12 – Vue générale de l'environnement de l'outil DAGARD [Bru04].

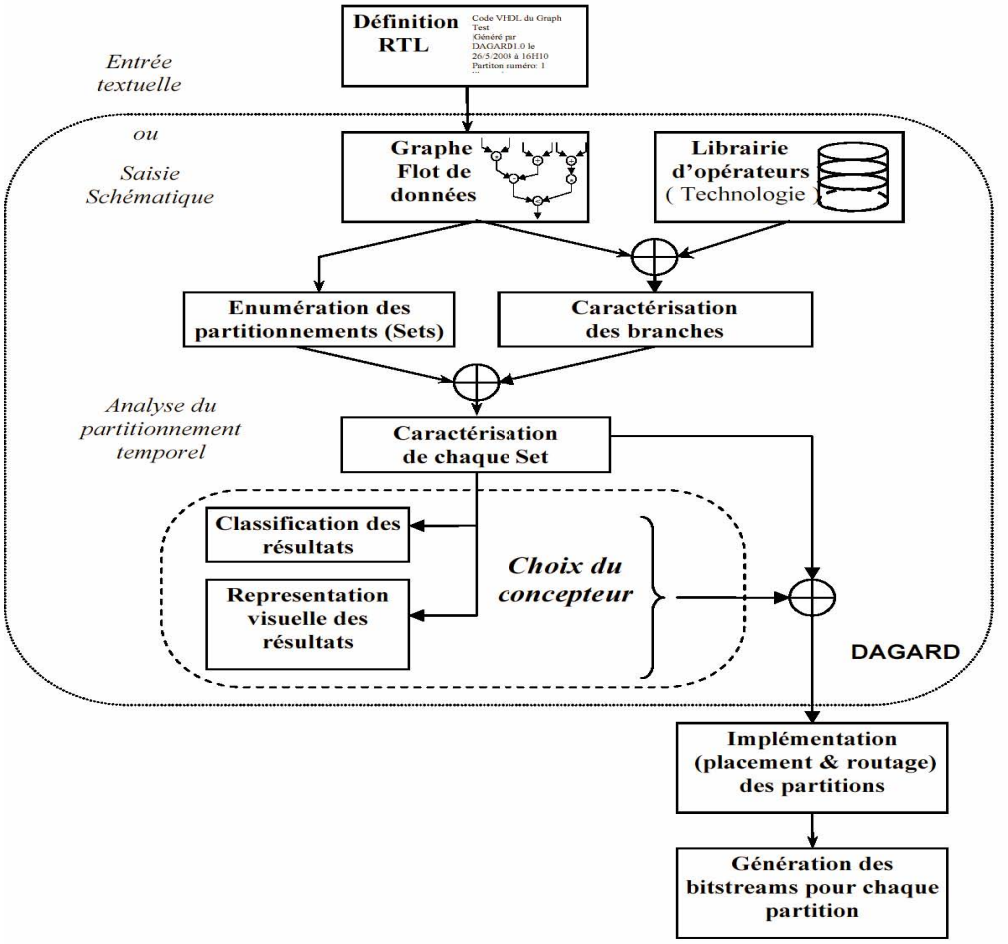

FIG. 2.13 – Flot de conception de l'Outil DAGARD [Bru04].

réalisable sous des contraintes. L'originalité de l'outil est qu'il présente une vue d'ensemble de toutes les possibilités de partitionnement. Du premier partitionnement qui correspond à une implémentation statique de l'algorithme (toutes les branches du graphe flot de données sont incluses dans une partition unique) au dernier partitionnement possible qui est le plus dynamique de tous (chaque branche du graphe flot de données peut correspondre alors à une nouvelle partition). Afin d'identifier le partitionnement répondant au mieux à ses besoins, DAGARD permet une navigation à travers les différents partitionnements possibles. Nous obtenons ainsi les caractéristiques de chacun des partitionnements possibles pour une application en reconfiguration dynamique.

La figure 2.14 présente un exemple de cartographies d'exploration de partitionnement généré par l'outil DAGARD. Les avantages et inconvénients des différentes possibilités sont rendus visibles, permettant ainsi au concepteur d'orienter l'optimisation de son application suivant diffé-

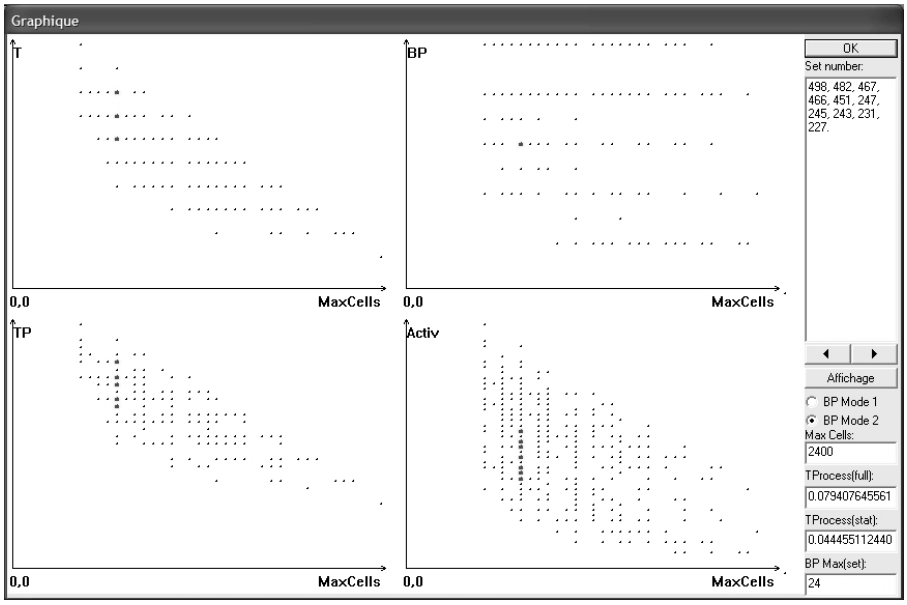

FIG. 2.14 – Vue graphique des possibilités de partitionnement [Bru04].

rents critères retenus. Le concepteur peut alors définir l'importance relative de chaque paramètre en fonction de ses besoins. En permettant une comparaison entre ces derniers suivant diverses caractéristiques, l'outil DAGARD aide et accélère le choix du partitionnement afin de créer les reconfigurations successives de la surface logique du FPGA. Ces caractéristiques sont : la surface logique nécessaire à l'implantation de l'application, la bande passante mémoire nécessaire, le temps d'exécution maximum de l'application, le temps d'exécution moyen de l'application (prenant en compte les probabilités d'exécution des branches), l'activité totale de la partition.

Dans ces conditions, un concepteur peut trouver la meilleure adéquation entre l'architecture ciblée et l'implantation de son application en reconfiguration dynamique tout en prenant en compte les ordonnancements possibles suivant les paramètres principaux d'une implémentation (surface logique totale nécessaire, temps de traitement, bande passante maximale nécessaire, consommation relative de ces différentes possibilités).

2.3 Limitations

Par la suite, nous considérons la reconfiguration dynamique comme le partitionnement temporel par reconfiguration successive et dynamique d'une structure FPGA et la synthèse architecturale comme la factorisation ou mutualisation d'unités fonctionnelles (unités fonctionnelles partagées) de la phase d'***Optimisation*** de l'étape de ***Allocation-Assigntion*** du flot de conception par synthèse architecturale.

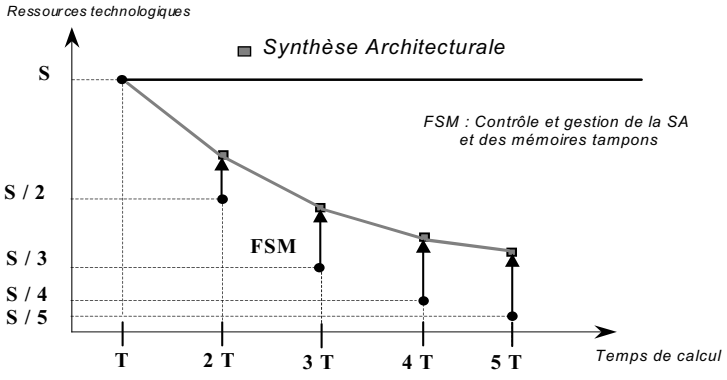

FIG. 2.15 – Evolution des ressources spatio-temporelles par la synthèse architecturale.

2.4 Analyse comparative : L'apport de la SA et de la RD

La synthèse architecturale et la reconfiguration dynamique sont deux approches qui ont pour objectif de trouver la meilleure adéquation entre un algorithme et une architecture (adéquation algorithme - architecture). Contrairement à la synthèse architecturale (**SA**) qui consiste, à travers des repliements architecturaux, à réutiliser des opérateurs qui doivent s'exécuter à des instants différents par l'addition de logique de contrôle, la reconfiguration dynamique (**RD**) des FPGAs consiste à implanter successivement sur le même circuit les opérateurs d'une application. Si l'on considère d'une part l'apport de la reconfiguration dynamique à travers le partitionnement temporel par un nombre de partition croissant, d'autre part, la synthèse architecturale par un degré croissant d'optimisation par la mise en œuvre d'unités fonctionnelles partagées, nous pouvons mettre en évidence les avantages et limites de ces deux approches. Les figures 2.15 et 2.16 montrent les évolutions attendues des ressources spatio-temporelles pour les deux approches méthodologiques. En effet, si on représente la surface nécessaire à l'implantation d'une application en fonction de la durée totale de traitement selon l'une ou l'autre des stratégies, on constate qu'au mieux, la synthèse architecturale peut diminuer par deux les ressources nécessaires pour un temps de cycle de calcul supplémentaire (figure 2.15).

Dans ce cas, vis à vis d'une synthèse architecturale, l'apport caractérisant la **RD** en matière d'implantation sur FPGA peut être illustré par le graphe en figure 2.16. Dans le cas de la reconfiguration dynamique, si on considère les temps critiques identiques à ceux de la synthèse architecturale et malgré une division successive des ressources par le nombre de partitions, il est nécessaire de comptabiliser les ressources logiques et mémoires liées au contrôle de la **RD** et à la gestion des mémoires tampons (enfouies ou non au FPGA) nécessaires à l'exécution d'une **RD**. Cependant, contrairement à la SA qui voit son contrôle augmenter avec le niveau de réutilisation des opérateurs, les ressources nécessaires au contrôle de la gestion des mémoires restent relati-

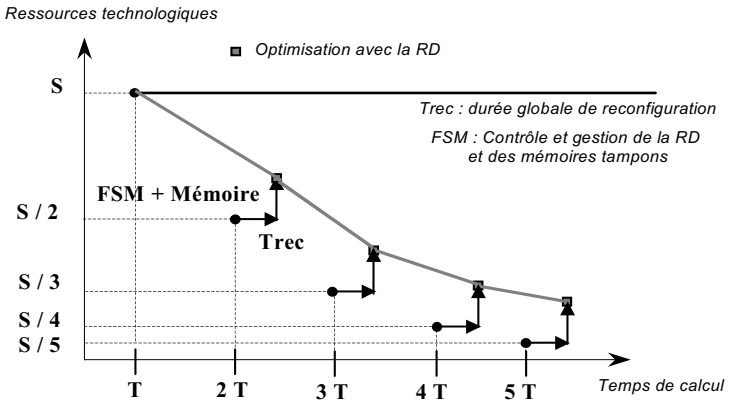

FIG. 2.16 – Evolution des ressources spatio-temporelles en RD.

vement identiques quelque soit le nombre de partitions. Par contre, un temps supplémentaire lié au temps nécessaire pour reconfigurer le FPGA entre chaque partition s'ajoute au temps de calcul (figure 2.16). Ce temps global de reconfiguration est également constant quelque soit le nombre de partitions si on suppose la possibilité de reconfiguration partielle et que la durée de reconfiguration est proportionnelle aux ressources implémentées.

Il est important de préciser que la **RD** présente des avantages dans le cas où les durées de configurations préjudiciables pour traiter une seule donnée sont amorties par le nombre élevé de données à traiter au cours d'une seule configuration. Sinon la **RD** perd de son intérêt, car le temps dédié à un traitement est principalement alloué aux phases de reconfiguration.

Si on effectue une analyse comparative du gain et du coût de l'utilisation de la reconfiguration dynamique par rapport à une synthèse architecturale classique, on constate que ces deux approches permettent de minimiser les ressources matérielles tout en respectant d'éventuelles contraintes de temps. Cependant, de grandes différences de résultats en terme de performances et d'optimisation des ressources peuvent apparaître selon l'application et les contraintes visées. Une façon d'optimiser l'implantation des algorithmes grâce à la **RD**, consiste à trouver les partitions qui conduisent à un même nombre de ressources logiques utilisées dans chaque partition [Ta03].

De ces observations, nous pouvons conclure que suivant la structure, la taille et le type des opérateurs à exécuter, la meilleure optimisation sera obtenue soit par une solution **RD**, soit par une solution **SA**. Ainsi, selon l'application et les contraintes de traitement ou de conception imposées, il s'agit de trouver quand il est préférable d'utiliser la reconfiguration dynamique ou la synthèse architecturale lors d'une implémentation sur technologie reconfigurable. Sur les figures 2.15. et 2.16, cela se traduit par la détermination des zones au-dessus de la **RD** indiquant pour cette dernière une meilleure optimisation des ressources vis-à-vis d'une solution par synthèse

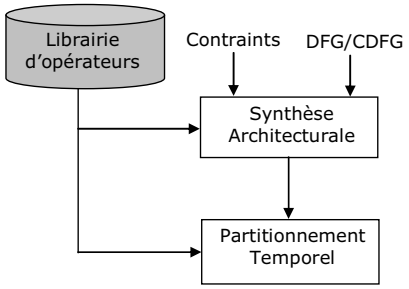

FIG. 2.17 – Etapes classiques d'une méthodologie optimisation basée RD - SA.

architecturale.

2.5 Méthodologies combinant le partitionnement temporel et la synthèse architecturale

A l'heure actuelle, nous identifions les travaux de Cardoso [Car03] et de Zhang [ZN00b] comme principaux travaux de développement méthodologique de conception combinant le partitionnement temporel et la synthèse architecturale. D'une manière générale, nous constatons que les approches méthodologiques des approches proposées reposent sur une l'application simultanée sur un graphe flot de données d'un partitionnement temporel et d'une optimisation individuelle des partitions par une approche de synthèse architecturale [Oa98, DGL92b]. Ce principe est illustré par la figure 2.17 suivante.

– L'orientation méthodologique proposée par Cardoso [Car03] consiste à combiner simultanément la technique de partitionnement temporel et la technique de réutilisation d'unités fonctionnelles de la synthèse architecturale. L'objectif de la méthodologie proposée par cette approche, consiste à trouver des partitions temporelles aboutissant à une solution de partitionnement dont le critère retenu ait une latence d'exécution minimum. La figure 2.18 présente le flot de conception globale de l'approche méthodologique proposée dans [Car03].

  A partir d'un graphe flot de données (GFD) correspondant à une description de l'application à implémenter, l'algorithme proposé dans [Car03] réalise une étape d'optimisation des UFs à assigner dans des partitions par réutilisation des unités fonctionnelles, en intégrant un multiplexage dans le temps des opérateurs factorisés. Cette optimisation repose sur une bibliothèque de composants et une caractérisation de chaque unité fonctionnelle mise en œuvre. Cette caractérisation intègre le nombre de ressources (CLB ou cellule logique) et la latence des unités fonctionnelles dans l'architecture reconfigurable cible. La méthode

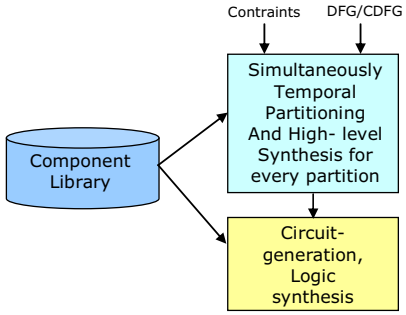

FIG. 2.18 – Le flot de conception SA - RD proposée dans [Car03].

consiste à rechercher par itération, à partir d'un nombre de partitions, les solutions possibles d'implémentation des UFs d'un graphe avec ou sans factorisation en respectant une contrainte de surface, d'ordonnancement et de temps d'exécution des opérations. La figure 2.20 suivante illustre le procédé méthodologique. La figure 2.20.a. présente un exemple de graphe flot de données contenant quatre additions et deux multiplications. Chaque opérateur est caractérisé par un nombre de cellules logiques et d'une latence en terme de nombre de cycles d'horloge (phase d'annotation).
L'approche combinée partitionnement et synthèse débute par un nombre initial de partitions vides obtenu par analyse des chemins critiques du GFD décrivant l'application. En fonction de ces derniers et à partir d'un ordonnancement des UFs contenues dans le graphe (topologique), les tâches sont affectées successivement dans les partitions selon d'une part le nombre de ressources et de latence des unités fonctionnelles exécutant l'application, et d'autres part, en respectant la contrainte de surface de technologie FPGA utilisée et de la latence maximum autorisée [Car03]. Le procédé de recherche est réalisé selon les phases ci-dessous :

– Définition d'un nombre de partitions de départ, initialement considérées vides, correspondant soit au nombre d'UFs constituant le chemin critique du graphe soit défini par le concepteur.

– Assignation des UFs dans les partitions selon l'ordonnancement des opérations, de la contrainte de surface et de la latence d'exécution des UFs considérées.

– À chaque assignement, une recherche par factorisation ou par réassignation des UFs dans les partitions permettant d'optimiser la latence de calcul, est réalisée tout en respectant la contrainte de surface.

Approach	Considering the sharing of FUs?	(+, ×)	#TPs	Execution latency	Resources used
Optimal latency	adders and multipliers	(1, 1)	1	4	3
Optimal latency	multipliers	1st TP: (1, 1) 2nd TP: (2, 0) 3rd TP: (1, 1)	3	5	3
ASAP leveling [17]	No	(4, 2)	3/4[3]	6	3
Without Temporal Partitioning	No	(4, 2)	1	4	8

FIG. 2.19 – Analyse des optimisations par réutilisation d'unité fonctionnelle [Car03].

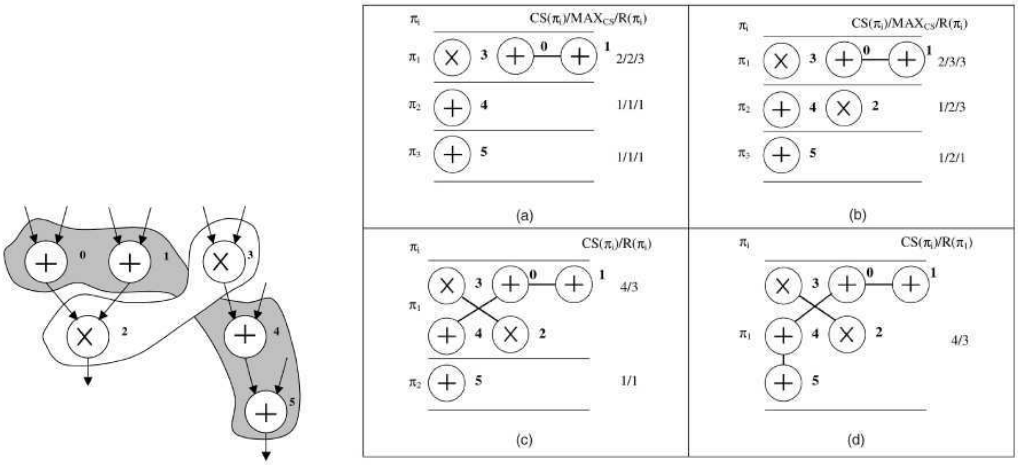

FIG. 2.20 – Illustration d'une caractérisation de la méthode proposée dans [Car03].

La figure 2.20.b. illustre les étapes progressives de l'approche proposée dans le cas du graphe flot de données représenté par la figure 2.20.a.

Plusieurs résultats d'optimisation sont présentés sur la figure 2.19 suivante selon le degré de factorisation ou réutilisation des unités fonctionnelles. Les solutions 1 et 2 (lignes 2 et 3, figure 2.19) sont considérées comme optimales par la réutilisation des unités fonctionnelles. Avec la solution 1, un additionneur et un multiplieur sont réutilisés. Dans ce cas, il n'y a qu'une partition (#TPs = 1, figure 2.19) avec une latence de 4 cycles d'horloge. La solution 2 ne réutilise qu'un seul multiplieur. Dans ce cas, le GFD est coupé en 3 partitions avec une latence de 5 horloges. La solution 3 est une méthode classique ASAP. Ici, aucune opération n'est réutilisée. Le nombre de partitions peut être 3 ou 4 avec des latences correspondantes à 5 ou 6. L'approche 4 est une solution sans partitionnement temporel. Cette solution possède une latence de 4 mais elle représente la solution qui nécessite le plus de ressources (8 cellules).

Une analyse comparative montre que les solutions optimales sont basées sur la réutilisation des unités fonctionnelles en offrant un meilleur compromis entre le nombre de partitions,

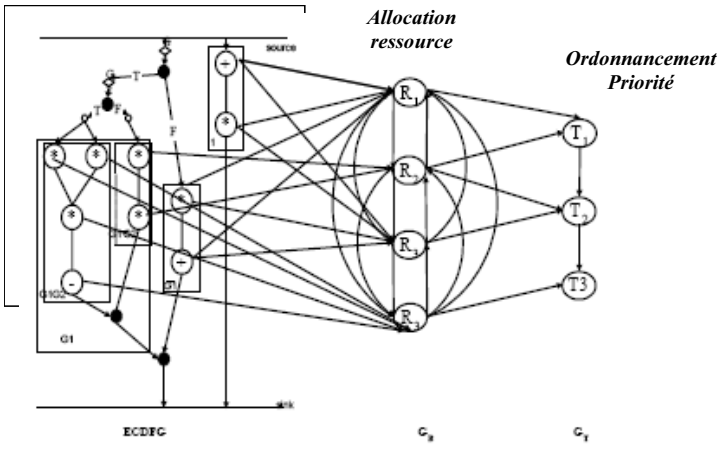

FIG. 2.21 – Exemple d'un graphique de synthèse [ZN00b].

la latence d'exécution et les ressources. Cependant la méthode proposée dans [Car03] ne justifie pas le nombre de partitions de départ nécessaires à l'application de la méthode proposée.

– Zhang [ZN00b] propose une approche par ordonnancement simultané de la synthèse architecturale et du partitionnement par exploration de l'espace de conception. L'objectif de la méthode proposée est une optimisation des phases de reconfiguration lors d'une implémentation dans un système reconfigurable dynamiquement. L'approche repose sur un algorithme basé sur une représentation graphique théorique de la problématique d'implémentation combinée synthèse architecturale - partitionnement temporel. À partir d'une représentation de l'application à mettre en œuvre sous forme d'un graphe flot de données/contrôle [KwNY98, XjZY97], un ensemble de configurations possibles $C = \{C_1, C_2, ..., C_n\}$ caractérisant les allocations des unités fonctionnelles $F = \{F_1, F_2, ..., F_m\}$ spécifiant l'application à implémenter [NZ00], une solution d'implémentation optimisée par synthèse et partitionnement est générée. L'étape d'optimisation des unités partagées est réalisée dans le but de minimiser le temps d'exécution global du GFD tout en considérant une surface totale du système reconfigurable dynamiquement A_{total} et des temps d'exécution des unités fonctionnelles. Le choix du partitionnement temporel ordonnancé s'effectue en vue de minimiser les durées allouées aux phases de reconfiguration du système pris en compte dans la latence globale d'exécution du graphe flot de données [NZ00]. L'approche fournit un graphique temporel de synthèse, spécifiant l'allocation des unités fonctionnelles partagées dans des partitions ordonnées au cours du temps. La figure 2.21 montre un exemple d'un graphique de synthèse identifiant les unités fonctionnelles partagées suivant un partitionnement ordonné.

2.6 Discussion

La principale caractéristique des approches combinées proposées dans la littérature est qu'elles sont basées sur analyse simultanée d'optimisation de la synthèse architecturale et d'un partitionnement temporel. L'avantage d'une telle stratégie est que l'étape d'optimisation par unités fonctionnelles partagées est performante dans la mesure où elle s'effectue en considérant la globalité du graphe flot de données décrivant l'application à implémenter. Cependant, une stratégie de réutilisation des unités fonctionnelles peut conduire à une augmentation des ressources globales par rapport à l'utilisation d'unités fonctionnelles indépendamment. En effet, il peut arriver que la mise en œuvre des éléments logiques auxiliaires permettant un multiplexage dans le temps des unités fonctionnelles soit pénalisante d'une part en terme de ressources supplémentaires (contrainte de ressource), d'autre part en terme de performance. De plus, la complexité du contrôle introduite par la factorisation augmente de façon importante la complexité de recherche d'un partitionnement optimal par ajout de contraintes supplémentaires de découpage et en complexifiant la répartition des unités fonctionnelles dans des partitions à déterminer.

2.7 Conclusion

Nous avons présenté les différentes approches méthodologiques que sont le partitionnement temporel et la synthèse architecturale pour optimiser l'Adéquation Algorithme - Architecture lors d'une implémentation d'une application sous contraintes de traitement ou de conception (temps réel, surface logique, consommation, bande passante, etc.). Nous avons défini la reconfiguration dynamique comme le partitionnement temporel et la synthèse architecturale comme une factorisation d'unités fonctionnelles (unités fonctionnelles partagées). Nous montrons également que la reconfiguration dynamique ne doit pas se substituer aux autres moyens d'optimisation et principalement à la synthèse architecturale. Bien au contraire, il est montré que ces deux approches sont complémentaires dans un objectif d'Adéquation Algorithme - Architecture. En effet, si la reconfiguration dynamique diminue les ressources nécessaires pour l'implémentation d'une application (et donc un gain sur le coût du composant FPGA), elle entraîne une augmentation du temps d'exécution global de l'application en augmentant les phases de reconfiguration et d'intercommunication entre les partitions. D'un autre côté, selon l'application visée, le degré de réutilisation d'unités fonctionnelles par une approche synthèse architecturale peut aussi bien réduire qu'augmenter les ressources d'implémentation sans pour autant diminuer la complexité de la logique de contrôle et de gestion supplémentaire. Le meilleur résultat demeure alors dans un compromis entre les différents apports de la reconfiguration dynamique et de la synthèse architecturale. Dans cet esprit, la littérature présente des méthodes principalement basées sur une analyse simultanée de la synthèse architecturale et de la reconfiguration dynamique au détriment

d'une complexité de mise en œuvre. Une nouvelle approche méthodologique consiste à définir un partitionnement initial, ensuite d'évaluer le degré d'optimisation possible par une synthèse basée sur la réutilisation des unités fonctionnelles entre les partitions. Ainsi, le partitionnement qui en découle ne nécessite pas un ré-ordonnancement des unités fonctionnelles tout en limitant la réduction de performance et en cherchant un compromis sur le nombre de partitions nécessaires à une implémentation. Ceci fait l'objet du travail essentiel présenté dans cette thèse. Une nouvelle approche méthodologique basée sur la synthèse architecturale et le partitionnement temporel est proposée dans ces travaux de thèse et présentée dans le chapitre suivant.

Chapitre 3

Méthodologie de Partitionnement temporel optimisée par Synthèse architecturale

3.1 Introduction

Dans la mise en œuvre de la reconfiguration dynamique, en plus d'un partitionnement spatial, nous avons à effectuer un partitionnement temporel, qui consiste à diviser en plusieurs parties une application à implémenter tout en satisfaisant des contraintes technologiques et de traitement. Les partitions déduites caractérisant le partitionnement spatio-temporel sont donc des fractions du traitement et correspondent chacune à une configuration du circuit. Or, l'utilisation de la reconfiguration dynamique peut conduire à une augmentation des besoins en temps de traitement, en consommation d'énergie, en mémoire temporaire.

Pour une plate-forme cible existante, les ressources matérielles sont fixées. L'application devra donc être capable de se contenter des ressources disponibles, en tentant d'en tirer le meilleur parti. De plus, différentes contraintes sont induites par les partitionnements possibles et l'application visée : la surface logique, la bande passante maximum, les mémoires ou zone de stockage, le temps d'exécution, les temps de reconfiguration, la consommation électrique, etc.. C'est pourquoi, dans une méthodologie du partitionnement temporel combinant une solution de synthèse architecturale, il est important de chercher à minimiser le nombre total de partitions temporelles.

Dans ce chapitre, à partir d'un ensemble de définitions, nous présentons et formulons une nouvelle approche méthodologique de partitionnement temporel optimisée par synthèse architecturale basée sur la réutilisation d'unités fonctionnelles combinées ou partagées. Notre objectif est d'apporter une contribution méthodologique d'implémentation optimisant l'implémentation

(Adéquation Algorithme-Architecture) sur architecture reconfigurable dynamiquement.

3.2 Formulation générale

3.2.1 Partitionnement temporel et contraintes

Le partitionnement temporel d'une application temps réel peut être considéré comme un problème spatio-temporel sous contraintes de temps et d'allocation dynamiques de ressources. Notre approche repose sur une spécification comportementale d'entrée sous forme d'un graphe flot de données (GFD) et d'un graphe flot de contrôle (GFC) modélisant respectivement les chemins de données de traitement (data-path) et de contrôle de l'application à implémenter [Tan01, Ta03]. Les sommets du graphe flot de données dénotent un ensemble d'opérateurs ou de tâches. Les dépendances entre ces opérateurs ou ces tâches sont représentées dans le graphe par des bords ou des arcs entre les sommets. Nous formulons le partitionnement temporel de la manière suivante :

- Un GFD/GFC dénoté par $G(V, E)$ est constitué d'un ensemble de noeuds $V = \{v_1, v_2, ..., v_i\}$ et un ensemble d'arcs $E = \{e_1, ..., e_c\}$ Les noeuds représentent les opérateurs arithmétiques et logiques d'un chemin de données et de contrôle (data-path et control-path). Les arcs représentent les dépendances de données entre les noeuds. Ces opérateurs sont caractérisés par un temps d'exécution Γv_i, une surface logique $Aear(v_i)$ et une taille des données de traitement $Size(v_i)$. Le partitionnement temporel correspond alors, pour une unité de calcul reconfigurable (par exemple une matrice FPGAs), à l'exécution ordonnée d'un ensemble de sous-graphes $\{P_1, P_2, ..., P_n\}$, représentant des partitions temporelles de G tel que :

$$\bigcup_{n=1}^{N} P_n = G \tag{3.1}$$

  où N est le nombre total de partitions. Ces partitions, correspondant à des phases de reconfiguration d'unité de calcul reconfigurable et exécutant des étapes ou des fractions de G. Les partitions sont caractérisées individuellement par un temps d'exécution Γ_{execuP_n} et une surface logique d'implémentation $Area(P_i)$. Parmi les objectifs d'optimisation d'un partitionnement temporel d'un GFD on trouve : la création de partitions homogènes en terme de ressources, l'optimisation de la bande passante entre partitions, la minimisation du temps d'exécution globale, etc.[Ta03, Car03, Bru04].
- Un ou plusieurs noeuds associés peuvent constituer une tâche de traitement parmi l'ensemble T de tâches du graphe G. L'ensemble des tâches de T modélise les unités fonctionnelles de traitement de G. Ces tâches sont caractérisées par leurs fonctions et permettent

d'exécuter l'application spécifiée par G selon un ordonnancement et une allocation spécifiques. On définit une tâche par une combinaison de noeuds exprimée par la relation suivante :

$$\forall m, \quad O_m = \bigcup_k v_{m,k} \quad , \quad k \geq 1 \tag{3.2}$$

où k est le nombre de noeuds du graphe G constituant la tâche O_m tel que :

$$\forall k, \quad v_{m,k} \in V \tag{3.3}$$

Chaque partition P_n est donc constituée et caractérisée par un sous-ensemble de tâches $\{O_{n,1}, O_{n,2}, ..., O_{n,m}, ..., O_{n,Mn}\}$ de l'ensemble T du GFD/GFC de l'application :

$$P_n = \bigcup_{m=1}^{M_n} O_{n,m} \tag{3.4}$$

où M_n est le nombre total de tâches dans la partition considérée P_n. L'ordonnancement des tâches $O_{n,m}$ dans une partition P_n décide de leurs positions spatiales et temporelles dans la partition.

La surface logique de chaque partition P_n peut s'exprimer à partir de la surface des tâches la constituant, elles-mêmes pouvant être exprimées par la surface des noeuds les constituants : :

$$\text{Area}(P_n) = \sum_{m=1}^{M_n} \text{Area}(O_{n,m}) \tag{3.5}$$

$$\text{Area}(O_{n,m}) = \sum_{i=1}^{I_{n,m}} \text{Area}(v_{m,\,i}) \tag{3.6}$$

$Area(O_{n,m})$ est la surface logique d'implémentation de la tâche $O_{n,m}$ de la partition P_n. A partir des équations 3.5 et 3.6, la surface logique de P_n peut également s'exprimer en fonction des surfaces logiques des noeuds v_i présents physiquement dans l'unité de calcul reconfigurable :

$$\forall v_i \in P_n \,, \ \text{Area}(P_n) = \sum_{i=1}^{I_n} \text{Area}(v_i) \tag{3.7}$$

Le temps de traitement global de l'application visée (Γ_{total}) s'exprime par l'égalité suivante :

$$\Gamma_{total} = \sum_{n=1}^{N} \Gamma_{execu P_n} + \sum_{n=1}^{N} \Gamma_{reconf P_n} \tag{3.8}$$

où Γ_{execuP_n} et $\Gamma_{reconfP_n}$ sont respectivement le temps d'exécution et le temps de reconfiguration de la surface logique de la partition P_n. Ce temps global est donc proportionnel au nombre total de partitions N. Si l'on considère N partitions homogènes, en terme de temps d'exécution et de ressources (donc en temps de reconfiguration) [Ta03], ce temps de traitement global devient :

$$\Gamma_{total} = \sum_{n=1}^{N} \Gamma_{execuP_n} + N \cdot \Gamma_{reconf} \tag{3.9}$$

où Γ_{reconf} est le temps de reconfiguration constant de la surface logique d'une partition dans l'unité de calcul reconfigurable .
La mise en œuvre d'un partitionnement temporel par reconfiguration dynamique consiste à exécuter les parties de l'algorithme en intégrant simultanément des conditions de performance ou de temps d'exécution globale, de temps de reconfiguration, de ressources disponibles, de bandes passantes entre les tâches, etc. Ces conditions nous amènent à définir plusieurs contraintes sur le partitionnement :

– **Contrainte surfacique d'implémentation**

Dans le cas d'une approche système, la surface logique de la partition P_n ou la surface logique S_{UCR} de l'unité de calcul reconfigurable dimensionne la surface logique nécessaire pour la mise en œuvre du partitionnement temporel. Chaque partition doit être inférieure ou égale à la surface S_{UCR} de l'unité de calcul reconfigurable. Nous définissons alors une contrainte de surface tel que :

$$\forall n \in [1, N] \ , \ Area(Pn) \leq S_{UCR} \tag{3.10}$$

– **Contrainte du taux d'exploitation surfacique**

En pratique, une synthèse logique sur une cible reconfigurable ne peut exploiter 100% de la surface logique disponible de l'unité reconfigurable. C'est pourquoi, on considère un pourcentage réel d'exploitation d'une surface logique reconfigurable, appelé coefficient d'exploitation surfacique λ, situé entre $80\% - 85\%$. Ce taux d'exploitation surfacique entraîne une contrainte de synthèse et d'implémentation dans l'unité de calcul reconfigurable tel que :

$$\max_{n\in\{1...N\}}(Area(P_n)) = \lambda \cdot S_{UCR} \quad avec \ \lambda < 1 \tag{3.11}$$

où λ est le coefficient d'exploitation surfacique de l'unité reconfigurable.

– **Contrainte temporelle**

Une contrainte de temps de l'application visée impose une contrainte d'exécution totale du partitionnement temporel tel que :

$$\Gamma_{total} \leq \Gamma \tag{3.12}$$

où Γ est la contrainte de temps de l'application. Cette contrainte de temps impose une contrainte d'exécution de chaque partition s'exprimant à partir de 3.8 par la relation suivante :

$$\sum_{n=1}^{N} \Gamma_{execuP_n} \leq \Gamma - \sum_{n=1}^{N} \Gamma_{reconfP_n} \tag{3.13}$$

Dans le cas d'une approche conception, cette contrainte de temps permet le dimensionnement de la surface optimale $Area(P_n)$ [Ta03]. On aboutit alors à la stratégie d'un partitionnement temporel homogène. L'expression de la contrainte temporelle précédente devient alors :

$$\sum_{n=1}^{N} \Gamma_{execuP_n} \leq \Gamma - N \cdot \Gamma_{reconf} \tag{3.14}$$

– **Contrainte de performance**

Le partitionnement temporel s'effectue par phases alternées de calcul/reconfiguration. Les phases de reconfiguration sont des étapes non productives en terme de traitement mais sont inclues dans son temps d'exécution globale. Vis-à-vis d'une contrainte de temps, les phases de reconfiguration sont compensées par des contraintes de performances plus élevées par rapport à une exécution statique (sans reconfiguration dynamique) [Gue97, Tan01, Bru04]. Cette contrainte de temps nécessite l'exécution des partitions avec une performance de traitement supérieure à celle envisagée lors d'une implémentation statique.

