Exploration Architecturale au Niveau Comportemental - Application aux FPGAs

Sébastien Bilavarn

N° d'ordre :

THÈSE

présentée devant

L'Université De Bretagne Sud

pour obtenir le grade de

Docteur De
L'Université De Bretagne Sud

Mention :

Électronique et Informatique Industrielle

École Doctorale Pluridisciplinaire

Composante Universitaire :

UFR Sciences

ET

Sciences de l'Ingénieur

par

Sébastien Bilavarn

Exploration Architecturale au Niveau Comportemental - Application aux FPGAs

soutenue le 28 Février 2002 devant la commission d'examen composée de :

M. :	J. L. Philippe	LESTER	Directeur de thèse
M. :	M. Auguin	I3S	Rapporteurs
M. :	D. Lavenier	IRISA	
M. :	M. Corazza	ENSSAT	Examinateurs
M. :	T. Collette	CEA	
M. :	G. Gogniat	LESTER	
M. :	J. P. Diguet	LESTER	Invité

Laboratoire d'Électronique des Systèmes TEmps Réel

Remerciements

Ce travail a été effectué au sein du groupe Adéquation Application Système (AAS) du Laboratoire d'Électronique des Systèmes TEmps Réel (LESTER), à Lorient. Je souhaiterais donc tout d'abord remercier M. E. Martin de m'avoir accueilli dans son laboratoire et M. J. L. Philippe, mon directeur de thèse, pour l'opportunité de thèse qui m'a été proposée il y a trois ans. Je le remercie aussi pour son soutien et ses conseils au cours de ces trois années de recherche.

Je tiens ensuite à remercier tous les membres du groupe AAS, les enseignants chercheurs, les thésards ainsi que les stagiaires avec qui j'ai été amené à travailler, parmi lesquels J. P. Diguet, Y. le Moullec, F. Bertrand, D. Heller, L. Bossuet, ...

Je souhaiterais remercier tout particulièrement Guy Gogniat, avec qui j'ai beaucoup apprécié travailler, pour son encadrement, ses conseils et encouragements.

Je remercie Messieurs M. Auguin, et D. Lavenier, pour avoir accepté d'être rapporteurs de ce travail ainsi que le président et les membres du jury.

Enfin, je remercie tous les membres du LESTER et leur souhaite à chacun beaucoup de réussite dans leurs projets.

Ce travail est dédié à la mémoire de ma mère.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

Motivation

La conception des systèmes électroniques n'a cessé d'évoluer vers l'intégration d'applications plus complexes pour des temps de conception plus courts. Cette progression a été rendue possible grâce aux spectaculaires avancées effectuées dans le domaine technologique, mais est ralentie par le fort décalage avec les environnements et outils de conception permettant de les exploiter. Afin de réduire ce gap technologie/outil, les méthodologies de conception se sont naturellement orientées vers des niveaux d'abstraction supérieurs, où fonctionalités et compromis sont plus faciles à discerner. Aujourd'hui, l'évolution des applications vers des systèmes à la fois plus performants et plus complexes conduit à l'élaboration de dispositifs hétérogènes, c'est à dire constitués d'unités de nature différente (i.e. logicielle, matérielle). Parmi les unités logicielles, on peut distinguer deux grandes classes de circuits numériques : les processeurs d'usage général (GPP) et les processeurs à usage spécifique (ASIP, DSP). Les processeurs d'usage général peuvent être programmés pour exécuter n'importe quelle classe d'application, alors que les processeurs à usage spécifique sont dédiés à une classe d'application (e.g. traitement d'image, traitement du signal, cryptographie) où directement à une application précise (e.g. console de jeux). Un certain nombre d'applications numériques peuvent être réalisées de façon uniquement logicielle, solution intéressante par la flexibilité qu'elle apporte due à la possibilité de reprogrammation. La motivation principale de l'utilisation de processeurs dédiés réside elle dans le respect des contraintes de performances ou dans la confidentialité de la solution implémentée. Toutefois, bien que ces solutions

uniquement logicielles soient préférables, un grand nombre d'applications, notamment dans le domaine des télécommunications et du multimédia, impliquent une architecture hétérogène logicielle / matérielle. Dans ce cas, de nouvelles méthodologies de conception qualifiées de conception conjointe logiciel/matériel (codesign) sont nécessaires. L'approche codesign consiste alors à définir l'ensemble des sous tâches d'une application à intégrer et à effectuer leur répartition sur des cibles logicielles ou matérielles. L'intérêt majeur de cette méthodologie réside dans la recherche d'une adéquation application / architecture satisfaisant les nombreuses contraintes de conception telles que le coût, les performances, la surface, la consommation, les temps de conception et de développement (*Time To Market*), l'évolutivité, ...La conception efficace de ces systèmes hétérogènes nécessite une approche globale dans laquelle les parties matérielles et logicielles sont conçues en parallèle et de façon interactive.

Un autre facteur important dans l'évolution des systèmes modernes est l'apparition de nouvelles architectures exploitant la synergie entre le matériel et le logiciel, basées sur la programmation de circuits matériels tels que les composants FPGAs (*Field Programmable Gate Array*). Ces composants sont actuellement principalement utilisés pour l'accélération de calculs spécifiques ou pour faire du prototypage d'ASIC (*Application Specific Integrated Circuit*). Leur introduction comme unité de calcul alternative et la flexibilité dont ils font preuve introduit une nouvelle dimension au problème de la conception, en élargissant un peu plus l'ensemble des choix d'intégration possibles. De plus, les récentes évolutions des différentes familles autorisent aujourd'hui l'intégration de systèmes de plus en plus complexes avec des contraintes de performances de plus en plus fortes. En effet, de nombreux fondeurs proposent aujourd'hui des puces électroniques intégrant sur un même substrat un ou plusieurs coeurs de processeurs et une matrice programmable (ex : *Excalibur* d'Altera). Par ailleurs, tout un champ technologique émerge actuellement dans le domaine de la reconfiguration dynamique (*run time reconfiguration*). Ces nouveaux circuits permettront de modifier, en cours d'exécution, partiellement ou complètement la configuration (donc la fonctionnalité) du circuit. Aussi, l'utilisation des composants programmables ne se limite plus à une étape dans un cycle de conception (prototypage) mais constitue véritablement une finalité d'implantation. Il est donc nécessaire

d'étendre ou de repenser les approches de conception actuelles afin de les adapter aux possibilités offertes par les technologies matérielles programmable.

OBJECTIFS

L'évaluation des performances d'une application sur une technologie de ce type est un problème peu étudié à ce jour, l'essentiel des travaux porte sur l'optimisation des architectures (dimensionnement des ressources de routage par exemple). En effet, jusqu'à présent les chercheurs ont principalement porté leurs efforts sur l'amélioration des architectures programmables et des outils de synthèse / placement / routage associés afin de les rendre plus performantes et ainsi constituer une réelle alternative aux ASICs. Aujourd'hui, l'intérêt des technologies programmables n'est plus à démontrer et plusieurs projets de recherche visant à imaginer des méthodes de conception conjointe spécifiques à ce domaine sont actuellement en cours. L'objectif du travail

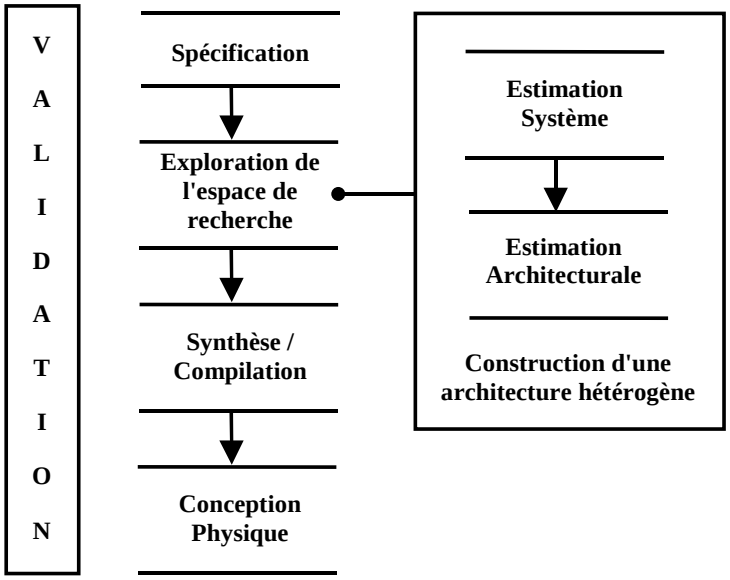


FIG. 1.1 – Flot de Codesign

présenté dans ce mémoire consiste à proposer des techniques et les outils associés permettant l'évaluation rapide de plusieurs compromis performances / occupation pour des applications candidates à une implantatation sur des architectures programmables. L'estimation de la surface et du temps d'exécution à partir d'une description comportementale telle qu'elle est abordée

ici s'intègre dans un flot de conception de systèmes hétérogènes mixtes (co-design), et se situe plus particulièrement à l'étape d'exploration de l'espace de recherche (figure 1.1). Le but de cette exploration est d'évaluer un grand nombre de solutions d'intégration afin de sélectionner la ou les meilleures. L'approche développée au laboratoire LESTER consiste à étudier le partitionnement (répartition des fonctions de l'application sur les ressources logicielles et matérielles) en partie dès la phase de spécification (estimation système), c'est à dire sans aucune notion d'architecture. Les estimations au niveau comportemental (aussi appelées estimation architecturale) qui font l'objet de ce mémoire, permettent ensuite de valider le choix des solutions d'implantation (actuellement en terme de surface et de temps d'exécution) pour les fonctions issues de l'analyse système en effectuant une estimation rapide de la faisabilité et des performances sur un composant candidat. La conjonction des estimations système et architecturale permet ainsi la définition d'une architecture mieux adaptée aux besoins de l'application.