Le temps d'exécution d'une partition est caractérisé par le temps de parcours le plus lent parmi les chemins critiques de la partition considérée. Dans l'optique d'atteindre

les meilleures performances, nous considérons une structure pipeline intégrée dans le GFD de l'application à implémenter. Ainsi, les sorties de chaque noeud successif du graphe sont pipelinées par des registres de tailles correspondante à la taille de données des noeuds considérés ($Size(v_i)$).
Le temps de traitement élémentaire (Γv_i) d'un noeud (opérateur) v_i du graphe G correspond au temps de parcours critique du chemin de données entre deux registres de pipeline successifs. L'un est situé parmi les registres de pipeline en entrée du noeud et le second parmi les registres de pipeline situés en sortie du noeud. La fréquence de fonctionnement ou de travail de chaque partition est dépendante du chemin de données élémentaire le plus critique dans la partition. Le temps d'exécution élémentaire le plus critique parmi les noeuds (opérateurs) physiquement présents dans une partition P_n impose la fréquence de travail maximale à la partition considérée :

$$\forall v_i \in P_n \ , \ \mathrm{F}_{P_n} \leq \frac{1}{\max(\Gamma v_i)} \tag{3.15}$$

où F_{P_n} est la fréquence de traitement associée à la partition P_n.

– **Contrainte de latence élémentaire de traitement**

La latence d'exécution d'une donnée dans une partition temporelle correspond au chemin le plus critique (le plus long) parmi tous les chemins de données situés entre l'entrée et la sortie de la partition. Les délais des chemins de données des partitions sont caractérisés et quantifiés par les temps d'exécutions élémentaires des noeuds de la partition. La figure 3.1 illustre la détermination des latences des chemins critiques de données (latence de traitement d'une seule donnée dans la partition) de deux partitions temporelles constituant une application. Dans cet exemple, on identifie respectivement trois et deux chemins de données dans la première et la deuxième partition. Ainsi, la latence de la première partition est de 400 ns et 300 ns pour la deuxième.
Si on considère l'ensemble Φ_n des chemins de données de la partition P_n tel que :

$$\xi_k \in \Phi n, (k \in [1, \ K_n]) \tag{3.16}$$

où K_n le nombre total de chemins de données possibles dans la partition P_n. Le temps d'exécution total d'une partition P_n s'exprime par l'expression suivante :

$$\Gamma_{execP_n} = \max(\Gamma(\xi_k)) + \ (D-1) \cdot \frac{1}{F_{P_n}} \tag{3.17}$$

où $max\Gamma(\xi_k)$ correspond à la latence du chemin critique, c'est-à-dire le temps de parcours du chemin de données ayant la plus grande latence dans la partition P_n. D est le nombre

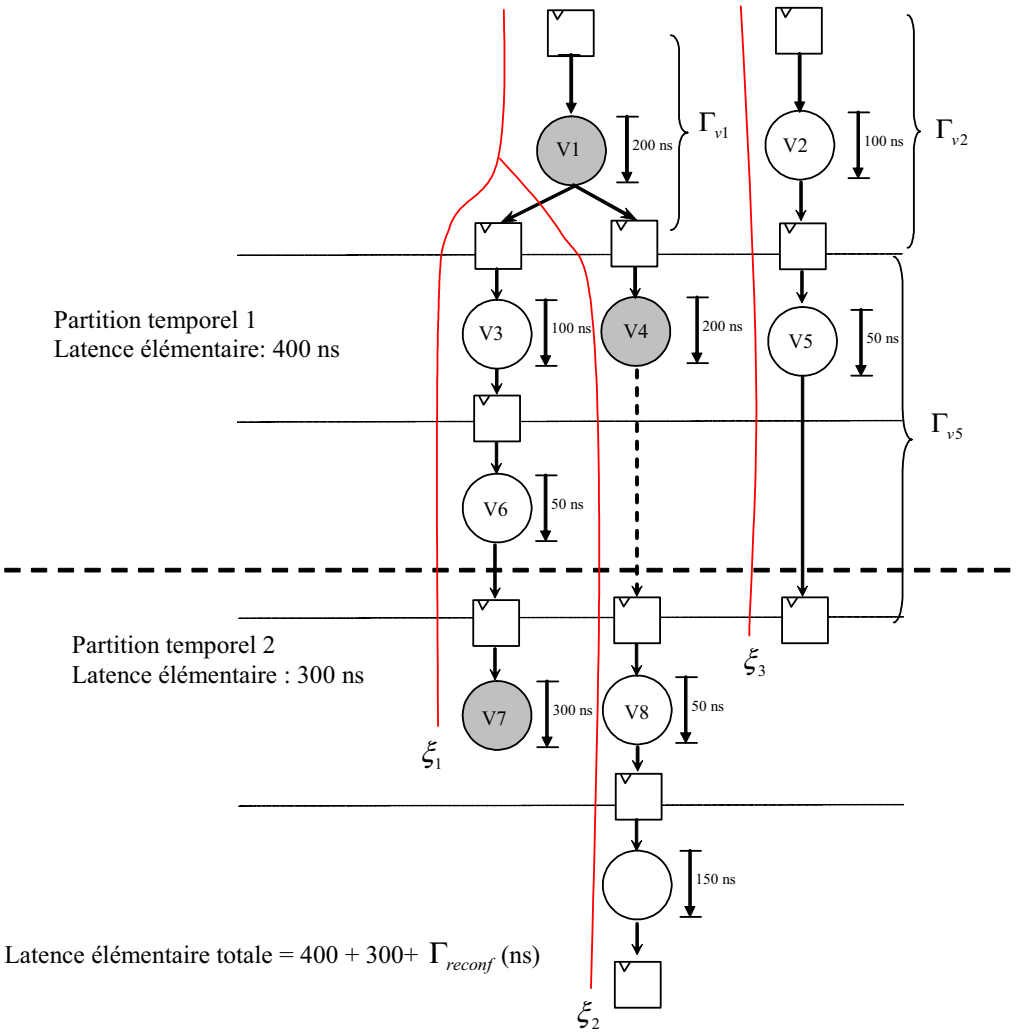

FIG. 3.1 – Estimation de latence élémentaire totale de traitement.

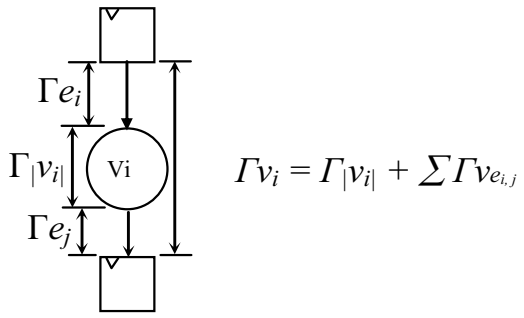

FIG. 3.2 – Estimation du temps de traitement élémentaire d'un noeud v_i.

de données traitées par la partition P_n.
Nous caractérisons le temps de parcours d'un chemin donné ξ_k par les temps de traversé des noeuds et les temps de propagation des données entre les noeuds constituant ξ_k :

$$\Gamma(\xi_k) = \sum \Gamma v_{\xi_k,i} = \sum \Gamma|v_{\xi_k,i}| + \sum_{i,j} \Gamma e_{v_i,v_j} \quad avec \quad v_i, v_j \in \xi_k \tag{3.18}$$

où $\Gamma|v_{\xi_k,i}|$ et $\Gamma e_{v_i,v_j}$ sont respectivement le temps de calcul élémentaire d'un noeud v_i et le temps de propagation lié au routage entre deux noeuds successifs parcourus par le chemin de données ξ_k dans la partition P_n. La figure 3.2 illustre la modélisation du temps d'exécution élémentaire d'un noeud parcouru dans le chemin critique ξ_k.
En pratique, la fréquence de travail associée à une partition est déterminée par la condition 3.15. La latence du chemin critique correspond alors au prologue pipeline de la partition et est exprimée à partir de l'équation suivante :

$$max(\Gamma(\xi_k)) = \frac{k}{F_{P_n}} \tag{3.19}$$

où k est le nombre d'étages pipeline dans la partition P_n considérée.

– **Contrainte de consommation d'énergie**

La consommation électrique dynamique d'une unité de calcul reconfigurable est proportionnelle à son activité, sa capacité, sa fréquence de travail et son alimentation électrique [Gar00, Poo02, LSB02]. Cette puissance dynamique électrique consommée peut être exprimée de façon proportionnelle en fonction du nombre d'opérateurs et de la fréquence de fonctionnement ou de travail de ces opérateurs [Abe06]. Par analogie, nous pouvons exprimer la puissance dynamique P_{dn} consommée par une partition P_n par la relation suivante :

$$P_{dn} = Area(P_n) \cdot [F_{P_n}]^3 \tag{3.20}$$

L'augmentation de la puissance de calcul peut être obtenue indifféremment en augmentant le nombre d'opérateurs (parallélisme de traitement) et/ou la fréquence de traitement. D'un point de vue énergétique, à puissance de calcul identique, il est préférable d'augmenter le nombre d'opérateurs plutôt que la fréquence de traitement [Abe06]. Cela justifie les bonnes performances énergétiques des structures parallèles par rapport à une solution optimisée par synthèse architecturale. En effet, lors d'une implémentation optimisée par synthèse architecturale, la consommation électrique de l'unité de contrôle peut être aussi importante que celle des unités de calcul [Abe06], entraînant alors un surcoût en terme de consommation électrique par rapport à une solution à structures d'opérateurs parallèles. De même, d'un point de vue énergétique et à puissance de calcul identique, il est préférable d'augmenter le nombre d'opérateurs plutôt que la fréquence de traitement dans le cas d'un partitionnement temporel, dans la mesure ou le nombre d'opérateurs ne se répercute pas fortement sur la puissance électrique. Cependant, les phases de reconfiguration sont très pénalisantes et entraînent un surcoût en terme de consommation par rapport à une implémentation statique. Dans ce contexte, une minimisation de la puissance électrique se traduit par une minimisation du nombre de reconfigurations nécessaires, tandis que dans le cas d'une synthèse architecturale, une minimisation d'un surcoût en terme de consommation électrique se traduira par une minimisation de l'unité de contrôle proportionnelle au degré de synthèse. La recherche d'un optimum de consommation lors d'un partitionnement temporel se traduit alors par une minimisation du nombre de partitions tout en minimisant les unités de contrôle.

– **Contrainte d'unicité**

Une contrainte d'unicité caractérise chaque tâche du GFD/GFC. En effet, les tâches doivent être identifiées dans une seule et unique partition lors du partitionnement temporel. Par conséquent, les partitions ne peuvent pas se recouvrir pour réaliser le GFD/GFC. La contrainte d'unicité s'exprime alors par les relations suivantes :

$$P_n \cap P_{n+1} = \emptyset \tag{3.21}$$

$$\forall\ O_m \in T,\ \ [O_m \subset P_n]\ \&\ \ [O_m \not\subset (G - P_n)] \quad (1 \leq n \leq N) \tag{3.22}$$

– **Contrainte d'ordonnancement**

Le partitionnement temporel met en œuvre l'exécution dans le temps de l'ensemble des tâches du GFD/GFC avec un ordonnancement déterminé. L'ordonnancement dans chaque partition est caractérisé par le respect d'exécution des tâches de l'ensemble des chemins de données de la partition considérée, selon un ordre défini par la spécification comportementale de l'application. L'ordonnancement définit alors une liste de tâches classées par ordre d'exécution parmi l'ensemble des tâches des chemins de données Φ_n de la partition P_n. En général, l'établissement de la liste d'ordonnancement des tâches est établi selon un algorithme de type ASAP (*As Soon As Possible*) ou ALAP (*As Late As Possible*) [Ka99]. Nous définissons alors l'ordonnancement par :

$$ASAP(\Phi_n) \quad ALAP(\Phi_n) \tag{3.23}$$

avec

$$\Phi n = \{\xi_k, k \in [1..K_n]\} \tag{3.24}$$

et

$$\xi_k = \sum O_{\xi_k,m} \quad , \quad O_{\xi_k,m} \in \xi_k \tag{3.25}$$

– **Contrainte de dépendances de données**

Les dépendances de données spécifiées par un ordonnancement des tâches des chemins de données d'une partition doivent être respectées. Chacune des tâches spécifiées par l'ordonnancement peut être visualisée comme une ou plusieurs unités fonctionnelles modélisées par un ou plusieurs noeuds ou opérateurs pouvant être indissociables dans une partition temporelle. Plus précisément, deux tâches O_i et O_j, dépendantes dans le temps l'une de l'autre, doivent alors être placées soit dans une même partition, soit dans deux partitions successives. Nous avons donc une contrainte de dépendance de données exprimée de la manière suivante :

$$\forall O_j, \forall O_i \rightarrow tO_j \tag{3.26}$$

$$\forall O_i \in P_i, O_j \in P_j \quad avec \quad 1 \leq i \leq j \leq N \tag{3.27}$$

Où P_i et P_j sont respectivement les partitions qui incluent les tâches O_i et O_j.

– **Contrainte sur la bande passante et la taille mémoire**

Lors d'un partitionnement temporel, les données transferées entre les tâches de deux partitions successives correspondent aux données intermédiaires entres partitions. Ces dernières caractérisent une contrainte en terme de bande passante (débit et taille du bus inter-partition) et de quantité de données inter-partitions de stockage nécessaire à la mise en œuvre du partitionnement. Une bande passante est liée à la capacité de transmission entre l'ensemble des tâches de deux partitions successives ayant des dépendances de données. Nous avons alors les conditions de dépendances de données suivantes :

$$\forall O_j, \forall O_i \rightarrow O_j \tag{3.28}$$

$$\forall O_i \in P_i, O_j \in P_j \quad avec \quad i \leq j \tag{3.29}$$

Cette contrainte spécifie à la fois les ressources de mémorisation (bande passante et taille de la zone de stockage) et le temps communication ou latence inter-partition avant une nouvelle phase de reconfiguration. Ce temps de communication est proportionnel à la quantité de données à transmettre entre deux partitions successives. Dans le cas d'une approche système, c'est-à-dire une bande passante et une taille de stockage fixées, le partitionnement temporel est contraint tel que :

– D'une part la quantité mémoire inter-partitions ($M_{Pi,\ Pi+1}$) n'excède pas l'espace de stockage autorisé :

$$\forall\, i \ \in \ N\,, \quad M_{Pi,\ Pi+1} \leq M_{ucr} \tag{3.30}$$

Où M_{ucr} est la quantité de mémorisation maximume disponible par l'unité de calcul reconfigurable. Pour une implantation sur cible FPGA, la zone temporaire de stockage peut correspondre soit à une mémoire externe ou embarqués dans le FPGA.

– D'autre part, les communications entre deux partitions successives respectent la bande passante autorisée, définie par l'expression suivante :

$$\forall O_m \leftrightarrow O_n\,,\, O_n \in P_n\,,\, O_m \in P_{n+1} \tag{3.31}$$

$$\text{Bandwidth}(\text{P}_{\text{n,n+1}}) = \sum_{\text{m=1,n=1}}^{m=N_n, n=N_{n+1}} \text{Bandwidth}(\text{O}_\text{n}, \text{O}_\text{m}) \ \leq \ BP_{ucr} \tag{3.32}$$

Où BP_{ucr} est la bande passante disponible par unité de calcul reconfigurable. Le terme $O_m \leftrightarrow O_n$ signifie que la tâche O_m et la tâche O_n ont une dépendance de données et sont indépendamment placées respectivement dans les partitions successives P_n

et P_{n+1}. L'expression $Bandwidth(O_n, O_m)$ correspondant à la bande passante entre les tâches O_n et O_m, est également fonction des tailles de communication entre les opérateurs réalisant mutuellement ces tâches :

$$\mathrm{Bandwidth(O_n, O_m)} = \sum \mathrm{Bandwidth(v_i, v_j)} \tag{3.33}$$

et

$$\forall v_i \leftrightarrow v_j \, , \, v_i \in P_n \, , \, v_j \in P_{n+1} \tag{3.34}$$

avec

$$\mathrm{Bandwidth(v_i, v_j)} = \sum size(e_{i,j}) \tag{3.35}$$

Où $size(e_{i,j})$ est la taille des données transférées entre les noeuds interdépendants v_i et v_j.

Il est à noter que cette contrainte de bande passante peut être un critère d'optimisation lors d'un partitionnement temporel [Bru04, Liu04].

3.2.2 Synthèse architecturale basée sur unités fonctionnelles partagées et contraintes

Chaque graphe GFD/GFC caractérisant une application peut être optimisé par synthèse architecturale basée unités fonctionnelles partagées des tâches de traitement des constituants. Dans ce contexte, on distingue deux sous-ensembles de tâches U et F dans un GFD/GFC tel que :

- Le sous-ensemble U de tâches de traitement à instanciation unique O_u. Ce sous-ensemble caractérise des opérateurs indépendants $\{O_1, O_2, ..., O_u, ..., O_U\}$, où O_u est une tâche non-factorisable et donc ne pouvant être mutualisée ou réutilisée. Les opérateurs constituant le sous-ensemble U sont des tâches non-réductibles. Concrètement, toutes les opérations appartenant à U sont de nature fonctionnelle différente et ne permettent pas une optimisation en terme de ressources par une approche de factorisation ou réutilisation de tâches communes.
- Le sous-ensemble $F = \{O_1, O_2, ..., O_m, ..., O_M\}$, de tâches à instanciation multiple et donc factorisables (O_m) et optimisables, vis-à-vis d'une synthèse architecturale par unités fonctionnelles partagées, en un nombre réduit de tâches. Concrètement, l'ensemble des tâches de F peut être remplacé par une seule tâche partagée par une réutilisation. Cette factorisation peut être illustrée par l'exemple simple suivant. On considère quatre modules connectés en cascade et exécutant une même fonction opérative sur des données différentes (figure 3.3). La surface logique à l'implantation de chaque module est $Area(i)$.

Une mise en œuvre possible d'une approche réutilisation d'opérateur est d'intégrer une logique d'aiguillage et de contrôle associée à de la mémoire. Le but de cette approche est de

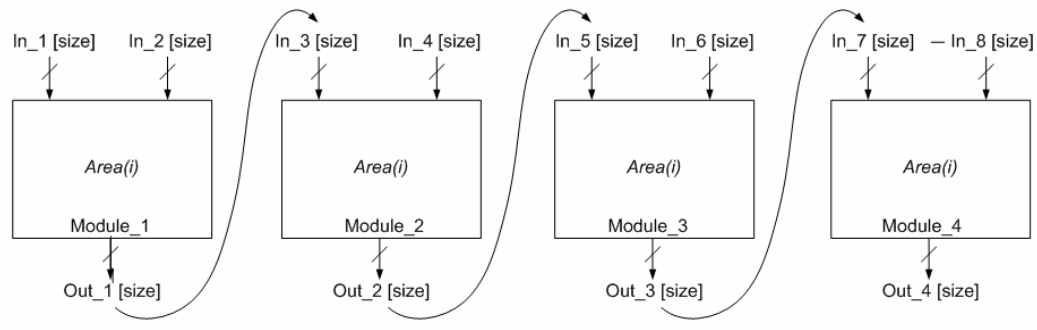

FIG. 3.3 – Modules à instanciation multiple connectés en cascade.

multiplexer temporellement sur au mieux un seul opérateur pour traiter les différentes données à des instants différents successifs. Le principe consiste donc à supprimer les opérations similaires jusqu'à obtenir la plus faible occurrence possible (de valeur 1) de la tâche opérative considérée en les substituant par des ressources logiques d'aiguillage, de contrôle et de stockage supplémentaires. Dans l'exemple considéré, l'architecture 4 modules montés en cascade a été remplacée par une nouvelle architecture constituée d'un seul module associé à deux multiplexeurs (figure 3.4).

La viabilité de l'approche doit être considérée selon le facteur de gain possible en terme de ressources ou surface logique. Dans notre exemple, par comparaison, un intérêt est de s'assurer que la surface logique de la nouvelle architecture exprimée par la somme suivante $Area(Controleur)+Area(Multiplexeur2)+Area(Multiplexeur4)+Area(i)$, reste inférieure au gain en surface correspondant au nombre de réductions de modules mutualisés $3 \times Area(i)$. Ce gain est d'autant plus fort, que les tâches factorisées sont soit de type " gros grains " (dans ce cas, on considère que les ressources de la logique de mise en œuvre de la mutualisation sont beaucoup plus faibles devant celles d'un module ($Area(Controleur)+Area(Multiplexeur2)+Area(Multiplexeur4) \ll Area(i)$) ou bien que le nombre de réduction par factorisation est important.

Le sous-ensemble F rassemble donc les tâches dont la périodicité d'apparition de leurs fonctionnalités (occurrence) est supérieure ou égale à deux, tel que :

$$F = \bigcup_{m=1}^{M} O_m^{q_m} \quad avec \quad (q_m \geq 2) \tag{3.36}$$

où quelque soit m, $O_m^{q_m}$ sont des tâches indépendantes entre-elles. q_m représente le nombre d'occurrence de la tâche O_m dans le GFD et définit la condition d'appartenance aux sous-ensembles réductible F et non-réductible U :

$$q_m \geq 2\,, \quad O_m^{q_m} \in F \tag{3.37}$$

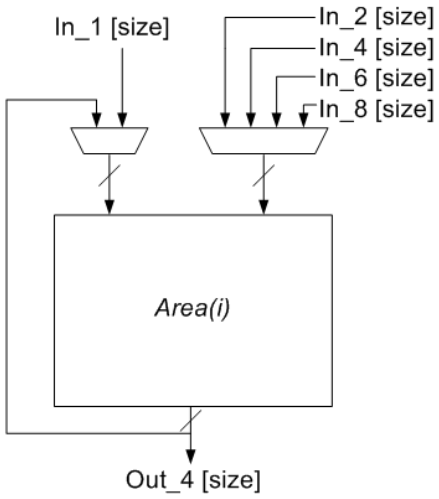

FIG. 3.4 – Illustration d'une logique d'unités fonctionnelles partagées par factorisation.

$$q_m = 1\,, \quad O_m^{q_m} \in U \tag{3.38}$$

On peut alors exprimer tout graphe flot de données et de contrôle d'une application à partir des tâches d'opérateurs le constituant par l'expression suivante :

$$G = \bigcup_{u=1}^{U_G} O_u \cup \bigcup_{m=1}^{M_G} O_m^{q_m} \quad avec \quad \forall m, \quad (q_m \geq 2) \tag{3.39}$$

où M_G et U_G sont respectivement le nombre total de tâches réductibles et non-réductibles de G. A partir des expressions 3.1 et 3.39, les partitions temporelles d'une application G en N partitions identifiables, peuvent être modélisées par leurs sous-ensembles de tâches U et F par l'expression suivante :

$$G = \bigcup_{n=1}^{N} (\bigcup_{u=1}^{U_n} O_{n,u} \cup \bigcup_{m=1}^{M_n} O_{n,M}^{q_m}) \quad avec \quad (q_m \geq 2) \tag{3.40}$$

où M_n et U_n sont respectivement le nombre de tâches à instanciation unique non-réductible ($O_{n,u}$) et multiple et factorisé ($O_{n,m}$) dans une partition P_n considérée. La mise en œuvre d'une factorisation des tâches ou unités fonctionnelles du sous-ensemble à instanciation multiple F conduit un nouveau sous-ensemble de tâches réduites F' constitué de tâches majoritairement indépendantes dont l'occurrence a été réduite dans la mesure du possible vers 1. On représente l'exécution d'une telle synthèse architecturale sur toute partition P_n par l'expression suivante :

$$AS(P_n) = U_n \cup F'_n \tag{3.41}$$

Où $AS()$ représente l'exécution de la synthèse architecturale basée sur la réutilisation de tâches communes et $AS(P_n)$ représente la partition réduite incluant les tâches à instanciation unique, les tâches réduites ainsi que les ressources de mise en œuvre de la factorisation possible des tâches à instanciation multiple.

– **Contraintes de l'association partitionnement temporel et synthèse architecturale**

La logique de contrôle et d'aiguillage nécessaire à la mise en œuvre de la factorisation doit respecter également les contraintes exposées dans la sous-section précédente § 3.2.1. Ainsi, la contrainte surfacique (équations 3.10 et 3.11) impose un dimensionnement de chaque partition inférieure ou égale à la surface logique de l'unité de calcul reconfigurable S_{UCR} :

$$\forall P_n, \quad \lambda \cdot S_{ucr} \geq max(Area(U_n \cup F'_n)) \tag{3.42}$$

3.2.3 Optimisation d'un partitionnement temporel par Synthèse Inter-partition

3.2.3.1 Introduction

Dans une méthodologie du partitionnement temporel combinant une solution de synthèse architecturale par réutilisation d'unités fonctionnelles, il est important de minimiser le nombre total de partitions temporelles. Une technique appropriée permettant d'optimiser un partitionnement temporel consiste à appliquer la synthèse architecturale entre deux partitions successives afin de minimiser le nombre de partitions tout en recherchant un compromis de réduction entre la surface logique et le temps d'exécution total. Nous proposons de développer cette approche que nous considérons comme une optimisation par synthèse inter-partition.

3.2.3.2 Définitions

Nous caractérisons la notion de synthèse inter-partition d'une conception partitionnée temporellement à l'aide des définitions ci-dessous :

Définition 1 *On appelle " partitions initiales ou originales ", les partitions obtenues par un algorithme de partitionnement temporel initialement exécuté sur une application décrite sous forme de graphe flot de données et de contrôle (GFD/GFC) sans aucun raffinement.*

La définition 1 considère l'algorithme de partitionnement temporel dans un sens général, c'est-à-dire que le ou les critères du partitionnement temporel est à l'initiative du concepteur selon des méthodes existantes ou envisagées.

Définition 2 *Une synthèse architecturale basée sur une réutilisation de tâches communes, appliquée sur un sous-graphe GFD/GFC correspondant à deux partitions successives (séquentielles) suivant un ordonnancement défini lors d'un partitionnement temporal initial est appelée Synthèse Inter-Partition (Inter-Partition Synthesis :IPS).*

Définition 3 *Les partitions résultantes d'une synthèse inter-partition d'un partitionnement temporel initial sont appelées " partitions réduites ".*

3.2.3.3 Formalisation : objectifs et conditions d'une synthèse inter-partition

Une synthèse architecturale inter-partition s'applique sur un partitionnement initial dans le but de réduire le nombre de partitions. L'objectif est d'optimiser le temps global d'exécution tout en recherchant une meilleure adéquation entre le temps d'exécution et les ressources matérielles exploitées.

Le résultat d'une synthèse inter-partition est un nouveau partitionnement temporel composé de N_{redu} partitions et tel que $N_{redu} \leq N$. Les partitions réduites définissent alors un partitionnement temporel optimisé. La synthèse architecturale inter-partition s'apparente à une factorisation de tâches similaires ou à instanciation multiple ou commune entre deux partitions telle que :

$$O_m^{q_m} \in F_i \, , \, F_i = F_n \cup F_{n+1} \tag{3.43}$$

où F_n et F_{n+1} sont les sous-ensembles des tâches à instanciation multiple de deux partitions successives d'un partitionnement initial. On exprime par $AS(P_n, P_{n+1})$ la modélisation de la synthèse inter-partition entre les deux partitions successives P_n et P_{n+1} d'un partitionnement temporel initial et P'_n est la partition réduite résultante. La validité ou la considération de cette dernière est conditionnée par la surface logique pour l'implantation de la partition $P'_n = AS(P_n, P_{n+1})$. Elle doit être inférieure ou égale à la surface logique de l'unité de calcul reconfigurable. C'est-à-dire à partir de la 3.42 :

$$Area(p'_n) = Area(U_{n,n+1} \cup F'_{n,n+1}) \leq \lambda \cdot S_{ucr} \quad avec \quad U_{n,n+1} = U_n \cup U_{n+1} \tag{3.44}$$

Le partitionnement temporel réduit est alors composé de partitions réduites et de partitions initiales pour lesquelles l'application IPS ne respecte pas la condition exprimée par l'expression 3.44.

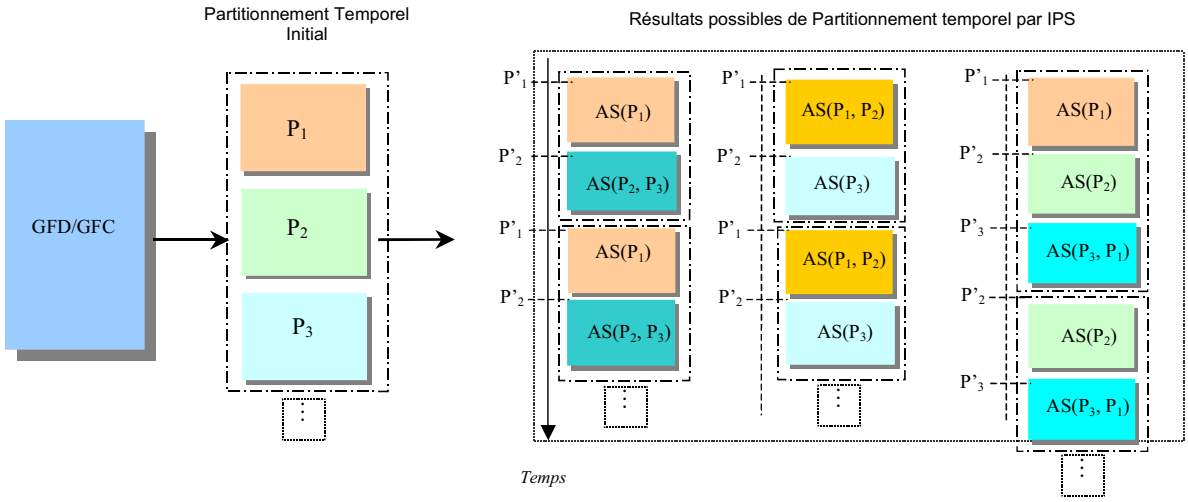

FIG. 3.5 – Illustration générale de la synthèse inter-partition.

Les partitions initiales non considérées par une optimisation IPS peuvent être également optimisées individuellement par une synthèse architecturale ($AS(P_n)$). Ainsi, le partitionnement temporel réduit peut alors être considéré de la manière suivante :

$$P'_n = \begin{cases} AS(P_n) \\ \quad et/ou \qquad (n \in N,\, n' \in N_{redu},\, N \geq N_{redu}) \\ AS(P_n, P_{n+1}) \end{cases} \tag{3.45}$$

où N et N_{redu} sont respectivement le nombre total de partitions initiales et réduites.

La figure 3.5 illustre la stratégie de raffinement et d'optimisation d'un partitionnement temporel initial par synthèse inter-partition . A partir d'un ensemble de N partitions originales $P_1, P_2, ..., P_n$, l'optimisation par IPS aboutit à un nouveau partitionnement temporel composé de partitions réduites définies par l'expression 3.45.

Le partitionnement temporel final doit également respecter les principales contraintes du partitionnement temporel initial :

– L'ensemble des partitions réduites doivent réaliser l'application G telle que :

$$G = \bigcup_{n'=1}^{N_{redu}} P_{n'} \tag{3.46}$$

– Respecter la contrainte de temps de traitement initialement spécifiée par les équations (3.10) et (3.12).

$$N_{redu} \cdot \sum_{n'} \Gamma_{P'_n} + N_{redu} \cdot \Gamma_{reconf} \leq \Gamma \tag{3.47}$$

– Respecter la contrainte surfacique dans le cas d'une approche développement de système :

$$\forall n' \in N_{redu}, \quad Area(P'_{n'}) \leq \lambda \cdot S_{ucr} \tag{3.48}$$

De plus, si on considère les liens existants entre la consommation électrique aux paramètres caractérisant un partitionnement (équations 3.20, 3.9 et 3.28), la réduction des phases de reconfiguration par minimisation du partitionnement initial justifie également le choix d'une optimisation de la reconfiguration dynamique par une approche synthèse architecturale en terme de réduction de consommation électrique. Cependant, une optimisation d'un partitionnement par IPS ne doit pas pénaliser la consommation électrique par rapport à un partitionnement initial. En effet, si une minimisation du nombre de partitions permet un éventuel gain de consommation électrique (réduction des phases de reconfiguration gourmandes en consommation électrique), elle peut néanmoins nuire à la consommation électrique par augmentation de l'unité de contrôle.

3.2.3.4 Considérations de mise en œuvre d'une synthèse inter-partition

Une stratégie basée sur la synthèse inter-partition nécessite la prise en compte de certaines considérations pratiques à mettre en œuvre dont on propose des solutions de résolution.

1. Pour un ensemble de partitions originales, une évaluation du degré d'efficacité d'une synthèse inter-partition par factorisation ou mutualisation de tâches ou des opérateurs sur des partitions originales successives doit être déterminée. Une solution consiste alors à définir un critère permettant une telle évaluation en vue d'obtenir une quantification possible de l'optimisation d'un partitionnement initial par une solution de synthèse architecturale IPS.
2. L'établissement d'un critère d'évaluation du degré d'efficacité d'une synthèse IPS, nécessite le développement d'un procédé d'identification de l'ensemble des tâches factorisables du graphe.

C'est pourquoi, nous proposons un critère prenant en compte l'aspect de la réutilisation des unités fonctionnelles et un algorithme permettant d'identifier les unités fonctionnelles à instanciation multiple et unique. L'identification des sous-ensembles U et F permettra de caractériser et identifier les partitions initiales dont l'application d'une synthèse inter-parttion parait judicieux en permettant d'atteindre une optimisation selon les critères définis en § 3.2.3.3. C'est l'objet des paragraphes suivants.

3.2.4 Le taux de ressemblance mutuelle

L'approche basée sur la réutilisation de l'unité fonctionnelle n'est pas une méthode générale. Son efficacité dépend de la structure et des opérateurs caractérisant l'application à implémenter. D'une manière générale, plus les éléments élémentaires de traitement organisant la mise en

œuvre de l'application sont de nature identique ou " ressemblante ", plus les conditions d'une synthèse par factorisation sont réunies. Le problème est d'évaluer la possibilité de la synthèse sur les différentes partitions d'une application. La caractérisation d'une synthèse inter-partition est importante car elle nous informe si une optimisation basée sur la réutilisation de l'unité est une solution appropriée dans l'approche proposée. Pour la conception RSoC, l'approche de réutilisation d'unités fonctionnelles (UFs) peut être une solution efficace pour les tâches synthétisables à instanciation multiple [Ha06].

Nous proposons un indicateur empirique décrivant le taux de ressemblance ou bien inversement d'exclusion des opérateurs ou fonctions entre les partitions successives d'un GFDC. On parle alors du ***T.R.M : le taux de ressemblance mutuelle ou inversement du T.E.M : le taux d'exclusion mutuelle.*** La figure 3.6 illustre un exemple de notre évaluation du taux de ressemblance entre deux partitions successives. On considère ici deux partitions successives A et B de tailles respectives S_A, S_B et contenant des tâches ou des unités fonctionnelles à instanciation multiple simultanément présentes dans les deux partitions. La taille d'une partition correspond au nombre total d'opérateurs ou de noeuds. On considère que dans la partition A, les tâches à instanciation multiple ($O_m^{q_m} \in A \cap B$, $q_m > 1$) présentent dans le sous-ensemble F_A représentent $37,5\%$ de l'ensemble des tâches formant la partition ($U_A \cup F_A$). Par exemple, la partition A contient 3 instanciations de tâches multiples composées au total de 6 noeuds parmi 16 noeuds contenus dans la partition A. De même, nous considérons par exemple que le sous-ensemble F_B de la partition B, est constitué de 5 tâches similaires à instanciation multiple, par rapport au sous-ensemble F_A, composées de 10 noeuds parmi 20 et représentant donc 50% des tâches de la partition $B(U_B \cup F_B)$. Afin d'obtenir une juste évaluation du taux de ressemblance entre les partitions A et B, une normalisation de la taille des partitions par la différence du nombre total de noeuds entre ces partitions (figure 3.6), doit être réalisée. Cette normalisation est mise en œuvre par remplissage d'UFs dans la partition A constituées de noeuds " vides " jusqu'à égaler la taille de la partition en terme de nombre de noeud contenu dans plus grande partition B. Cette normalisation permet donc une comparaison des parties " ressemblantes " ou " similaires " entre les partitions A et B. L'évaluation de la ressemblance au niveau UFs entre les deux partitions A et B, peut être alors exprimée par l'expression suivante :

$$\frac{0.375 \times S_A}{S_B} \tag{3.49}$$

Soit dans notre exemple un taux de ressemblance équivalent à 30%.

Cet indicateur est un paramètre permettant de caractériser en globalité les opérateurs ou tâches dans les partitions considérées de l'application à traiter. Nous définissons ce paramètre à partir du nombre et des types d'opérateurs ou fonctionnalités mis en œuvre dans les partitions analysées. Pour formaliser le **T.R.M**, nous supposons qu'il existe deux partitions successives P_n

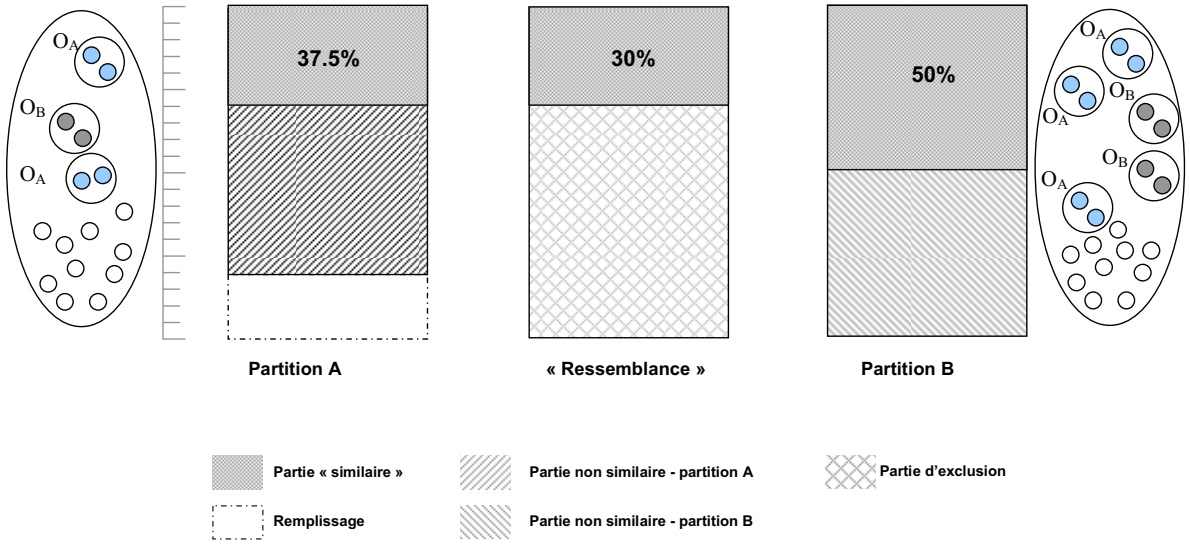

FIG. 3.6 – L'évaluation pour décrire la caractéristique similaire.

et $P_{n+1} \in G$. Dans ces partitions, les éléments de plus bas niveau correspondent aux opérateurs logiques et arithmétiques. Les nombres totaux de ces opérateurs dans les partitions n et $n+1$ sont notés respectivement I_n et I_{n+1}.

Plusieurs opérateurs peuvent être regroupés dans le but de les considérer comme des tâches ou des unités fonctionnelles mettant en œuvre une fonction de calcul. Pour les différents types d'UF, nous avons un nombre différent d'opérateurs bas niveau la constituant. La taille d'une unité fonctionnelle dépend du nombre d'opérateurs composés réalisant son type et sa fonction de calcul ainsi que ses liens de dépendances avec son voisinage. Dans le cas où une unité fonctionnelle ne contient qu'un seul opérateur, cet opérateur présente lui-même un type et une fonction de calcul. Le taux de ressemblance mutuelle entre les partitions P_n et P_{n+1} est calculé à partir de l'équation suivante :

$$T.R.M. = \frac{min[\sum_{m=1}^{M_n}(M_{n,m} \cdot I_{n,m}), \sum_{m=1}^{M_n+1}(M_{n+1,m} \cdot I_{n+1,m})]}{max(I_n, I_{n+1})} \tag{3.50}$$

où $M_{n,m}$ est le nombre d'unités fonctionnelles à instanciation multiple prioritaire dans la partition n ; m classifie et identifie le type de fonction exécutée par les UFs communes dans les partitions successives n et $n+1$. I_n et $I_{n,m}$ sont respectivement le nombre total d'opérateurs et le nombre d'opérateurs constituant la fonction de calcul de type m dans la partition n.

La valeur de ce taux en pourcentage permet de décrire le degré de ressemblance entre deux partitions différentes. Dans un flot de conception méthodologie de partitionnement temporel, l'évaluation du **T.R.M.** préalable permet d'estimer la possibilité d'une solution intégrant une

synthèse architecturale. La valeur du **T.R.M.** augmente proportionnellement avec la ressemblance. Dans ce cas, une synthèse inter-partition apparaît judicieuse et les deux partitions associées peuvent alors être fusionnées par une réutilisation des unités fonctionnelles pour former une seule partition.

La difficulté d'utilisation de ce taux de ressemblance réside sur la définition d'un seuil de ressemblance permettant d'aboutir à une solution optimisée appropriée respectant la condition exprimée par l'équation 3.44. Selon une comparaison, à un seuil **T.R.M.** prédéfini, une solution d'optimisation basée synthèse architecturale est considérée ou non appropriée. De plus, cet indicateur montre une efficacité en terme de résultat de synthèse sous certaines conditions à respecter à la fois au niveau :

- de la granularité des unités fonctionnelles ou tâches à instanciation multiple présentes dans les partitions considérées par une évaluation du **T.R.M.**
- de l'identification et l'extraction des UFs à partir des noeuds simultanément présents dans les partitions I_n et I_{n+1}.

En effet, selon les différents niveaux de granularité d'unités fonctionnelles considérées, les résultats d'une synthèse peuvent être différents. Si on considère un niveau UF de type de " grain fin ", la phase d'identification et d'extraction des tâches peut fournir un plus grand nombre UFs à instanciation multiple. Cependant, le niveau d'optimisation par synthèse peut ne pas être efficace, car en général, une granularité de plus en plus fine nécessite une complexité et des ressources supplémentaires proportionnées. Plus précisément, il peut avoir une consommation plus importante en terme de ressources supplémentaires mettant en œuvre la logique d'aiguillages et de contrôle réalisant la mutualisation des UFs. En revanche, si on considère un niveau de granularité élevé UF de type de " gros grain ", le processus de synthèse est plus efficace en terme d'optimisation des ressources supplémentaires réalisant le procédé de factorisation des UFs au détriment d'une réduction du degré de factorisation et d'une plus grande complexité d'extraction et d'identification des UFs, car, une granularité élevée réduit le dégré de factorisation et nécessite moins de ressources supplémentaires de contrôle.

A partir de ces considérations, la détermination du **T.R.M.** doit s'exécuter en priorité sur les tâches les plus nombreuses de plus haut niveau de granularité. Ensuite, il faut définir un compromis entre le niveau de granularité en terme du noeud définissant une tâche et son nombre d'instanciations multiples. Il s'agit alors de rechercher la limite d'efficacité d'une synthèse par unités fonctionnelles partagées entre un faible nombre de tâches " gros grain " et un nombre plus élevé d'UFs " grain fin " pouvant être constituées par les mêmes noeuds du graphe G.

Dans ce cadre, on définit des règles de priorité d'élaboration des unités fonctionnelles possibles constituées de plusieurs opérateurs (noeuds) selon un ordre de plus haute priorité de sélection d'un noeud vis-à-vis d'une élaboration d'une UF. D'une manière générale, les règles de priorité sont établies selon l'ordre d'importance suivant : plus haute priorité à la taille d'une

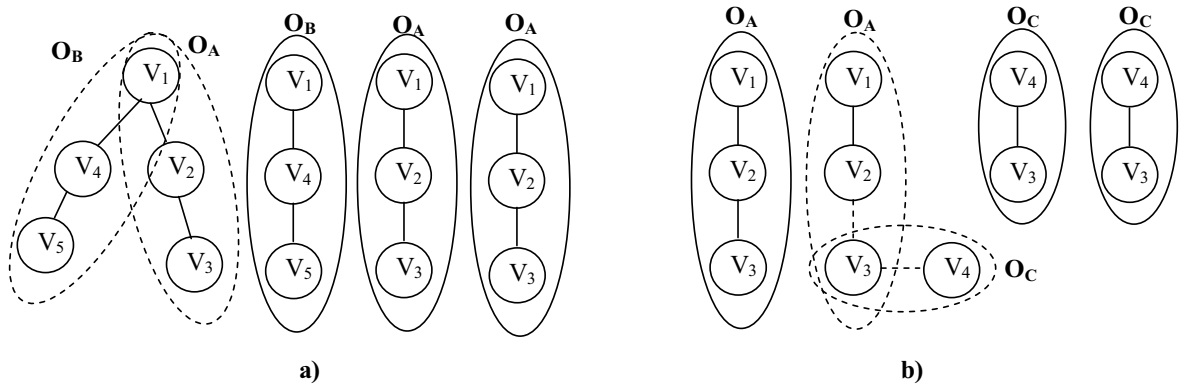

FIG. 3.7 – Règles prioritaires de formation des UFs.

unité fonctionnelle, puis au nombre d'instanciations d'une unité fonctionnelle, ensuite à l'ordonnancement des noeuds ou opérateurs. La figure 3.6, illustre les deux règles prioritaires de formation d'unités fonctionnelles vis-à-vis des opérateurs ou noeuds indépendants :

Règle 1 *Pour un opérateur ou noeud v_i présentant plusieurs possibilités d'association à la formation de plusieurs types d'UFs à instanciation multiple, la priorité est donnée au type d'UFs de plus grosse granularité (UF - " gros-grain "). Par exemple, nous avons une possibilité d'association pour former trois type d'UFs A, B ou C. Si on suppose que la taille de l'unité A est supérieure à celle de l'unité B, et que celle de l'unité B est supérieure à elle de l'unité C, alors l'unité A bénéficie d'une plus haute priorité à sa formation par rapport aux autres types d'UFs vis à vis d'un opérateur indépendant v_i (figure 3.7.b).*

Règle 2 *Dans le cas d'une association possible d'un noeud indépendant v_i à deux UFs de taille identique, la priorité sera donnée à l'UFs avec le nombre d'instanciations le plus élevé (figure 3.7.a)*

Règle 3 *La formation des UFs doit s'effectuer selon l'ordonnancement temporel et les dépendances de données entre les noeuds.*

L'exécution du **T.R.M.** nécessite donc l'identification et l'extraction selon des règles de formation de tâches ou UFs à instanciation multiple et unique dans le GFD visé (sous-ensembles U et F). L'objectif est de trouver les tâches identiques gros-grain (définition de la taille maximum) des unités fonctionnelles permettant une approche de synthèse par réutilisation efficace pour une optimisation. Pour cela, nous proposons un algorithme d'identification des unités fonctionnelles respectant les conditions d'une utilisation efficace du **T.R.M.** C'est l'objet du paragraphe suivant.

3.2.5 Algorithme d'identification et d'extraction d'unités fonctionnelles factorisables

Une unité fonctionnelle peut être considérée à différents niveaux de grains selon leur composition d'un ou plusieurs opérateurs. La définition de la taille de tâche dépend du choix différent du niveau de la taille de grains. Dans le paragraphe précédent (§ 3.2.4), nous avons présenté le taux de ressemblance mutuelle qui est réalisé par un processus d'évaluation de ressemblance des partitions par comparaison basée sur les noeuds/tâches ou UFs modélisant les chemins de données et de contrôle d'un graphe G. Cependant, l'approche adoptée nécessite donc l'identification de ces tâches communes entre partitions initiales selon des règles de formation précédemment définies en § 3.2.4

Nous proposons un algorithme d'identification et d'extraction des unités fonctionnelles communes respectant les règles de formation pour une optimisation en terme de ressources de la phase de synthèse inter-partition basée sur la réutilisation des UFs à instanciation multiple. L'algorithme prend comme entrée la description GFD/GFC des partitions initiales à optimiser par IPS. Il agit à partir d'une définition du type (ou du genre) de noeuds du graphe d'extraire les UFs factorisables selon les règles 1, 2 et 3 tel que mentionnés précédemment, le noeud est l'élément basique de toutes les UFs. Nous considérons les définitions suivantes permettant de caractériser tous les noeuds d'un graphe selon un type, la taille (size) et le nombre d'entrées et sorties.

Définition 4 *Soit un ensemble de sommets V d'un graphe G. Chaque noeud v_i est un élément dans l'ensemble V et a les attributs suivants :*

- *$genre(v_i)$ dénote le type de noeud v_i,*
- *$\pi.input.(v_i)_{ne}$ dénote le ne^{eme} entrée du noeud v_i, ou $ne \in N_{ne}.v_i$, $N_{ne}.v_i$ est le nombre total d'entrées de noeud v_i,*
- *$\pi.output.(v_i)_{ns}$ dénote le ns^{eme} sortie du noeud v_i, ou $ns \in N_{ns}.v_i$,$N_{ns}.v_i$ est le nombre total de sorti de noeud vi,*
- *$\sigma.(v_i)_{ne}$ dénote la taille de ne^{eme} entrée de noeud v_i,*
- *$\sigma.(v_i)_{ns}$ dénote la taille de ns^{eme} sortie de noeud v_i.*

La définition 4 nous permet d'identifier tous les noeuds selon une caractérisation par type et taille. L'ensemble V représente les opérateurs du graphe ou sous-graphe de l'application cible. Ensuite, à partir de cette définition, nous identifions et classifions tous les différents types de noeuds par sous-ensembles spécifiés selon la définition 5.

Définition 5 *Soit un ensemble de genre opérateur $\{B_1, ..., B_n\}$, Chaque noeud v_i du graphe doit être placé dans un sous-ensemble B_n selon la condition $genre.v_i = genre(n)$. L'indice n indique le type d'opérateurs ou noeuds différents. $Nomb.B_n$ est le nombre d'opérateurs dans le*

sous-ensemble B_n. Chaque noeud v_i doit être identifié et considéré que dans un seul type de noeud ou opérateur. Soit :

$$\sum_{1}^{N_g} Nomb.B_n = N_v \tag{3.51}$$

ou N_g est le nombre total de types d'opérateurs pour tous les v_i : $\{v_i | i \in N_v\}$, N_v est le nombre total d'opérateurs de l'ensemble V constituant G et pouvant être décrit par l'ensemble de genre possible de noeuds ou opérateurs :

$$V = \bigcup_{1}^{N_g} B_n \, et \, B_n = \bigcup v_i \,|\, genre.v_i = genre(n) \tag{3.52}$$

Définition 6 *Chaque Unité fonctionnelle (UF) ou tâche d'un graphe G est composée de noeuds (opérateurs). Les UFs identiques sont caractérisées par un même nombre de noeuds de même type possédant les mêmes relations de dépendance de données parmi les opérateurs ou noeuds les constituant.*

Les tâches représentant les unités fonctionnelles d'un graphe sont composées par au moins un opérateur (noeud v_i). Le nombre d'UFs pouvant être formé et trouvé dépend des règles prioritaires de formation des UFs selon le niveau de granularité. En général, la taille de granularité dépend du nombre d'opérateurs. Une règle de formation orientée bas niveau des UFs permet de trouver un plus grand nombre de UFs identiques possibles parmi les noeuds constituant l'ensemble V cible. Cependant, l'exécution d'une synthèse architecturale basée sur la réutilisation d'UFs, n'aboutit pas forcement à une optimisation principalement dû à une augmentation des ressources supplémentaires nécessaire à la factorisation. Cependant, les UFs " grains fins " sont faciles à définir par rapport aux UFs " gros grains ". En revanche, ces derniers sont souvent difficiles à définir selon les applications.

Définition 7 *Soit un ensemble de sous-ensembles de tâches $OS\{OS_1, ..., OS_m\}$ représentant les différentes classes (types) de tâches. OS_m est un sous-ensemble contenant des tâches identiques. Une tâche particulière $OS_m(x)$ est définie telle que $\{OS_m(x) | 1 \leq x \leq N_m, genre.OS_m(x) = genre.OS_m\}$, où N_m est le nombre de tâches en type m.*

La définition 7 définit l'ensemble des tâches regroupées en sous-ensemble de tâches identiques caractérisées par la définition 8 qui doivent déterminer parmi l'ensemble de noeuds v_i de V ceux représentant le graphe G.

Définition 8 *La relation entre deux noeuds v_i et v_j est présentée par un attribut de connexion $connex(v_i, v_j)$. Les regroupements en UFs entre les noeuds et leurs successeurs, pouvant être considérés comme des tâches identiques, sont conditionnés par le même attribut décrivant les*

dépendances de données entre deux noeuds et la taille de leur connexion tel que $\{connex(v_i, v_j) | \pi.output.(v_i)_{ns} \rightarrow \pi.input.(v_j))_{ne}$ *,* $\sigma.(v_i)ns = \sigma.(v_j)ne\}$ *où* v_j *correspond au noeud successeur du noeud* v_i*.*

La définition 8 spécifie les conditions d'identification des tâches identiques ou similaires à partir des types et dépendances de données, plus précisément, grâce à une comparaison de connexion entre deux noeuds successifs. A partir de ces définitions et de l'ensemble des noeuds V, nous avons l'objectif de déterminer et placer chaque tâche identifiée dans un sous-ensemble $OS_m()$ de tâches de même type selon la condition d'attribut spécifiée par la définition 8.

La figure 3.8 présente la fonction principale de l'algorithme d'identification et d'extraction d'unités fonctionnelles factorisables. L'identification débute par la fonction $Main\_kernel()$, qui analyse le graphe DFG de l'application cible pour créer l'ensemble des noeuds V et examiner les proprietés et attributs de chaque noeuds (type opérateur, successeurs, etc.). Tous les noeuds doivent être assignés dans un sous-ensemble B_n en fonction de leurs types d'opérations arithmétiques ou logiques (additionneur, multiplexeur, opérateur OU, etc.) ; Cette étape est spécifiés par la ligne 4 dans la figure 3.8 (Classify the type B_n). Pour tous types de noeud dans chaque sous-ensemble B_n, les phases d'identification et d'extraction des UFs sont réalisées grâce à deux fonctions $Create\_task()$ et $Thread\_main()$ (ligne 8 et 9 - figure 3.8).

Les pseudo-codes des fonctions $Create\_task()$ et $Thread\_main()$ sont détaillés respectivement dans les figures 3.9 et 3.10. Nous expliquerons brièvement les sous-fonctions utilisées par les fonctions $Create\_task()$ et $Thread\_main()$:

- La sous fonction $P.add()$ permet de créer une nouvelle tâche O_p pour un sous-ensemble de type de tâche OS_m lors de l'étape initiale de recherche des tâches (voir la fonction $Create\_task()$ dans la figure 3.9) ou lors de la détection d'une nouvelle tâche au cours du processus d'identification,
- La sous-fonction $P.create()$ permet de créer un nouveau sous-ensemble de type de tâches OS_m,
- La sous fonction $Alias()$ permet d'assigner le type d'un opérateur v_i et la mise à jour de son ordre d'indice dans le sous-ensemble de type d'opérateur B_n,
- La sous fonction $Succ()$ fournit le noeud successeur d'un opérateur ou noeud v_i donné en argument.

La fonction $Create\_task()$ est chargée de créer des nouvelles tâches O_p de même type à partir des ensembles V et B_n. Les opérateurs d'un ensemble de type d'opérateur B_n sont initialement et individuellement placés dans chaque nouvelle tâche O_p initialement crée d'un sous-ensemble de tâches OS_m et où p est l'indication d'ordre de tâche dans ce sous-ensemble. Dans cette première étape, ces tâches sont similaires puisqu'elles ont un même type de noeud. Typiquement, le choix

```
1.  //This main function
2.  Main_kernel{
3.    Create list v_i∈ V // number for all node in V
4.    Classify the type B_n//Create the all subset of operators' type
5.     begin
6.       int n=1; int m=0;
7.       for n=<Ng {   // for each subset of operator's type B_n,
8.         Creat_task();  // create the task which
                          begin with this type operator.
9.         Thread_main(); // complete and classify the created task
10.        OS_final= OS_final + OS_m;
11.        n++;
12.        }
13. }
```

FIG. 3.8 – Algorithme d'identification et d'extraction d'unités fonctionnelles factorisables.

du type d'un opérateur ou un noeud dépend de son ordre d'apparition dans l'ensemble V. Ainsi, si le premier noeud v_1 de V est contenu dans le sous-ensemble B_1, le sous-ensemble de type de tâche OS_m débutera avec tous les noeuds contenus dans le sous-ensemble B_1 pour former les nouvelles tâches initiales vides O_p et dans lesquelles vont être placés les noeuds de même type que v_1 parmi les noeuds inclus dans le sous-ensemble B_1 (voir lignes 5-8 - figure 3.9).

En général, chaque noeud dans B_1 va être placé individuellement dans chaque tâche vide O_p comme le premier noeud de la tâche à déterminer (ligne 9 dans la figure 3.9). L'indice p est l'indicateur d'ordre de placement. Dans cette première étape, toutes les tâches O_p sont alors dans un premier temps similaires car ces tâches contiennent un même type d'opérateur. Les noeuds assignés dans les tâches sont alors déconsidérés de l'ensemble V (ligne 10 - figure 3.9). Le type d'opérateur et l'ordre du noeud dans le sous-ensemble B_n sont assignés par la sous fonction $Alias()$ qui permet d'indiquer la position du noeud dans la tâche O_p (ligne 11- figure 3.9). Ainsi, toutes les tâches nouvellement crées forment un nouveau sous-ensemble de type de tâches (lignes 16, 18 - figure 3.9).

Après la création d'un sous-ensemble de tâches initiales OS_m par la fonction $Create\_task()$, la fonction $Thread\_main()$ réalise la recherche des UFs similaires par comparaison de connexion pour des noeuds, instanciés dans les tâches initiales O_p, avec leurs noeuds successeurs dans V. Toutes les tâches qui ont le même type de connexion sont considérées comme des tâches similaires et donc à instanciation multiples dans G (ligne 11 - figure 3.10). Les successeurs d'un noeud déterminés par la sous fonction $Succ()$ sont intégrés dans la tâche considérée ou le noeud prédécesseur est placé (lignes 14, 16 et 30 - figure 3.10). Ces noeuds successeurs sont ensuite déconsidérés de l'ensemble V (lignes 15, 17 et 31 - figure 3.10). Toutes les UFs identiques sont ajoutées dans le sous-ensemble de type de tâches OS_m (ligne 19 - figure 3.10).

Un sous-ensemble de type de tâches valable contient au moins deux tâches similaires (tâches à instanciation multiple). Si au moins deux UFs (ou tâches) sont identifiées et différentes des

```
1.  // create task function main kernel
2.  Create_task() {
3.     int p=1; // the index of tasks
4.     int q=1; // order of node in current task
5.     for vi ∈ V do
6.       if (genre(vi) = genre.n) then
7.            //if the node type corresponds to the selected type
8.         P.add(Op) ; //create a new task
9.         Op= Op+vi ; //add the current node to this task
10.        V=V-vi ; //remove the current node from V
11.        Alias (vi)=vp.q ; //assign info to current node
12.         i++;
13.         p++ ;
14.           if (OSm∈ OSfinal ) then
15.             m++;
16.             OSm=OSm+Op;
17.           Else
18.             OSm=OSm+Op;
19.           End if;
20.        Else
21.         i++;
22.        End;
23. }
```

FIG. 3.9 – Description de la fonction $Create\_task()$.

UFs précédemment identifiées, l'algorithme crée alors un nouveau sous-ensemble de type de tâche OS_{m+z} afin de les classer et recenser. Ces tâches sont alors déplacées du sous-ensemble OS_m vers le sous-ensemble OS_{m+z} (lignes 22-24 - figure 3.10). Si par contre une tâche ne peut être associée à un sous-ensemble de type de tâche, elle est alors considérée localement ou temporairement comme une tâche à instanciation unique. Dans ce cas, la tâche est " libérée " et les noeuds constituants cette tâche sont alors reconsidérés parmi les noeuds de V pour les détections et les identifications suivantes (lignes 45, 46 - figure 3.10). Lorsque l'algorithme termine l'assignement d'un noeud v_i à une tâche O_p, les successeurs $Succ(v_i)$ deviennent des noeuds courants pour le niveau de détection suivant (lignes 26, 27, 37 - figure 3.10).

Durant la phase de détection de tâche, l'algorithme cesse la détection d'une tâche courante, si le ou les successeurs d'un noeud v_i ne sont plus considérés dans l'ensemble V. En effet, cela signifie alors que ce ou ces successeurs ont été assignés avec d'autre tâches O_p. De même dans le cas où l'algorithme ne trouve plus des connexions similaires entre les noeuds, la détection est alors terminée (lignes 11, 42 - figure 3.10). Après la phase de détection de tâches, l'algorithme supprime toutes les tâches contenant qu'un seul noeud en reconsidérant tous les noeuds associés à ces tâches à l'ensemble V (lignes 45-46 - figure 3.10).

A travers la fonction $Main\_kernel()$, l'algorithme recense toutes les tâches similaires identifiées dans la phase de détection précédente par un sous-ensemble final de type de tâches OS_{final} . Ensuite la détection suivante est préparée en considérant les noeuds courants suivants (lignes

```
1. //main seek kernel for similar tasks
2. Thread_main(){
3.   int x=1;
4.   int p=0; // the indication of task
5.   int z=0;
6.   for all q; // the order of all node in task
7.   for p=1 to Nomb.Bn do // Nomb.Bn is the total number of created subset for all node in subset Bn
8.     vp.present = vp.q;
9.     vp+x.present = vp+x.q;  //set the present node for Op and Op+x
10.    while x=<(Nomb.Bn - p) do {
11. Loop A: If (Connx(vp.present, succ(vp.present))==Connx(vp+x.present, succ(vp+x.present))& succ(vp+x.present))∈ V then
12.              //judge the connections from two tasks are similar or not
13.               If succ(vp.present) ∈ V then // if the first Op of subset of task type OSm
14.                  Op = Op + succ(vp.present) ; // add current node to an existent task
15.                  V = V- succ(vp.present) ; // remove current node from V
16.                  Op+x = Op+x + succ(vp+x.present) ; // add current node to an existent task
17.                  V = V- succ(vp+x.present) ;
18.                   if Op,Op+x ∈ OSm then
19.                        OSm = OSm + (Op + Op+x ); // add current task to the subset of task type
20.                     else
21.                        z++;
22.                        P.create(OSm+z);
23.                        OSm = OSm - (Op + Op+x );
24.                        OSm+z = OSm+z + (Op + Op+x ); // add current task to a new subset of task type
25.                   end if;
26.                   vp.present = succ(vp.present);
27.                   vp+x.present = succ(vp+x.present);
28.                   x++;
29.                Else
30.                  Op+x = Op+x + succ(vp+x.present) ;
31.                  V = V- succ(vp+x.present) ;
32.                  if Op,Op+x ∈ OSm then
33.                        Om = Om + Op+x ;//add current task to subset of task type
34.                     else
35.                        OSm+z = OSm+z + Op+x ;//add current task to subset of task type
36.                  end if;
37.                  vp+x.present = succ(vp+x.present);
38.                  x++;
39.               End if;
40.               If Nomb.Op > Nomb.level then break loop; //define the level of FUs by designer
41.            Else
42.               break loop
43.            END if; }
44.  p++;
45.  If Op ∉ Om then  // if a task is not assigned to any existent subset of task type (for example only one node in this task).
46.     V=V+allnodein Op ; // free all node in current task to V
47.  End if; }
```

FIG. 3.10 – Description de la fonction $Thread\_main()$.

10, 11 - figure 3.8). La fonction $Main\_kernel()$ permet également de définir une recherche de tâches similaires selon un niveau de granularité des UFs pouvant être spécifié par le concepteur (ligne 40 - figure 3.10).

Une nouvelle phase détection du processus d'identification est répétée pour les noeuds restant de l'ensemble V en considérant un sous-ensemble B_n suivant. Ce processus est répété pour tous les sous-ensembles de type d'opérateurs B_n et tant que l'ensemble V n'est pas vide.

Nous illustrons notre algorithme d'identification et d'extraction d'unités fonctionnelles factorisables avec un exemple de description de GFD. L'exemple considéré est un algorithme de détection de contour en traitement d'images. La figure 3.11 donne le GFD de cet exemple contenant 28 noeuds. Dans une première étape, l'algorithme vérifie les attributs de tous les noeuds dénotés dans l'ensemble V du graphe. Il attribut également un ordre d'apparition du noeud dans

le graphe. Ensuite, les sous-ensembles de type de noeuds B_n sont crées dans lesquels les noeuds seront classés (noeuds arithmétiques, logiques, etc.) selon la définition 4.

Chaque sous-ensemble B_n signifie un ensemble de type différent de noeud. Dans l'exemple considéré, tous les sous-ensembles de type d'opérateurs de tous les noeuds du graphe sont donnés dans la figure 3.12. Les noeuds de même type sont recensés et classés dans un même sous-ensemble B_n et sont assignés avec un nouveau indice d'ordre à travers la sous-fonction $Alias()$. Par exemple, le noeud $v_4(\beta_1.v_2)$ dans le sous-ensemble B_1 (voir figure 3.12), les attributs β_1 et v_2 présentent respectivement le type du noeud et le nouvel ordre dans le sous-ensemble B_1. Chaque noeud ne peut être classifié que dans un seul type d'opérateur et doit être placé dans un unique sous-ensemble de type de noeud B_n. Dans notre exemple, sept sous-ensembles (les comparateurs, multiplexeurs, soustracteurs, additionneurs, multiplieurs, valeurs absolues et diviseurs) sont distingués et constitués. Ces sous-ensembles sont listés dans la figue 3.12.

Le processus d'identification des UFs identiques débute à partir des noeuds du sous-ensemble de type d'opérateur dans lequel le premier noeud v_1 est placé (B_1). La figure 3.13 détaille le processus global d'identification et extraction des tâches constitué de plusieurs itérations. Le sous-ensemble B1 contient sept opérateurs ou noeuds de même type (voir figure 3.12). Dans une première itération, sept tâches vides sont crées et correspondent aux " *tâches initiales* " nommées $O_0(x)$. Les noeuds du sous-ensemble B_1 sont placés individuellement dans chaque tâche $O_0()$ respectivement comme un noeud initial. Ils sont ensuite déconsidérés de l'ensemble de noeuds V du graphe. Le nombre total de " *tâches initiales* ", correspondant à un premier sous-ensemble de tâches, est égal au nombre total de noeuds dans le sous-ensemble B_1. Dans notre cas, nous avons un nombre de 7 tâches initiales : $O_0(1)$ à $O_0(7)$ (voir figure 3.13). Ensuite, l'algorithme commence l'identification des tâches communes de façon comparative. L'étape de " *détection* " du processus d'identification fournit les successeurs des noeuds associés selon les connexions possibles de chaque noeud dans une tâche $O_0()$. Par exemple, le noeud initial $v_1(v_2, v_3)$ contenu dans l'UF initiale $O_0(1)$ possède deux successeurs v_2 et v_3.

La détection de la tâche $O_0(1)$ commence par les opérateurs v_1, v_2 et v_3 avec v_1 comme noeud courant. La détection de la tâche $O_0(2)$ commence par l'opération v_4 de même type que v_1, et dont le noeud v_5 est le noeud successeur. L' étape " *détection* " de la première itération (voir figure 3.13), s'effectue à partir des noeuds initiaux dans chaque tâche $O_0(x)$ et leurs successeurs. Une fois les successeurs de noeuds initiaux trouvés et annotés, l'algorithme liste et compare toutes les connexions de même hiérarchie entre les noeuds des tâches $O_0()$ et leurs successeurs. Le but est d'identifier les regroupements possibles des noeuds en UF. Afin de distinguer les différents types de connexion (par l'attribut connex()), nous utilisons le symbole "⇒" pour signifier un " *type* " de connexion existant dans au moins deux tâches $O_0()$ différentes ; et le symbole "−" pour signifier un type de connexion exclusive existant uniquement dans une seule tâche parmi les tâches $O_0()$.

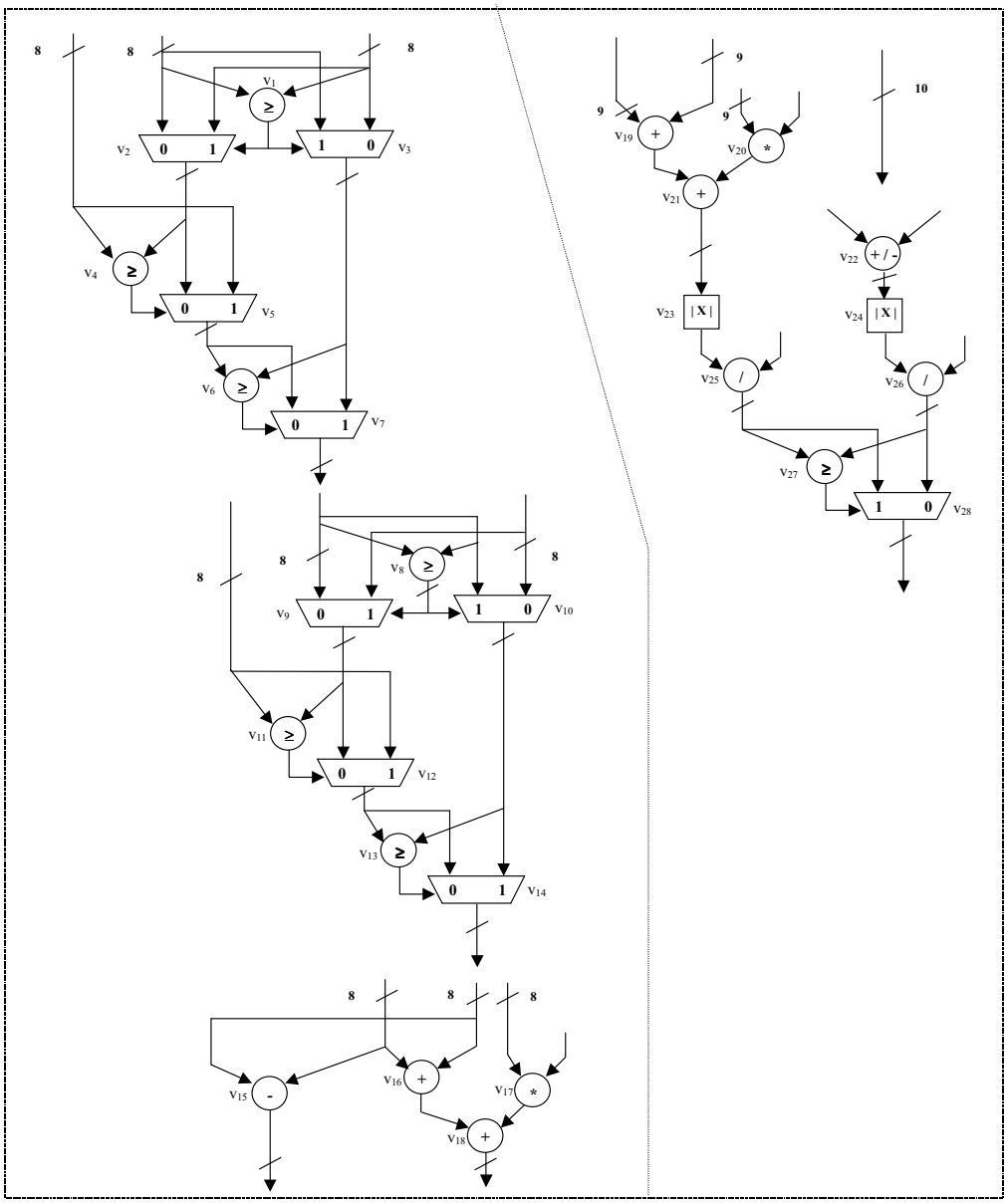

FIG. 3.11 – Le chemin de données d'exemple.

B_1 : {7 | v_1(β1.v1), v_4(β1.v2), v_6(β1.v3), v_8(β1.v4), v_{11}(β1.v5), v_{13}(β1.v6), v_{27}(β1.v7)} *comparateur*
B_2 : {9 | v_2(β2.v1), v_3(β2.v2), v_5(β2.v3), v_7(β2.v4), v_9(β2.v5), v_{10}(β2.v6), v_{12}(β2.v7), v_{14}(β2.v8), v_{28}(β2.v9)} *multiplexeur*
B_3 : {2 | v_{15}(β3.v1), v_{22}(β3.v2)} *soustracteur*
B_4 : {4 | v_{16}(β4.v1), v_{18}(β4.v2), v_{19}(β4.v3), v_{21}(β4.v4)} *additionneur*
B_5 : {2 | v_{17}(β5.v1), v_{20}(β5.v2)} *multiplieur*
B_6 : {2 | v_{23}(β6.v1), v_{24}(β6.v2)} valeur *absolu*
B_7 : {2 | v_{25}(β7.v1), v_{26}(β7.v2)} *diviseur*

FIG. 3.12 – Liste de sous ensemble de type de noeuds ou d'opérateurs.