CONTRIBUTION

Le travail exposé ici porte sur l'estimation en temps et en surface de fonctions candidates à une implantation matérielle, plus précisément sur des composants programmables. L'estimateur peut donc être vu comme un outil permettant de vérifier la faisabilité de l'intégration d'un système sur un FPGA, à partir d'une spécification comportementale issue d'un code de haut niveau. Ce travail est original en plusieurs points : Tout d'abord, peu d'études sont menées dans le domaine, en partie pour les raisons précédemment citées, à savoir l'intérêt nouveau des chercheurs pour la définition de méthodes de conception conjointe appliquées aux technologies matérielles programmables. A notre connaissance, seules trois études visant à définir des méthodes d'estimation qui tiennent compte de l'application à intégrer ont été publiées (ce point sera repris en détail dans l'état de l'art). De plus, ces études ne prennent pas en compte les structures de contrôle intervenant dans les applications complexes, ce qui limite leur utilisation à l'estimation de chemins de données. Dans l'approche présentée ici, la spécification supporte différentes structures de contrôle telles que les boucles et les structures conditionnelles. Ainsi, l'estimation d'applications complexes (incluant par exemple de la hiérarchie, des données multidimentionnelles) peut être effectuée.

La méthode développée intègre l'estimation des unités de traitement, de

mémorisation et de contrôle, ce qui constitue un point original dans la mesure où il existe très peu d'approches s'intéressant à ces trois aspects simultanément. Ce point est nécessaire pour obtenir une estimation globale d'une application sur une architecture matérielle programmable. La validation de la méthode s'est donc portée sur des systèmes représentatifs de deux classes d'applications (orientées vers du traitement, du contrôle ou de la mémorisation de données) : codage audiofréquence et traitement d'image. Enfin, bien que l'étude présentée dans ce mémoire vise essentiellement les composants programmables, elle constitue un point de départ intéressant dans le cadre de l'élaboration d'une technique d'estimation matérielle qui puisse aussi s'appliquer aux ASICs. Un point également fondamental est la définition d'une méthode d'estimation qui soit au maximum indépendante d'une technologie donnée, afin de rendre son utilisation la plus large possible. Aussi, dans le cadre de cette thèse, nous nous sommes efforcé de rendre l'approche la plus générique possible.

Le rapport précision / complexité est une caractéristique importante pour un estimateur puiqu'elle conditionne l'exploration d'un vaste espace de recherche (exploration du parallélisme, pour plusieurs fréquences d'horloge, plusieurs allocations) et l'évaluation de plusieurs composants cibles en un temps raisonnable. D'autre part, étant données les récentes avancées dans le domaine de la synthèse de haut niveau et le degré de maturité atteint par les outils de synthèse architecturale, la méthode d'exploration développée ne se justifie que si elle apporte une amélioration significative du cycle de conception. Aussi, lors de la définition des techniques d'estimation, la complexité des algorithmes a été un critère important. Nous verrons dans la dernière partie de ce mémoire au travers des exemples choisis pour la validation que l'exploration et la synthèse architecturale ne s'opposent pas. Au contraire, leur complémentarité laisse entrevoir les perspectives d'une méthodologie de conception prometteuse.

Plan du mémoire

Le plan de cette étude est le suivant : Dans le chapitre 2, après une présentation du codesign et de sa problématique, une introduction de l'approche effectuée au laboratoire est réalisé afin de présenter le contexte dans lequel s'inscrit la méthode d'estimation. Un bilan des nombreuses techniques d'estimations classées par niveau d'abstraction est ensuite présenté afin de bien

dégager l'utilisation potentielle des différents algorithmes. Puis, les composants ayant servi à valider les travaux, à savoir la famille Virtex de Xilinx et Apex d'Altera sont abordés. Cette présentation permet au lecteur non familier avec ces technologies d'avoir une vision synthétique des deux familles les plus largement utilisées. Le chapitre 3 introduit les hypothèses à prendre en compte pour garantir la pertinence des résultats et propose la définition d'un flot d'estimation. Celui-ci est composé de deux parties, les estimations au niveau structurel (RTL ou architectural) où sont définies un certain nombre de solutions architecturales, et les estimations au niveau physique (layout) qui permettent d'évaluer la surface et les performances de ces solutions. Ces étapes sont détaillées dans les deux chapitres suivants. Le chapitre 6 présente les résultats de cette méthode sur deux applications représentatives dans le domaine de la vidéo et de l'audio. Enfin, le dernier chapitre conclut ce mémoire et évoque les différentes perspectives possibles par rapport au travail mené et aux résultats obtenus.

Chapitre 2

Méthodologies de conception

Les techniques et les capacités d'intégration des circuits électroniques n'ont cessé de s'accroître au cours de ces dernières années. L'intégration de systèmes toujours plus complexes nécessite la définition de nouvelles méthodes de travail afin d'exploiter pleinement les évolutions technologiques. C'est dans ce cadre que s'inscrivent les travaux présentés dans ce mémoire.

Dans ce chapitre, nous effectuons tout d'abord une brève présentation du domaine de la conception des systèmes mixtes logiciels / matériels, afin d'introduire et de positionner l'approche développée au laboratoire. Cette dernière est basée sur l'utilisation d'estimateurs très tôt dans le flot de conception (avant l'étape de partitionnement), ce qui constitue un point original. En effet, deux étapes interviennent successivement : l'estimation "Système" et l'estimation "Architecturale". Ces deux étapes sont complémentaires et consistent en un raffinement progressif des estimations. Aussi, la deuxième partie de ce chapitre est consacrée aux estimateurs afin de donner au lecteur une vision claire des méthodes d'estimation existantes et de positionner le travail présenté dans ce mémoire, à savoir une méthode d'estimation architecturale. Enfin, la technique proposée visant principalement les composants programmables, une présentation des deux architectures les plus représentatives du domaine est effectuée.

2.1 La conception conjointe logiciel / matériel

2.1.1 Problématique du codesign

Le problème de la conception conjointe logiciel / matériel provient de la nécessité de trouver une répartition équilibrée des fonctions d'une application sur les composants d'un système numérique dans le but d'aboutir à une architecture aux performances maximales sous contraintes de coût et de consommation d'énergie. L'évolution des technologies au cours de ces dernières années permet aujourd'hui l'intégration d'applications très complexes au sein d'une même puce (on parle de *SOC*, System On Chip). L'avènement des systèmes portables ou embarqués (téléphonie mobile, multimédia, communications sans fil ...) imposent des contraintes de plus en plus fortes car leur conception fait appel à des domaines variés tels que l'électronique analogique, numérique, les hyper fréquences, ... Si on se limite au seul domaine du numérique, les solutions d'intégration sont déjà multiples. La plus économique consiste à utiliser des processeurs. Qu'ils soient d'usage généraux ou spécifiques (*DSP*, Digital Signal Processor), leur emploi constitue généralement la solution la plus rapide en terme de temps de développement puisqu'elle consiste en la réalisation d'un programme, mais c'est aussi la moins intéressante en terme de performances. À l'opposé, la solution matérielle consiste à développer un circuit intégré entièrement dédié à l'application, ce qui permet toutes sortes d'optimisations que ce soit en performances, surface ou consommation d'énergie. Par rapport à une intégration logicielle, cette solution est plus chère en temps et en coût car la définition de l'architecture d'un circuit intégré est généralement beaucoup plus longue que l'écriture d'un programme, et qu'il faut en plus rajouter les délais de fonderie et de test. Ainsi, la solution intermédiaire qui consiste à combiner les avantages de ces deux approches est un challenge intéressant car elle permet l'élaboration d'architectures mieux adaptées aux contraintes de conceptions (figure 2.1). Typiquement, les parties évolutives et peu critiques sur le plan des performances sont placées sur un ou plusieurs processeurs alors que les fonctionalités coûteuses en temps et en consommation d'énergie sont elles candidates à une intégration matérielle. Les considérations d'ordre économique peuvent aussi entrer en compte en favorisant par exemple une architecture logicielle

pour une mise à disposition rapide sur le marché (*Time To Market*). Le problème revient donc à choisir les composants logiciels / matériels ainsi que leur nombre (allocation), et trouver la répartition des fonctionalités du système sur ces composants (partitionnement).

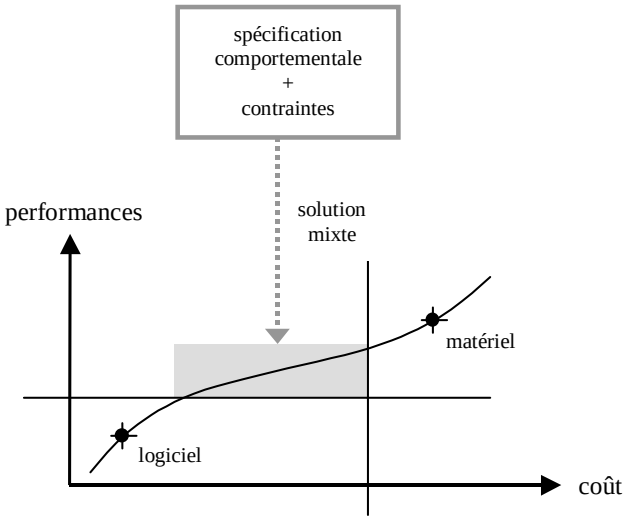


FIG. 2.1 – Les différentes solutions d'intégration d'un système

Il est clair aujourd'hui que les outils de CAO sont indispensables pour une conception sûre de ces systèmes électroniques tout en réduisant les coûts et les temps de conception. Cela explique l'intérêt grandissant des industriels et du monde de la recherche dans le domaine de la conception conjointe (codesign) depuis les premiers travaux sur l'analyse et la synthèse assistée par ordinateur de systèmes mixtes à l'université de Stanford en 1990 [34].

2.1.2 Limites des approches classiques

Un rapide survol de quelques outils et méthodes [34][35][36] révèle deux grandes constatations. Premièrement, beaucoup d'entre eux trouvent leur limites dans une classe d'application et ceci à cause du modèle d'architecture et du modèle de spécification. D'autre part, toutes ces techniques reposent sur le principe de partitionnement, qui consiste à segmenter la spécification dans le but de réaliser l'intégration du système sur plusieurs composants. C'est lui qui constitue le coeur du processus de la conception conjointe. Les

fonctions composant le système sont réparties sur les différentes unités du modèle d'architecture et la qualité de ce partitionnement, qu'il soit manuel ou automatique, est évalué au moyen d'une fonction de coût, d'estimateurs ou directement par la vérification des performances après intégration du système. La qualité de l'intégration repose donc essentiellement sur celle du partitionnement et finalement, les propriétés intrinsèques de l'application n'ont pas une influence directe sur la topologie de l'architecture. L'approche originale développée au LESTER cherche au contraire à guider la construction de l'architecture en fonction des propriétés intrinsèques de l'application. L'architecture est élaborée de façon pragmatique en précisant progressivement les hypothèses architecturales à mesure que l'on progresse dans l'analyse du système.