Si la connexion n'est pas exclusive, les successeurs de chaque noeud dans la tâche sont retenus et intégrés dans la tâche considérée $O_0()$ où le noeud prédécesseur est placé. En effet, si les liaisons entre les noeuds sont identiques, les noeuds concernés peuvent être groupés. L'identification du type de connexion est réalisée par l'étape " *connexions similaires* " (voir figure 3.13). Dans notre exemple, les connexions $connex(v_1, v_2)$: "$v_1 \Rightarrow v_2$", $connex(v_1, v_3)$, $connex(v_4, v_5)$, $connex(v_6, v_7)$, $connex(v_8, v_9)$, $connex(v_8, v_{10})$, $connex(v_{11}, v_{12})$, $connex(v_{13}, v_{14})$, $connex(v_{27}, v_{28})$ sont identiques. Tous les successeurs associés à ces connexions seront alors considérés et associés aux tâches $O_0()$ dans lesquelles se trouvent respectivement leurs prédécesseurs (voir l'étape "*Regroupage & détection*" dans la figure 3.13).

L'algorithme poursuit sa comparaison des connexions des noeuds successeurs de chaque tâche $O_0()$ avec leurs noeuds successeurs suivants. Si une connexion est exclusive ou bien les successeurs ont déjà été assignés, la connexion considérée est non valide et ignorée (voir " *connexions non-similires* " figure 3.13). Par exemple, la connexion exprimée par $connex(v_2, v_4) = v_2 - v_4$ dans laquelle le noeud v_4 a déjà été assigné dans la tâche $O_0(2)$ n'est pas valide. Dans ce cas, l'algorithme mémorise et considère la tâche courante comme l'état final de la tâche si plus aucune connexion n'est trouvée avec des noeuds successeurs. Dans notre exemple, les noeuds successeurs des tâches initiales $O_0(1) - O_0(5)$ ne sont plus disponibles et donc plus présents parmi les noeuds de l'ensemble V. Les successeurs de la tâche $O_0(6)$ possèdent des connexions exclusives tandis que la tâche $O_0(7)$ ne possède pas de noeud successeur. En résumé, les deux tâches initiales $O_0(1)$ et $O_0(4)$ (voir figure 3.13) sont considérées comme similaires. En effet, la connexion $connex(v_1, v_2)$ correspond à celle $connex(v_8, v_9)$ et la connexion $connex(v_1, v_3)$ correspond à la connexion $connex(v_8, v_{10})$. Nous obtenons alors une nouvelle mise à jour de la tâche $O_0(1)\{v_1|v_2, v_3\}$ qui est composée des noeuds v_1,v_2 et v_3 et est identique à la tâche mise à jour $O_0(4)\{v_8|v_9, v_{10}\}$ composée des noeuds v_8,v_9 et v_{10}. Les 4 tâches initiales $O_0(2)$, $O_0(3)$, $O_0(6)$, $O_0(5)$ et $O_0(7)$ sont considérées également similaires puisque la connexion $connex(v_4, v_5)$ correspond aux connexions $connex(v_6, v_7)$, $connex(v_{11}, v_{12})$, $connex(v_{13}, v_{14})$ et $connex(v_{27}, v_{28})$. Par conséquent, la tâche mise à jour $O_0(2)\{v_4|v_5\}$ composée des noeuds v_4 et v_5 est identique aux nouvelles tâches $O_0(3)\{v_6|v_7\}$ composée des noeuds v_6 et v_7, $O_0(5)\{v_{11}|v_{12}\}$ composée des noeuds v_{11} et v_{12}, $O_0(6)\{v_{13}|v_{14}\}$ composée des noeuds v_{13} et v_{14} et $O_0(7)\{v_{27}|v_{28}\}$ composée des noeuds v_{27} et v_{28}. Ainsi, dans la première itération du processus d'identification des UFs à instanciation multiple, deux types de tâches identiques O_1 et O_2 sont trouvés (voir ligne " *Tâche finale de la 1ère itération* ", figure 3.13).

L'exécution de la seconde itération s'effectue à partir des noeuds du sous-ensemble B_3. En effet, tous les noeuds du sous-ensemble B_2 ne sont plus considérés dans V puisque ils ont tous déjà été assignés dans la première itération du processus d'identification. Dans l'étape de " *détection* " de cette seconde itération, et en fonction des noeuds pouvant être considérés du sous-ensemble B_3, deux tâches initiales sont créées $O_0(1)\{v_{15}|v_{19}, v_{20})\}$ et $O_0(2)\{v_{22}|v_{24})\}$. Cependant,

Processus	Étapes	Détail						
Première itération	*Tâches initiales*	O_0 (1) {[v_1]}	O_0 (2) {[v_4]}	O_0 (3) {[v_6]}	O_0 (4) {[v_8]}	O_0 (5) {[v_{11}]}	O_0 (6) {[v_{13}]}	O_0 (7) {[v_{27}]}
	Détection	v_1 (v_2,v_3)	v_4 (v_5)	v_6 (v_7)	v_8(v_9,v_{10})	v_{11} (v_{12})	v_{13} (v_{14})	v_{27} (v_{28})
	Connexions similaires	$v_1 \Longrightarrow v_2$ $v_1 \Longrightarrow v_3$	$v_4 \Longrightarrow v_5$	$v_6 \Longrightarrow v_7$	$v_8 \Longrightarrow v_9$ $v_8 \Longrightarrow v_{10}$	$v_{11} \Longrightarrow v_{12}$	$v_{13} \Longrightarrow v_{14}$	$v_{27} \Longrightarrow v_{28}$
	Regroupage et détection	{[v_1][v_2,v_3]}	{[v_4][v_5]}	{[v_6][v_7]}	{[v_8][v_9, v_{10}]}	{[v_{11}][v_{12}]}	{[v_{13}][v_{14}]}	{[v_{27}][v_{28}]}
		v_2(~~v_3~~,~~v_4~~,~~v_5~~), v_3(~~v_7~~)	v_5(~~v_6~~,~~v_7~~)	v_7(~~v_8~~,~~v_9~~,~~v_{10}~~,~~v_{11}~~,~~v_{12}~~)	v_9(~~v_{10}~~,~~v_{11}~~,~~v_{12}~~), v_{10}(~~v_{13}~~,~~v_{14}~~)	v_{12}(~~v_{13}~~,~~v_{14}~~)	v_{14}(v_{15},v_{16},v_{17})	v_{28}()
	Connexions non similaires	v_2-~~v_3~~ v_2-~~v_4~~ v_2-~~v_5~~ v_3-~~v_7~~	v_5-~~v_6~~ v_5-~~v_7~~	v_7-~~v_8~~ v_7-~~v_9~~ v_7-~~v_{10}~~ v_7-~~v_{11}~~ v_7-~~v_{12}~~	v_9-~~v_{10}~~ v_9-~~v_{11}~~ v_9-~~v_{12}~~ v_{10}-~~v_{13}~~ v_{10}-~~v_{14}~~	v_{12}-~~v_{13}~~ v_{12}-~~v_{14}~~	v_{14}-v_{15} v_{14}-v_{16} v_{14}-v_{17}	v_{28} est le dernier noeud dans le V
		Tous les noeuds successifs ont été assignés aux autres tâches, il n'y a pas d'autres connexions similaires entre deux tâches, fin de la 1ère itération.						
Tâche finale de la 1ère itération	*Type de tâche 1*	O_1 (1) {[v_1][v_2,v_3]}			O_1 (2) {[v_8][v_9, v_{10}]}			
	Type de tâche 2		O_2 (1) {[v_4][v_5]}	O_2 (2) {[v_6][v_7]}		O_2 (3) {[v_{11}][v_{12}]}	O_2 (4) {[v_{13}][v_{14}]}	O_2 (5) {[v_{27}][v_{28}]}
	Dans la 1ère itération, deux types de tâches O_1() et O_2() sont trouvés.							
Deuxième itération	*Tâches initiales*	O_0 (1) {[v_{15}]}	O_0 (2){[v_{22}]}					
	Détection	v_{15} (v_{19},v_{20})	v_{22} (v_{24})					
	Connexions non similaires	v_{15}-v_{19} v_{15}-v_{20}	v_{22}-v_{24}					
	Il n'y a pas d'autres connexions similaires entre deux tâches, aucun type de tâche n'est trouvé dans la 2ième itération.							
Troisième itération	*Tâches initiales*	O_0 (1) {[v_{16}]}	O_0 (2) {[v_{18}]}	O_0 (3) {[v_{19}]}	O_0 (4) {[v_{21}]}			
	Détection	v_{16} (v_{18})	v_{18} (v_{22})	v_{19} (v_{21})	v_{21} (v_{23})			
	Connexions similaires	$v_{16} \Longrightarrow v_{18}$	v_{18}-v_{22}	$v_{19} \Longrightarrow v_{21}$	v_{21}-v_{23}	Les connexions similaires sont trouvées. Continue jusqu'à la prochaine détection		
	Regroupage et détection	{[v_{16}][v_{18}]}		{[v_{19}][v_{21}]}				
		v_{18}(v_{17})		v_{21}(v_{20})				
	Connexions similaires	$v_{18} \Longrightarrow v_{17}$		$v_{21} \Longrightarrow v_{20}$				
	Regroupage et détection	{[v_{16}][v_{18}][v_{17}]}		{[v_{19}][v_{21}][v_{20}]}	Tous les noeuds successifs ont été assignés, fin de la 3ième itération.			
		v_{17}(~~v_{18}~~)		V_{20}(~~v_{21}~~)				
Tâche finale de la 3ième itération	*Type de tâche 3*	O_3 (1) {[v_{16}][v_{18}][v_{17}]}		O_3 (2) {[v_{19}][v_{21}][v_{20}]}				
	Dans la 3ième itération, le type de tâche O_3() est trouvé.							
Quatrième itération	*Tâches initiales*	O_0 (1) {[v_{23}]}	O_0 (2) {[v_{24}]}					
	Détection	v_{23}(v_{25})	v_{24}(v_{26})					
	Connexions similaires	$v_{23} \Longrightarrow v_{25}$	$v_{24} \Longrightarrow v_{26}$					
	regroupage et détection	{[v_{23}][v_{25}]}	{[v_{24}][v_{26}]}	Tous les noeuds successifs ont été assignés, fin de la 4ière itération.				
		v_{25}(~~v_{27}~~)	v_{26}(~~v_{27}~~,~~v_{28}~~)					
Tâche finale de la 4ième itération	*Type de tâche 4*	O_4 (1) {[v_{23}][v_{25}]}	O_4 (2) {[v_{24}][v_{26}]}					
	Dans la 4ième itération, le type de tâche O_4() est trouvé.							

FIG. 3.13 – Le détail du processus d'identification.

Type de tâches	Liste détaillée des tâches par type				
Type 1	O_1 (1) {[v_1][v_2,v_3]}	O_1 (2) {[v_8][v_9, v_{10}]}			
Type 2	O_2 (1) {[v_4][v_5]}	O_2 (2) {[v_6][v_7]}	O_2 (3) {[v_{11}][v_{12}]}	O_2 (4) {[v_{13}][v_{14}]}	O_2 (5) {[v_{27}][v_{28}]}
Type 3	O_3 (1) {[v_{16}][v_{18}][v_{17}]}	O_3 (2) {[v_{19}][v_{21}][v_{20}]}			
Type 4	O_4 (1) {[v_{23}][v_{25}]}	O_4 (2) {[v_{24}][v_{26}]}			

FIG. 3.14 – Liste détaillée des ensembles finaux par type de tâche.

les connexions de ces deux tâches sont exclusives. Ces deux UFs ne sont donc pas considérés comme identiques. Les noeuds de ces tâches ne sont pas assignés dans une UF et sont alors reconsidérés parmi les noeuds disponibles dans l'ensemble V. Le processus d'identification est réitéré. Ainsi, pour la troisième itération, une tâche est identifiée tandis que pour la quatrième itération deux tâches sont identifiées (voir figure 3.13).

Le processus d'identification s'arrête après réalisation de toutes les itérations possibles et pour tous les sous-ensembles B_n. Dans notre exemple, quatre itérations ont été nécessaires et ont permis d'identifier et d'extraire quatre différents types d'UFs. Toutes ces UFs trouvées sont détaillées dans la figure 3.14 et localisées dans le graphe flot de données de notre exemple sur la figure 3.15.

3.2.6 Méthodologie de partitionnement temporel optimisée par synthèse architecturale inter-partition

3.2.6.1 Principe méthodologique

Notre méthode nommée IPS (Inter-Partition Synthesis - Synthèse Inter-Partition) consiste à effectuer une synthèse architecturale entre deux partitions initiales successives d'une application complète représentée sous forme d'un graphe flot de données (GFD). Le principe méthodologique général de notre approche est présenté par la figure 3.16. Il se résume de la manière suivante :

Phase (a) : Partitionnement temporel initial

A partir d'une description en entrée de l'application visée sous forme de GFD/GFC, un partitionnement temporel générant des partitions initiales matérialisant des sous- graphes est réalisé. L'approche méthodologique proposée est indépendante de la méthode de partitionnement temporel initial. En effet, cette première étape peut être mise en œuvre avec toutes les méthodologies de partitionnement temporel basées sur un ou plusieurs critères d'optimisation (bande passante, temps d'exécution, surface, etc.) et ayant comme entrée une description de type GFD [Cha98, ea03, Car03].

Phase (b) : Evaluation d'une Synthèse Inter-partition

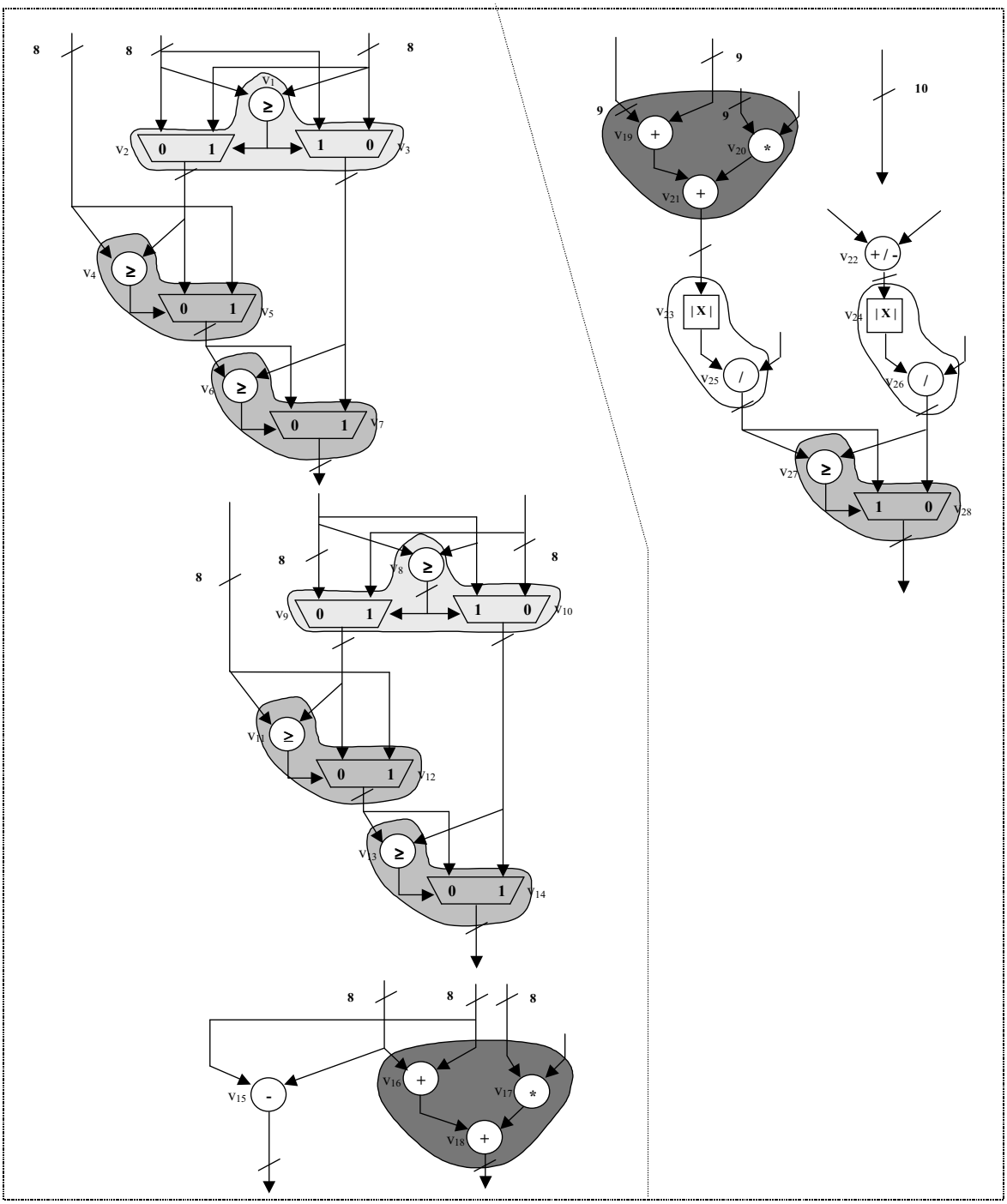

FIG. 3.15 – Illustration des UFs identifiées dans le GFD.

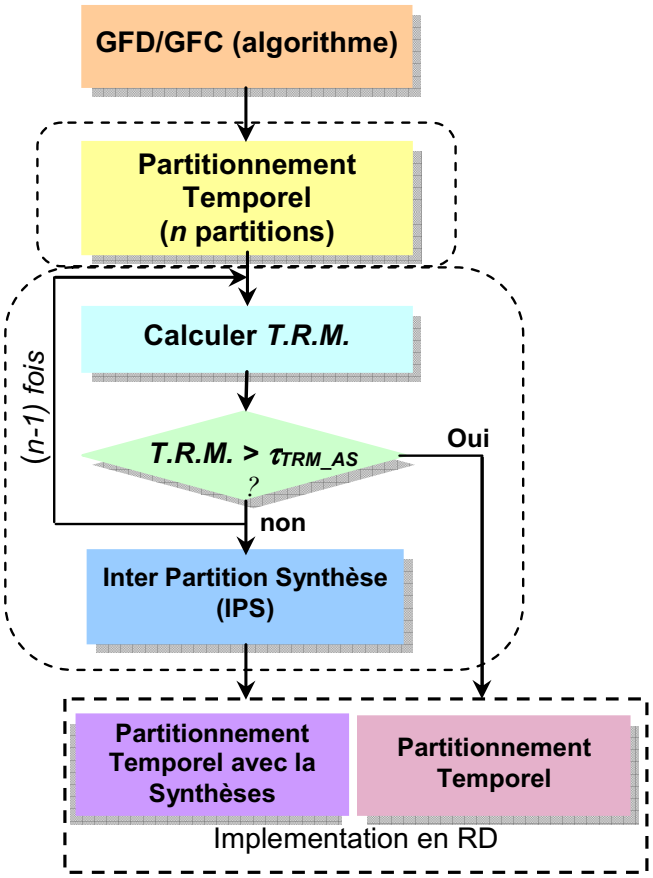

FIG. 3.16 – Principe de l'approche méthodologique proposée.

Ensuite, dans la seconde phase, nous proposons de calculer un indicateur décrivant le degré de ressemblance entre les partitions initiales successives. Cet indicateur est décrit par un paramètre qui définit le taux de ressemblance mutuelle (**T.R.M**). Les résultats de comparaison entre les **T.R.M** calculés pour l'ensemble des partitions initiales successives, et une valeur de seuil τ_{TRM_AS} permet d'orienter une implantation vers une solution appropriée à exécuter d'une application en reconfiguration dynamique.

Dans le schéma de principe de la méthodologie proposée, la valeur du seuil τ_{TRM_AS} est définie par la contrainte de surface logique. La principale condition formulée consiste à conserver une surface logique pour les partitions réduites inférieure à la surface logique initiale de l'unité de calcul reconfigurable. Une formulation de cette condition est déduite par les équations suivantes. A partir de la modélisation de deux partitions initiales selon l'équation 3.39 :

$$P_i = U_i \cup F_i \quad et \quad P_j = U_j \cup F_j \quad ou \quad i < j \tag{3.53}$$

La nouvelle partition combinée IPS peut être formulée par la relation suivante :

$$P_{i,j} = AS(P_n, P_{n+1}) = U_i \cup U_j \cup F'_{i,j} \tag{3.54}$$

A partir de la condition 3.11, nous pouvons alors en déduire l'expression de la condition sur la surface logique de la partition réduite $P_{i,j}$:

$$Area(P_{i,j}) < \lambda \cdot S_{ucr} \tag{3.55}$$

où λ est le coefficient d'exploitation surfacique de l'unité de calcul reconfigurable

A partir des équations 3.54 et 3.55, nous pouvons donc formuler la condition sur la surface de la logique de contrôle et d'aiguillage mettant en œuvre la factorisation des UFs à instanciation multiple :

$$Area(U_i \cup U_j \cup F'_{i,j}) + Area(R_{supp}) < \lambda \cdot S_{ucr} \tag{3.56}$$

$$Area(F'_{i,j}) + Area(U_i \cup U_j) + Area(R_{supp}) < \lambda \cdot S_{ucr} \tag{3.57}$$

D'où

$$Area(F'_{i,j}) < \lambda \cdot S_{ucr} - Area(U_i \cup U_j) - Area(R_{supp}) \tag{3.58}$$

où R_{supp} correspond aux ressources supplémentaires et $Area(R_{supp})$ représente la surface logique de ces ressources supplémentaires mettant en œuvre physiquement la factorisation des UFs à instanciations multiples.

$Area(F'_{i,j})$ est la surface logique de la partie qui peut être réutilisée soit la partie qui compose des unités fonctionnelles et qui correspond également à l'expression suivante :

$$Min\left[Area(\sum_{k=1}^{p}(Nf_i^{(p)}\cdot Sf_i^{(p)})), Area(\sum_{k=1}^{p}(Nf_j^{(p)}\cdot Sf_j^{(p)}))\right] \tag{3.59}$$

qui est dans l'équation de T.R.M (équation 3.50). Où p est le nombre d'instanciations. Donc,

$$\frac{Area(F'_{i,j})}{Max(Area(P_i), Area(P_j))} < \frac{\lambda\cdot S_{ucr} - Area(U_i\cup U_j) - Area(R_{\text{supp}})}{Max(Area(P_i), Area(P_j))} \tag{3.60}$$

Donc la valeur du seuil est :

$$\tau_{TRM\_AS} = Min\left[\frac{\lambda\cdot S_{ucr} - Area(U_i\cup U_j) - Area(R_{\text{supp}})}{Max(Area(P_i), Area(P_j))}\right]^{i,j\in N} \tag{3.61}$$

Phase (c) : Synthèse Inter-partition

A partir des contraintes, des résultats de la phase précédente et des critères d'application imposés par le concepteur, une solution de partitionnement temporel réduit par synthèse architecturale définie par l'équation 3.45 est générée pour l'application visée.

3.2.6.2 Flot de conception d'un partitionnement temporel optimisé par IPS

A partir des formulations considérées, l'approche méthodologique proposée s'insère dans un nouveau flot de conception spécifique basée IPS développé pour l'implémentation optimisée RD-SA d'une application. Les différentes phases détaillées du flot de conception d'un partitionnement temporel optimisé IPS sont représentées dans la figure 3.17. Les étapes de fonctionnement des phases du flot de conception de la méthode proposée sont modélisées de la manière suivante :

Bloc A : Entrée du flot de conception

Une description de l'application saisie sous forme graphe flot de données et de contrôle (DFG/CDFG) est choisie comme entrée du flot de conception. Ce niveau de description en noeuds d'opérations et arcs de communication entre opérateurs, correspondant à la modélisation et formulation adoptées, facilite la mise en œuvre d'une recherche de solution optimale d'un partitionnement temporel en permettant les associations possibles des éléments de calcul constituant l'application selon différents niveaux de granularité (noeuds, tâches, etc.).

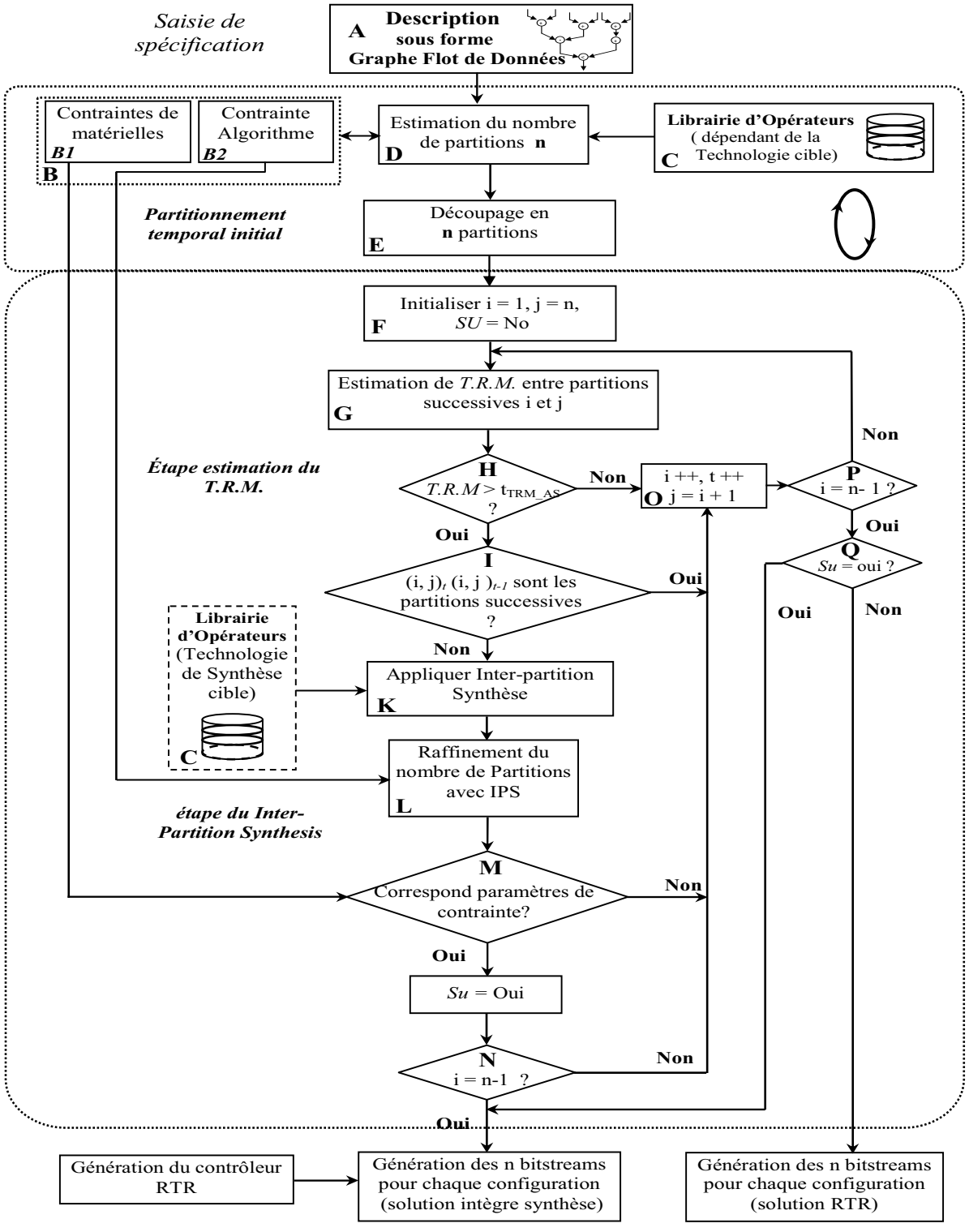

FIG. 3.17 – Flot de conception détaillé du partitionnement temporel optimisé IPS.

Phase (a) : Partitionnement temporel initial

La méthode de partitionnement temporel initial adoptée dans le flot de conception proposé ici, repose sur la méthode constructive de partitions homogènes (voir chapitre 2 § 2.2.2.4.3) [Ta03]. Cette phase (a) est automatisée par l'outil DAGARD [Bru04]. L'intérêt de cette approche heuristique est qu'elle permet une estimation de la surface moyenne optimale par partitions tout en exploitant au mieux la contrainte de temps. Ce qui la rend complémentaire à l'approche proposée cherchant à réduire le temps d'exécution total par réduction du nombre de partitions initiales. De plus, le flot de conception de ce type de partitionnement facilite l'intégration des phases d'ordonnancement et d'optimisation des unités fonctionnelles partagées par synthèse architecturale.

Notre approche n'est pas liée spécifiquement à cette méthode de partitionnement temporel initial. Elle peut être mise en œuvre avec tout autre méthodologie de partitionnement temporel comme entrée de notre approche d'optimisation [Cha98, ea03, Car03].

Les étapes du partitionnement temporel adopté sont globalement similaires au flot de conception présenté dans la figure 3.17 mais adaptées à notre approche de conception. A partir du graphe flot de données de l'algorithme à partitionner (bloc A), les paramètres de contrainte (bloc B) et une caractérisation de la technologie reconfigurable cible (bloc C), les étapes de la méthode de partitionnement sont mis en œuvre à travers les blocs D et E.

Bloc D : Estimation du nombre de partitions

Une approximation du nombre de partitions initiales est calculée [Ta03]. L'estimation est basée d'une part sur une bibliothèque d'opérateurs (bloc C) dont les caractéristiques dépendent de la technologie cible et d'autre part sur la définition de contraintes.

Bloc C : Librairie d'opérateurs(technologie de synthèse)

Cette librairie de la technologie cible est une bibliothèque des modèles de tous les opérateurs. Ces derniers sont caractérisés pour une technologie FPGA. La caractérisation permet d'évaluer la surface logique à l'implantation selon les modélisations à la fois des surfaces logiques des noeuds vi présents physiquement dans l'unité de calcul reconfigurable et formulée par l'équation 3.7 ; des temps de traversée des noeuds et de propagation des données entre les noeuds formulés par l'équation 3.18.

Bloc B : Définition des contraintes

Pour assurer le fonctionnement et garantir l'application visée, l'implémentation doit répondre à toutes les contraintes. On distingue les contraintes matérielles et les contraintes de traitement. Une saisie des contraintes matérielles (sous-bloc B1) est réalisé. Les contraintes matérielles sont souvent la surface logique de l'unité de calcul reconfigurable (ressources fixées dans le cas d'une approche développement système), la fréquence maximum de travail supportée par la cible, la taille et bande passante des ressources mémoires, la taille d'entrée-sortie, la consommation, certaines caractéristiques du FPGA telle que la vitesse de reconfiguration maximum, le support d'une reconfiguration partielle, etc. Les contraintes de traitement (Sous-bloc B2), correspondent à la contrainte de temps, le temps d'exécution total, la latence de traitement autorisée, la fréquence maximale autorisée dans le cas d'une application à consommation réduite, la surface logique maximum du FPGA requis pour les partitions, etc.

Bloc E : Découpage en n partitions

À partir de l'estimation du nombre de partitions originales (bloc D), les limites ainsi que le Floorplan de chaque partition temporelle initiale du graphe sont déduites selon différents critères telles que la surface/partition, la bande passante inter-partition, etc.

Phase (b) : Synthèse inter-partition

Bloc G : Estimation du T.R.M. (T.E.M.)

L'estimation du paramètre de T.R.M. entre deux partitions successives est effectuée par l'algorithme de T.R.M. (bloc F). Toutes les valeurs de T.R.M. sont alors comparées à une valeur seuil τ_{TRM_AS} qui peut être soit prédéfinies par le concepteur, soit précalculée selon certains critères (blocs H et I). Dans notre cas, ce seuil est évalué selon la contrainte surfacique de l'unité de calcul reconfigurable (expression 3.61.

Bloc K : Exécution IPS

En fonction des résultats de comparaison (blocs H et I), une synthèse inter-partition est appliquée entre deux partitions initiales successives [La06]. Dans cette étape, sont mises en œuvre l'identification et la factorisation des UFs à instanciation multiple (§. 3.2.5) Ce processus de synthèse architecturale est effectué manuellement ou à l'aide d'un outil de synthèse par factorisation d'UFs (bloc K). Dans cette étape, les paramètres de caractérisation technologique cible spécifiés dans la librairie d'opérateur (bloc C) permet également l'évaluation des ressources de la

logique d'aiguillage et de contrôle nécessaires à la mise en œuvre d'une synthèse architecturale par UFs partagées appliquée aux partitions initiales.

Phase (c) : Partitionnement temporel réduit

En fonction des conditions spécifiées par l'équation 3.40, les optimisations par IPS du nombre de partitions initiales sont retenues ou non.

Bloc L : Validation IPS et raffinement des partitions

A partir des résultats du bloc K, le flot méthodologique IPS reconstruit de nouvelles partitions en remplaçant les couples de partitions successives initiales avec l'unité de traitement et de contrôle associée. Ensuite l'ensemble des partitions incluant les nouvelles partitions générées formant le partitionnement réduit sont re-numérotée selon un nouvel ordre respectant la contrainte d'ordonnancement (bloc L).

Bloc M : Vérification des contraintes respectées

Une vérification générale finale sur le respect des conditions de contraintes est réalisée (bloc L). Ces vérifications correspondent principalement aux respects des contraintes de surface du FPGA selon l'équation 3.10 et de temps d'exécution total selon l'équation 3.14. En effet, il s'agit de contrôler si les tailles surfaciques des nouvelles partitions sont supérieures ou non à la surface de l'unité de calcul reconfigurable (taille maximale de la matrice FPGA) ou si le temps d'exécution total intégrant les partitions IPS est supérieur à la contrainte de traitement de l'application. En fonction de ces conditions, le partitionnement final optimisé IPS sera ou non accepté (voir figure 3.17).

Enfin, après le processus de vérification soit examiné, les phases de placement - routage et génération de fichier de configuration (bitstream) des partitions sont réalisées avec les outils de technologie du FPGA cible considéré.

Développement Algorithmique de l'approche Méthodologique

Une exécution algorithme typique dans le flot de conception basé sur notre approche repose sur les étapes suivantes :

1. Définition des contraintes : contrainte de temps, dispositif de cible et technologie de cible.
2. Capture du DFG/CDFG de l'application.

3. Détermination des partitions initiales.
4. Estimation du paramètre auxiliaire T.R.M. (ou bien T.E.M.)
5. Application de la synthèse inter-partition.
6. Validation du partitionnement réduit par vérification des contraintes.
7. Résultat final de l'optimisation du nombre de partitions réduites.
8. Génération des fichiers de configuration de la conception finale basée RD-SA
   - Génération des unités de contrôle des partitions
   - Génération des floorplans des partitions réduites

Le processus du traitement ci-dessus nous permet de donner une solution progressive globale pour effectuer une implantation optimisée en reconfiguration par partitionnement et factorisation d'UFs de l'application visée. Le pseudo code du processus principal de l'algorithme mettant en œuvre notre approche méthodologique est décrit dans la figure 3.18 ci-dessous.

La fonction $TempPartitioning(int\, N_{TP}, int\, T_{maxexec}, int\, S_{ucr})$) réalise le partitionnement temporel initial. Cette fonction prend pour argument un graphe flot de données G et des contraintes spécifiées telle que le temps d'exécution total $T_{maxexec}$ autorisé, le temps de reconfiguration T_{reconf} spécifié par la technologie cible, la surface logique de l'unité de calcul reconfigurable (FPGA cible) S_{ucr} considérée éventuellement selon l'approche adoptée comme la taille de la surface optimale pour toutes les partitions. La fonction détermine alors les partitions préliminaires correspondant au nombre de partitions initiales N_{TP} comme entrée du processus d'optimisation inter-partition. Ce nombre devient alors le point de départ de la mise en œuvre algorithmique d'optimisation inter-partition ($N_{AS} = N_{TP}$, line 12 - figure 3.18).

Ensuite à chaque itération de l'algorithme correspondent une estimation du degré de synthèse inter-partition et une restructuration des unités fonctionnelles/opérateurs pour former des unités de calculs associées à des unités de contrôle mettant en œuvre la fusion par factorisation des UFs des partitions successives. Ce processus est réalisé pour toutes les combinaisons de partitions successives dans le but de réduire le nombre de partitions initiales de départ $N_{AS} = N_{TP}$.