Peu d'approches font intervenir les estimations avant l'étape de partitionnement. Or l'emploi d'estimateurs peut se faire dès les phases de spécification et permettre ainsi de guider les choix de conception d'une architecture hétérogène pour une meilleure adéquation avec l'application. L'architecture peut être élaborée par un raffinement progressif de l'analyse de la spécification au moyen d'estimations opérant à différents niveaux d'abstraction. Il n'y a alors plus de modèle d'architecture prédéfini, ni de notion de partitionnement tel qu'elle a été précédemment définie, ce qui simplifie à priori le processus de conception. La réduction de complexité résultant de l'emploi d'estimateurs peut alors permettre l'exploration d'un espace de recherche plus important.

2.2 Codesign : l'approche du LESTER

2.2.1 Principe de l'approche

L'idée consiste à commencer la conception d'une architecture ad hoc dès les phases de spécification et de conception préliminaire des applications. L'intérêt de se situer à un tel niveau d'abstraction reside dans le fait que c'est à ce stade de la conception que les choix effectués ont le plus d'impact sur les performances finales du système. On peut distinguer deux points essentiels dans ce flot (figure 2.2) : l'estimation "Système" et l'estimation "Architecturale".

Les estimations au niveau système permettent de caractériser un ensemble

de fonctions et de les classer par ordre de criticité. La criticité traduit le poids du choix d'implantation d'une fonction sur les performances finales de l'ensemble des fonctions. Les estimations sont décorrélées de toute hypothèse architecturale, c'est à dire que seule la spécification comportementale est prise en compte pour cette analyse. Le coût et les performances de chaque

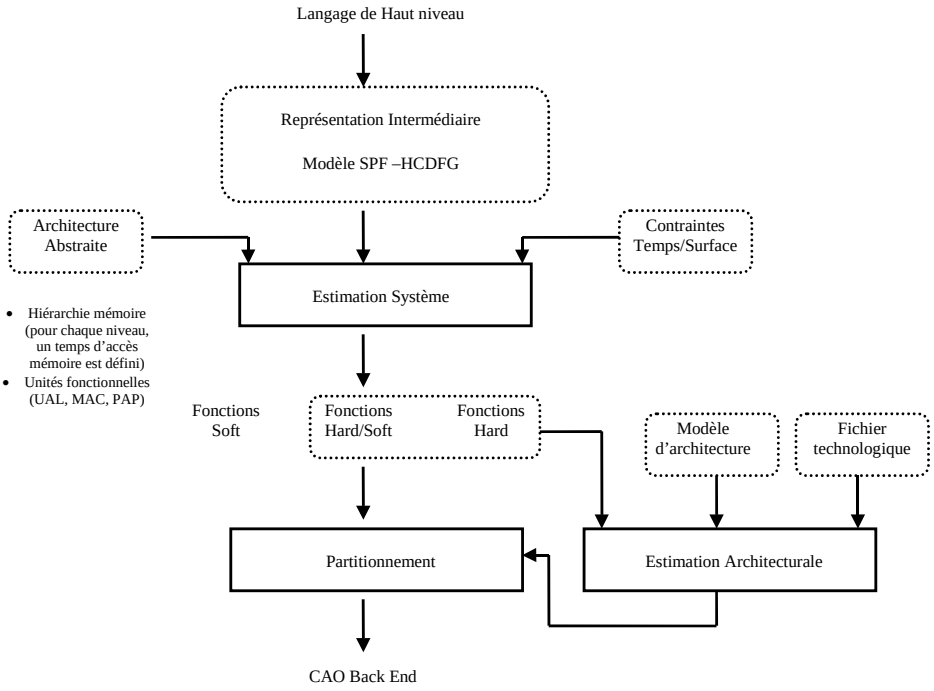


FIG. 2.2 – Flot de conception : "Design Trotter"

fonction sont ainsi estimés en tenant compte des intéractions avec les fonctions plus critiques et déjà estimées. La dernière étape de l'estimation système consiste en une projection de chacune des fonctions de l'application sur une cible modélisée afin de définir d'une part les fonctions candidates à une implantation logicielle, d'autre part les fonctions préssenties pour une implantation matérielle et enfin celles pour qui le choix ne peut être fait à ce niveau d'abstraction et nécessite par conséquent d'être traitées par les étapes "aval". Ainsi, le concepteur dispose dès l'étape de partitionnement d'une décomposition logicielle / matérielle à partir de laquelle il peut affiner la solution finale.

L'estimation architecturale intervient afin d'évaluer les performances des fonctions candidates à une implantation matérielle et celles pour qui le choix

n'est pas encore établi. L'étape d'estimation architecturale a donc pour rôle de déterminer une solution d'implantation pour toutes les fonctions pouvant être implantées sur une cible matérielle, en l'occurrence, ici un FPGA. À partir de ces informations plus précises, l'étape de partitionnement effectue les choix définitifs d'implantation et définit les fonctions associées aux unités logicielles, celles relevant des unités matérielles ainsi que les supports de communication de l'architecture finale.

L'originalité de cette approche par rapport aux autres outils de codesign tient dans la stratégie qui permet une construction incrémentale de l'architecture et une exploration de l'espace de recherche à partir d'un très haut niveau d'abstraction.

2.2.2 Spécification : le modèle SPF - HCDFG

La description comportementale de l'application est saisie dans un langage de haut niveau et est ensuite traduite dans un modèle de représentation intermédiaire : le modèle SPF [29]. C'est sur ce modèle que les différents outils travaillent.

Le principal objectif de ce modèle est la spécification de systèmes numériques à un haut niveau d'abstraction pour permettre l'exploration de l'espace de recherche. Il permet la prise en compte de toutes les informations telles que les opérations, le contrôle, les opérandes multi-dimensionnels nécessaires à l'estimation des unités de traitement, contrôle, mémoire et au partitionnement de l'application sur une architecture hétérogène. Le modèle SPF est basé sur cinq concepts de base qui sont la concurrence, la hiérarchie, les communications, la synchronisation et le temps. La concurrence permet d'exprimer les calculs et accès aux données en parallèle. La hiérarchie permet de gérer la complexité due à l'importance du traitement et du contrôle des applications modernes. La communication traduit les échanges de données et de contrôle entre différentes tâches (*process*). La synchronisation est nécessaire pour modéliser les dépendances d'exécution entre processus concurrents et le temps lui, sert à spécifier les contraintes temporelles.

SPF est un modèle graphique basé sur trois vues (figure 2.3) : *Système*, *Process* et *Fonction*. Ces vues sont hiérarchiques et on passe à la vue suivante quand la granularité de la vue courante ne permet plus de décomposition.

La vue *système* correspond au plus haut niveau de spécification possible. Elle permet la capture des différentes configurations du système à concevoir

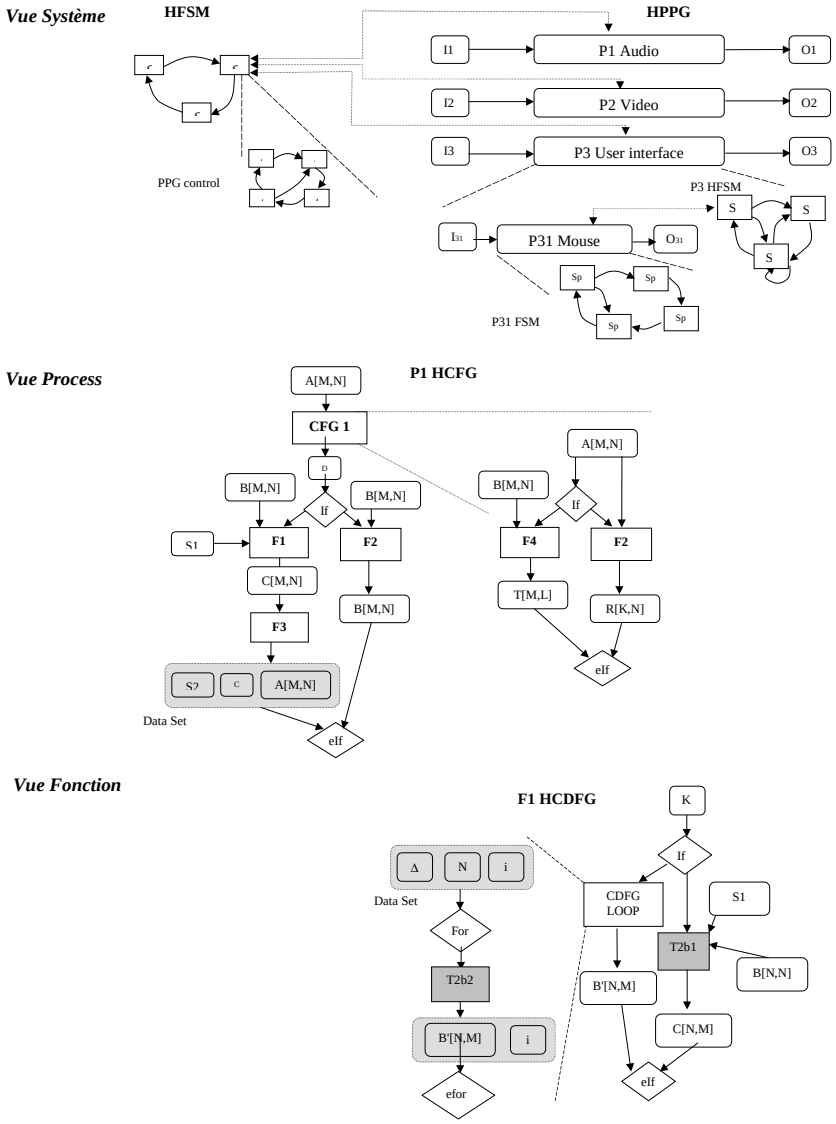


FIG. 2.3 – Le modèle SPF

(set top box par exemple), en montrant tous les différents processus à déclencher pour fonctionner dans une configuration, et la structure de contrôle nécessaire à leur mise en route. Elle utilise un modèle à états finis où un flot de données est associé à chaque état. La hiérarchie permet de modéliser les sous-états du système. La vue *process* permet de décrire chaque processus

du système. C'est un graphe hiérarchique de fonctions (autocorrélation ou filtrage par exemple) liées par des structures de contrôle et qui échangent des données. Ces données peuvent être scalaires ou multi-dimensionnelles. Dans ce dernier cas, les modes d'adressage sont explicitement représentés afin de permettre l'estimation mémoire. Le plus petit noeud de traitement dans cette vue correspond à une fonction. Le modèle utilisé dans la vue *process* est un graphe flot de contrôle hiérarchique (*HCFG*). Aucun traitement n'y apparait explicitement : ils sont encapsulés dans les fonctions. La vue *fonction* décrit chaque fonction associée aux processus à l'aide d'opérations élémentaires ou de graphes hiérarchiques d'opérations élémentaires. Les structures de contrôle et les échanges de données sont aussi présents dans cette vue. Le modèle utilisé à ce niveau est donc un graphe flot de contrôle et de données hiérarchique (*HCDFG*). Signalons ici que la méthode d'estimation architecturale présentée travaille uniquement à ce niveau (l'estimation architecturale vise à estimer une fonction sur une cible matérielle).