La fonction $CalcMer(int\, Nf_i^{(p)}, int\, Nf_j^{(p)}, intNc_i, intNc_j)$ calcule le taux de ressemblance mutuelle $M_{i,j}$ entre deux partitions successives i et j. Les paramètres de la fonction $Nf_i^{(p)}$, $Nf_j^{(p)}$, Nc_i et Nc_j correspondent aux paramètres du T.R.M. défini en §. 3.2.4 et spécifié par l'équation 3.50. Ils correspondent respectivement aux nombres d'opérateurs de chaque unité fonctionnelle constituant les partitions i et j ; p est l'ensemble de UFs dans les partitions i et j. Nc_i et Nc_j sont respectivement les nombres d'opérateurs totaux dans les partitions i et j.

La figure 3.19 présente le pseudo code détaillé de l'algorithme mettant en œuvre la synthèse inter partition. Le taux de ressemblance mutuelle $M_{i,j}$ est calculé entre les deux partitions i et j et est mis en comparaison avec une valeur seuil τ_{TRM_AS}. Ce calcul permettant de valider l'exécution ou non de la phase de factorisation des tâches à instanciation multiple préalablement identifiées

```
1.  // begin main process
2.  int N_TP = 0; // initial the number of temporal partitions
3.  int N_AS = 0; // initial the number of partitions for architectural synthesis
4.  int Sucr = 0; // initial the FPGA surface for temporal partitioning
5.  TempPartitioning(int N_TP, int T_maxexec, int Sucr ) // temporal partitioning
6.  {...
7.    N_TP <= n
8.    T_maxexec <= Σ_{i=1}^{n} Tpi  +(n-1)T_reconf  // calculate the total execution time
9.    Sucr <= Max(Pi) // calculate the Fpga surface for partitions
10.    StartAS = true; // start the architectural synthesis
11.  } // end TempPartitioning
12.  N_AS = N_TP;
13.  LOOP B :  While( StartAS == true ) {
14.                        InitialForIPS ( int i, int j, Boolean Su ) { // initial for calculation of parameter MER
15.                            i =1;
16.                            j =N_AS;
17.                            t = 1;
18.                            Su = false; // initial synthesis using state
19.                            } // end InitialForIPS
20.            LOOP A : while  ( i !== n-1) {
21.                            CalcMer ( int Nf_i^(p), int Nf_j^(p), int Nc_i, int Nc_j ) // calculate the parameter MER
22.                            if M_i,j ≤ τ_mer_AS then
23.                                  { if ( (i, j)_t ∩ (i, j)_t-1) == 0  then { // does the current partitions have been used in last step
24.                                        tryIPS (N_CP )  // inter-partitioning synthesis
25.                                        Checkcondition ( )  // checking the constrained condition
26.                                        endofpartition (int i, Boolean Su, int N_AS, int N_CP, int N_P) } // if the last two successive .partitions
27.                                    else  {  NextPartition ( int i, int j, int t ) } } // to the next pair successive partitions
28.                             else {  NextPartition ( int i, int j, int t )
29.                                    endofpartition (int i, Boolean Su, int N_TP, int N_AS, int N_P)
30.                                  }
31.                            }
32.              }
33.  // end main process
```

FIG. 3.18 – Pseudo code du processus principal de l'algorithme IPS.

```
1.  if M_i,j ≤ τ_mer_AS then
2.        { if ( (i, j)_t ∩ (i, j)_t-1) == 0 then {
3.              // does the current partitions have been used in last step
4.              tryIPS (N_CP ) // inter-partitioning synthesis
5.              Checkcondition ( ) // checking the constrained condition
6.              endofpartition (int i, Boolean Su, int N_AS, int N_CP, int N_P) }
7.              // if the last two successive partitions
8.          else {
9.              NextPartition ( int i, int j, int t ) } }
10.             // to the next pair successive partitions
11. else {  NextPartition ( int i, int j, int t )
12.           endofpartition (int i, Boolean Su, int N_AS, int N_CP, int N_P) }
```

FIG. 3.19 – Pseudo-code de l'algorithme IPS.

```
1.  Checkcondition( )
2.  { if T_exc ≤ T_maxexc && area(P_ij) ≤ S_ucr then {
3.        Ncp = N_AS – 1;
4.        Su = true;
5.        endofpartition(int i, Boolean Su, int N_AS, int N_CP, int N_P)}
6.    else {
7.        NextPartition(int i, int j, int t)}
8.    }
```

FIG. 3.20 – Description de la fonction $Checkcondition()$.

et extraites selon l'algorithme développé est présenté en §. 3.2.5 entre les deux partitions i et j. Dans l'algorithme proposé, la valeur seuil τ_{TRM_AS} est prédéfinie par le concepteur. L'expression $((i,j)_t \cap (i,j)_{t-1}) == 0$ est une contrainte qui permet d'éviter la fusion entre partitions non successives.

La fonction $Checkcondition()$ (détaillée par la figure 3.20) de l'algorithme vérifie la validité de la nouvelle partition réduite. Le temps d'exécution total T_{exc} pour les partitions résultantes par une exécution IPS ne doit pas être supérieure à la contrainte temps $T_{maxexec}$. De même la surface $area(P_{i,j})$ doit être inférieure à la surface S_{ucr}. L'indicateur de la synthèses Su validera

```
1.  endofpartition ( int i, Boolean Su, int N_AS, int N_CP, int N_P )
2.  { if i == n-1 && Su = true then {
3.      N_AS <= N_CP;
4.      continue LOOP B;
5.    elsif i == n-1 && Su = false then {
6.      N_P <= N_CP;
7.      genecotroller (N_P) } } // generate the controller for each partition
8.    else {
9.      NextPartition ( int i, int j, int t ) }
10.   return;
11. }
```

FIG. 3.21 – Description de la fonction $endofpartition()$.

par une valeur booléenne true une partition réduite retenue. L'algorithme termine par la fonction $endofpartition(int\, i, Boolean\, S_u, int\, N_{TP}, int\, N_{CP}, int\, N_P)$, mettant à jour le partitionnement temporel réduit final. L'argument N_{CP} est le nombre de partitions qui est dans le processus de la synthèse. N_P est le nombre de partitions réduites finales.

3.3 Conclusion

Ce chapitre est structuré autour de la formalisation d'une modélisation combinée de la reconfiguration dynamique et de la synthèse architecturale. Nous avons présenté une méthodologie d'implémentation optimisée par combinaison du partitionnement temporel et du principe de réutilisation d'unités fonctionnelles partageables lors d'une conception sur technologie reconfigurable de type FPGA. Plus précisément, nous avons développé une nouvelle approche de conception permettant d'effectuer une synthèse architecturale entre deux partitions temporelles successives. La méthode proposée fusionne les tâches dans une méthode unifiée et globalement à tel point qu'une optimisation de ressource matérielle s'applique à la partition temporelle. L'originalité de cette proposition est une optimisation du nombre total de partitions (diminution) préalablement déterminé au cours d'un partitionnement temporel initial. L'objectif est d'aboutir à un bon compromis entre le temps d'exécution total, les temps de reconfiguration et les ressources d'implémentation.

La méthode d'optimisation consiste à estimer le niveau de ressemblance entre les différentes partitions initiales en vue de fournir une valeur quantitative et qualitative permettant d'intégrer une optimisation de synthèse architecturale basée unités fonctionnelles partagés. Un paramètre appelé " le taux de ressemblance mutuelle" évaluant le degré de factorisation des unités fonctionnelles (UFs) entre les partitions successives et l'applicabilité d'une synthèse architecturale a été proposé. Ce paramètre permet d'intégrer une phase de synthèse inter-partition (IPS) dans un flot méthodologique de conception. Grâce à notre proposition du taux de ressemblance, la méthode proposée permet une décision rapide de l'exécution ou non d'une optimisation de partitions par une solution de synthèse architecturale. Une fois le choix défini selon une valeur T.R.M estimé et comparé, une synthèse inter-partition est appliquée sur les partitions temporelles.

Nous avons proposé une évaluation préliminaire empirique fiable du taux de ressemblance ou d'exclusion mutuelle. D'une manière générale, il est difficile d'obtenir une estimation parfaite d'un tel critère, notamment à cause des différentes techniques possibles de synthèse architecturale par factorisation d'unités fonctionnelles partagées. C'est pourquoi des règles d'extraction des UFs d'un sous graphe modélisant deux partitions successives ont été proposées dans le but d'aboutir à une juste quantification du taux de ressemblance (**T.R.M**) entre ces partitions.

Nous avons également proposé et formalisé une méthode d'identification et d'extraction des unités fonctionnelles constituant un graphe flot de données selon des règles de priorités. Cela

en vue d'aboutir à des unités fonctionnelles à instanciation multiple permettant à la fois une évaluation juste du T.R.M et une optimisation " efficace " en terme de factorisation des UFs possibles. L'objectif est de trouver les UFS " identiques " possibles à partir des noeuds et selon les chemins de données dans les partitions successives considérées. Un algorithme mettant en œuvre cette détection et extraction de tâches synthétisables a été proposé, détaillé et illustré sur un exemple de graphe de flot de données d'un algorithme de détection de contours (§. 3.2.5). Les parties "identiques" entre deux partitions sont trouvées et peuvent être optimisées par une technique de factorisation. Il est évident, que les résultat d'une mise en œuvre d'une technique de factorisation par l'intégration d'une logique d'aiguillage et de contrôle sont très variables en fonction des choix du concepteur.

L'approche méthodologique formulée et proposée dans ce chapitre doit être exécutée, évaluée et validée sur des applications concrètes. C'est l'objet du chapitre suivant.

Chapitre 4

Application et Validation

4.1 Introduction

En vue de valider notre méthodologie d'optimisation d'un partitionnement temporel initial par synthèse architecturale basée sur la réutilisation d'unités fonctionnelles, nous mettons en œuvre notre approche sur deux applications test. La première application, de type "*chemin de données* ", consiste à un détecteur de contour pour traitement d'images. Pour l'évaluation de la robustesse de notre approche, il nous faut l'appliquer également sur un algorithme possédant un niveau de contrôle plus important qu'un algorithme typiquement "chemin de données". C'est pourquoi, le choix de la deuxième application de type mixte " *contrôle-chemin de données* ", s'est porté sur l'algorithme de chiffrement de donnée AES. L'objectif de ce choix est de mettre en évidence les besoins et considérations à prendre pour des algorithmes autres que tout "chemin de données" lors d'une implémentation en reconfiguration dynamique. Ensuite, il s'agit d'évaluer les conditions d'optimisation pour lesquelles la reconfiguration dynamique est une solution viable par rapport à d'autres approches telle que la réutilisation d'unités fonctionnelles.

Nous présentons l'application de notre approche méthodologique et les résultats expérimentaux des implémentations réalisées sur technologies FPGA Xilinx et Atmel, à partir d'un partitionnement temporel initial basé sur la méthode constructive par estimation et raffinement et son outil associé DAGARD [Ta03, Ba03, Bru04]. ***Notre approche n'est pas liée spécifiquement à cette méthode de partitionnement temporel initial. Elle peut être mise en œuvre avec tout autre méthodologie de partitionnement temporel comme entrée de notre approche d'optimisation*** [Cha98, ea03, Car03]. Une fois le partitionnement initial obtenu et créé, le processus de notre méthode d'optimisation est mis en œuvre à travers une analyse et le calcul du taux de ressemblance mutuelle (T.R.M) parmi toutes les partitions initiales préalablement déterminées. Nous présenterons les résultats expérimentaux des implantations réalisées. Les performances comme le temps d'exécution et la quantité de ressources utilisées sont décrits.

À partir de ces expérimentations, des solutions sont proposées concernant l'implémentation en reconfiguration dynamique mettant en œuvre le partitionnement temporel optimisé par synthèse architecturale selon une valeur de seuil prédéfinie du taux de ressemblance mutuelle permettant d'adapter une solution appropriée finale d'implémentation. Ensuite, une implémentation de type synthèse en réutilisation des unités fonctionnelles est réalisée. Ce type de synthèse entre les partitions successives permet de réduire le nombre de partitions total. Cette dernière permet une comparaison par rapport à l'implantation en reconfiguration dynamique sans l'optimisation de synthèse.

Enfin, nous conclurons sur les résultats obtenus selon l'approche proposée.

4.2 Application I : Détection de contour d'images

4.2.1 Présentation générale

L'application initiale consiste à un algorithme de détection de contours pour le traitement d'images par calcul de gradient. L'algorithme calcule le gradient de l'intensité de chaque pixel. Ceci indique la direction de la plus forte variation du clair au sombre, ainsi que le taux de changement dans cette direction. On connaît alors les points de changement soudain de luminosité, correspondant à des contours d'images, ainsi que l'orientation de ces bords.

La détection de contour par un calcul de gradient d'images est basée sur un filtre médian 3×3 suivie par un détecteur de Sobel 3×3. Le schéma général est représenté sur la figure 4.1. Dans cet exemple nous considérons un filtre médian séparable [Dem91]. C'est un opérateur de filtrage non linéaire. Ce filtre permet d'éliminer les bruits impulsionnels tout en préservant la qualité des contours d'une image. En pratique, nous calculons d'abord la valeur du médian horizontal de trois pixels successifs (figure 4.2). Ensuite la valeur des médians verticaux des trois valeurs des médians horizontaux successifs. Les retards lignes, implantés comme FIFOs ou par lecture de la mémoire trois fois plus rapidement que le traitement, ne sont pas pris en compte ici. Cet opérateur est globalement constitué de comparateurs et de multiplexeurs 8 bits.

Le calcul du gradient est réalisé par un opérateur de Sobel. L'algorithme de Sobel est un opérateur utilisé en traitement d'image pour la détection de contours. Le traitement correspond à la convolution de l'image par l'application successive de deux filtres mono dimensionnels sur les lignes et colonnes de la fenêtre de traitement correspondant respectivement au gradient vertical (First Sobel) et horizontal de Sobel (Second Sobel) (figure 4.3). Ils constituent la structure séparable du masque de Sobel. La première partie de la décomposition de l'opérateur de Sobel est calculée et retardée pour calculer la seconde partie (figure 4.1).

Le gradient final, qui sera affecté au pixel central de la fenêtre de traitement, correspond au maximum des valeurs absolues du gradient vertical et horizontal (figure 4.4). Cela est une

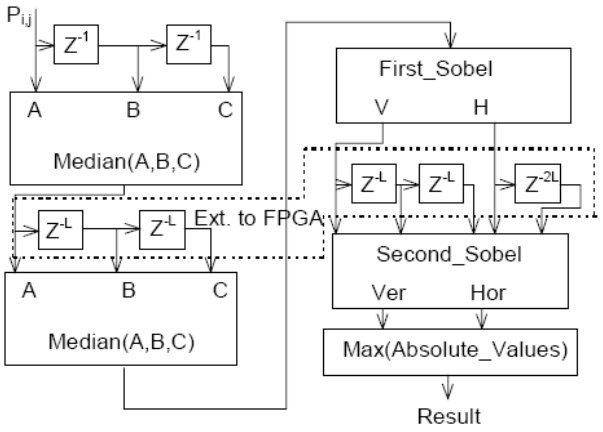

FIG. 4.1 – Schéma général du chemin de données d'un gradient images.

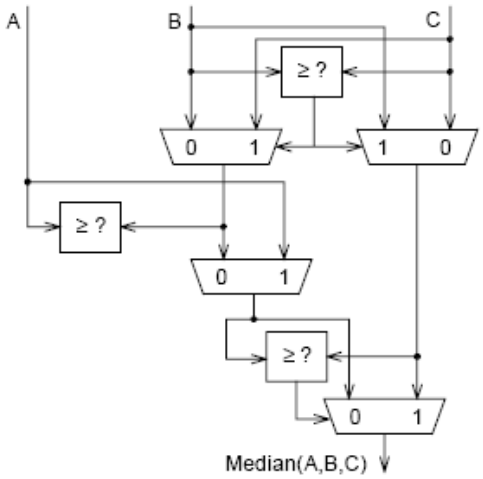

FIG. 4.2 – Détail de la fonction Médian (A, B, C).

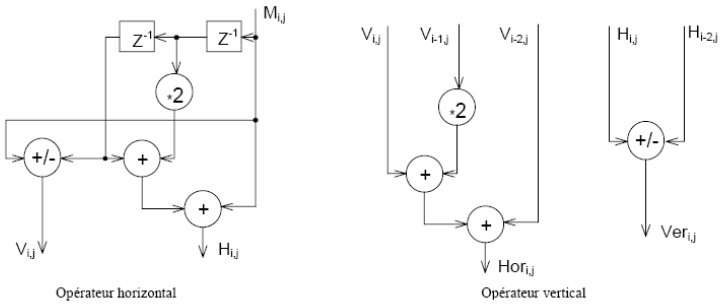

FIG. 4.3 – Opérateur horizontal et vertical de Sobel.

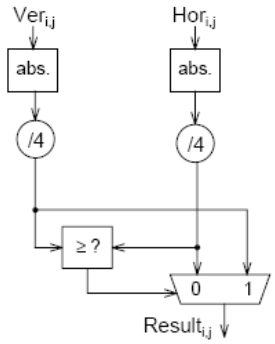

FIG. 4.4 – Détail de la fonction maximum de la valeur absolue.

approximation de l'amplitude du gradient.

4.2.2 Partitionnement temporel initial

L'illustration d'approche d'optimisation par synthèse architecturale inter-partition, est démontré avec un partitionnement temporel initial basé sur le critère d'optimisation surface respectant une contrainte temporelle (voir chap. 2 §. 2.2.2.4.3). Son fonctionnement repose sur une méthode de partitionnement qui recherche le découpage d'une application en un maximum de sous applications de tailles homogènes[Ta03]. Il délivre un nombre de partitions **n** réalisables sous une contrainte de temps donnée. Connaissant **n**, deux stratégies sont possibles. La première minimise la surface nécessaire à l'application en créant des partitions homogènes et tente ensuite de minimiser les besoins en bande passante induite par le découpage. La seconde a le fonctionnement inverse : minimisation de la bande passante mémoire, puis un raffinement tente d'homogénéiser les surfaces des différentes partitions.

En pratique, l'implantation des partitions impose la condition : $\underline{\mathbf{n}} \in IN$. À partir de l'analyse de la valeur de **n**, nous pouvons extraire certaines informations sur les paramètres nécessaires du système permettant l'implémentation du graphe flot de données G. Nous décrivons ces différents cas ci-dessous :

1. $\underline{\mathbf{n} > 1}$: Il est possible de réaliser un partitionnement en reconfiguration dynamique. Dans ce cas, une optimisation est possible et permet une réduction de la surface logique de l'implémentation avec la technologie utilisée. **n** correspond au nombre de partitions que l'on est sûr d'obtenir.

2. $\underline{\mathbf{n} \leq 1}$: Dans ce cas, un partitionnement en reconfiguration dynamique ne permet pas de respecter la contrainte de temps. Ici, seule une implémentation statique avec ou sans

parallélisme de traitement, ou bien avec ou sans optimisation par synthèse architecturale répondra à la contrainte de temps.

Si $\mathbf{n} = 1$: Seul une implémentation statique est suffisante avec ou sans optimisation par synthèse architecturale pour respecter les contraintes avec la technologie FPGA utilisée.

Si $\mathbf{n} = 0$: Pour assurer les contraintes, il est nécessaire de réaliser une implémentation statique avec un parallélisme de traitement. $1/x$ donne le degré de parallélisme. En pratique $1/x \in IN$. Dans ce cas, il s'agit de savoir si le choix de la cible technologique était judicieux. Il se pourrait qu'une cible à grain plus épais [Sa02, RDS02] ou à grain mixte (par exemple FPGA virtex intégrant des fonctions arithmétiques figées) soient plus adaptées au traitement considéré.

Par la suite, nous nous situerons dans le premier cas ($\mathbf{n} > 1$), avec pour objectif d'introduire en complément d'un partitionnement reconfiguré, une optimisation par synthèse architecturale.

DAGARD (Découpage Automatique de Graphes pour Architectures Reconfigurables Dynamiquement) est un outil automatisant (voir chap. 2 § 2.2.2.5) la méthodologie de partitionnement temporel initial adopté pour notre illustration [Ba03, Bru04]. Cet outil permet de réaliser le découpage d'une application, représentée sous forme d'un graphe flot de données (GFD), en plusieurs sous-applications successivement exécutées sur une surface logique optimisée par reconfiguration dynamique. Il peut aussi calculer la bande passante et le nombre de cellules (ressources logiques) par étage d'opérateurs du graphe flot de données de l'application à implémenter.

DAGARD s'insère dans le flot de conception d'une application (voir figure 2.11). L'objectif du DAGARD est de trouver la meilleure utilisation des ressources nécessaires à l'application en mettant en œuvre la Reconfiguration Dynamique et en visant différents objectifs :

- Minimisation des ressources logiques nécessaires à une implantation sous contrainte de temps.
- Minimisation de la bande passante lors de l'utilisation de la RD.

Au final, DAGARD permet de spécifier les caractéristiques d'une architecture SoC reconfigurable nécessaire pour l'implantation d'un algorithme particulier. La figure 4.5 montre la fenêtre du DAGARD[Bru04].

4.2.2.1 Modélisation et caractérisation technologique

A partir de la technologie utilisée, les paramètres tels que la consommation en terme de ressources logiques, les temps d'exécution des opérateurs de l'algorithme, la vitesse de reconfiguration de la technologie FPGA utilisée ont été évalués [Ta03, Liu04] et intégrés dans l'outil DAGARD. L'objectif ici est de modéliser à la fois :

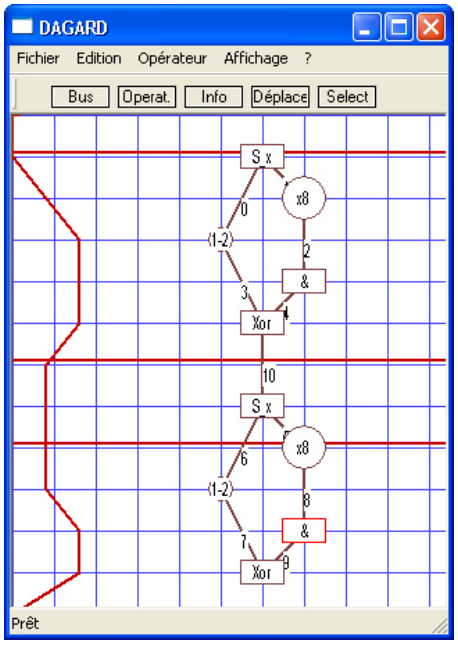

FIG. 4.5 – La fenêtre de saisie GFDC de l'outil DAGARD [Bru04].

– Les surfaces logiques des noeuds v_i présents physiquement dans l'unité de calcul reconfigurable sont formulées selon l'équation 3.7.

– Les temps de traversée des noeuds et de propagation des données entre les noeuds formulés par l'équation 3.18 en vue d'en déduire une évaluation de la fréquence de travail et la latence du chemin critique (prologue pipeline) associée à chaque partition et exprimée par l'équation 3.19.

Les technologies AT40K de ATMEL et Virtex de Xilinx permettent de réaliser de la reconfiguration totale ou partielle. Elle offre la possibilité pour chaque nouvelle configuration de générer les données de configuration limitées aux ressources exploitées. La table 4.1.suivante donne un exemple d'estimation de vitesse de configuration de circuit Atmel et Xilinx [Atm, Xil07b, Liu04].

TAB. 4.1 – Vitesses de reconfiguration des circuits FPGAs AT40k et Virtex XCV200E.

Famille de circuit	Possibilité de RD partielle	Tailles (Portes)	Délai de reconfiguration (ms)	Vitesse de reconfiguration (Mportes/s)
AT40K(ATMEL)	Oui	45000	0,54	83,39
Virtex(Xilinx)	Oui	63504	2,67	23,81

L'étude de la structure des cellules logiques de la technologie utilisée permet l'évaluation du nombre de cellules nécessaires pour réaliser chaque opérateur du GFD. Cela permet d'évaluer les ressources logiques nécessaires pour chaque étape de l'application à partir du graphe flot de données. Une estimation des performances et des ressources a été réalisée pour la technologie FPGA Atmel [Tan01]. Ces estimations ont été réalisées à partir de l'architecture des cellules logiques de la technologie considérée et des différents modes de configuration possible d'une cellule logique. Les tables 4.2 et 4.3 rappellent quelques estimations d'opérateurs arithmétiques et logiques. L'outil DAGARD utilise ces caractérisations pour la détermination des partitions possibles d'un algorithme en vue d'une implémentation sur la technologie FPGA Atmel AT40K [Atm].

TAB. 4.2 – Estimations des ressources d'opérateurs logiques et arithmétiques sur technologie FPGA Atmel AT40K.

Opérateur D bits synchronisé	**Nombre de cellules**
Multiplication ou division par 2^k	0
Additionneur ou soustracteur	D+1
Multiplexeur	D
Comparateur	2D
Valeur absolue	D
Registre de synchronisation supplémentaire	D
Opérateur logique	D

*D est la taille maximale de données à traiter

TAB. 4.3 – Temps d'exécution des opérateurs synchronisés (FPGA AT40k).

Opérateur D bits	**Temps d'exécution**
Multiplication ou division par 2^k	0
Additionneur ou soustracteur	D*(T_C + T_R) +T_{Setup}
Multiplexeur	T_C + T_{Setup}
Comparateur	(2*D-1)*(T_C) + 2*T_R+T_{Setup}
Valeur absolue	D*(T_C + T_R) + T_{Setup}
Registre de synchronisation supplémentaire	T_C + T_{Setup}
Opérateur logique	T_C + T_{Setup}

*D est la taille maximale de données à traiter, T_C =1,7 ns temps de traversée d'une cellule logique
T_R = 0,17 ns temps inter-routage entre cellule, T_{Setup} = 1,5 ns temps de pré-positionnement des registres

Nous avons réalisé une estimation de certains opérateurs arithmétiques et logiques sur la technologie FPGA Xilinx. Cette dernière utilise une structure de cellule logique différente de celle de Atmel. Dans le FPGA Atmel les plus petits éléments sont les cellules (figure 4.6.a).

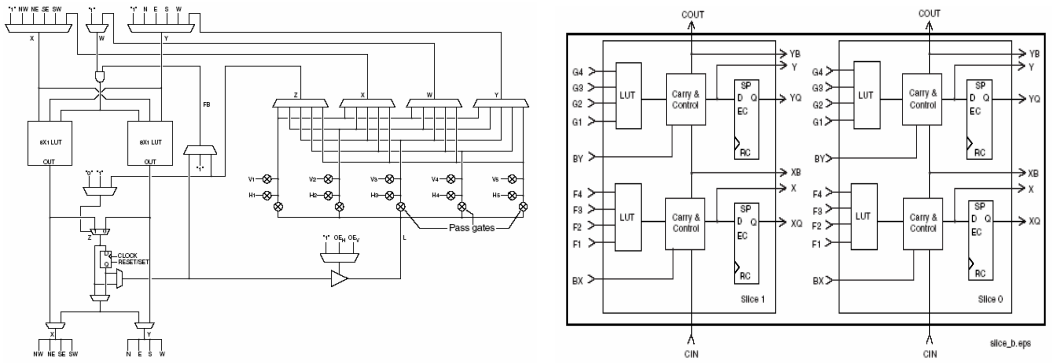

FIG. 4.6 – (a) La cellule de Atmel (b) CLB Virtex composé de 2-slice de Xilinx.

La structure d'une cellule logique de la technologie Atmel est composée de deux LUT à trois entrées et une sortie qui peut être vue comme une seule LUT à quatre entrées [Atm]. Avec la technologie Xilinx, on dispose d'une autre structure de cellule logique (CLB) constituée de deux slices eux-mêmes constitués de deux LUT à quatre entrées (figure 4.6.b) [Xil07b].

A partir de la structure d'un CLB, nous pouvons estimer les ressources et performances d'opérateurs logiques et/ou arithmétiques. Par comparaison, pour réaliser un même opérateur tel qu'un multiplexeur 4 bits, avec la technologie Atmel il faut 4 cellules logiques contre seulement 1 CLB avec la technologie Xilinx. Les tables 4.4 et 4.5 donnent les estimations avec la technologie FPGA Virtex en termes de temps d'exécution et de ressources d'opérateurs logiques et arithmétiques synchronisés ou non.

TAB. 4.4 – Estimation des ressources d'opérateurs logiques et arithmétiques sur technologie Virtex.

Opérateur D bits synchronisé	**Nombre de cellules**
Multiplication ou division par 2^k	0
Additionneur ou soustracteur	(D/4)+1
Multiplexeur	D/4
Comparateur	D/2
Valeur absolue	D/4
Registre de synchronisation supplémentaire	D/4
Opérateur logique	D/4

*D est la taille maximale de données à traiter

Ces caractérisations ont été intégrées dans l'outil DAGARD afin de permettre une estimation automatique des fréquences et des ressources d'un partitionnement temporal initial en vue d'une implémentation sur la technologie FPGA Virtex.

TAB. 4.5 – Temps d'exécution des opérateurs synchronisés (FPGA Virtex).

Opérateur D bits	**Temps d'exécution**
Multiplication ou division par 2^k	0
Additionneur ou soustracteur	(D/4)*(T_C + T_R) +T_{Setup}
Multiplexeur	T_C + T_{Setup}
Comparateur	((D/2)-1)*(T_C) + 2*T_R+T_{Setup}
Valeur absolue	(D/4)*(T_C + T_R) + T_{Setup}
Registre de synchronisation supplémentaire	T_C + T_{Setup}
Opérateur logique	T_C + T_{Setup}

*D est la taille maximale de données à traiter, T_C =0,7 ns temps de traversée d'une cellule logique
T_R = 10 ns temps inter-routage entre cellule, T_{Setup} = 0,6 ns temps de pré-positionnement des registres

4.2.2.2 Caractérisation et annotation du GFD

Le partitionnement initial par l'outil DAGARD, s'effectue dans un premier temps par une annotation et une caractérisation du graphe flot de données de l'algorithme de détection de contours en fonction de la cible technologique FPGA [Ta03, Ba03]. Cette caractérisation et annotation permettent d'estimer les fréquences et ressources nécessaires pour l'exécution des partitions initiales possibles d'un algorithme. En effet, le temps d'exécution d'une partition dépend à la fois du temps d'exécution de l'opérateur le plus lent et du routage entre les différents opérateurs ou cellules exploitées dans cette partition. DAGARD permet une caractérisation en terme de performances (temps d'exécution des opérateurs synchronisés) et des ressources des opérateurs arithmétiques et logiques pour la technologie Atmel et Xilinx [Liu04, Bru04]. La figure 4.7. illustre la phase d'annotation et caractérisation par DAGARD du GFD du détecteur de contour pour la technologie AT40K d'indice de vitesse -2. Les fonctions médianes sont réalisées sur des opérateurs 8 bits. Nous avons des multiplexeurs 8 bits qui consomment 8 Cells et sont exécutés en 5 ns, des comparateurs 8 bits qui consomment 16 Cells et sont exécutés en 41 ns. La première partie de Sobel nécessite des registres 8 bits qui consomment 8 Cells et sont exécutés en 4 ns. Les multiplieurs et diviseurs par puissance de deux sont uniquement basés sur du routage et ne consomment aucune cellule. Les additionneurs/soustracteurs 8 bits consomment 9 Cells et sont exécutés en 24 ns. Les additionneurs 9 bits consomment 10 Cells et sont exécutés en 27 ns. Dans la seconde partie de l'algorithme de détection de contours (calcul de composante horizontale de Sobel), le seul nouveau opérateur est un additionneur/soustracteur de taille 10 bits qui consomme 11 Cells et est exécuté en 30 ns. Dans la fonction valeur absolue maximale, les nouveaux opérateurs sont des opérateurs de calcul de la valeur absolue de taille 11 bits qui consomment 10 Cells et sont exécutés en 29 ns.

4.2.2.3 Estimation performances et ressources

En sommant les ressources des opérateurs du GFD tout en prenant en compte de l'accroissement de la taille des données au cours du calcul, DAGARD estime une surface totale de cellules logiques et une fréquence maximale de fonctionnement qui dépendent de l'opérateur le plus lent et des propagations de routage entre les opérateurs (délai inter-opérateur). Par exemple, à taille identique, les comparateurs sont les opérateurs les plus lents de graphe flot de données du détecteur de contours (figure 4.7). DAGARD calcule ensuite le nombre de partitions possibles en considérant la taille du bloc de données à traiter et une contrainte de traitement. La table 4.6. résume l'étape de caractérisation et de calcul préliminaire du partitionnement initial par l'outil DAGARD sur technologie AT40K du détecteur de contours dans le cas d'un bloc de données de taille image de 512×512 pixels (codé en niveaux de gris sur un octet) et en considérant une contrainte de temps de traitement correspondant à une contrainte de temps image de 40 ms [Ta03, Bru04].

TAB. 4.6 – Estimation des ressources d'implantation en RD du détecteur de contours.

Nombre total de ressources	Temps d'exécution critique (ns)	Nombre de partitions	Ressources / partition	Durée de reconfiguration / partition (us)
467	41	3	156	114

4.2.2.4 Partitionnement temporel initial

De l'étape précédente, DAGARD propose un partitionnement initial homogène du GFD (approximativement le même nombre de cellules logiques par partition correspondant à une taille optimale). Le graphe flot de données du détecteur de contours est alors parcouru jusqu'à obtenir la taille optimale qui correspond à la surface minimale suffisante. En pratique, DAGARD permet de modifier manuellement le partitionnement proposé dans le but de prendre en compte la bande passante mémoires de stockage des données inter-partition [Bru04]. En fonction du bloc de données images à traiter et de la contrainte de temps imposée, nous obtenons un partitionnement initial composé de trois partitions pour le détecteur de contours. Les tables 4.7 et 4.8 récapitulent les résultats d'implantation du partitionnement initial proposé par DAGARD respectivement sur les technologies Atmel At40K dans le cas d'un bloc de données 512×512 et Xilinx Virtex dans le cas d'un bloc de données 1024×1024. Dans les deux cas, nous avons volontairement considéré un facteur de gain en densité fonctionnelle de 3 [Wir97].

La figure 4.8 illustre le partitionnement initial du détecteur de contours composé de trois partitions initiales.

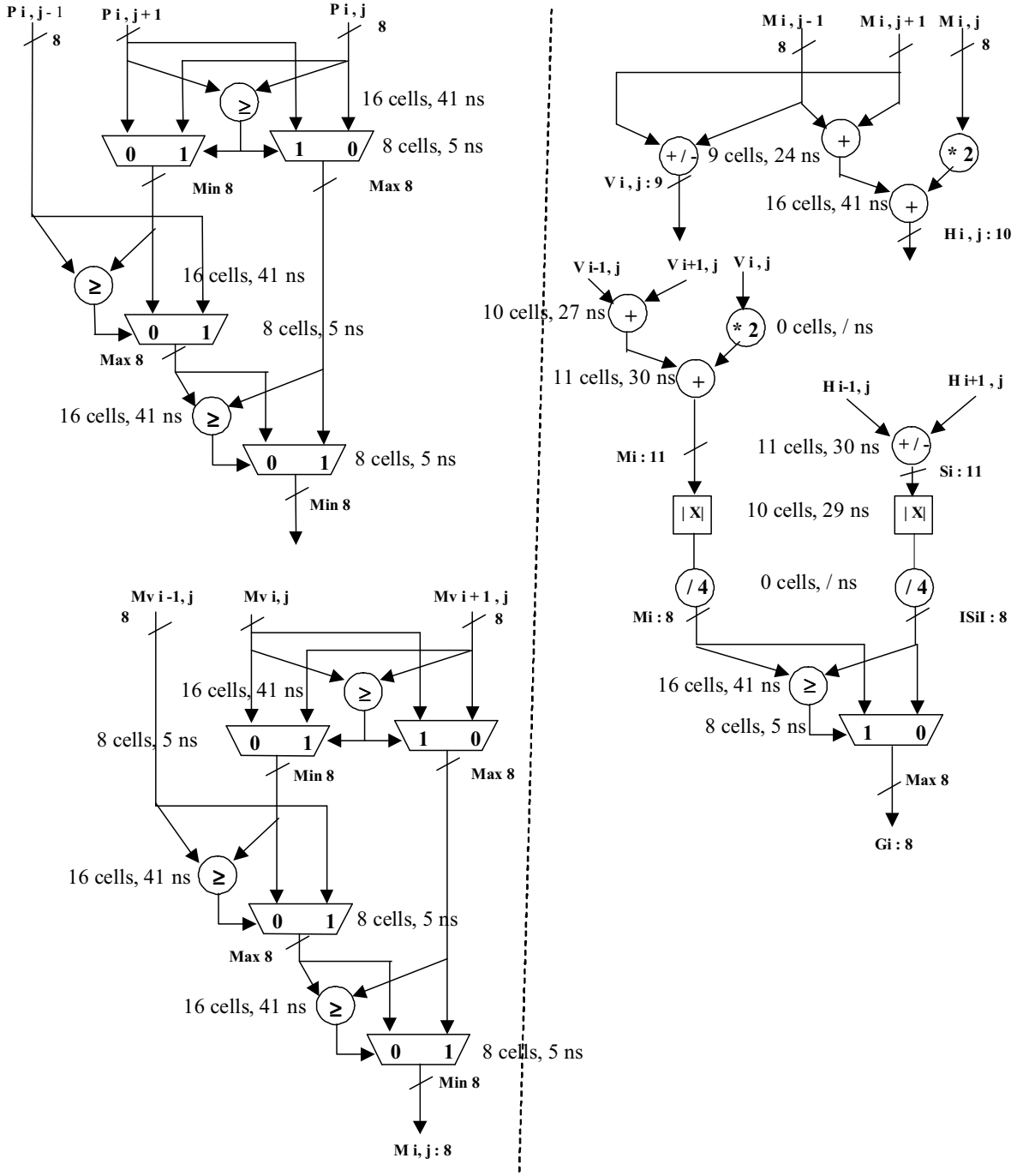

FIG. 4.7 – GFD annoté et caractérisé du chemin de données du détecteur de contours.

TAB. 4.7 – Résultats d'implantation sur AT40K du détecteur de contours pour image 512^2.

Partition	Nombre de cellules	Fréquence de travail (Mhz)
1	152	24, 9
2	156	24, 8
3	159	27, 2

TAB. 4.8 – Résultats d'implantation sur VIRTEX du détecteur de contours pour images 1024^2.

Partition	Nombre de cellules (CLB)	Fréquence de travail (Mhz)
1	44	95, 7
2	68	95, 7
3	49	86, 2

4.2.3 Optimisation par synthèse architecturale inter-partitions

4.2.3.1 Taux de ressemblance mutuelle

La table 4.9. recense et caractérise les principaux opérateurs bas niveaux (multiplexeur, comparateur, registre, additionneur, soustracteur, diviseur, valeur Absolue, etc.) pour l'ensemble des trois partitions initiales du détecteur de contours. La table 4.10. spécifie et caractérise les principales unités fonctionnelles (UFs) partagées possibles dans les trois partitions considérées. Quatre UFs sont déterminées, identifiées et annotées selon la méthode d'identification et d'extraction d'unités fonctionnelles factorisables développée et présentée en chapitre 3 (voir Chap. 3 §. 3.2.5). La figure 4.9 illustre le résultat de la phase d'identification et d'extraction des unités fonctionnelles partagées de l'algorithme de détection de contours.

TAB. 4.9 – Caractérisation des opérateurs de l'algorithme de détection de contours.

Opérateur	Partition 1*	Partition 2	Partition 3
Multiplexeur	4	4	1
Comparateur	4	2	1
Registre	1	4	4
Additionneur	0	2	2
Multiplieur	0	1	4
Soustracteur	0	1	1
Diviseur	0	0	2
Valeur Abs	0	0	2

*Partition et ci-après part.

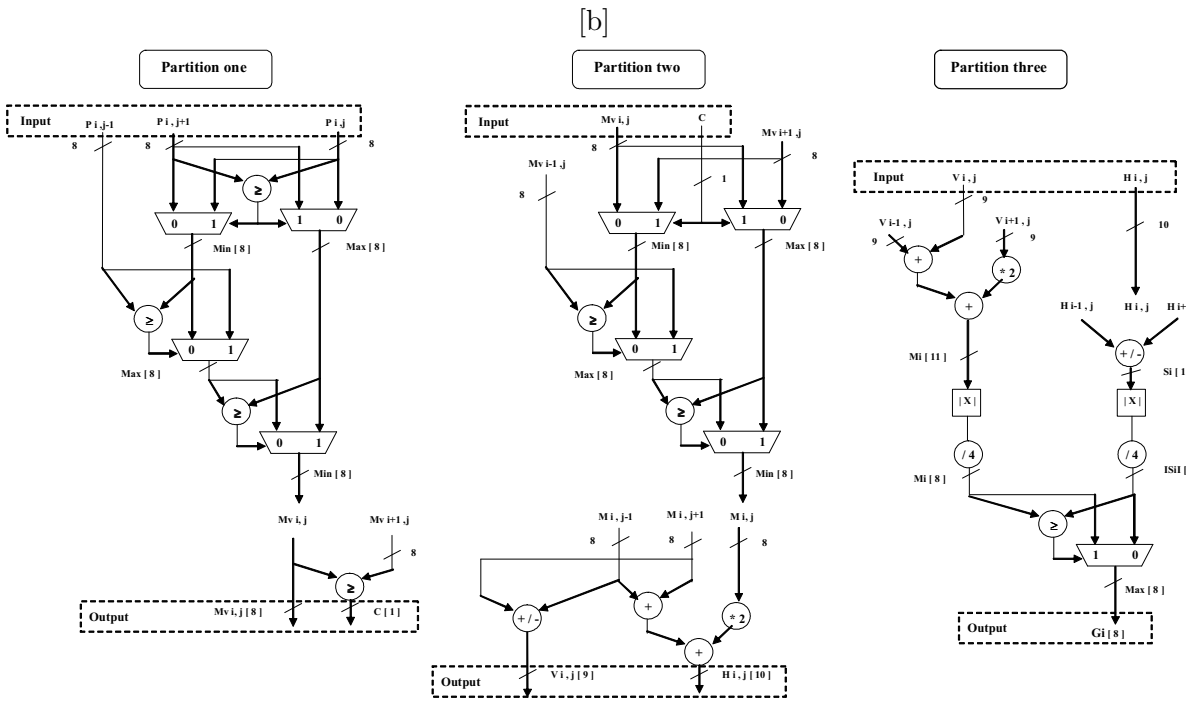

FIG. 4.8 – Partitionnement temporel initial du détecteur de contours.

TAB. 4.10 – Identification, extraction et annotation des UFs partagées du détection de contours.

Opérateur (UF)	Partition 1	Partition 2	Partition 3
Macro 1 (UF)	1	1	0
Macro 2	2	2	1
Macro 3	0	1	1
Macro 4	0	0	2

Cette caractérisation et la phase d'extraction des UFs factorisables nous permettent de calculer le taux de ressemblance mutuelle (T.R.M) à partir de l'équation 3.50 du chapitre précédent en vue d'une évaluation du degré d'optimisation du partitionnement initial par synthèse architecturale inter-partition. La table 4.11 détaille les résultats de calcul du T.R.M appliqués sur les partitions initiales du détecteur de contour.

4.2.3.2 Optimisation par synthèse architecturale inter-partition

Les valeurs calculées des taux de ressemblance mutuelle inter-partition, montre une ressemblance fonctionnelle entre les partitions 1 et 2. Tandis que les partitions 3 et 1 présentent la plus forte exclusion. En effet, on constate que la partition initiale 2 intègre quasiment la partition initiale 1 en terme de fonctionnalité et d'opération de calcul (figure 4.8). Cette analyse permet

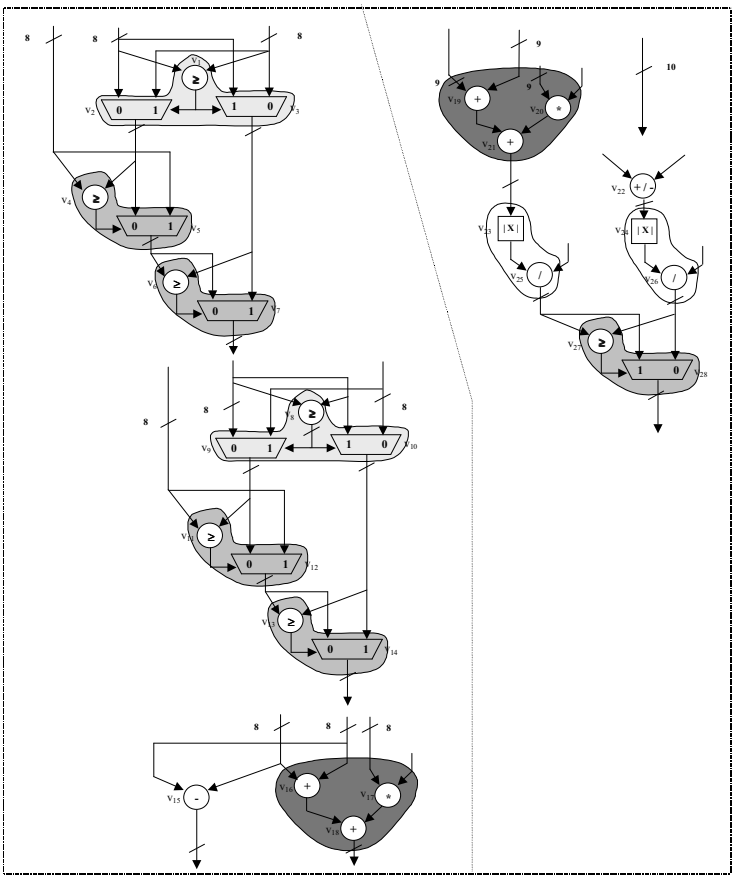

FIG. 4.9 – Les unités fonctionnelles du détecteur de contours.

TAB. 4.11 – des Taux de ressemblance Mutuelle Inter-partition du détecteur de contours.

TRM_{p1-p2}	TRM_{p2-p3}	TRM_{p3-p1}
56 %	47 %	27 %

d'envisager une optimisation du nombre de partitions initiales de l'algorithme de détection de contours. L'application d'une synthèse architecturale inter-partition basée sur la factorisation des unités fonctionnelles semble donc judicieuse entre les partitions 1 et 2. A l'issue, ces deux partitions se trouvent fusionnées en une seule partition réduite. Au final, l'ensemble des partitions réduites permettant de réaliser globalement l'algorithme de détection de contours avec un nombre de deux partitions réduites P'_1 et P'_2 correspondantes à :

$$P'_1 = AS(P_1, P_2) \qquad et \qquad P'_2 = P_2 \tag{4.1}$$

Notre approche propose donc une réduction de trois partitions initiales en deux partitions réduites. La figure 4.10 représente le partitionnement temporel final résultant de cette optimisation par une synthèse architecturale inter-partition manuelle et donne les résultats d'implémentation sur la technologie Xilinx Virtex des partitions réduites de la détection de contours.

TAB. 4.12 – Partitionnement temporel optimisé du détecteur de contours.

Partition	Nombre de cellules (CLB)	Fréquence de travail (Mhz)
1	99	88, 2
2	49	86, 2

4.2.3.3 Discussion et analyse

Le procédé de factorisation de synthèse architecturale repose sur le remplacement de toutes les unités fonctionnelles identiques par une seule UF associée à une logique d'aiguillage de données pilotée et commandée par un contrôleur logique. C'est principalement l'ajout d'une logique de multiplexage qui permet cette réutilisation des unités fonctionnelles communes. Selon le niveau de grain des unités fonctionnelles, la logique de multiplexage peut devenir contraignante et coûteuse en terme de surface logique par rapport à la surface même des UFs. En effet, l'augmentation des ressources logiques dépendant principalement de la différence entre les grains de calcul des UFs et l'ajout d'un complément de la logique d'aiguillage - contrôleur.

Concernant l'application de détection de contours, le gain en ressources obtenu dans la factorisation des UFs ne compense pas les ressources de la logique d'aiguillage et de contrôle mettant

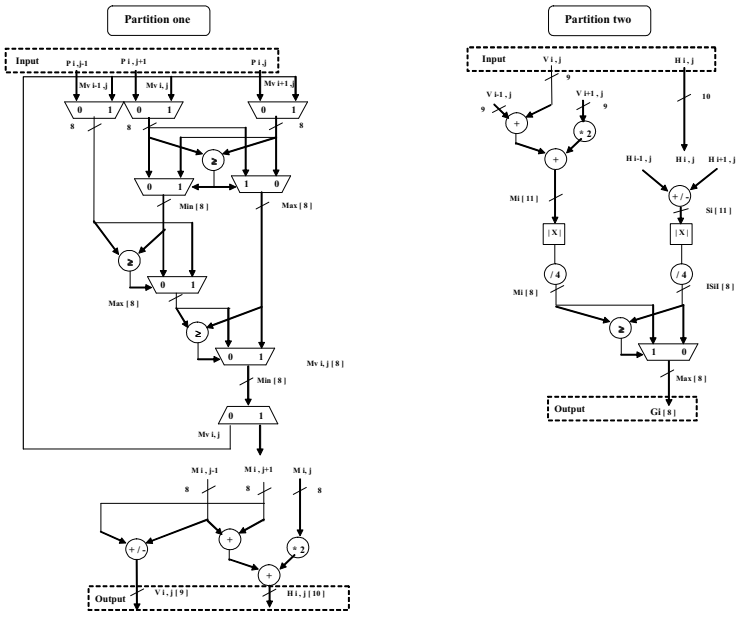

FIG. 4.10 – Partitionnement temporel optimisé du détecteur de contours.

en œuvre la mutualisation des UFs factorisables. En effet, nous avons obtenu les surfaces optimales $S_{ucr} = 68$ et $S'_{ucr} = 99$ CLB pour l'implémentation des partitions initiales et réduites respectivement. La réduction du nombre de partitions initiales à 2 grâce à la synthèse architecturale inter partition a engendré une augmentation de ressources en terme de surface logique de 45%. Cependant la surface optimale S'_{ucr} demeure inférieure à la somme des surfaces logiques des deux surfaces initiales 1 et 2. On peut donc considérer que plus les UFs sont de types "gros grains" plus l'optimisation des ressources des UFs communes supprimées par l'étape de factorisation est importante par rapport aux ressources supplémentaires nécessaires de la logique de contrôle et d'aiguillage mettant en œuvre l'optimisation architecturale des partitions réduites.

Cependant, bien que dans certain cas, la surface logique optimale des partitions combinées par IPS est augmentée, les temps de reconfiguration et de traitement global sont réduits par le nombre final de partitions réduites.

Si on considère le temps de reconfiguration proportionnel à la surface optimale S_{ucr} [Ta03], nous évaluons le temps de reconfiguration maximal T_{Rec} en fonction de la surface optimale par l'équation suivante :

$$T_{Rec} = n \cdot \Gamma_{reconf} = n \cdot \alpha \cdot S_{ucr} \tag{4.2}$$

Où n est le nombre de partitions réduites, Γ_{reconf} est le temps de reconfiguration d'une partition et α un coefficient de proportionnalité lié à la vitesse de reconfiguration (nombre de cellule/unités de temps) de la technologie considérée. Nous pouvons déduire le taux de réduction

des temps de reconfiguration ΔR_{rec} défini par l'expression suivante :

$$\Delta R_{rec} = \frac{T_{Rec} - T'_{Rec}}{T_{Rec}} \tag{4.3}$$

où T_{Rec} et T'_{Rec} sont respectivement les temps globaux de reconfiguration dans le cas du partitionnement temporel initial et réduit. En effet, en considérant d'une part le taux relatif absolu (augmentation ou diminution) de la surface logique optimale entre le partitionnement initial et réduit $\Delta S'_{ucr}$ tel que :

$$\Delta S_{ucr} = \frac{S'_{ucr} - S_{ucr}}{S_{ucr}} \tag{4.4}$$

D'autre part, l'expression du temps de reconfiguration global du partitionnement réduit par l'équation suivant :

$$T'_{rec} = n' \cdot \alpha \cdot S_{ucr} \cdot (1 + \Delta S_{ucr}) \tag{4.5}$$

où n' le nombre de partitions réduites. Le taux de réduction des temps de reconfiguration ΔR_{rec} peut être déduit et exprimé à partir des équations 4.2, 4.3 et 4.5 par l'expression suivante :

$$\Delta R_{rec} = 1 - \frac{n'}{n} \cdot (1 + \Delta S_{ucr}) \tag{4.6}$$

où n est le nombre de partitions initiales. Dans ce cas, l'exécution du partitionnement optimisé permet une diminution du temps des phases de reconfiguration de 3% en comparaison avec une exécution du GFD de l'algorithme de détection de contour par le partitionnement temporel initial. Nous obtenons une faible réduction étant donnée les surfaces logiques mises en jeu.

De même, pour la réduction du temps de traitement total. À partir de l'expression du temps d'exécution globale définie par l'équation 3.9 en fonction du temps de reconfiguration maximal T_{Rec} :

$$\Gamma_{total} = n \cdot \Gamma_{execuPn} + T_{rec} \tag{4.7}$$

Nous définissons le gain de réduction du temps d'exécution global de traitement ΔT_{Exe} de l'application par partitionnement réduit par rapport à un partitionnement initial par l'expression suivante :

$$\Delta T_{Exe} = \frac{\Gamma_{total} - \Gamma'_{total}}{\Gamma_{total}} \tag{4.8}$$

où Γ_{total} et Γ'_{total} sont respectivement les temps globaux de traitement dans le cas du partitionnement temporel initial et réduit. Si nous supposons d'une part une fréquence minimum de $65,1$ Mhz pour l'exécution des partitions initiales ou réduites, nous pouvons exprimer ce gain de réduction par l'équation suivante :

$$\Delta T_{exe} = 1 - \frac{n'}{n} \cdot (1 + k \cdot \Delta R_{rec}) \tag{4.9}$$

avec

$$k = \frac{\frac{f \cdot T_{rec}}{n' \cdot N}}{1 + \frac{f \cdot T_{rec}}{n \cdot N}} \tag{4.10}$$

où N est le bloc de données à traiter par partition. D'autres part, à partir de la condition suivante généralement vérifiée :

$$\Gamma_{execuPn} = \frac{N}{f} \gg \Gamma_{reconf} \tag{4.11}$$

nous déduisons alors, à partir des expressions 4.9 et 4.10 un gain maximal de réduction du temps de traitement équivalent à :

$$\Delta T_{exe\_max} \approx 1 - \frac{n'}{n} \tag{4.12}$$

En effet, l'approche d'optimisation d'un partitionnement temporel par IPS conduit à une réduction du nombre de partitions initiales aboutissant à un gain dépendant principalement des nombres de partitions initiales et réduites. Pour notre application de détection de contour, l'exécution du partitionnement optimisé nous permet d'obtenir une diminution du temps d'exécution maximal de 33% en comparaison avec le temps de traitement du GFD de l'algorithme par le partitionnement temporel initial. Cela nous permet d'aboutir à un compromis entre l'augmentation de la surface logique reconfigurable minimum (en CLB) et le temps d'exécution globale de l'application. En effet, nous avons au final un gain en terme de temps de traitement de plus de 32% au détriment d'un surcoût en terme de ressources logiques par rapport au partitionnement initial de 45%.

4.3 Application II : Algorithme de chiffrement de donnée “ AES ”

4.3.1 Présentation générale

L'algorithme AES (Advanced Encryption Standard) Rijndael est un standard de cryptage symétrique [JV03] destiné à remplacer le DES (Data Encryption Standard) qui est devenu trop faible (cryptage fonctionnant avec des clés de 56 bits) au regard des besoins de cryptage actuels. Cet algorithme possède les caractéristiques suivantes :

- AES est un standard, donc libre d'utilisation, sans restriction d'usage ni brevet.
- c'est un algorithme symétrique, c'est-à-dire que le déchiffrement s'obtient en effectuant les opérations inverses lors du chiffrement.

- c'est un algorithme de chiffrement par blocs de données.
- il supporte différentes combinaisons ([longueur de clé]-[longueur de bloc] : 128-128, 192-128 et 256-128 bits).

En termes décimaux, ces différentes tailles possibles signifient concrètement :

- 3, 4e38 clés de 128-bit possibles.
- 6, 2e57 clés de 192-bit possibles.
- 1, 1e77 clés de 256-bit possibles.

Pour avoir un ordre d'idée, les clés DES ont une longueur de 56 bits (64 bits au total dont 8 pour les contrôles de parité), ce qui signifie qu'il y a approximativement 7, 2e16 clés différentes possibles (256). Cela nous donne de l'ordre de 1021 fois plus de clés 128 bits pour l'AES que de clés 56 bits pour le DES. En supposant que l'on puisse construire une machine qui peut casser une clé DES en une seconde (donc qui puisse calculer 256 clés par seconde), alors cette machine mettrait encore 149 000 milliards d'années pour casser une clé AES. Pour conclure, on voit que le standard AES répond aux mêmes exigences que le DES, mais il est également beaucoup plus sûr et flexible que son prédécesseur.

Du point de vue de nos objectifs, le choix de cet algorithme repose sur les caractéristiques suivantes :

- présente une structure contenant des boucles de traitement nécessitant une grande rapidité de traitement.
- flexibilité d'implémentation : taille de clés et de blocs différents selon l'application visée.
- hardware et software : il est possible d'implémenter l'AES aussi bien sous forme logicielle que matérielle (câblé).
- simplicité de son architecture.

Si l'on se réfère à ces critères, on voit que l'AES est un algorithme particulièrement approprié pour les implémentations embarquées qui suivent des règles beaucoup plus strictes en matière de ressources, puissance de calcul, taille mémoire, etc.

L'algorithme Rijndael [JV03] procède par blocs de données de 128 bits, avec une clé de 128 bits également. Chaque bloc de données subit une séquence de 5 transformations répétées 10 fois. Les transformations sont les suivantes :

- Addition de la clé secrète (par un ou exclusif) : Sous-module " *XorRoundKey* "
- Transformation non linéaire d'octets : les 128 bits sont répartis en 16 blocs de 8 bits, eux mêmes orientés vers des tables de codages 4×4 (*S-Box*). Chaque octet est alors codé par une fonction non linéaire S : sous-module " *Sub_Byte* ".
- Transformation en lignes : les 3 dernières lignes sont décalées cycliquement vers la gauche : la 2ème ligne est décalée d'une colonne, la 3ème ligne de 2 colonnes, et la 4ème ligne de 3 colonnes : sous-module " *ShiftRows* "
- Transformation en colonnes : Chaque colonne est transformée par combinaisons linéaires

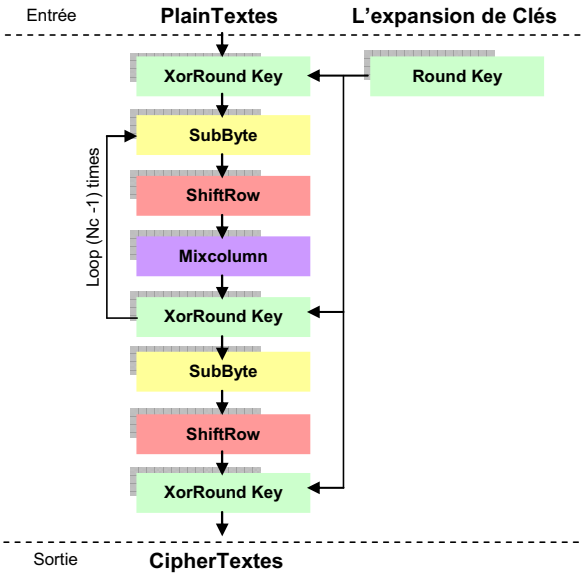

FIG. 4.11 – Schéma de principe de l'algorithme de cryptage AES.

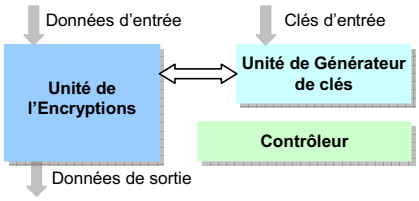

FIG. 4.12 – Architecture fonctionnelle de l'algorithme AES.

des différents éléments de la colonne (ce qui revient à multiplier la matrice 4×4 par une autre matrice 4×4) : sous-module " *MixColumn* ".

– Addition de la clé de tour : À chaque tour, une clé de tour est générée à partir de la clé secrète par un sous-algorithme (dit de cadencement). Cette clé de tour est ajoutée par un ou exclusif au dernier bloc obtenu : sous-module " *XorRoundKey* ".

La figure 4.11 illustre la séquence de transformation d'un bloc de données ainsi que le principe de fonctionnement du chiffrement AES.

L'architecture générale fonctionnelle de l'algorithme AES est représentée sur la figure 4.12. Les composants principaux de l'architecture d'AES sont le module de commande (*Controller*), le générateur de clé tour (*RoundKey*) et le module de chiffrement (*Encryption unit*). L'exécution du module de cryptage est exécutée en permanence en association avec le générateur de clé tour suivant les indications de la figure 4.12.

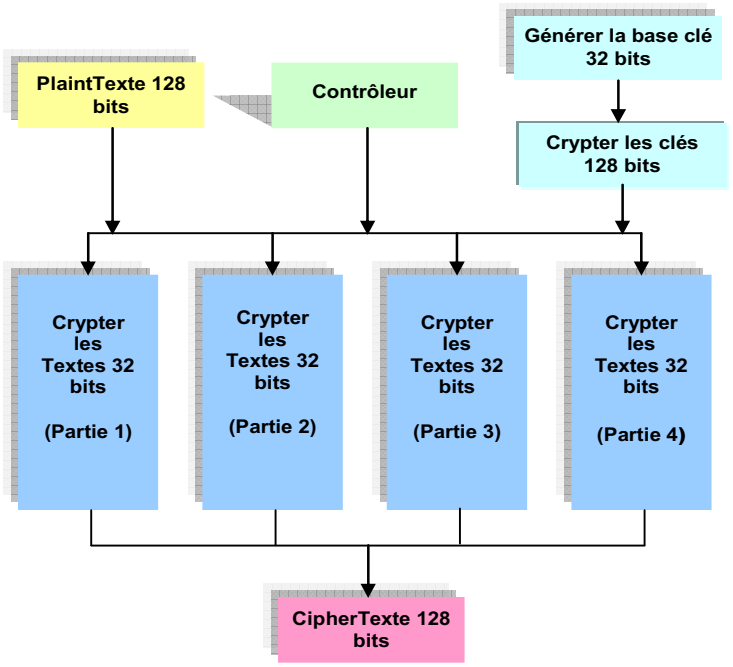

FIG. 4.13 – Structure par bloc du chemin de données de l'algorithme AES.

La structure par bloc de chemin de données et de contrôle du graphe flot de données de l'algorithme AES est décrite sur la figure 4.13. Chaque ***Macro*** reçoit 32 bits de texte et 32 bits de la clé cryptée. À l'intérieur de chaque partie ***Macro*** les données sont traitées par paquets de données 8 bits en vue d'exécuter la séquence des cinq transformations d'un bloc de données (voir figure 4.11).

La figure 4.15 présente la structure du générateur de clé de tour. Cette clé a une longueur de 128 bits (la longueur de la clé peut également être de 192 ou 256 bits selon le niveau de sécurité désiré). Les données de clé de tour sont obtenues par trois étapes successives. Premièrement, la clé 128 bits est décomposée en quatre mots de 32 bits. Ces derniers sont également décomposés en segment de mots 8 bits. Chaque segment 8 bits du mot 32 bits de poids faible de la clé est employé comme une adresse à des fonctions *SBOX* correspond à quatre tables de codage d'entrées 8 bits et de sortie 8 bits (détaillé figure 4.14). La valeur retournée de 8 bits avec les 3 autres segments constitue un nouveau mot de 32 bits intitulé " Subword ".

Dans l'étape suivante, une opération ou exclusif est réalisée entre " *Subword* " et la clé de chiffrement dont le mot 32 bits poids fort est transformé par un ou exclusif avec un mot résultant d'un tableau de codage intitulée " *RCON* ". Un pointeur est employé comme adresse pour accéder à ce tableau de codage. À chaque cycle d'horloge, le pointeur est incrémenté et permet une transformation pour chaque nouvelle clé. Enfin, les 128 bits résultant des transformations précédentes de la clé initiale sont décomposés en rangée de mots 32 bits. Chaque rangée subit

hex	y															
	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

x

FIG. 4.14 – Le Substitution Tableau - Sbox[xy] (en hexadécimal).

des opérations mutuelles ou exclusive par blocs d'octets avec le segment de mots 32 bits voisins précédent (voir figure 4.15). Le résultat final fournit une clé de tour (" *RoundKey* ") pour le module de chiffrement du bloc de données à crypter (Plaintext).

La figure 4.16 détaille le schéma fonctionnel du module de chiffrement de l'algorithme AES. Dans le cas d'un chiffrement de blocs de données 128 bits, la transformation " *SubByte* " de chaque ***Macro***, décompose chaque bloc de données à crypter (*Plaintext*) en 16 octets et considère un bloc de données comme une matrice 4×4 d'élément correspondant à un octet. Chaque élément est adressé dans un tableau de codage " S-box ". Le tableau de codage " S-box " fournit les valeurs de la transformation d'affinage définies par [JV03] et exprimée par l'équation suivante :

$$b'_1 = b_i \oplus b_{(i+4)mod8} \oplus b_{(i+5)mod8} \oplus b_{(i+6)mod8} \oplus b_{(i+7)mod8} \oplus c_i \tag{4.13}$$

avec $0 < i < 8$ où le b_i est le bit i de l'octet b et c_i est bit i de l'octet c avec la valeur décimale {63} ou binaire {01100011}. Plus de détails peuvent être trouvés dans [JV03].

Ensuite, les lignes de la nouvelle matrice 4×4 sont décalées et rangées en colonne. Le décalage est appliqué à chaque bloc de la rangée et de chaque rangée individuellement. Les quatre blocs sont décalés c fois où c est le nombre de la rangée. La dernière transformation est " *XorRoundKey* ". Elle consiste en un ou-exclusif entre la matrice résultante des deux transformations précédentes et la clé de tour " *RoundKey* " calculée par le module de génération de clé de tour. Le processus est réitéré 10 fois. À la fin de l'exécution de ce module, les textes chiffrés du " *Plaintext* " sont

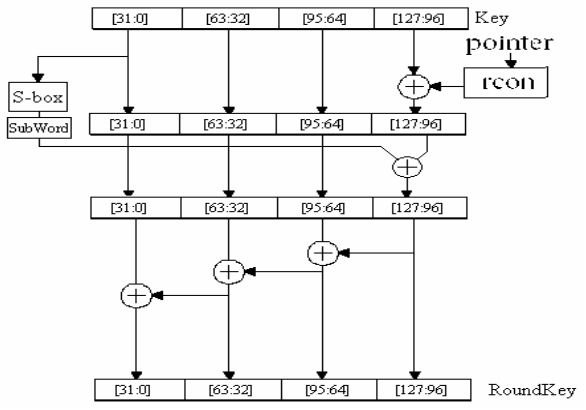

FIG. 4.15 – Chemin de données du module de génération de clé de tour.

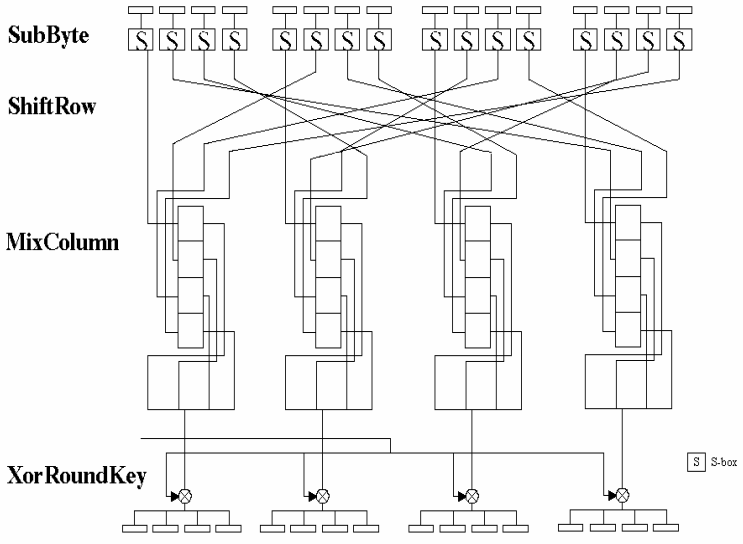

FIG. 4.16 – Module de chiffrement AES.

produits.

4.3.2 Partitionnement temporel initial

4.3.2.1 Caractérisation et annotation du GFD

A partir d'une description graphe flot de données de l'algorithme AES, une caractérisation, une annotation et une estimation des performances et ressources du graphe en vue de l'obtention d'un partitionnement temporel initial sont réalisées à l'aide de l'outil DAGARD. Cette première étape est réalisée en considérant la technologie FPGA Virtex Xilinx. La figure 4.17 présente l'architecture de l'algorithme AES sous forme de graphe flot de données détaillé dans l'outil DAGARD. La figure 4.18 suivante donne la légende relative à la figure 4.17.

L'outil DAGARD détermine de manière automatique le nombre de partitions, les performances attendues et les ressources nécessaires à l'implantation du graphe flot de données de l'algorithme considéré. Le partitionnement est obtenu à partir de la contrainte de temps de l'algorithme. Nous considérons un traitement de données à un débit de $10 Moctets/s$. Ce qui correspond au traitement de blocs de données $N = 100000$ octets en 10 ms.

Le tableau 4.13 résume ces données pour la technologie FPGA Xilinx Virtex [Xil07b]. Ainsi, à partir de la taille des données à traiter et des opérateurs saisis dans DAGARD, nous obtenons une surface totale de 1703 cellules logiques (CLB), un temps d'exécution global de $15,4 ns$ et un nombre de partitions égal à 5.

TAB. 4.13 – Estimation des ressources d'implantation en RD par l'outil DAGARD de l'algorithme AES.

Estimation du nombre total de CLB	Temps d'exécution opérateur(ns)	Nombre d'étape de reconfiguration	Nombre Moyen de ressources/étapes
1703	15,4	5	341

DAGARD a découpé le chemin de données en cinq partitions.

Le tableau 4.16 fournit une estimation automatisée de chaque partition avec la technologie Xilinx Virtex. La taille des partitions ainsi que la bande passante mémoire (Fréquence x taille des données) nécessaire par partition sont décrites. Il est à noter que la taille des données en sortie de la première partition n'est pas entièrement utilisée pour la partition suivante. Nous avons donc une bande passante plus faible en entrée et en sortie pour les partitions suivantes.

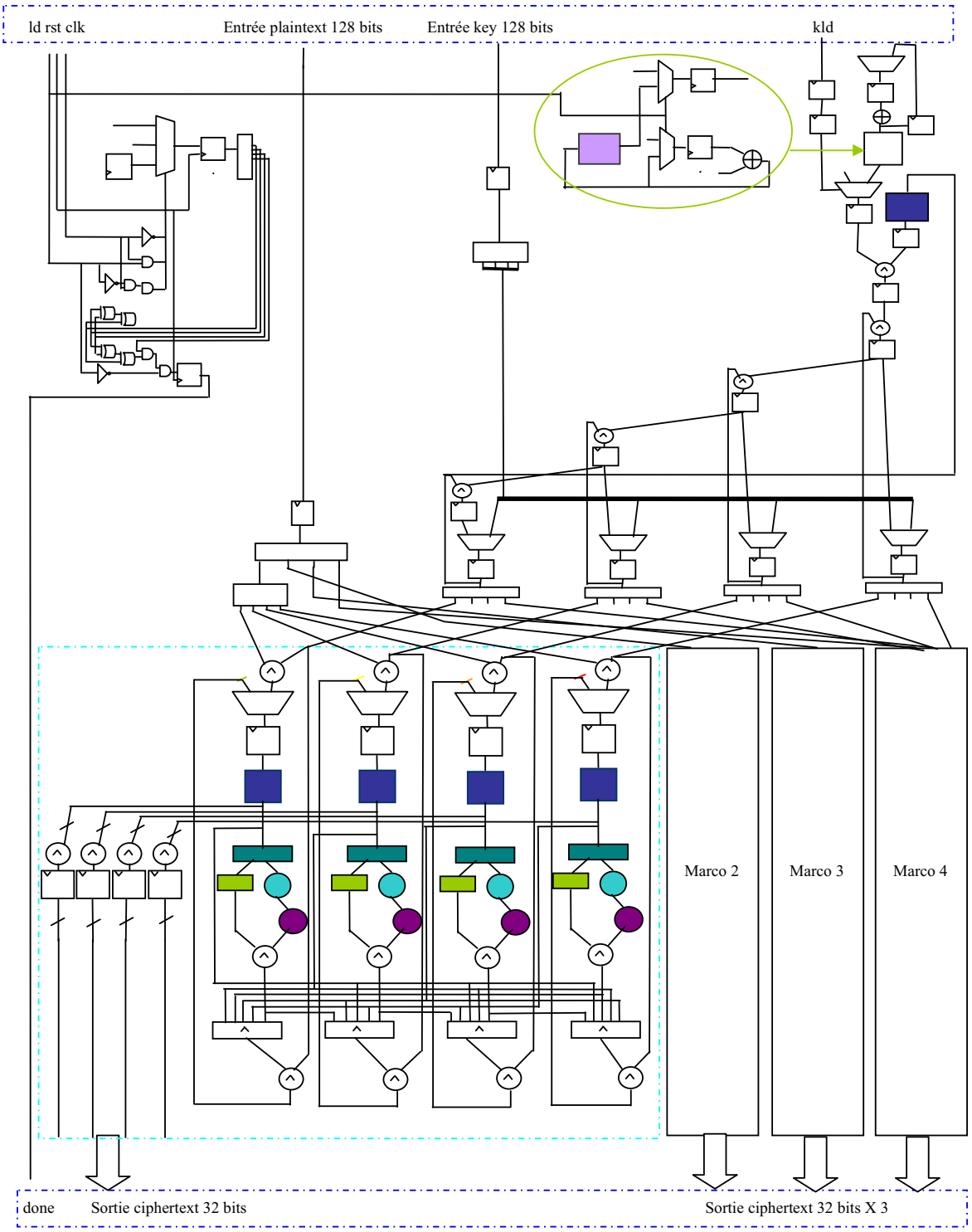

FIG. 4.17 – Graphe flot de données de l'algorithme AES.