La figure 2.3 montre un exemple de spécification utilisant le modèle SPF. La vue *système* est composée de trois processus dont la décomposition hiérarchique correspond à d'autres machines à états finis (P3 User interface) ou à des sous-process (P1 Audio). La vue *process* permet la représentation de ces derniers sous forme de graphe flot de contrôle hiérarchique (P1 HCFG associé au processus P1 Audio par exemple). Puis la vue *fonction* décrit les traitements associés aux sous-fonctions par un modèle flot de contrôle et de données, comme l'illustre la sous fonction F1.

La représentation graphique du modèle permet le passage entre les différentes vues et leur hiérarchie. Le concepteur peut ainsi facilement accéder à tous les niveaux de la description du système. Grâce à ce modèle, on peut représenter un large spectre d'applications orientées contrôle ou traitement. Il permet la spécification et la représentation de systèmes hétérogènes dans le but d'effectuer les étapes d'estimation, de partitionnement et est adapté à l'approche codesign. Le concepteur doit d'abord décrire les vues *système* et *process* car ces étapes reposent esentiellement sur son analyse du système à concevoir et font largement appel à son expérience de concepteur. Le modèle *HCDFG* correspondant à la vue *fonction* est obtenue à l'aide d'un parser sur la spécification en entrée, actuellement décrite en langage C.

Après avoir présenté le flot de conception développé au LESTER, ainsi

que le modèle de spécification des applications, nous présentons une vision synthétique des méthodes mises en oeuvre dans les étapes d'estimation système et architecturale. Ceci permettra de positionner plus clairement les travaux développés au LESTER par rapport à l'ensemble des techniques d'estimation existant à ce jour, techniques dont la présentation fait l'objet de la section 2.3.2.

2.2.3 L'Estimation Système

Il ne s'agit pas dans cette partie d'expliquer les techniques mises en oeuvre durant l'estimation système (ceci sort du cadre de ce mémoire), mais plutôt de donner au lecteur une vision globale de l'approche afin de bien comprendre la philosophie du flot développé au LESTER.

La méthodologie de l'estimation système repose sur quatre points originaux [37]. Premièrement, une caractérisation macroscopique de l'ensemble de l'application est réalisée afin de déterminer l'orientation des différentes fonctions en terme de traitement, contrôle ou mémoire (TCM par la suite). Le second point est le niveau d'abstraction élevé qui permet d'obtenir des valeurs d'estimation indépendantes de tout modèle architectural. Troisièmement, le coût est caractérisé pour différentes contraintes du nombre de cycles alloués à une fonction, ce qui conduit à exploiter plus ou moins le parallélisme intrinsèque de la fonction (figure 2.4). Ainsi, chaque fonction est caractérisée sous la forme d'une courbe (coût vs nombre de cycles), ce qui permet d'explorer un plus grand nombre de solutions architecturales et par conséquent, de retenir une solution en meilleure adéquation avec l'application. Le quatrième point porte sur l'inter-dépendance de l'estimation : le traitement d'une fonction prend en compte les interactions possibles avec le reste de l'application dans le but d'une optimisation efficace.

Le flot d'estimation système s'articule principalement autour des points suivants :

- La caractérisation macroscopique : elle permet d'extraire la nature de l'application spécifiée. Dans un premier temps, un comptage du nombre d'opérations de traitement, de contrôle et de mémorisation (TCM par la suite) est effectué pour chaque fonction séparément. Cette caractérisation fournit un critère de sélection d'un ensemble de "fonctions

critiques" de l'application (fonctions les plus coûteuses d'un point de vue des métriques TCM). Ceci signifie qu'on ne considère que les fonc-

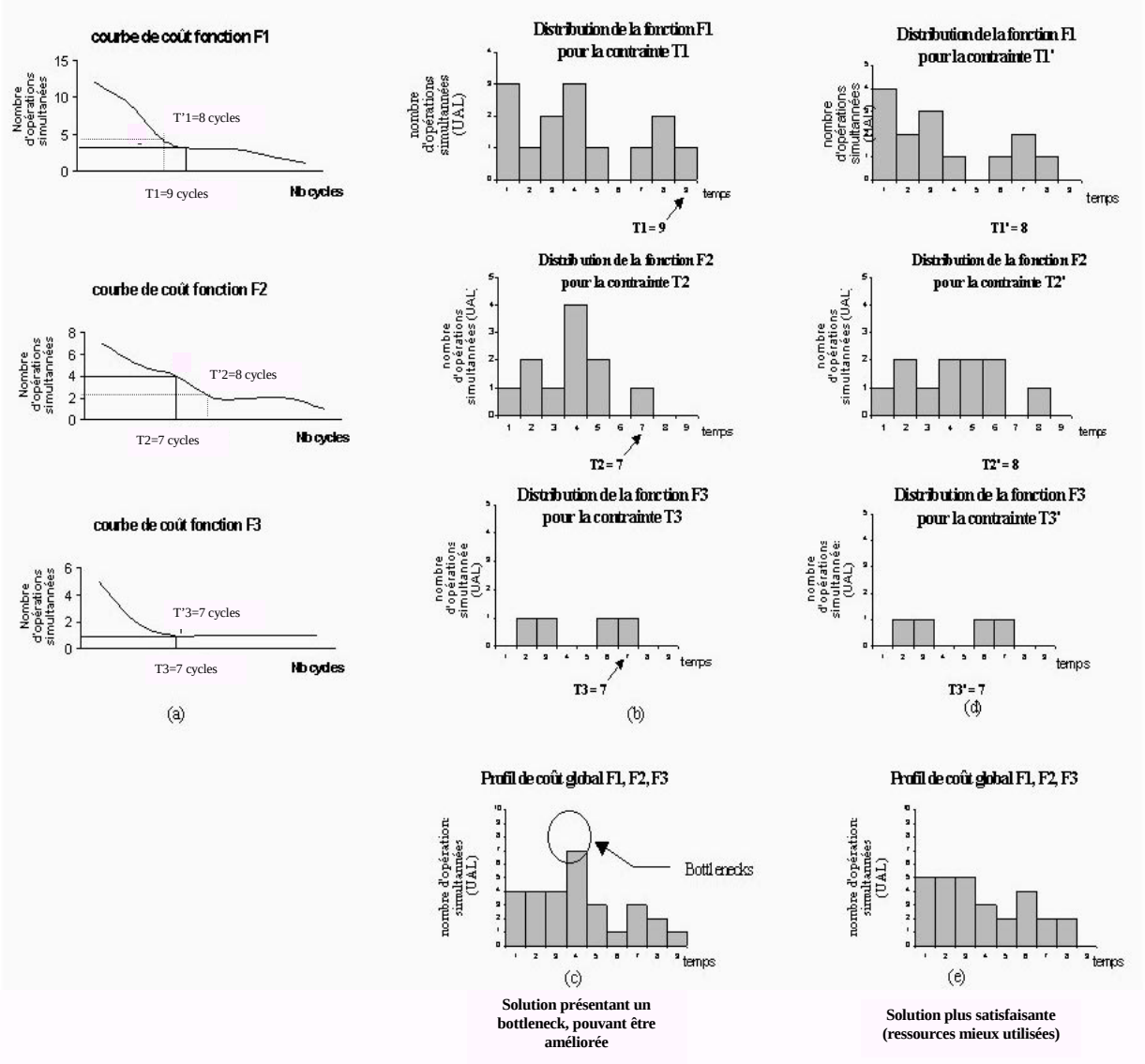


FIG. 2.4 – Exemple de profil de coût global

tions qui ont le plus d'impact sur l'architecture finale.

– La co-estimation intra-fonction : étant donné un ensemble de fonctions critiques, le but est de calculer pour chaque fonction la courbe de coût du nombre d'opérations simultanées pour plusieurs contraintes de

temps exprimées en nombre de cycles (figure 2.4.a.).

- L'ordre d'analyse des fonctions critiques : à partir de la classification issue du calcul des métriques TCM, un raffinement est effectué en tenant compte de la nature des liens et des potentiels d'optimisation (partage de ressources, ou minimisation d'accès à la mémoire, ...) existant entre les fonctions.
- La co-estimation inter-fonction : comme pour la co-estimation intra-fonction, des courbes de coût sont calculées pour chaque fonction et tiennent compte du partage de ressources et de données avec les fonctions précédemment estimées, donc plus critiques.
- Le profil de coût global : il traduit le coût de l'ensemble des fonctions critiques à chaque cycle, c'est à dire le nombre de ressources utilisées simultanément du premier au dernier cycle d'exécution (qui correspond à la contrainte de temps). Ces profils permettent de repérer les goulots d'étranglement qui correspondent à de fortes discontinuités dans la répartion des opérations (figure 2.4.c.). En redistribuant le parallélisme des fonctions concernées, on "lisse" les courbes de façon à obtenir une répartition plus homogène et donc un coût moins important (figure 2.4.e.).
- les transformations algorithmiques : une fonction primitive pouvant être décrite par plusieurs algorithmes (FFT par exemple), le concepteur peut modifier ses choix d'algorithmes pour améliorer l'adéquation.