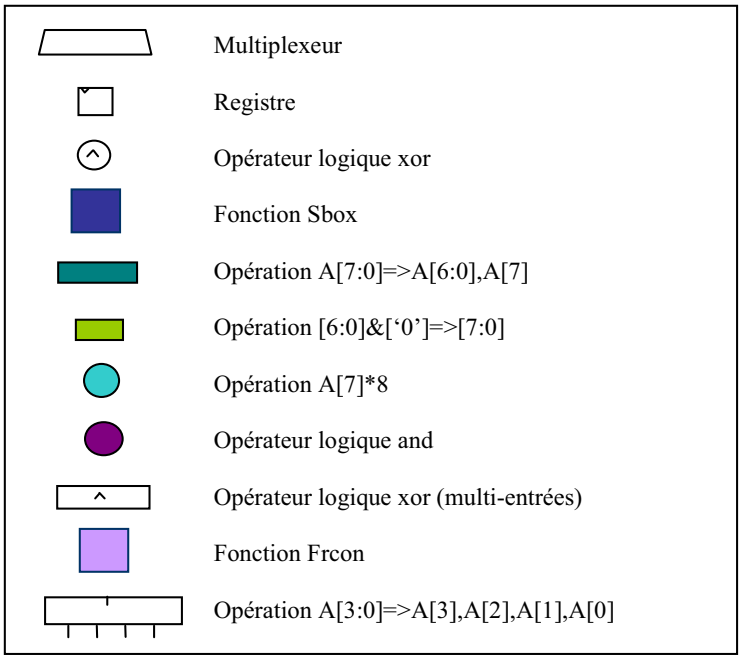

FIG. 4.18 – Légende du Graphe flot de données de l'algorithme AES.

4.3.2.2 Partitionnement initial du GFD

La figure 4.19 présente le Partitionnement automatisé obtenu par l'outil DAGARD de l'algorithme AES. Les macros de cryptage de textes 32×32 bits sont identifiés. La figure 4.20 présente le graphe flot de donnée détaillé de cette macro de traitement saisi dans DAGARD.

Le partitionnement proposé par DAGARD ne prend pas en compte la gestion des boucles de traitement. C'est pourquoi le partitionnement initialement proposé n'est pas réalisable puisqu'il chevauche une boucle de traitement (voir figure 4.19). Un raffinement manuel de la première partition s'impose. Cette dernière est agrandie afin qu'elle englobe la boucle de traitement ini-

TAB. 4.14 – Estimation des partitions de l'algorithme AES sur technologie Xilinx Virtex.

Numéro de partition	Nombre de Cellules	Fréquence maximale (Hz)	Taille de bit entrée	Taille de bit sortie	Bande passante nécessaire en entrée	Bande passante nécessaire en sortie
1	392	$42,3^e+007$	130	129	$5,49^e+008$	$5,45^e+008$
2	396	$54,6^e+007$	129	64	$7,04^e+008$	$3,49^e+008$
3	324	$57,4^e+007$	64	32	$3,67^e+008$	$1,83^e+008$
4	324	$57,4^e+007$	64	32	$3,67^e+008$	$1,83^e+008$
5	324	$57,4^e+007$	64	32	$3,67^e+008$	$1,83^e+008$

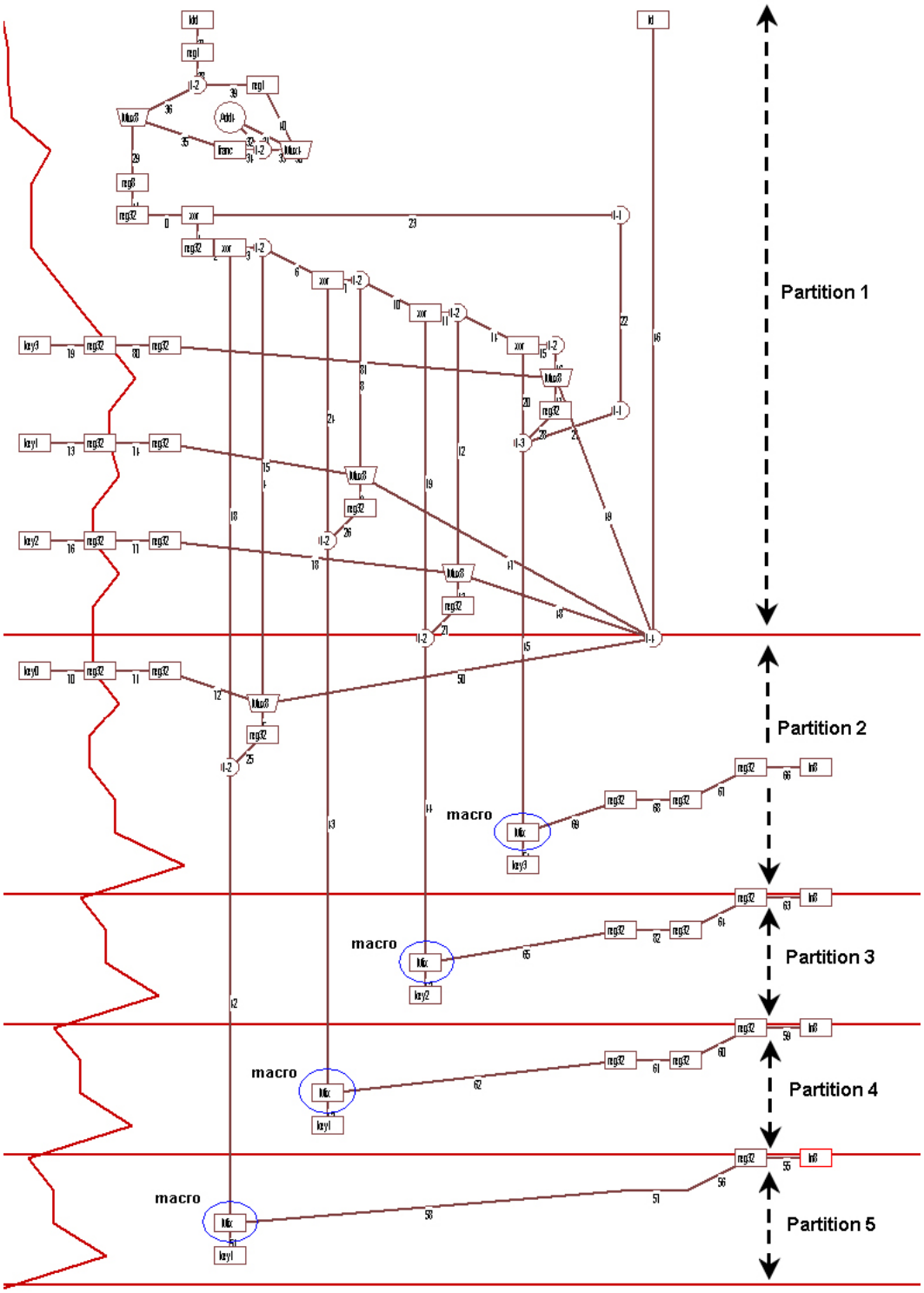

FIG. 4.19 – Partitionnement automatisé obtenu avec DAGARD de l'algorithme AES.

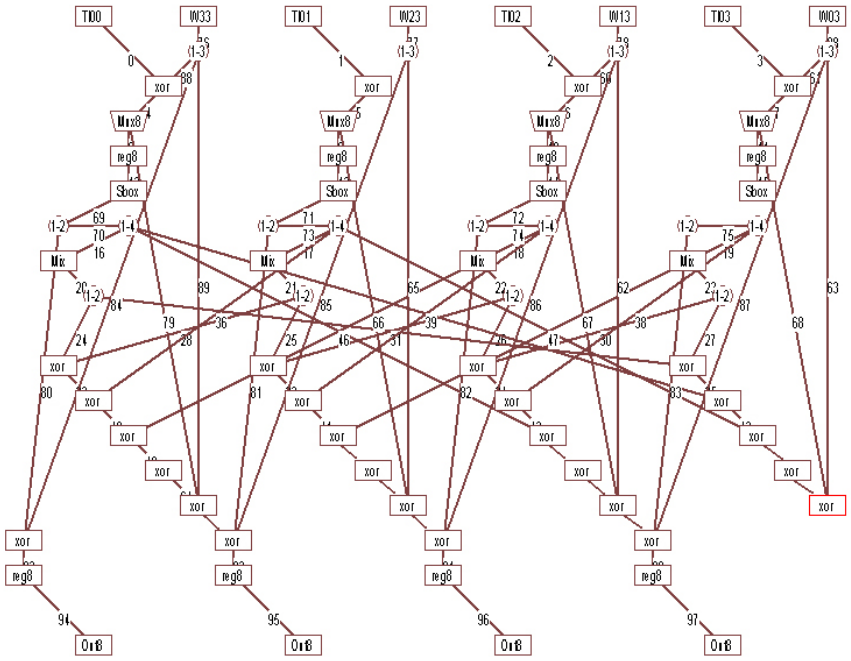

FIG. 4.20 – Flot de données détaillé et annoté de la macro de cryptage des données.

tialement découpée par le premier partitionnement (voir figure 4.19).

Nous avons implémenté sur cible FPGA Xilinx Virtex le partitionnement raffiné obtenu avec DAGARD. L'outil de synthèse utilisé est *FPGA Advantage Synthesis*. Les figures 4.21 et 4.22 représentent successivement les graphes flot de données de la partition 1 et des partitions 2, 3, 4 et 5 en rouge dans la figure. La première partition intègre entièrement la génération de clé de tour avec un nombre d'itération Nc égal à 10. On peut distinguer les quatre modules SBOX théoriquement factorisables. Les partitions suivantes sont une décomposition du module de chiffrement de données.. Cette partition prend en entrée le bloc de données à crypter (Plaintext) et les clés de tour (figure 4.22). Un module de contrôle existe dans chaque partition. Dans la première partition, le contrôleur charge la séquence de clé secrète et gère le chargement après calcul des clés de tour. Dans les autres partitions, les contrôleurs gèrent les sous blocs Plaintext et le chargement des clés de tour pour le chiffrement par bloc réalisé dans chaque partition.

Le tableau 4.15 montre les résultats d'implantation du partitionnement final avec un chiffrement de bloc de données 128 bits en entrée. On constate que l'estimation des résultats attendus des partitions est cohérente avec les résultats d'implantation.

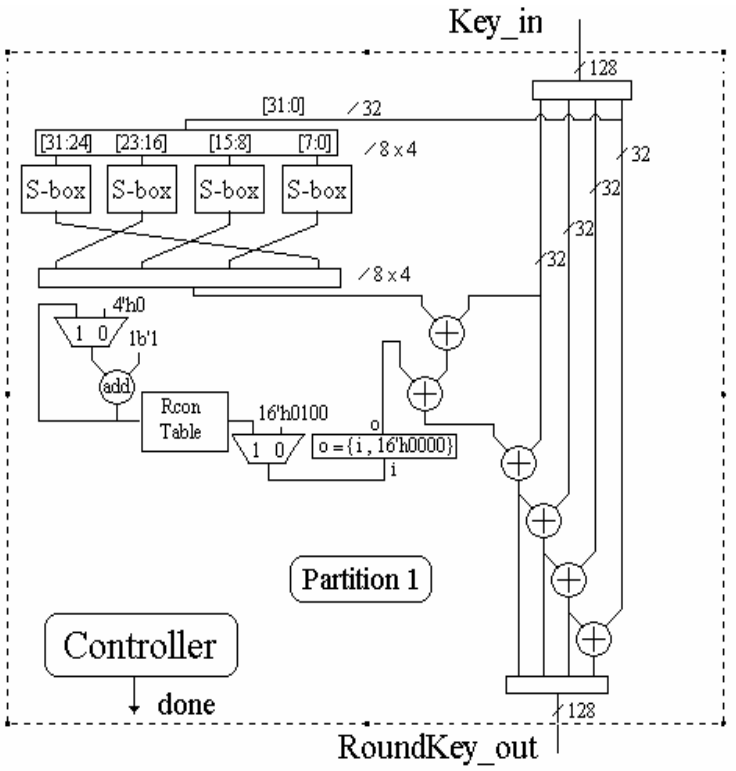

FIG. 4.21 – Partition initiale 1 du graphe flot de données de l'algorithme AES.

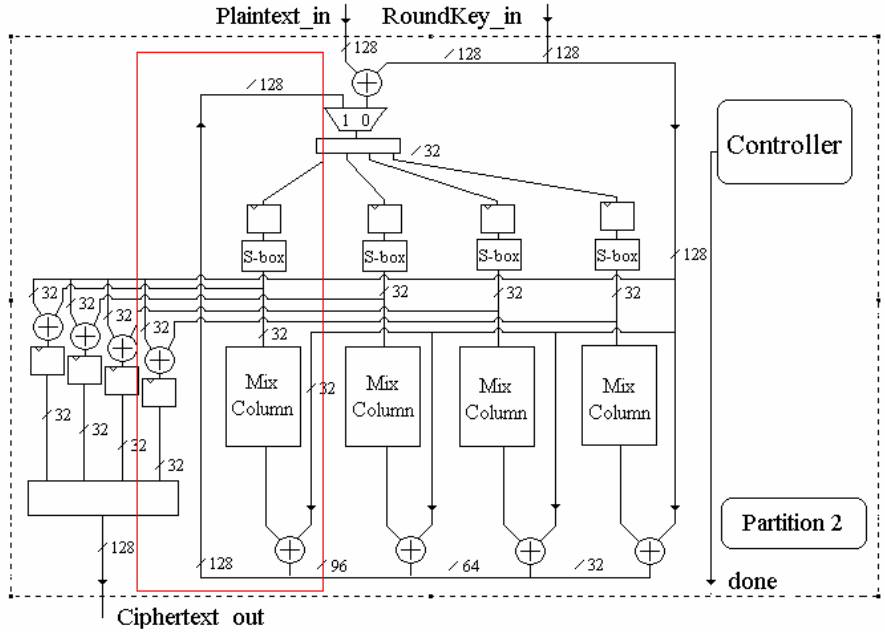

FIG. 4.22 – Partitions initiales 2, 3, 4, 5 du graphe flot de données de l'algorithme AES.

TAB. 4.15 – Implantation des partitions initiales de l'algorithme AES (128 bits) sur la technologie FPGA Xilinx Virtex.

Numéro de partition	Nombre de CLBs	Fréquence maximale (MHz)
1	423	65,1
2	320	77,8
3	320	77,8
4	320	77,8
5	320	77,8

TAB. 4.16 – Estimation des partitions de l'algorithme AES sur technologie Xilinx Virtex.

Numéro de partition	Nombre de Cellules	Fréquence maximale (Hz)	Taille de bit entrée	Taille de bit sortie	Bande passante nécessaire en entrée	Bande passante nécessaire en sortie
1	812	$7.424^{e}+007$	130	129	$9.653^{e}+009$	$9.577^{e}+009$
2	832	$9.259^{e}+007$	129	64	$11.94^{e}+009$	$5.925^{e}+009$
3	672	$9.259^{e}+007$	64	32	$5.925^{e}+009$	$2.963^{e}+009$
4	672	$9.259^{e}+007$	64	32	$5.925^{e}+009$	$2.963^{e}+009$
5	672	$9.259^{e}+007$	64	32	$5.925^{e}+009$	$2.963^{e}+009$

4.3.3 Optimisation par synthèse architecturale inter-partition

4.3.3.1 Taux de ressemblance mutuelle

La tableau 4.17. recense et caractérise les principale des opérateurs bas niveaux (multiplexeur, comparateur, registre, additionneur, etc.) pour l'ensemble des cinq partitions initiales constituant le graphe flot de données de l'algorithme AES. Tous les opérateurs et les unités fonctionnelles sont identifiés dans le but de calculer le taux de ressemblance mutuelle. Nous avons classifié 9 opérations en opérateurs et 2 opérations en unité fonctionnelle.

La table 4.18 donne les résultats de calcul du paramètre T.R.M pour chaque partition initiale successive à partir des résultats de caractérisation des opérateurs du graphe flot de données mettant en œuvre l'algorithme AES (table 4.17). Nous obtenons une grande valeur de T.R.M pour les partition 1 et 2 et les partition 5 et 1 signifiant ainsi des conditions non favorables pour une optimisation par synthèse inter-partition entre les partitions 1, 2 et 5, 1.

Par contre, nous avons des valeurs fortes de T.R.M pour les partitions 2 et 3, les partitions 3 et 4 et les partitions 4 et 5. Nous obtenons des valeurs de TRM_{p2-p3}, TRM_{p3-p4}, TRM_{p4-p5} identiques égales à 1. Ces valeurs fortes signifient un fort taux de ressemblance entre les parti-

tions considérées. En effet, ces 4 partitions sont identiques et donc des phases de reconfigurations entre les partitions 2 et 3, 3 et 4, 4 et 5 ne sont pas judicieuses. Cela signifie que nous avons des conditions favorables à une optimisation par synthèse architecturale inter-partition basée réutilisation d'unités fonctionnelles. Nous en déduisons donc qu'une partition réduite correspondant à la fusion des partitions initiales 2, 3, 4 et 5 peut donc assurer la mise en œuvre de ces partitions.

TAB. 4.17 – Caractérisation des opérateurs de l'algorithme AES.

Opération	Partie 1	Partie 2	Partie 3	Partie 4	Partie 5
Multiplexeur 1	8	4	4	4	4
Multiplexeur 2	1	0	0	0	0
Registeur	23	8	8	8	8
Addeur	2	0	0	0	0
Multiplieur	0	4	4	4	4
Logique &	5	4	4	4	4
logique xor 1	9	16	16	16	16
logique xor 2	0	4	4	4	4
Not	3	0	0	0	0
S-box	1	4	4	4	4
Frcon	1	0	0	0	0

TAB. 4.18 – Taux de ressemblance mutuelle Inter-partition de l'algorithme AES.

TRM_{p1-p2}	TRM_{p2-p3}	TRM_{p3-p4}	TRM_{p4-p5}	TRM_{p5-p1}
36%	100%	100%	100%	36%

Nous concluons qu'une exécution optimale dans la limite de la surface logique optimale déterminée par DAGARD correspond à une exécution en RD entre la partition 1 et une partition réduite mettant en œuvre les quatre partitions initiales suivantes et obtenue par synthèse architecturale de ces quatre partitions initiales. L'ensemble des deux partitions réduites permettent de réaliser globalement l'algorithme AES et correspond alors à :

$$P'_1 = P_1 \quad et \quad P'_2 = AS(P_2, P_3, P_4, P_5) \tag{4.14}$$

Cette solution propose donc une réduction du nombre de partitions initiales de trois en

modifiant également les partitions initiales 1 et 2.

4.3.3.2 Synthèse architecturale Inter-partition

La partition réduite 1 correspond à la partition initiale 1 (voir figure 4.21). Nous avons réalisé une optimisation par synthèse architecturale inter-partition par réutilisation d'unités fonctionnelles des partitions initiales 2, 3, 4 et 5 pour obtenir une partition réduite 2. Dans les partitions initiales les données étaient cryptées avec les clés de tour par quatre transformations (voir figure 4.16). Les 128 bits de données et les clés de tour étaient individuellement séparés en quatre segments de blocs 32 bits. Chacun de ces blocs est chiffré avec une clé de tour de 32 bits. Ces chiffrements utilisent les même opérateurs pour réaliser les mêmes fonctions (figure 4.22). Une solution architecturale basé sur la réutilisation de blocs d'opérations est réalisé selon le principe de fonctionnement illustré dans la figure 4.23. Cette architecture traite des mots de données de 32 bits et nécessite quatre cycles d'exécution pour traiter les blocs de données 128 bits. L'architecture fonctionnelle proposée est composée de blocs de registre A et B, de registres de travail, d'un contrôleur, de multiplexeurs et des fonctions de transformation SubByte, MixColumn, et XorRounKey pour des chemins de données de 32 bits. Les registres A et B peuvent être vus comme des tables 4×4 de mots de taille 8 bits. La structure des chemins de données de l'architecture associée à un contrôleur spécifique permet une combinaison alternée des mots de données afin de réaliser la transformation ShiftRows (figure 4.23).

L'architecture optimisée par synthèse architecturale a le fonctionnement suivant : dans un premier temps, le Plaintext (128 bits) ayant subi une opération XOR avec une clé de tour est stocké dans le registre A. Ensuite une première itération de cryptage est exécutée. Dans cette première itération, les données sont lues à partir du registre A et les résultats sont stockées dans le registre B. Le contrôleur charge le bloc de données $S_{0,0}, S_{1,1}, S_{2,2}, S_{3,3}$ du registre A dans les registres de travail. Ensuite les transformations SubByte, MixColum, XorRounKey sont exécutées séquentiellement. Les résultats sont stockés dans la première colonne du registre B (block $S'_{0,0}, S'_{1,0}, S'_{2,0}, S'_{3,0}$). Similairement, les blocs $S_{0,1}, S_{1,2}, S_{2,3}, S_{3,0}$, $S_{0,2}, S_{1,3}, S_{2,0}, S_{3,1}$ et $S_{0,3}, S_{1,0}, S_{2,1}, S_{3,2}$ du registre A sont également cryptés et stockés dans les colonnes suivantes (block $S'_{0,1}, S'_{1,1}, S'_{2,1}, S'_{3,1}$, $S'_{0,2}, S'_{1,2}, S'_{2,2}, S'_{3,2}$ et $S'_{0,3}, S'_{1,3}, S'_{2,3}, S'_{3,3}$) du registre B.

Dans une seconde itération de cryptage, le procédé de traitement est répété dans une direction de transfert de données inverse (chargement à partir du registre B et stockage des résultats dans le registre A). Ainsi de suite, pour les cycles de cryptage suivants, la direction de transfert de données alterne de façon à ce que le dernier résultat d'itération de cryptage est toujours chargé pour l'itération suivante. Après N_c itérations, on obtient en sortie un bloc de données crypté. Cette solution architecturale minimise les ressources logiques. La figure 4.24 présente l'architecture microélectronique de la seconde partition réduite résultant de la phase manuelle

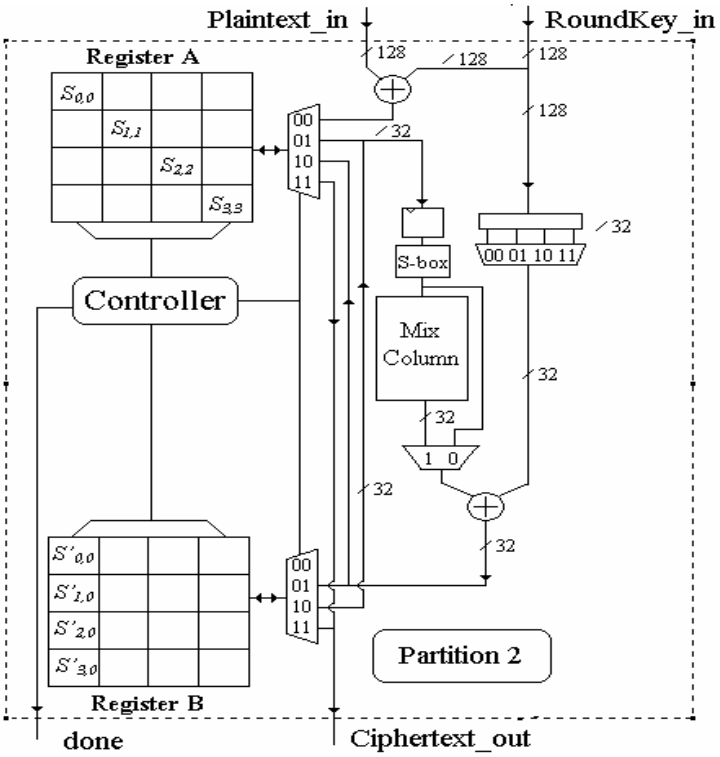

FIG. 4.23 – Architecture fonctionnelle optimisée de la seconde partition réduite de l'algorithme AES.

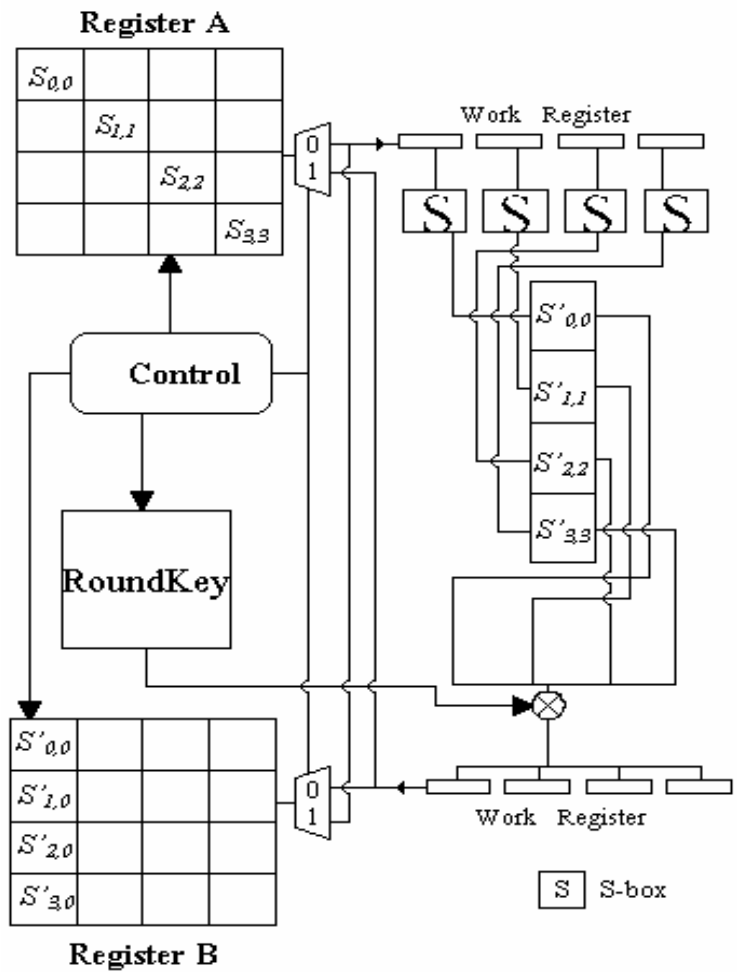

FIG. 4.24 – Illustration de la partition réduite 2 de l'algorithme AES après optimisation par synthèse architecturale inter-partition.

d'optimisation par synthèse architecturale inter-partition (IPS) entre les partitions initiales 2, 3, 4 et 5

Nous avons implémenté les partitions réduites obtenues au cours des phases successives par notre approche méthodologique sur la technologie Xilinx Virtex XCV200E sans utilise les blocs BRAMs disponibles. Les résultats de ces implémentations sont présentés dans les tables 4.19.

TAB. 4.19 – *M.E.R.*Partitionnement temporel réduit par optimisation IPS de l'algorithme AES.

Partition	Nombre de CLBs	Fréquence maximal (Mhz)
1	464	65,1
2	360	77,8

La première partition réduite correspondant à la première partition initiale dans le graphe flot de données intègre le calcul de la clé de tour. Elle utilise un nombre de ressources logiques totales de 464 CLBs avec une fréquence de fonctionnement maximum de 65,1 MHz. Nous obtenons donc une surface logique optimale S'_{ucr} équivalente à la surface S_{ucr} définie lors du partitionnement temporel initial et correspondant à la surface maximale nécessaire à l'implémentation du partitionnement réduit. Toutefois, la deuxième partition réduite correspond à une fusion des quatre partitions initiales 2, 3, 4 et 5 réalisant principalement le chiffrement d'un bloc de données. Cette seconde partition réduite occupe un nombre de cellules logiques de 360 CLBs, soit un surcoût de $12,5\%$ par rapport aux ressources logiques des partitions initiales (voir table 4.15) mais inférieure à la surface logique optimale spécifiant la surface nécessaire à la mise en œuvre du partitionnement temporel initial. Dans notre cas, l'implémentation utilise seulement $15,31\%$ de cellules logiques disponibles sur une FPGA de Xilinx Virtex XCV200E-6.

La réduction du nombre de partitions initiales à 2 grâce à la synthèse architecturale inter partition a engendré une forte diminution des temps de reconfiguration. En effet, si on considère le temps de reconfiguration proportionnel à la surface optimale S_{ucr} (équation 4.6), l'exécution du partitionnement optimisé permet une diminution du temps des phases de reconfiguration de plus de 56% en comparaison avec une exécution du GFD de l'algorithme AES par le partitionnement temporel initial. Nous aboutissons alors à une réduction de plus de moitié de la consommation générée par les phases de reconfiguration de la plateforme cible reconfigurable.

De même, l'approche d'optimisation d'un partitionnement temporel par IPS conduit à une réduction du nombre de partition initiale aboutissant à un gain de réduction du temps d'exécution global de traitement ΔT_{Exe} (équation 4.12) dépendant principalement des nombres de partitions initiales et réduites. Pour notre application AES, l'exécution du partitionnement optimisé nous permet d'obtenir une diminution du temps maximal d'exécution proche de 60% en

comparaison avec le temps de traitement du GFD de l'algorithme AES par le partitionnement temporel initial. Cela nous permet d'obtenir un excellent compromis entre la logique surface reconfigurable minimum (CLB) et le temps d'exécution global de l'application.

4.3.4 Bilan et discussion

Nos résultats expérimentaux sur la technologie Xilinx Virtex ont démontré que l'application de notre approche méthodologique sur l'algorithme AES permet d'atteindre une bonne adéquation entre la plus petite surface logique et le temps d'exécution totale en comparaison. La table 4.20 permet une analyse comparative entre les résultats de notre implémentation et les principaux implémentations de l'algorithme AES proposés dans la littérature. Toutes ces implémentations mettent en œuvre l'algorithme AES en mode exécution ECB [JV03]. Seule notre approche propose une solution en reconfiguration dynamique par partitionnement optimisé IPS.

L'exécution de notre approche utilise seulement $15,31\%$ des ressources logiques disponibles dans un FPGA Xilinx Virtex XCV200E-6 sans ressources RAM. Notre approche est plus flexible tout en évitant un surdimensionnement des ressources FPGA. En effet, dans la majorité des cas, nous avons un gain en ressources logiques situé entre plus de 2,5 à 27 fois plus petit et sans utilisation de bloc mémoire interne FPGA. À partir de la fréquence de travail maximale, la partition deux correspondant au module de chiffrement permet un débit de 205 Mbps. En terme d'application, notre approche nécessite un temps de latence de 110 μs afin de calculer les clés de tour par la mise en œuvre de la première partition et une phase de reconfiguration partielle pour la configuration de la seconde partition. Ces phases sont exécutées autant de fois qu'il y a de changement de clé secrète au cours du traitement.

L'implémentation proposée dans [CG03]repose entièrement sur une synthèse architecturale et permet d'utiliser moins de ressources logiques. Cependant elle fournit le plus faible débit de chiffrement pouvant se traduire selon les applications à un non respect de contrainte de temps.

Les résultats obtenus à partir de l'application de chiffrage d'AES prouvent que notre approche produit une solution assez efficace même si la complexité du problème est imprévisible. En effet, dans notre exemple, notre méthode permet au final de remplacer les 5 partitions initiales par deux partitions réduites qui sont effectuées séquentiellement.

4.4 Conclusion

Nous avons présenté et détaillé l'implantation de deux applications (algorithme de détection de contour et de cryptage AES) en reconfiguration dynamique par partitionnement temporel optimisée par une synthèse inter-partition. Les implémentations ont été réalisées selon notre flot de conception basé IPS (figure 3. 14), et mis en œuvre selon phases successives suivantes :

TAB. 4.20 – Tableau de comparaison en terme de ressources et débit de chiffrement de données.

Référence	Technologie	Ressources (CLB)	BRAMs	Throughput ECB mode (Mbps)
Chodowiec and al. [Ca01]	Xilinx XC1000	12,600	80	12,160
Mcloone and al. [MM01]	Xilinx XCV812E	2,222	100	6,956
Dandalis and al. [Da04]	Xilinx XCV1000	5,673	?	353
Pramstaller and al. [PW04]	Xilinx XCV1000E	1,125	/	215
Chodowiec and al. [CG03]	Xilinx XC2S30	222	3	166
Notre Proposition	Xilinx XCV200	464	/	205

A partir d'une description graphe flot de données de l'application à implémenter et de la cible reconfigurable technologique un partitionnement temporel initial est réalisé. Dans l'illustration de notre approche, ce partitionnement repose sur la détermination automatisée de partitions initiales optimisées et homogènes en terme de ressources matérielles à partir d'une contrainte de traitement (Outil DAGARD). Ce partitionnement temporel initial n'est qu'une base d'entrée possible pour l'application de l'approche proposée parmi d'autres méthodes de partitionnement temporel.

Ensuite, à partir des partitions initiales, le calcul du degré d'optimisation par factorisation des unités fonctionnelles parmi l'ensemble des partitions est réalisé. Cette phase est assurée grâce à la définition du taux de ressemblance mutuelle entre les partitions initiales. Ce taux permet d'orienter une exploration d'une synthèse architecturale inter-partition. Après identification des UFs factorisables, une logique de contrôle et d'aiguillage permettant la mutualisation des opérations est mise en œuvre. Cette logique complémentaire peut être pénalisante en terme de coût en ressources logiques. En effet, une augmentation de ressources doit être considérée lorsque les partitions contiennent plusieurs types d'unités fonctionnelles (dimension au niveau grain) devant répondre à toutes les contraintes. Plus précisément, dans le cas où les partitions contiennent plusieurs unités fonctionnelles de "grains fins", une vérification sur la contrainte de surface est nécessaire car la factorisation d'UFs "grains fins" peut générer une consommation

plus importante en ressources supplémentaires en comparaison avec la mise en œuvre des UFs sans factorisation. Dans ce cas, de telles factorisations sont pénalisantes pour une solution de synthèse si la réduction du nombre de partitions initiales est faible. En effet, le gain en terme de temps d'exécution n'aboutit pas à un bon compromis entre les ressources d'implémentation et le temps de traitement global de l'application.

Ces travaux expérimentaux montrent l'intérêt de l'approche méthodologique proposée afin d'atteindre une bonne solution d'adéquation algorithme - architecture basée conjointement sur la synthèse architecturale (réutilisation d'unités fonctionnelles) et la reconfiguration dynamique (partitionnement temporel). Nos résultats expérimentaux basés sur les technologies FPGA reconfigurables partiellement Xilinx Virtex et Atmel AT40K ont démontré que l'application de notre méthodologie d'implémentation conduit à un bon compromis entre la surface logique, la durée de traitement, la latence de calcul et de reconfiguration et par conséquent dans une certaine mesure à une réduction de la consommation par rapport à une exécution basée uniquement sur une approche reconfiguration dynamique. Ce compromis permet d'aboutir à une optimisation adaptée au cours d'une conception ciblée RSoC d'applications embarquées sous contrainte de temps. Finalement, la méthode proposée permet de spécifier les besoins de l'architecture nécessaire à l'implantation d'un algorithme, au regard des différentes contraintes de temps et contraintes technologiques.

Chapitre 5

Conclusion Générale et Perspectives

Le partitionnement temporel consiste à utiliser la reconfiguration dynamique des FPGA afin d'augmenter la densité fonctionnelle d'une application, tout en conservant une flexibilité et sans pénaliser la surface globale d'implémentation. Pour augmenter cette densité fonctionnelle, il est nécessaire de replier temporellement différentes parties de l'algorithme à implémenter sur la même surface logique FPGA. Grâce à ce procédé, la reconfiguration dynamique permet d'apporter plusieurs avantages lors de la conception de systèmes embarqués, tel que la réduction du nombre de circuits, poids et volume au détriment d'un accroissement du temps d'exécution global et parfois d'une consommation d'énergie supplémentaire. La technique de synthèse architecturale, limité à l'optimisation par réutilisation d'unités fonctionnelles, peut contribuer à l'atténuation des points faibles qu'apporte une exploitation de la reconfiguration dynamique.

L'adéquation architecture algorithmes (A^3) à travers une conception exploitant le potentiel de la reconfiguration dynamique et de la synthèse architecturale dans le domaine du calcul reconfigurable est un travail complexe et difficile à mettre en œuvre. Il repose d'abord sur l'amélioration de l'efficacité et des performances des FPGAs, qui a permis de dégager suffisamment de ressource pour comprimer tous les traitements sur une application visée.

Nous avons lancé une recherche dans ce thème dans le but de développer une méthodologie efficace combinant simultanément le partitionnement temporel et la synthèse architecturale basée sur la réutilisation d'unité fonctionnelle pour l'aide à la conception sur la partie FPGA d'un RSoC en vue d'améliorer l'adéquation algorithme - architecture pour des applications sous contraintes de traitement et/ou architecturale. Notre stratégie consiste à satisfaire au plus juste une optimisation des ressources matérielles en minimisant, par une approche de synthèse architecturale, le nombre de partitions temporelles nécessaires pour une application exécutée en reconfiguration dynamique.

La formulation de la problématique d'une combinaison entre le partitionnement temporel et la synthèse architecturale sous contraintes imposées par la technologie ainsi que par l'application

visée nous permet de proposer une nouvelle approche méthodologique originale. Cette méthode décrit comment on peut aborder un partitionnement d'un algorithme particulier en plusieurs étapes ainsi qu'une synthèse architecturale entre les partitions obtenues. Ce qui nous permet de réduire le nombre total de partitions pour une exécution en reconfiguration dynamique sur FPGA. On obtiendra ainsi un compromis du nombre de reconfigurations minimal. Notre approche consiste à synthétiser les partitions successives avec pour objectif de réduire le nombre de partitions temporelles. Il s'agit alors de mettre en œuvre une optimisation de ressource implémentée par une approche synthèse architecturale inter-partition (IPS) et dont l'exécution ou non repose une estimation préalable du degré de " ressemblance " favorisant une factorisation des unités fonctionnelles (UFs) entre les partitions successives.

L'estimation traduisant la ressemblance entre les partitions successives est réalisée à partir d'une comparaison en terme de nombre et de type caractérisant les opérateurs mis en œuvre dans les partitions. Cette étape est réalisée à partir d'une description sous forme graphe flot de données des partitions initiales. Cette forme de description permet de faire une analyse au niveau granularité de calcul et d'évaluer le degré d'optimisation attendu avec la technique de synthèse par factorisation d'UFs. Afin de mieux prendre en compte cette ressemblance, on introduit un paramètre quantitatif : le taux de ressemblance mutuelle ($T.R.M$). Celui-ci est défini comme un taux en pourcentage du degré de "ressemblance" ou de similitude des UFs présentes entre les deux partitions à analyser. Cette approche aide à choisir l'implantation appropriée d'un algorithme par comparaison du $T.R.M$ avec une valeur seuil prédéfinie par le concepteur. Plus précisément, il permet d'orienter une exploitation de la synthèse inter-partition pour une implémentation en complément de celle exploitant la reconfiguration. Nous aboutissons alors à des couples de partitions optimisées en une seule partition par une synthèse inter-partition au lieu de les reconfigurer.

L'algorithme permettant l'identification et l'extraction des UFs à instanciation multiple représentant les parties ressemblantes entre deux partitions a également été proposé et développé tout en prenant en compte les conditions d'efficacité d'une quantification de ressemblance par $T.R.M$.

L'originalité de l'approche proposée est que les partitions obtenues du partitionnement sont optimisées dans le sens d'une approche synthèse. En comparaison avec d'autres travaux existants, principalement focalisés sur la mise en œuvre d'un processus d'optimisation par synthèse appliqué simplement à chaque partition individuellement et indépendant des partitions voisines, cette méthode applique une factorisation possible des UFs à deux partitions successives. L'objectif est de réduire le nombre total de partitions d'un partitionnement temporel initial afin de minimiser le temps d'exécution global de l'application. Il s'agit d'effectuer les traitements avec un nombre minimal de reconfiguration et un débit de donnée maximal permis sur la plus petite surface tout en respectant la contrainte de temps. Celui-ci aboutit à une meilleur adéquation

algorithme et implantation en prenant en compte les besoins réels en terme de ressources et par rapport à une contrainte de temps calcul. Cette approche de synthèse inter-partition est une solution efficace pour améliorer une implantation en reconfiguration dynamique. Cette démarche de conception originale semble très prometteuse.

Nous avons illustré et montré la validité de notre nouvelle méthode d'exploitation de la reconfiguration associée à une optimisation par synthèse architecturale sur deux applications réelles : un algorithme de détection de contour d'images et l'algorithme de cryptage de bloc de données AES. L'implantation des exemples a été évaluée sur les technologies FPGA Atmel et Xilinx et les résultats obtenus nous ont permis de démontrer la validité du concept de l'approche proposée. Les deux exemples présentés sont issus de domaines d'applications différents. Néanmoins, aucune optimisation de ces algorithmes n'avait été entreprise préalablement à la conception de notre méthode. De ces travaux, nous pouvons retenir plusieurs points importants :

- La reconfiguration dynamique n'est pas la principale réponse à une implémentation optimisée. Elle permet effectivement d'augmenter virtuellement la surface logique disponible sur le composant reconfigurable. Mais elle présente également un certain nombre de contraintes qui peuvent devenir très pénalisantes dans certains cas. Cependant, la reconfiguration dynamique combinée à une solution de synthèse architecturale permet de fusionner des partitions initiales successives dans un but de minimiser les phases de reconfiguration et d'améliorer le temps d'exécution global d'une application reconfigurée.
- Les premiers résultats obtenus à partir de l'application de chiffrage d'AES prouvent que notre approche produit une solution assez efficace même si la complexité du problème est imprévisible. En effet, dans notre exemple, notre méthode permet de remplacer les 5 partitions initiales par deux partitions réduites qui sont effectuées séquentiellement. Notre solution montre comment nous pouvons combiner les avantages de l'exécution en reconfiguration dynamique aussi bien que l'avantage de la synthèse architecturale.
- En pratique, les résultats présentés pour une telle approche montre que celle-ci reste tributaire de plusieurs facteurs techniques, tels que la variété des fréquences de travail d'une étape de traitement, le format de chemin de données, les différentes contraintes de traitement ou technologique ou la variété des techniques de mise en œuvre de la factorisation ou la réutilisation des unités fonctionnelles.

Ce travail présenté sur l'exploitation potentielle de la reconfiguration et de la synthèse architecturale pour le domaine du calcul reconfigurable reste préliminaire. Cependant, nous avons proposé une approche montrant une meilleure Adéquation Algorithme - Architecture. Une implémentation sur structure reconfigurable dynamiquement peut être améliorée par réduction du nombre de partitions temporelles grâce à une optimisation par synthèse architecturale basée

sur une caractérisation et la réutilisation d'unités fonctionnelles. C'est pourquoi, une telle méthodologie qui permet d'accomplir un partitionnement temporel optimisé par la réutilisation d'unités fonctionnelles communes inter-partitions, est intéressante en terme d'accroissement de l'Adéquation Algorithme - Architecture.

De futurs travaux sont encore nécessaires pour améliorer la méthodologie proposée. Nous restons attentifs à une amélioration du calcul du $T.R.M$ dans le flot de conception afin d'obtenir une précision supérieure. Par exemple, développer une évaluation du taux de ressemblance mutuelle ($T.R.M$) en fonction des différents niveaux de granularité des unités fonctionnelles à instanciation multiple. En effet, il existe différentes solutions de mise en œuvre d'une synthèse par factorisation des UFs selon le niveau de granularité (plus ou moins "gros-grains" ou "grains fins"). Une piste consiste alors à analyser pour les différents types d'UFs, leurs niveaux de granularité et de spécifier pour chaque niveau une technique appropriée, efficace et optimisée de mise en œuvre d'une synthèse par réutilisation ou factorisation, de même que pour l'algorithme d'identification et d'extraction des unités fonctionnelles identiques que nous proposons dans le troisième chapitre. Cet algorithme peut être renforcé par une exploration des différents niveaux de granularité possible d'UFs pouvant être extraites. Suivi ensuite par une analyse et une évaluation en terme d'optimisation et de ressources attendues avec une technique de synthèse par factorisation. En résumé, il faut déterminer dans quelles limites cela peut être profitable au maximum dans l'approche d'optimisation proposée.

D'autres perspectives à ces travaux seraient de vérifier, de compléter et de mettre en œuvre le flot de conception de la méthodologie proposée. En effet, il n'existe pas d'outils mettant en œuvre le partitionnement temporel et la synthèse architecturale simultanément. Nous pensons qu'il est toutefois possible de confronter les résultats obtenus en intégrant ce travail dans l'outil de partitionnement générique DAGARD dans le but de disposer d'un environnement complet qui permette d'intégrer une exploration automatique de solutions architecturales pour des applications variées. Cet outil peut nous donner des indications et contribuer à la pertinence d'un partitionnement temporel que nous proposons. En effet, cela permettra de faciliter la tâche des concepteurs ainsi que d'améliorer l'efficacité d'une implémentation. Ainsi, le flot de conception de la méthodologie que nous proposons doit être intégré afin d'améliorer cet outil logiciel. Cependant, certaines parties du flot ne sont effectivement pas encore développées, comme la technique de la réutilisation d'unités fonctionnelles. De même, pour l'automatisation de la recherche des tâches synthétisables à instanciation multiple entre les partitions à partir d'une description graphe flot de données annotée. L'objectif est de fournir les outils nécessaires au concepteur afin qu'il puisse agir le plus efficacement et le plus intuitivement possible. Une autre perspective serait d'étendre cette étude à la possibilité d'intégration d'autres techniques de synthèse architecturale dans le flot de conception proposée.

Il serait bien sûr intéressant de continuer l'évaluation de notre approche sur des applications

du domaine du traitement de signaux ou d'images. En effet, notre approche semble efficace étant données les blocs de données à traiter qui existent souvent pour les applications de cryptographie et les applications de traitement d'image.

En conclusion, ce travail vise à contribuer au développement et à la réalisation d'outils de conception pour des technologies reconfigurables. Il accentue le développement des outils adaptés à ce type de technologie qui sont de plus en plus présents dans les systèmes de traitement numérique.

Bibliographie

[Abe06] N. Abel : *Outils et méthodes pour les architectures reconfigurables dynamiquement à grain fin, Synthèse et gestion automatique des flux de données.* Laboratoire des Équipe de Traitement des Images et du Signal, UMR 8051 CNRS. Cergy-Pointoise, 2006.

[ABM98] J.L. Philippe A. Baganne et E. Martin : A formal technique for hardware interface design. *IEEE Trans. on Circuits And Systems*, 45:584–591, 1998.

[Aic94] M. Aichouchi : *Etude des liens entre la synthèse architecturale et la synthèse au niveau transfert de registres.* Thèse de doctorat, Institut National Polytechnique de Grenoble, France, 1994.

[AK01] R. Allen et K.Kennedy : *Optimizing compilers for modern architectures.* Morgan Kaufmann, 2001.

[Alt] Altera : *Data Book.* http ://www.altera.com/products/ip/processors/nios2/ni2-index.html.

[APM86] J.T. Pizarro A.C. Parker et M. Mlinar : Maha : a program for data path synthesis. *In Proc. of the 23rd DAC*, pages 462–466, June 1986.

[Arm03] Joe Armstrong : *Making reliable distributed systems in the presence of software errors.* Thèse de doctorat, The Royal Institute of Technology, Stockholm,Sweden, 2003.

[Arm07] Joe Armstrong : Programming erlang, software for a concurrent world. *Pragmatic Programmers*, 2007.

[Atm] Atmel : *FPSLIC (AVR with FPGA) Product Overview.* http ://www.atmel.com/products/FPSLIC/overview.asp.

[Atm98] Atmel : *Datasheet of ATMEL AT40K FPGA.* http ://www.atmel.com/dyn/resources/prod-documents/DOC1132.PDF, 1998.

[Ba03] P. Brunet et al. : Hardware partitioning software for dynamically reconfigurable soc design. *In Proc. the 3rd IEEE International Workshop on System-on-chip for*

real time Applications, IEEE Circuits and Systems Society's Technical Committee on VLSI and on Communication, pages 106–111, Calgary, Canada, Jun 2003.

[Bag97] A. Baganne : *Methodologie de synthèse des unités de communication matérielles dans une approche de conception mixte logiciel/matériel (codesign).* Thèse de doctorat, Université de Rennes I, France, Dec 1997.

[BB05] Sophie Bouchoux et Elbey Bourennane : An application based on dynamic reconfiguration of fpgas : Jpeg2000 arithmetic decoder. *Optical Engineering*, Vol.44(Issu 10), October 2005.

[Bel04] A. Beletska : Finding coarse grained parallelism in arbitrarily nested loops. *Electronic Modelling, No.6*, 2004.

[Ber97] E. Berrebi : *Méthodologie pour l'application industrielle de la synthèse comportementale.* Thèse de doctorat, Institut National Polytechnique de Grenoble, France, 1997.

[Bom04] P. Bomel : *Plate-forme de prototypage rapide fondée sur la synthèse de haut niveau pour applications de radiocommunications.* Thèse de doctorat, Université de Bretagne Sud, France, Dec 2004.

[Bos04] L. Bossuet : *Exploration de l'espace de conception des Architectures Reconfigurables.* Thèse de doctorat, Université de Bretagne Sud, France, Décembre 2004.

[Bou04] N. Boudouani : *Architectures reconfigurables dynamiquement : synthèse matérielle d'opérateurs de détection et d'estimation de mouvement temps réel.* Thèse de doctorat, Université Cergy-Pontoise, France, 2004.

[BP97] Keith D. Cooperand Simpson L. Taylor Briggs Preston : Value numbering. *Software-Practice and Experience*, 27(6):701–724, June 1997.

[BP06] Anna Beletska et Pierluigi San Pietro : Extracting coarse-grained parallelism with the affine transformation framework and its limitations. *Dipartimento di Elettronica ed Informazione, Politecnico di Milano, Via Ponzio, 34/5 - 20133 Milano ITALY*, 2006.

[Bru04] Philippe Brunet : *Exploration multicritères d'architectures à Reconfiguration Dynamique.* Thèse de doctorat, Université Henri Poincaré, France, Dec 2004.

[Ca01] P. Chodowiec et al : Fast implementation of secret-key block ciphers using mixed inner-and outer-round pipelining. *In Proc. of the Symposium on Field Programmable Gate Array - FPGA 2001, ACM*, pages 94–102, 2001.

[Car03] João M.P. Cardoso : On combining temporal partitioning and sharing of functional units in compilation for reconfigurable architectures. *IEEE Trans. ON COMPUTERS*, 52(10), Oct 2003.

[CDR96] S.K. Pattanam C. Dawson et D. Roberts : The verilog procedural interface for the verilog hardware description language. *In Proc. of IEEE International Verilog HDL Conference, 26-28 Feb.*, pages 17–23, Feb 1996.

[CF02] Haskell B. Curry et Robert Feys : Haskell 98 language and libraries : The revised report. *Jozef Stefan Institute, Technical Report CSD-94-3, http ://haskell.org/onlinereport/preface-jfp.html*, December 2002.

[CG03] P. Chodowiec et K. Gal : Very compact fpga of the aes algorithm. *In Proc. of the workshop on Cryptographic Hardware and Embedded Systems - CHES 2003, LNCS 2779, Springer Verlag*, pages 319–333, 2003.

[Cha98] S. Chandrasekhar : Partitioning method and algorithms for configurable compution machines. *Master of Science in Electrical Engineering. Faculty of the Virginia Polytechnic Institut and State University*, August 1998.

[Coc70] John Cocke : Global common subexpression elimination. *In Proc. of Symposium on Compiler Construction, ACM SIGPLAN Notices 5(7)*, pages 850–856, July 1970.

[Cou03] P. Coussy : *Synthèse d'Interface de Communication pour les Composants Virtuels.* Thèse de doctorat, Université de Bretagne Sud, France, Dec 2003.

[CW91] R. Camposano et W. Wolf : *High level VLSI synthesis.* Kluwer Academic, 1991.

[Da04] A. Dandalisn et al : A comparative study performance of aes final candidates using fpgas. http ://csrc.nist.gov/encryption/aes/round2/conf3/papers/23-adandalis.pdf, 2004.

[DD98] M. Paindavoine et S. Weber D. Demigny : Architecture à reconfiguration dynamique pour le traitement temps réel des images. *Revue technique et Science de l'information, Numéro Spécial programmation des Architectures Reconfigurables*, 1998.

[Dem91] N. Demassieux : *Architecture VLSI pour le traitement d'images : une contribution à l'étude du traitement matériel de l'information.* Thèse de doctorat, Ecole nationale supérieure des télécommunications (ENST), France, Décembre 1991.

[DGL92a] A. Wu D. Gajski, N. Dutt et S. Lin : *High-Level Synthesis, Introduction to chip and system design.* Kluwer Academic Publisher, 1992.

[DGL92b] N. Dutt D. Gajski, A. Wu et S. Lin : *High level Synthesis : Introduction to Chip and System Design.* Kluwer academic publishers, 1992.

[Din96] H. Ding : *Synthèse Architecturale Interactive et Flexible.* Thèse de doctorat, Institut National Polytechnique de Grenoble, France, April 1996.

[ea03] K.B. Chehida et al. : Partitionnement pour la conception de systèmes réactifs à flots de données sur architectures reconfigurable. *In Proc. of SympAAA'2003*, Nice, 2003.

[Ell00] P. Ellervee : *High Level Synthesis of Control and Memory Intensive Applications.* Thèse de doctorat, Royal Institute of Technology, Stockholm, 2000.

[EM93] J.L. Philippe E. Martin, O. Santieys : Gaut : An architecture synthesis tool for dedicated signal processors. *In Proc. of IEEE Intl. European Design Automation Conference (Euro DAC'93)*, pages 14–19, Paris, France, 1993.

[Est00] G. Estrin : Computer network- based scientific collaboration in the energy research community, 1973-1977 :a memoir. *IEEE Annals of the History of Computing*, 22(4): 42–52, Apr.-Jun. 2000.

[Est02] G. Estrin : Reconfigurable computer origins : The ucla fixed-plus-variable (f+v) structure computer. *IEEE Annals of the History of Computing*, 24(4):3–9, Oct.-Dec. 2002.

[ET63] G. Estrin et R. Turn : Automatic assignment of computations in a variable structure computer system. *IEEE Trans. Electronic Computers*, 12(5):755–773, Dec. 1963.

[Fea92a] P. Feautrier : Some efficient solutions to the affine scheduling problem, part i, one dimensional time. *International Journal of Parallel Programming, No.21*, pages 313–348, 1992.

[Fea92b] P. Feautrier : Some efficient solutions to the affine scheduling problem, part ii, multidimensional time. *International Journal of Parallel Programming, No.21*, pages 389–420, 1992.

[Fea94] P. Feautrier : Toward automatic distribution. *Journal of Parallel Processing Letters, No.4*, pages 233–244, 1994.

[Fis90] J. P. Fishburn : Clock skew optimization. *IEEE Trans. On Computers*, 39:945–951, 1990.

[Gar00] A. D. Garcia : *Etude sur l'estimation et l'optimisation de la consommation de puissance des circuits logiques programmables du type FPGA.* Thèse de doctorat, Ecole Nationale Supérieure des Télécommunications, France, 2000.

[GCM03a] E. Senn G. Corre, N. Julien et E. Martin : Contraintes mémoire et solution architecturale pour applications tdsi. *In Proc. of Actes du colloque GRETSI*, Paris, France, Septembre 2003.

[GCM03b] E. Senn G. Corre, N. Julien et E. Martin : Ordonnancement sous contraintes de mémorisation : une optimisation efficace des ressources lors de la synthèse d'architecture. *In Proc. of Actes des Journées Francophones d'Etudes Faible Tension Faible Consommation (FTFC'03)*, Paris, France, May 2003.

[GL99] S.A. Guccione et D. Levi : Design advantages of run-time reconfiguration. *In Proc. In : Schewel, J., Athanas, P.M., Guccione, S.A., Ludwig, S., McHenry, J.T. (Eds.),*

vol. 3844, SPIE, pages 87–92, The International Society for Optical Engineering, Bellingham, WA, September 1999.

[Gov95] S. Govindarajan : Scheduling algorithms for high level synthesis. *Term paper ECE 834, University of Cincinnati*, 1995.

[Gov00] S. Govindarajan : *Algorithms for Design Space Exploration and High Level Synthesis for Multi-FPGA Reconfigurable Computers*. Thèse de doctorat, University of Cincinnati, march 2000.

[Gue97] H. Guermoud : *Architecture reconfigurable dynamiquement dédiées aux traitements en temps réel des signaux vidéo*. Thèse de doctorat, Université Henri Poincaré Nancy 1, France, 1997.

[Ha06] Yuko Hara et al. : Function cell optimization in behavioral synthesis. *In Proc. of the 9th EuroMicro Conference on Digital System Design (DSD'06)*, IEEE Computer Society Press, 2006.

[Hei96] M. J. M. Heijligers : *The application of Genetic Algorithms to High-Level Synthesis*. Thèse de doctorat, University of Eindhoven, 1996.

[H.G97] E.Tisserand et S.Weber H.Guermoud, Y.Berviller : Architecture à base de fpga reconfigurable dynamiquement dédiée au traitement d'image sur flot de données. *In Proc. of 16° colloque GRETSI*, septembre 1997.

[HZC02] Z. Lin H. Zhou et W. Cao : Research on vhdl rtl synthesis system. *In Proc. the First IEEE International Workshop on Electronic Design, Test and Applications (DELTA.02)*, 2002.

[INR93] INRIA : Les formats communs des langages synchrones, projet synchrone rapport no. 157. *Institut National de Recherche en Informatique et Automatique (INRIA), Rocquencourt, France*, June 1993.

[IRI01] IRISA : Cours master : Optimisations de code indépendantes de l'architecture cible. *http ://www.irisa.fr/master/COURS/CAPS/CoursCD/HTML/Compilation/OptimisationIndependanteMachine/OptimIndep.htm*, 2001.

[Ive62] Kenneth E. Iverson : *A Programming Language*. Wiley, 1962.

[JPa06] J-Ph.Diguet et al. : Epicure : A partitioning and codesign framework for reconfigurable computing. *Journal of Microprocessor and Microsystems, Special issue on FPGA-based Reconfigurable Computing, Elsevier Ed.*, Vol.30(No.6):367–387, Sep 2006.

[JPM95] J.P. Diguet J.L. Philippe, O. Sentieys et E. Martin : From digital signal processing specification to layout. *Logic and Architecture Synthesis : state of-the-art and novel approaches*, pages 307–313, 1995.

[JV03] D. Joan et R. Vincent : *A Specification for the AES Algorithm, Dr. Brian Gladman, V3.11.* http ://fp.gladman.plus.com/cryptography_technology/rijndael/aes.spec.311.pdf, 2004, 12th Sept 2003.

[Ka99] M. Karthikeya et al. : Temporal partitioning and scheduling data flow graphs for reconfigurable computers. *IEEE Transactions on Computers*, Vol.48(No.6), June 1999.

[KASO90] T. N. Mudge K. A. Sakallah et O. A. Olukotun : Check tc and min tc : Timing verification and optimal clocking of digital circuits. *In Proc. of ICCAD'90*, pages 552–555, IEEE Computer Society, 1990.

[Kis96] P. Kission : *Exploitation de la hiérarchie et de la réutilisation de blocs existants par la synthèse de haut niveau.* Thèse de doctorat, Institut National Polytechnique de Grenoble, France, 1996.

[KPB07] S. Bayar K. Paulsson, M. Hübner et J. Becker : Exploitation of run-time partial reconfiguration for dynamic power management in xilinx spartan iii-based systems. *In Proc. Of 3rd International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC07), ISBN 2-9517461-3-X*, pages 1–6, Yokohama, Japan, June 2007.

[Kuc98] K. Kuchcinski : An approach to high level synthesis using constraint logic programming. *In Proc. 24th EUROMICRO '98 Conference on Engineering Systems and Software for the Next Decade*, Vesteras, SWEDEN, August 1998.

[KV98] M. Kaul et R. Vemuri : Optimal temporal partitioning and synthesis for reconfigurable architectures. *Design, Automation, and Test in Europe*, Feb 1998.

[KwNY98] X.-J. Zhang K.-w. Ng et G. H. Young : Design representation for dynamically reconfigurable systems. *In Proc. of the 5th Annual Australasian Conference on Parallel And Real-Time Systems(PART'98)*, pages 14–23, 1998.

[La00] P. Lakshmikanthan et al. : Behavioural partitioning with synthesis for multi-fpga architectures under interconnect, area, and latency constraints. *In Proc. 15th IPDPS 2000 Workshops, ISSN 0302-9743*, page 924, Cancun, Mexico, May 2000.

[La06] Ting Liu et al. : Toward a methodology for optimizing algorithm-architecture adequacy for implementation reconfigurable system. *In Proc. of 13th IEEE International Conference on Electronics, Circuits and Systems (ICECS2006)*, IEEE Computer Society Press, December 2006.

[LD94] P. Lysaght et J. Dunlop : Dynamic reconfiguration of fpgas. *in More FPGA's, W. Moore and W. Luk, Eds.Oxford, England : Abingdon*, pages 82 – 94, June 1994.

[LES06] LESTER : *High Level Synthesis for Digital Signal Processors User's Manual, Version 4.1*, 2006.

[Lin97a] Y.L. Lin : *Recent Development in High Level Synthesis.* KNational Science Council of R.O.C, 1997.

[LIN97b] Yong-Loug LIN : Recent developments in high level synthesis. *ACM trans, Design Automation of Electronic systems*, 2(1):2–21, Jan 1997.

[Liu04] T. Liu : Implantation d'un algorithme de cryptage en reconfiguration dynamique et synthèse architecturale. Mémoire de D.E.A., Université Henri Poincaré, France, Juillet 2004.

[LKD06] N. Abel L Kessal et D. Demigny : Real-time image processing using dynamic reconfiguration paradigm : architecture and programming. *Traitement du signal*, vol. 23(No.1), 2006.

[LSB02] A. S. Kaviani L. Shang et K. Bathala : Dynamic power consumption in virtex-ii fpga family. *In Proc. of International Symposium on Field- Programmable Gate Arrays*, pages 157–164, Mars 2002.

[Ma05] Y.Le Moullec et al. : Design trotter : System-level dynamic estimation task a 1st step towards platform architecture selection. *Journal of embedded computing (JEC), IOS Press*, Vol.4(No.4), 2005.

[Ma06] Abdellatif Mtibaa et al. : An iterative method for algorithms implementation on a limited dynamically reconfigurable hardware. *Journal of Computer Science, Science Publications, USA*, Vol.2(No.5):422–430, May 2006.

[MAM03] J.P. Diguet. X. Fornari. A.M. Fouilliart. C.Gamrat. G. Gogniat. P. Kajfasz M. Auguin, K. Ben Chehida et Y Le Moullec : Partitionning and codesign tolls and methodology for reconfigurable computing : the epicure philosophy. *In Proc. of the Third International Workshop on Systems, Architectures, Modeling Simulation, SAMOS 03*, Samos, Greece, July 2003.

[MM01] M. McLoone et J. McCanny : High performance single chip fpga rijdael algorithm implementation. *In Proc. of Proceedings of the Workshop on Cryptographic Hardware and Embedded systems-CHES 2001, LNCS 2162, Springer Verlag*, pages 65–76, http ://csrc.nist.gov/encryption/aes/round2/conf3/papers/23-adandalis.pdf, 2001.

[MMC90] A.C. Parker M.C. McFarland et R. Camposani : The high-level synthesis of digital systems. *In Proc. of the IEEE. 78(2)*, pages 301–318, IEEE Computer Society, February 1990.

[MT93] R. Micallef-Trigona : Datapath intensive asic design-synthesis from vhdl. *IEEE*

Colloquium on VHDL (Very High Speed Integrated Circuits Hardware Description Language) - Applications and CAE Advances, 6(1993/076), April 1993.

[Muc97] S. S. Muchnick : *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[Nar98] M. Narasimhan : Exact scheduling techniques for high level synthesis. *Master of Science in Electrical Engineering, Lousiana State University*, 1998.

[NZ00] K.W. Ng et X.-J. Zhang : Module allocation for dynamically reconfigurable systems. *In Proc. of the 7th Reconfigurable Architectures Workshop (RAW 2000)*, Cancun, Mexico, 2000.

[Oa98] I. Ouaiss et al. : An integrated partitioning and synthesis system for dynamically reconfigurable multi-fpga architectures. *In Proc. of Fifth Reconfigurable Architectures Workshop (RAW'98)*, pages 31–36, Mars 1998.

[PBAS97] Alain Darte Pierre Boulet et Georges Andr-Silber : Loop parallelization algorithms : from parallelism extraction to code generation. *Laboratoire de l'informatique du parallélisme, Reaserache report*, August 1997.

[PK87] P.G. Paulin et J.P. Knight : Force directed scheduling in automatic data path synthesis. *In Proc. of the 24th ACM/IEEE Design Automation Conference*, pages 195–202, ACM and IEEE Computer Society, Miami, June 1987.

[Poo02] K. K. W. Poon : Power estimation for field programmable gate arrays. Mémoire de D.E.A., University of British Columbia, 2002.

[PW04] N. Pramstaller et J. Worlerstorfer : *A Universal and Efficient AES Co-processor for Field Programmable Logic Arrays.* J. Becker, M. Platzner, S. Vernalde (Eds.) : FPL 2004, LNCS 3203, pp. 565-574, 2004. © Springer-Verlag Berlin Heidelberg, 2004.

[RBK98] M. Karabernou R. Bourguiba, D. Demigny et L. Kessal : Designing a new architecture for real time image analysis with dynamically configurable fpgas. *In Proc. of IEEE IMACS Computational Engineering in Systems Applications*, Hammamet, Tunisia, 1998.

[RDS02] S. Pillement R. David, D. Chillet et O. Sentieys : Dart : a dynamically reconfigurable architecture dealing with future mobile telecommunications constraints. *In Proc. of In Parallel and Distributed Processing Symposium (IPDPS02)*, pages 156 – 163, 2002.

[RMH90] Mads Tofte Robin Milner et Robert Harper : *The Definition of Standard ML.* The MIT Press, ISBN-13 : 978-0-262-63132-7, 1990.

[Sa02] G. Sassatelli et al. : Highly scalable dynamically reconfigurable systolic ring-architecture for dsp applications. *In Proc. of Design, Automation and Test in Europe Conference and Exhibition (DATE02)*, pages 553 – 558, 2002.

[Sem] Lattice Semi. : *Datasheet : LatticeMico32 System.* http ://www.latticesemi.com/documents/doc20893x66.pdf.

[SGN02a] M. Kishinevsky. S. Rotem. N. Savoiu. N.D. Dutt. R.K. Gupta S. Gupta, T. Kam et A. Nicolau : Coordinated transformations for high-level synthesis of high performance microprocessor blocks. *In Proc. of Design Automation Conference*, 2002.

[SGN02b] Rajesh Gupta Sumit Gupta, Nikil Dutt et Alex Nicolau : Coordinated parallelizing compiler optimizations and high-level synthesis. *CECS(Center for Embedded Computer Systems)Technical Report No.02-35*, December 2002.

[SGN03] Rajesh Gupta Sumit Gupta, Nikil Dutt et Alex Nicolau : Loop shifting and compaction for the high-level synthesis of designs with complex control flow. *CECS(Center for Embedded Computer Systems)Technical Report No.03-14*, April 2003.

[Sil94] J. Silc : Scheduling strategies in high level synthesis. *Jozef Stefan Institute, Technical Report CSD-94-3*, 52, 1994.

[SJ93] A. Sharma et R. Jain : Estimating architectural resources and performance for high-level synthesis applications. *IEEE Trans. VLSI Syst.*, 1(2):175–190, Jun 1993.

[SN91] F. Catthoor et H. De Man S. Note, W. Guerts : Cathedral-iii : Architecture-driven high-level synthesis for high throughput dsp applications. *In Proc. of 28th ACM/IEEE Design Automation Conference*, 1991.

[Ste97] S. Muchnick Steven : *Advanced Compiler Design and Implementation.* The MIT Press, ISBN-13 : 978-0-262-63132-7, pp. 378-396, 1997.

[Sug00] Z. Sugar : *Synthèse comportementale basée sur l'ordonnancement.* Thèse de doctorat, Institut National Polytechnique de Grenoble, France, May 2000.

[Syn07] Synopsys : *Datasheet of Design Compiler(ASIC).* http ://www.synopsys.com/products/logic/dc-ultra-ds.pdf, 2007.

[Ta00] N. Togawa et al. : Automated design synthesis and partitioning for adaptive reconfigurable hardware. *In Proc. Of the ASP-DAC 2000, IEEE Circuits and Systems*, pages 309–312, Yokohama, Japan, 2000.

[Ta03] C. Tanougast et al. : Temporal partitioning methodology optimizing fpga resources for dynamically reconfigurable embedded real-time system. *Microprocessors and Microsystems, Elsevier*, 27(3):115–130, Apr 2003.

[Tan01] Camel TANOUGAST : *Méthodologie de partitionnement applicable aux systèmes sur puce à base de FPGA, pour l'implantation en reconfiguration dynamique d'algorithmes flot de données.* Thèse de doctorat, Université Henri Poincaré, France, Oct 2001.

[TCW00] J.R. Hauser T.J. CALLAHAN et J. WAWRZYNEK : The garp architecture and c compiler. *IEEE Trans. Comp.*, 3(4):62–69, 2000.

[TF69] S.C. Yang T.D. FRIEDMAN : Methods used in automatic design generator (alert). *IEEE Trans. ON COMPUTERS C-18 :593-614*, 1969.

[Tho04] J. THORVINGER : *Dynamic Partial Reconfiguration of an FPGA for Computational Hardware Support.* Thèse de doctorat, Lund Institute of Technilogy, Lund University, 2004.

[Vei98] Jim VEITCH : *A History and Description of CLOS.* Handbook of Programming Languages, Volume IV : Functional and Logic Programming Languages, ed. Peter H. Salus. (1st edition), Macmillian Technical Publishing, ISBN 1-57870-011-6, Pages 107-158., 1998.

[Vem01] R. VEMURI : Automated design synthesis and partitioning for adaptive reconfigurable hardware. *Hardware implementation of intelligent systems, Physica-Verlag GmbH, ISBN 3-7908-1399-0*, 2001.

[Wil97] John WILLOUGHBY : Synthesis support for design partitiong. *In the IEEE International Verilog HDL Conference (IVC'97)*, 1997.

[Wir97] M. J. WIRTHLIN : *Improving Functional Density Through Run-Time Circuit Reconfiguration.* Thèse de doctorat, Brigham Young University, 1997.

[WL94] M.S. Lam W. LIM : Communication-free parallelization via affine transformations. *In Proc. of the 7th workshop on languages and compilers for parallel computing*, pages 92–106, 1994.

[WL97] M.S. Lam W. LIM : Maximizing parallelism and minimizing synchronization with affine transforms. *In Conference Record of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1997.

[Xil] XILIX : *Datasheet : MicroBlaze Processor Reference Guide.* http ://www.xilinx.com/ise/embedded/mb-ref-guide.pdf.

[Xil07a] XILIX : *Datasheet of PowerPC 405 Processor Block Reference Guide.* http ://www.xilinx.com/bvdocs/userguides/ug018.pdf, 2007.

[Xil07b] XILIX : *Virtex-II Pro / Virtex-II Pro X Complete Data Sheet.* http ://www.xilinx.com/bvdocs/publications/ds083.pdf, 2007.

[XjZY97] K. W. Ng Xue-jie Zhang et G. Young : High-level synthesis using genetic algorithms for dynamically reconfigurable fpgas. *In Proc. of the 23rd Euromicro Conference (EUROMICRO' 97)*, Sep 1997.

[Zim79] G. Zimmermann : The mimola design system : A computer aided digital processor design method. *In Proc. of the Design Automation Conference*, 1979.

[ZN00a] X. Zhang et K. Ng : Module allocation for dynamically reconfigurable systems. *In Proc. 15th IPDPS 2000 Workshops*, page 932, Cancun, Mexico, mai 2000.

[ZN00b] X.-J. Zhang et K.-W. Ng : An effective high-level synthesis approach for dynamically reconfigurable systems. *In Proc. of the 4th International/Conference Exhibition on High Performance Computing in Asia-Pacific Region*, 2000.