Il résulte de cette analyse système un ensemble d'informations intéressantes pour l'aide à la conception d'une architecture ad hoc. Elle fournit à l'étape de partitionnement un graphe sur lequel sont annotées un certain nombre de contraintes à respecter concernant le parallélisme (degré de déroulage des boucles par exemple). Elle fournit de plus un ensemble de contraintes séquentielles exprimées sous la forme de dépendances d'optimisation (réutilisation, optimisation des accès mémoire) et une étape de pré-ordonnancement (ordonnancement simplifié utilisé pour le calcul des profils de coût). Ces informations permettent de réduire l'exploration de l'espace de recherche pendant le partitionnement. Les estimations architecturales interviennent alors pour fournir des valeurs d'estimation précises sur le coût et les performances attendus du placement des diverses fonctions sur des composants cibles.

Il existe différents degrés de couplage entre l'estimation système et l'es-

timation architecturale. Ces derniers conduisent à des compromis entre l'exploitation du parallélisme, la précision des résultats obtenus et le temps de calcul machine. Dans le cadre de ce mémoire, nous limitons la présentation à un couplage faible : l'estimation système fournit à l'estimation architecturale une liste de fonctions potentiellement candidates à une réalisation matérielle. L'ensemble des caractéristiques résultant de l'estimation système ne sont donc pas prises en compte, car comme nous le verrons dans les chapitres suivants, l'estimation architecturale effectue de nouveau un ordonnancement afin de mettre en évidence le parallélisme potentiel. Le choix de présenter uniquement ce lien entre les deux étapes d'estimation provient de la volonté de proposer une méthode d'estimation qui s'inscrive dans le flot de conception développé au LESTER, mais qui puisse également fonctionner de façon autonome.

2.2.4 L'Estimation Architecturale

L'architecture d'un système hétérogène est composée d'unités de nature différente sur lesquelles s'exécutent les diverses fonctionnalités du système. L'exploration de l'espace de recherche doit permettre de tester rapidement la

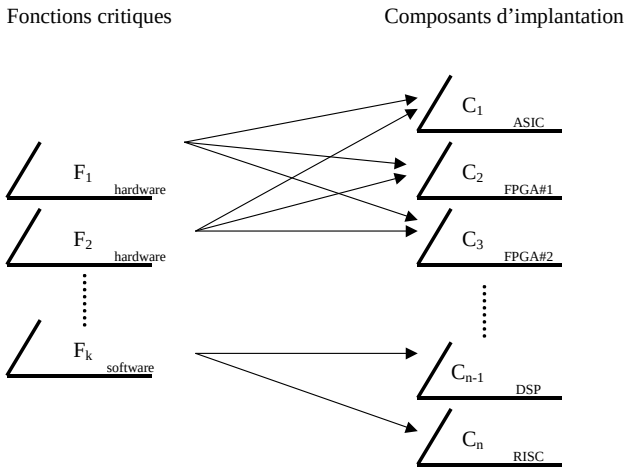


FIG. 2.5 – Allocation de composants

faisabilité et les performances de ces fonctions sur plusieurs types de technologies (ASICs, FPGAs, DSPs, ...) afin de sélectionner la meilleure solution d'implantation. La figure 2.5 montre un exemple de fonctions critiques et de

composants candidats pour l'intégration. Le choix des composants est guidé par les métriques de caractérisation macroscopique établies à l'étape des estimations système (par exemple, une fonction effectuant du calcul intensif pourra être candidate à une implantation matérielle). L'estimation architecturale permet ensuite d'évaluer le coût de l'intégration de chaque fonction sur plusieurs composants, de façon à ne garder que le plus satisfaisant. Le travail présenté ici ne s'intéresse qu'à l'estimation pour une implantation matérielle, plus particulièrement les FPGAs. La méthode doit donc pouvoir s'appliquer à toutes les familles de composants de ce type et constitue en outre un point de départ à partir duquel une technique d'estimation globale prenant en compte l'estimation des ASICs peut être développée.

L'estimation d'une fonction décrite au niveau comportemental par un graphe flot de données et de contrôle hiérarchique (*HCDFG*) sur un composant FPGA a pour but de vérifier la faisabilté de l'intégration, d'évaluer le coût en surface (taux d'occupation) et les performances en vitesse d'exécution. Une exploration basée sur l'analyse du parallélisme, l'influence de l'allocation de ressources et de la période d'horloge, ainsi que la prise en compte de la logique de contrôle et de l'unité de mémorisation est effectuée afin de caractériser le coût de manière complète et réaliste. Les objectifs suivants ont été fixés :

- proposer une méthode globale (prise en compte des aspects traitement, contrôle, mémoire).
- proposer une méthode qui exploite le parallélisme potentiel en caractérisant les fonctions selon une courbe surface vs vitesse d'exécution.
- proposer une méthode de complexité faible afin de pouvoir évaluer plusieurs composants.
- proposer une méthode qui puisse facilement s'étendre à d'autres composants et qui ne soit pas spécifique à une seule famille.
- proposer une méthode qui s'inscrive dans le flot de conception du LESTER, mais qui puisse aussi fonctionner de façon indépendante.
- proposer une méthode qui prenne en entrée une application intégrant de la hiérarchie, du parallélisme et des structures de contrôle.

Comme nous le verrons dans la suite du mémoire, ces objectifs ont été atteints. L'approche devait également inclure l'estimation de la puissance et une prise en compte fine du routage des composants FPGA. Malheureusement,

ces travaux n'ont pu être finalisés.

Afin de définir les méthodes mises en oeuvre, un état de l'art approfondi a été effectué. Il existe en effet un grand nombre de méthodes d'estimation dans la littérature, mais aucune à notre connaissance ne réunit toutes les conditions nécessaires pour une exploration rapide à partir d'un modèle de spécification complexe autorisant la hiérarchie et les structures de contrôle. Cependant, les techniques développées dans la littérature constituent une base riche à partir de laquelle une méthode globale peut être développée. La suite de ce chapitre présente ces différentes méthodes.

2.3 État de l'art dans le domaine des estimations

Les estimations jouent un rôle essentiel dans le processus de conception puisqu'ils permettent de renseigner le concepteur et de le guider dans ses différents choix. Un très grand nombre de méthodes d'estimation sont décrites dans la littérature, diversité qui s'explique par la complexité du processus de conception où les estimateurs sont nécessaires à tous les niveaux : de la spécification jusqu'au plan de masse, ceci pour chaque type d'unité composant l'architecture (unités de traitement, contrôle, mémoire, communication) pour plusieurs types de contraintes (temps, surface, consommation, ...) et de technologies (ASIC, FPGAs ...). Nous proposons donc de passer en revue différentes méthodes d'estimation à partir d'une description comportementale classées par le niveau d'abstraction auquel elles s'appliquent (système ou RTL), ainsi que celles plus spécifiques aux FPGAs (niveau logique).

2.3.1 État de l'art des estimations au niveau système

Les avancées dans le domaine technologique permettent aujourd'hui l'intégration d'un grand nombre de fonctions au sein d'un même système. Les contraintes de conception imposent des temps de développement plus courts et donc réduisent le temps disponible pour la recherche de la meilleure adéquation application - système. L'estimation système est donc nécessaire pour guider le concepteur dans ses différents choix à l'étape d'exploration de l'espace de recherche. Le niveau système consitue le niveau d'abstraction le

plus élevé. L'analyse au niveau système d'une description comportementale permet de définir un certain nombre de caractéristiques qui renseignent le concepteur sur les propriétés du système indépendamment de toute notion d'architecture ou d'implantation physique (découpage en sous-fonctions, degré de granularité, ...).

Vahid propose dans son approche [38], le calcul de métriques de proximité pour le partitionnement au niveau fonctionnel. Ces métriques sont définies au niveau système et permettent l'évaluation du rapprochement de deux sous parties de la spécification, en terme de calcul ou de communication. Elles sont utilisées dans des fonctions de coût basées sur des pondérations empiriques qui dépendent de l'application. Des métriques similaires sont utilisées, à un niveau de granularité plus faible, pendant la phase de regroupement d'opérations logiques [39], arithmétiques [40] et instructions [41]. Sur le plan architectural, un grand nombre de métriques sont définies dans [42] et [43] pour la communication entre opérateurs arithmétiques. Dans le cas du partitionnement sur une architecture cible donnée, des estimations sont utilisées par les fonctions de coût de l'algorithme de partitionnement [44] [45]. Le principe consiste à évaluer les conséquences en termes de surface et performances avant de modifier éventuellement les choix d'implantation. Dans [45], le but est de définir dynamiquement la granularité de l'application.

Ces estimations ont pour but de guider la conception en proposant un certain nombre de métriques à partir desquels des choix sont effectués. Plus ces métriques interviennent tôt dans le flot de conception, plus elles sont indépendantes de l'implémentation et permettent d'apprécier globalement, et non plus localement, le réel potentiel d'optimisation.

2.3.2 État de l'art des estimations au niveau RTL

Le niveau RTL correspond au niveau d'abstraction permettant de décrire une architecture. Les méthodes d'estimation opérant à ce niveau ont pour but d'évaluer le nombre de composants (les multiplieurs, les registres, les mémoires, ...) sous des contraintes telles que le temps d'exécution, la surface. Un grand nombre de méthodes ont été développées au cours de la recherche sur la synthèse architecturale (synthèse de haut niveau) pendant ces vingt dernières années. Beaucoup de travaux concernent l'estimation de

l'unité de traitement qui représente souvent la partie la plus coûteuse pour les algorithmes de traitement du signal. On note aussi l'apparition d'un grand nombre d'études sur les aspects mémorisation avec l'avênement des applications de type vidéo numérique.

Le coût d'une architecture dédiée au traitement du signal dépend principalement de l'unité de traitement et de l'unité de mémorisation. La réalisation de l'unité de traitement constitue souvent la première étape. Aussi, un grand nombre de ces techniques vise l'estimation du coût de l'unité de traitement, sous contrainte de temps réel. Le problème consiste à évaluer les besoins en terme d'unités fonctionnelles à partir de la description comportementale d'un algorithme. Le compromis temps / coût peut être formalisé de plusieurs façons.

Dans [6], les auteurs (Rabaey et Potkonjak) le caractérisent au moyen de bornes supérieures et inférieures. Le principe du calcul de la borne supérieure consiste à chercher entre $t = 0$ et $t = T$ (T étant la contrainte de temps) l'instant où le parallélisme est à son maximum ce qui se produit lorsque la superposition d'opérations appartenant à un même type est maximale. Il s'agit d'une limite surestimée reflétant le pire cas. La limite inférieure est obtenue en moyennant le temps nécessaire à l'exécution des n_i opérations de type i dans le graphe ($n_i * \Delta_i$, où Δ_i représente le délai de l'opérateur réalisant l'opération i) par la contrainte de temps T. Cette valeur n'est exacte que dans le cas où le taux d'utilisation de la ressource i est de 100%. L'étude est ensuite étendue en cherchant à inclure le problème des dépendances de données.

Sharma et Jain [5] ont élaboré une méthode permettant d'obtenir la borne inférieure du nombre d'opérateurs en tenant aussi compte des dépendances entre noeuds du graphe. Le principe repose sur l'analyse de l'activité des opérations du graphe dans un intervalle $[\delta_1, \delta_2]$ compris dans $[0, T]$ où T est la contrainte de temps. Ainsi, chacune des opérations susceptibles d'intervenir dans cet intervalle de temps est ordonnancée à sa date au plus tôt ($ASAP$) si elle minimise la durée d'activité entre δ_1 et δ_2 et à sa date au plus tard ($ALAP$) dans le cas inverse. Le nombre d'opérations obtenu est ensuite divisé par la longueur de l'intervalle considéré pour donner une valeur moyenne. Celle-ci traduit une équirépartition temporelle des opérations, donc un parallélisme minimal. La valeur maximale obtenue sur l'ensemble des intervalles

possibles fournit finalement la borne minimale du nombre d'opérateurs. Les auteurs estiment l'erreur moyenne à 5% pour une complexité en $O(NC^2)$.

Les méthodes probabilistes [2] (Diguet) permettent des estimations de complexité beaucoup plus réduite, de l'ordre de $O(N)$ selon le type d'estimation probabiliste. Comme dans les méthodes précédentes, les dépendances de données sont considérées au travers du calcul des dates *ASAP* et *ALAP*, et l'étude se fait indépendamment pour chaque opérateur. La méthode repose sur le principe suivant : une opération peut être exécutée aléatoirement (la loi est choisie uniforme) entre sa date *ASAP* et sa date *ALAP*. Pour chaque type d'opérateur, on cumule les probabilités d'exécution des opérations à chaque cycle. La somme maximale trouvée, arrondie à l'entier supérieur est le nombre d'opérateurs requis. Contrairement aux méthodes précédentes, les estimations probabilistes ne fournissent ni une borne inférieure, ni une borne supérieure. La réduction de la complexité permet en outre de caractériser le coût de manière dite "dynamique", c'est à dire sous forme de courbes en fonction d'une contrainte de temps variable.

Les méthodes présentées ci-dessus ne s'appliquent que dans le cas de spécifications qui ne comportent pas de structures de contrôle. En général, une description comportementale comporte aussi des structures conditionnelles et itératives, une méthode d'estimation réaliste doit donc en tenir compte.

Dans la méthode [3] (Narayan et Gajski), les auteurs s'intéressent à l'estimation temps / surface d'une application décrite dans un modèle de description qui accepte les structures de contrôle, ainsi que la spécification hiérarchique et l'exécution concurrente (SpecChart). Le principe repose sur l'emploi de probabilités pour pondérer les temps d'exécution de chaque branche d'une structure conditionnelle. Ces probabilités peuvent être définies par l'utilisateur, issues d'une analyse type "vecteurs de test" (*profiling*) ou tout simplement suivre une loi uniforme (équirépartition des probabilités en fonction du nombre de branches). Outre la prise en compte de l'aspect contrôle de la spécification, la méthode propose une technique d'évaluation de la surface de l'unité de contrôle basée sur l'estimation du nombre de pas de contrôle et de signaux nécessaires pour le séquencement des unités de traitement et de mémorisation. Des extensions aux cas des mémoires multi-ports, au traitement du pipeline et au calcul d'autres métriques de performances sont présentés par les mêmes auteurs dans [4].

Les techniques décrites ci-dessus sont basées sur des modèles simples de façon à obtenir des valeurs d'estimation rapide, ceci bien sûr au détriment de la précision. Une autre approche pour obtenir des valeurs d'estimations consiste à utiliser directement une technique d'ordonnancement. Par rapport à un algorithme d'estimation, une heuristique d'ordonnancement garantit la faisaibilité de la solution trouvée. Toutefois, l'augmentation de complexité qui en résulte est importante (typiquement $O(N^2)$ [4]) et l'emploi d'une technique de ce type pour l'évaluation d'un grand nombre de solutions d'implantation (exploration de l'espace de recherche) peut rapidement s'avérer impossible. Une évaluation de la complexité des techniques d'ordonnancement pour les graphe flot de contrôle a été proposée dans [33]. Le tableau 2.1 présente

Référence	Complexité	Commentaire
Rabaey & Potkonjak [6]	$\leq O(N^2)$	Bornes supérieure et inférieure à partir d'un DFG
Sharma & Jain [5]	$O(NC^2)$	Borne inférieure à partir d'un DFG
Probabiliste [2]	$O(N)$	Caractérisation du coût en fonction d'une contrainte de temps variable
Formulation ILP Rim & Jain [7]	$O(N+cC)$	Borne inférieure du nombre de pas de contrôle
Narayan & Gajski	–	Prise en compte du contrôle et de la de la hiérarchie (Specharts)
List Scheduling [8]	$O(NC)$	Technique d'ordonnancement d'un DFG
Force Directed Scheduling [9]	$O(N^2C)$	Technique d'ordonnancement d'un DFG
Path Based Scheduling [33]	Exponentielle	Technique d'ordonnancement prenant en compte le contrôle

TAB. 2.1 – Différentes techniques pour l'estimation au niveau comportemental

plusieurs techniques d'estimation et d'ordonnancement, ainsi que leurs complexités respectives. N représente le nombre de noeuds du graphe et C le nombre de pas de contrôle.

Dans les applications embarquées, une grande partie de la consommation d'énergie est due au stockage et au transfert des données. En outre, les mémoires ont souvent un coût important en consommation et en surface. Certains travaux se sont donc orientés vers une méthodologie de conception qui s'appuie d'abord sur l'optimisation de l'architecture mémoire, plutôt que

sur celle de l'unité de traitement (travaux effectués à l'IMEC [23]). Cette approche s'applique particulièrement aux applications à forte dominance pour le traitement de données comme dans le domaine du multimédia ou de la télécommunication. Les premiers travaux relatifs à l'estimation mémoire se situent au niveau scalaire, car les applications ciblées (dans le cadre de la synthèse de haut niveau) ne contiennent qu'un nombre limité de signaux. Les variables scalaires décrites dans une spécification au niveau RTL peuvent être assignées à un nombre de registre minimum au moyen de techniques telles que l'algorithme "left-edge" [10], la formulation ILP [24], la coloration de graphe [25] ou le clique partitionning [26].

Ces techniques sont difficilement utilisables dans les applications qui traitent d'un grand nombre de signaux multi-dimensionnels. Les données struc-

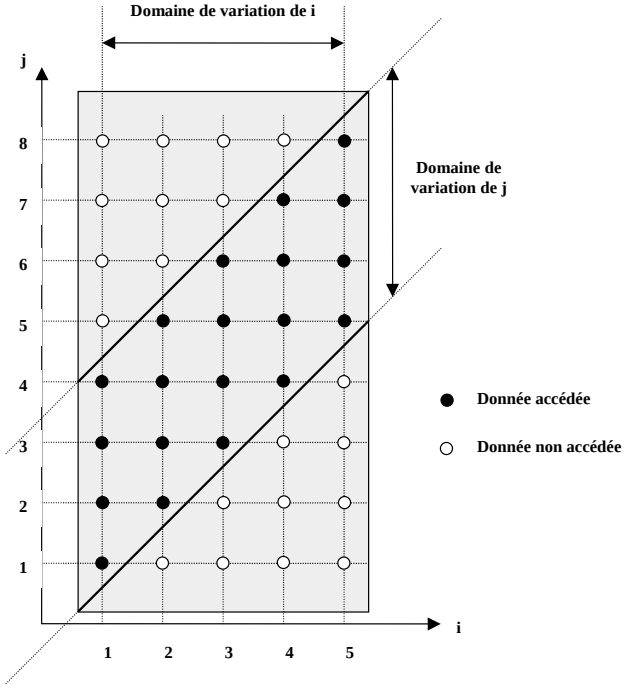


FIG. 2.6 – Modélisation mémoire sous forme de polyèdre

turées telles que les vecteurs et les matrices sont très utilisées et les traitements effectués sur ces types de données ont conduit au développement de nombreuses techniques. Lorsque l'on manipule des structures de données de type tableau, les traitements sont généralement décrits par des boucles imbri-

quées dont les indices sont bornés par deux fonctions affines. Une approche répandue pour l'estimation mémoire d'applications qui traitent de signaux multi-dimensionnels repose sur la modélisation sous forme de polyèdre et a été introduite dans [27]. Sur la figure 2.6 sont représentés les points du tableau qui sont accédés dans l'exemple suivant :

$$\begin{aligned} &do\, i := 1...5 \\ &\quad do\, j := i...i+3 \\ &\quad\quad f(x(i,j)); \end{aligned}$$

La mémoire nécessaire pour mémoriser le tableau x est la "boîte rectangulaire" contenant l'ensemble des éléments de x. Dans notre exemple, elle est de taille $5 * 8$, c'est à dire qu'il faut une mémoire de 40 points, alors que le nombre d'emplacements réellement utilisés est égale à 20. Cette valeur définit le volume du polyèdre. Le taux d'utilisation de la mémoire est de 50% dans ce cas. De nombreux travaux se basent sur cette représentation pour développer d'autres techniques telles que les transformations (pour trouver un compromis entre la taille mémoire et la complexité de la fonction d'adressage par exemple), ou pour étudier le problème d'allocation mémoire, assignation signal mémoire, ordonnancement des opérations, ... [23].

2.3.3 État de l'art des estimations spécifiques aux FPGAs

Les facteurs essentiels de performances pour les FPGAs sont la qualité des outils de CAO, la qualité du composant et les qualités électriques de l'architecture. La majorité des travaux relatifs aux FPGAS portent sur l'optimisation des architectures et des outils de placement / routage [13]. La prise en compte de l'application pour l'estimation du coût relève du problème de l'utilisateur et est un problème peu étudié à notre connaissance. Aujourd'hui, la complexité des systèmes électroniques est telle qu'elle nécessite l'utilisation d'architectures hétérogènes, composées d'unités de nature différente, ce qui rend la conception de ces systèmes difficile. Il est donc important de pouvoir proposer très tôt à l'utilisateur un certain nombre de métriques de performances (typiquement temps, surface, consommation) mais cette fois en tenant compte de l'application que l'on doit y intégrer. Il existe peu de

méthodes à notre connaissance qui traitent du problème d'estimation sous cet angle.

Xu et Kurdahi proposent une technique visant l'estimation en temps et surface des FPGAs de type SRAM, et proposent une application à la famille XC4000 de Xilinx [14][15][16]. La technique repose sur un modèle du processus de mapping et prend en compte les effets dus au routage. Elle consiste à prédire l'utilisation des LUTs, la construction des CLBs ainsi que le placement et la forme du plan de masse. À partir d'une netlist d'entrée au niveau logique, un regroupement des portes qui tient compte de la structure interne des LUTs est effectué (cette approche se situe après une étape de synthèse de haut niveau). Il résulte de cette étape une netlist au niveau LUT. Ces LUTs sont ensuite regroupées dans les CLBs en fonction des configurations possibles (F, G, H). La netlist de CLBs obtenue est ensuite analysée de manière à prédire la topologie du plan de masse. Le coût des interconnexions est pris en considération en ajustant la topologie de telle sorte qu'elle tienne compte des ressources de routage du composant. Ensuite, un modèle temporel permet de calculer les performances en analysant les délais dus aux CLBs, aux lignes d'interconnexions et aux points d'interconnexions. Cette méthode présente un compromis précision complexité intéressant (6.1% d'erreur moyenne pour une complexité de l'ordre de O(n)), mais elle est aussi très dépendante du composant cible, la famille XC4000 de Xilinx.

Une autre approche développée à l'Institut Fédéral de Technologie Suisse [17] permet l'estimation des paramètres temps et surface à partir d'un haut niveau d'abstraction. Cette technique cible des applications de type régulières où domine le traitement de données. Le principe repose sur la séparation de l'aspect spécification comportementale de l'influence des spécificités architecturales du FPGA. La phase spécification est réalisée sous la forme d'un graphe flot de données ramené à un ensemble d'opérations élémentaires (+, *, mux, opérations logiques et LUT). Une phase de caractérisation algorithmique est alors effectuée, elle consiste à définir les paramètres suivants : nombre d'entrées / sorties, nombre d'opérations élémentaires, nombre de registres (hors registres pipeline), facteur de parallélisme, et nombre d'itérations. Le composant lui, est caractérisé à l'aide d'un modèle spécifiant les caractéristiques de chaque type d'opération élémentaire. Une combinaison linéaire des paramètres de caractérisation algorithmique et des paramètres de caractérisation

du composant permet ensuite de calculer les performances du système. Les auteurs proposent en outre une exploration de l'espace de recherche basée sur l'utilisation du pipeline permettant ainsi de diminuer la longueur du chemin critique. Les applications révèlent une bonne précision de la méthode mais aussi quelques limites de l'aveu même des auteurs : les effets du routage introduisent une erreur importante et le contrôle n'est pas pris en compte. Ceci limite l'utilisation de l'estimateur au cas d'applications régulières constituées essentiellement d'interconnexions courtes (les deux applications test proposées dans [17]sont le filtre FIR et le calcul de ressemblance de blocs 16x16 pour l'estimation de mouvement). Cette étude est menée sur un composant de la famille XC4000E de chez Xilinx.

Il existe une autre approche [18] [19] (Miller, Owyang, Kliman) basée sur une bibliothèque de benchmarks. Ceux-ci constituent une sorte de base de données de circuits dont les caractéristiques de surface et de performances ont été mesurées sur une grande variété de FPGAs. Lorsque l'on souhaite évaluer une certaine application, celle-ci est partitionnée en sous-parties dont chacune est remplacée par le circuit benchmark le plus ressemblant. Connaissant les caractéristiques de ces circuits, il est alors possible d'en déduire des valeurs d'estimation pour l'ensemble de l'application. Cette approche est fortement limitée par le fait qu'elle dépend énormément de cette base de données qui doit être régulièrement mise à jour en fonction des évolutions à la fois des composants et des applications.

Ces trois méthodes sont les seules à notre connaissance qui s'intéressent à l'estimation d'une application sur une technologie FPGA. Les principales limitations qui en ressortent concernent les composants ciblés (XC4000 pour les deux premières) et les spécifications d'entrée, inconvénients qui sont en grande partie liés au domaine d'application de ces estimateurs. En ce qui nous concerne, il faut pouvoir traiter le cas des spécifications complexes (incluant les structures de contrôle) et l'approche doit aussi s'appliquer à plusieurs types de composants. C'est la raison pour laquelle cette étude a été validée sur deux composants représentatifs des technologies modernes employées par les deux plus importants constructeurs : Xilinx et Altera. Nous proposons maintenant une présentation des architectures de ces deux composants illustrant deux types d'architecture : la structure ilôt de calcul (Virtex) et la structure hiérarchique (Apex), pour reprendre la terminologie employée dans

[20].

2.4 Composants cibles : les FPGAs

Les circuits logiques programmables les plus anciens sont les PAL (*Programmable Array Logic*) et sont apparus sur le marché au début des années 70. La première génération de ces composants permettait uniquement la réalisation de fonctions combinatoires. Puis sont apparus les PALs à registres permettant la réalisation de systèmes synchrones. Ce type de composant de faible densité est aujourd'hui communément désigné sous l'appellation PLD (*Programmable Logic Device*). Face à l'augmentation croissante de la complexité des systèmes électroniques, les composants ont évolué vers l'intégration de plusieurs architectures de type PAL au sein d'un même boîtier. Ces nouveaux composants ont donc naturellement été baptisés CPLD (*Complex PLD*). Leur architecture peut être vue comme une multipication de modules type PAL associés à une matrice d'interconnexion programmable.

Parallèlement à l'apparition des PALs, l'augmentation de la complexité des systèmes a eu pour effet la généralisation de l'usage des composants à application spécifiques, à savoir les ASICs (*Application Specific Integrated Circuits*). Ces composants apportent une réponse satisfaisante aux problèmes d'intégration mais leur forte densité en limite toutefois l'application à des domaines de grande diffusion ou exigeant une grande confidentialité.

Apparus au milieu des années 80, les FPGAs correspondent à ce que l'on pourrait considérer comme la voie intermédiare entre l'approche générique (PAL, CPLD) et spécifique (ASIC). Ils autorisent à la fois de grandes capacités d'intégration et la définition de fonctions par l'utilisateur.

Les FPGAs ont connu d'importantes évolutions architecturales au cours de ces dernières années. Ces évolutions résultent essentiellement de l'augmentation des capacités d'intégration qui permettent aux différents contructeurs de proposer des composants à la fois gros et performants. Les caractéristiques essentielles de ces nouvelles familles sont d'intégrer dans leurs architectures des modules visant à augmenter les performances comme par exemple de la mémoire, des entrées / sorties rapides, asservissement d'horloge ... Une autre innovation importante dans le domaine des composants programmables repose sur la reconfiguration partielle ou totale du composant de manière dyna-

mique. Cet aspect ne sera pas traité dans nos travaux et nous nous limiterons dans cet état de l'art à l'étude des composants qui constituent le fer de lance des deux plus importants fournisseurs de composants classiques actuellement, la famille *Virtex* de Xilinx et *Apex* de chez Altera.

2.4.1 La famille Virtex (Xilinx)

L'architecture générale des FPGAs de chez Xilinx est basée sur une matrice carrée de cellules configurables pouvant être connectées entre elles par un réseau d'interconnexions [21]. La liaison vers l'extérieur se fait par des blocs d'entrées / sorties configurables en niveau logique, en impédance et en direction. La première génération XC2000 date de 1985 et Xilinx en est maintenant à sa septième génération de composants FPGA. La tendance des dernières générations est de cibler certains crénaux porteurs du marché, comme la solution "bas coût" ou à l'opposé la solution "haute performance". Ainsi, la panoplie présentée par le fabricant montre sa volonté de couvrir tous les segments du marché.

La dernière génération de composants baptisée Virtex vise les très fortes capacités (4 millions de portes) et les hautes vitesses ($>$ 100 MHz). La technologie employée est une technologie CMOS 0.18μm avec 6 niveaux de métallisation (2001). Dérivée de la famille des XC4000, les évolutions concernent le routage, les entrées / sorties et l'intégration des mémoires (figure 2.7). L'architecture du composant est basée sur une matrice carrée de "blocs logiques" dénommés CLBs (Configurable Logic Blocks) utilisés pour l'intégration de fonctions logiques combinatoires et synchrones. Chacun de ces blocs est constitué de quatre sous éléments regroupés par paires appelés *slices*. Ils comportent deux LUTs à 4 entrées permettant chacune la réalisation de fonctions logiques à 1, 2, 3 ou 4 entrées, de mémoires ou de registres à décalages. Ils disposent de plus d'un registre par générateur de fonction permettant ainsi la synchronisation de la sortie. Il est donc possible de réaliser un certain nombre de fonctions par slice : 2 fonctions logiques jusqu'à quatre entrées, 2 additionneurs 1 bit, 1 additionneur 2 bits, 2 registres à décalage 16 bits.

Chaque générateur de fonction peut être configuré en bloc mémoire RAM / ROM synchrone 16x1-bit, dite mémoire distribuée. Il est alors possible d'utiliser un CLB pour réaliser des mémoires simple port (16x8-bit, 32x4-bit,

64x2-bit, 128x1-bit) ou double port (16x4-bit, 32x2-bit, 64x1-bit). Les mémoires distribuées conviennent bien pour l'utilisation de petites mémoires. Pour des besoins plus importants, on trouve d'autres ressources de mémorisation, les blocs RAM, disposés en périphérie de la matrice de CLB. Ce sont des modules de mémoire RAM synchrone double port de 4096 bits configurables en mots de 1, 2, 4, 8 ou 16 bits. Chaque bloc est composé de deux

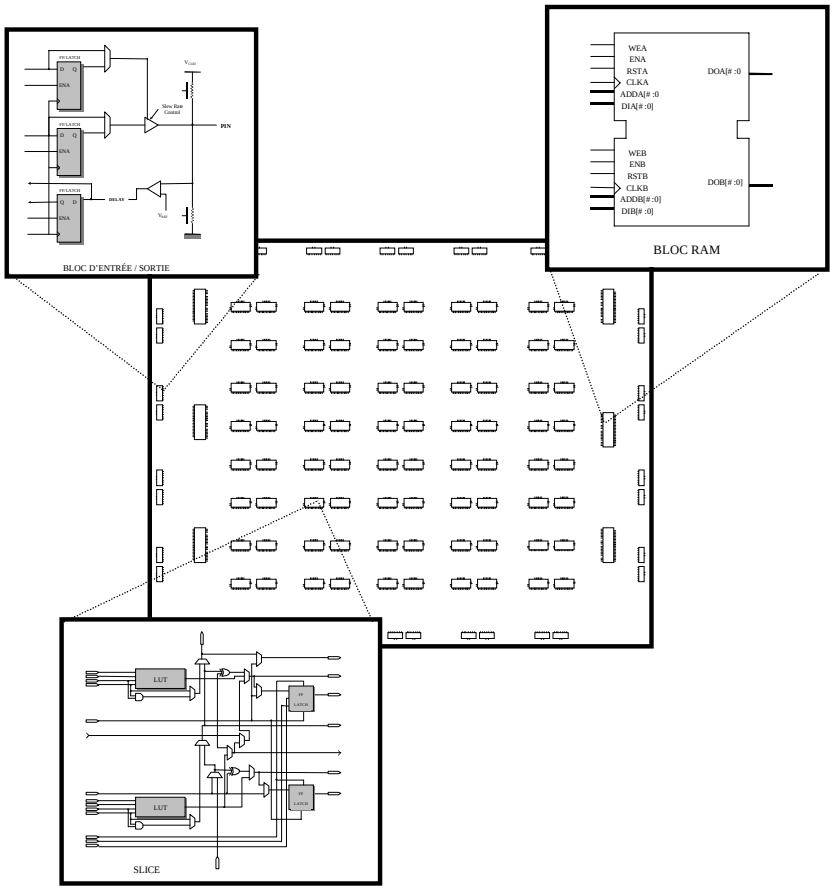


FIG. 2.7 – Architecture Virtex

ports dont les signaux de contrôle sont indépendants, ce qui permet une plus grande souplesse d'utilisation de ces mémoires.

Les blocs d'entrées / sorties, qui permettent de faire le lien entre les ressources logiques du circuit et l'extérieur du composant, sont regroupés à la périphérie de la puce. Ils disposent de 3 registres : un pour l'entrée, un pour la sortie et un pour le contrôle. Ces registres partagent la même horloge et

disposent d'une entrée de validation (*clock enable*) et de la possibilité de mettre des résistances de *pull-up*, *pull-down* ainsi qu'un programmateur de retard sur l'entrée. D'autre part, les plots d'entrées / sorties ont été conçus pour s'adapter à la plupart des standards d'entrées / sorties numériques en vigueur (LVTTL, PCI, GTL+, HSTL, AGP, ...). Pour cela, ils sont regroupés dans 8 bancs, chacun d'eux disposant de sa propre alimentation comprise entre 1.5 et 3.3 V, le coeur du composant étant alimenté en 1.8 V.

En ce qui concerne les ressources de routage, il est nécessaire de le hiérarchiser compte tenu de la taille des matrices utilisées. On trouve donc des ressources pour le routage local de chaque CLB avec ses voisins, ainsi que des connexions à des matrices de routage qui sont des canaux de longueur et de portées différentes. Il existe en outre des ressources de routage particulières entre les CLBs constituées de lignes 3-états segmentées. D'autre part, on trouve en périphérie du coeur de la puce un réseau de routage permettant d'augmenter la souplesse de placement / routage des entrées / sorties.

2.4.2 La famille Apex (Altera)

Fin 92, Altera, qui produit des EPLD et des MAX, lance sur le marché une famille de FPGA, FLEX 8000 (*Flexible Logic Element matriX*), dans le but de concurrencer les LCA de Xilinx et la famille AT6000 d'Atmel. Ce lancement coïncide avec l'arrivée sur le marché des EPLD de Xilinx. En 1995, Altera annonce le lancement de la famille FLEX10K qui constitue leur deuxième génération de FPGA. Cette famille vise le créneau de la forte densité d'intégration avec des capacités allant jusqu'à 250000 portes utilisables. En 1997, Altera lance sa troisième génération, la famille FLEX6K, qui vise le créneau du composant bas coût de capacité moyenne (jusqu'à 24000 portes utilisables). Quatrième génération de FPGA Altera, la famille Apex 20K lancée en 1999 apporte des évolutions architecturales permettant de réaliser des systèmes comportant plus d'un million de portes utilisables [22]. On retrouve les concepts utilisés dans les familles précédentes de FPGA du même constructeur puisque l'on trouve un réseau d'interconnexions X-Y (rows & columns) permettant de relier des LABs (ici appelés MegaLAB) aux cellules d'entrées/sorties (figure 2.8). En fait, l'évolution majeure réside dans la constitution des MegaLABs qui, du fait que l'on vise les très fortes

capacités, intègrent beaucoup plus de cellules.

Un *MegaLAB* est un groupe de 16 à 24 *LABs* (Logic Array Blocks) et d'un module *ESB* (Embedded System Block), qui est une évolution des EAB rencontrés dans le famille FLEX 10K. Le MegaLAB dispose d'un réseau

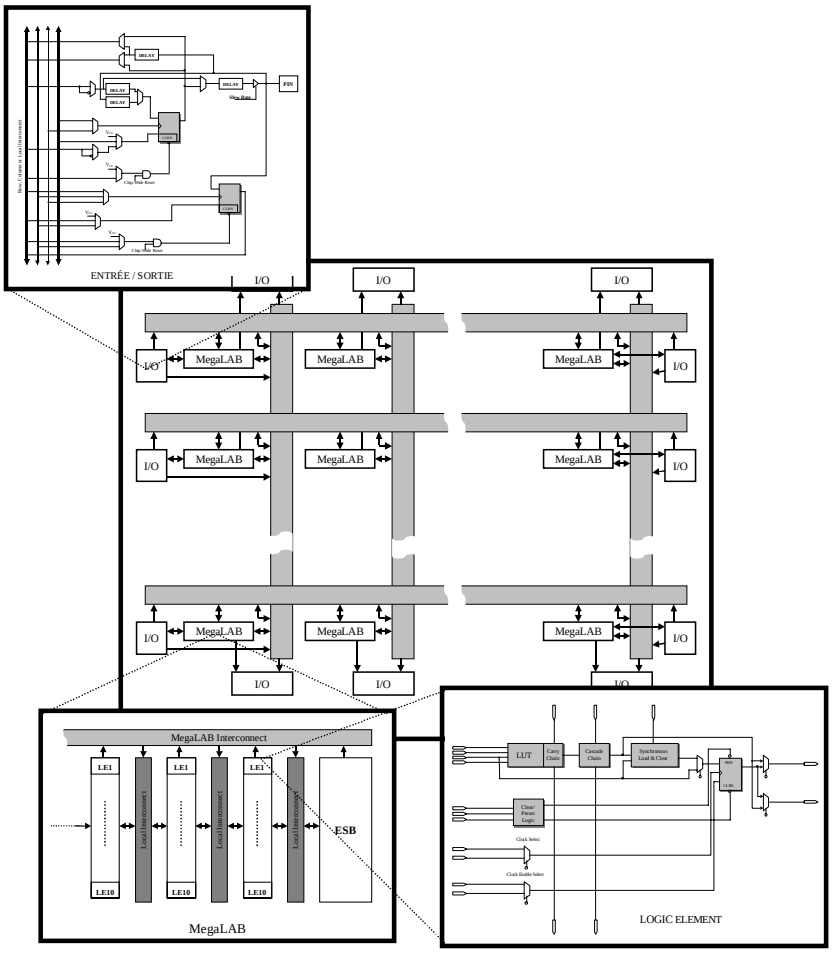


FIG. 2.8 – Architecture Apex

de routage interne qui permet de faire l'interface entre les réseaux locaux des LABs et le routage global X-Y, ce qui porte à 3 le nombre de couches d'interconnexions du circuit. Chaque LAB est composé de 10 cellules de base (*LE*, Logic Elements) et est connecté aux deux réseaux d'interconnexion locaux voisins.

L'élément logique est la plus petite cellule du composant. Il comprend un générateur de fonction mémoire capable de réaliser n'importe quelle fonction

logique à 4 entrées, et un élément séquentiel qui est une bascule pouvant être configurée en D, T, JK ou RS. Les cellules peuvent être mises en cascade pour réaliser des fonctions combinatoires à entrées multiples en utilisant un minimum de ressources et disposent de ressources facilitant la propagation de retenue arithmétique.

D'autre part, la famille Apex dispose de ressources spécifiques, les *ESBs* (Embedded System Blocks) qui ont pour rôle de remplir 2 types de mission : mémoire et synthèse CPLD. Un ESB est constitué d'un bloc de 2048 bits de mémoire RAM (mais aussi ROM et CAM) utilisable de manière synchrone et asynchrone, en simple ou double port. Il est alors possible de réaliser avec 1 ESB les mémoires double port suivantes : 128x16-bit, 256x8-bit, 512x4-bit, 1024x2-bit, 2048x1-bit. De plus, les ESBs offrent la possibilité de mise en cascade de façon à réaliser des mémoires plus grosses. Un autre mode de configuration de l'ESB consiste en 16 macrocellules de type EPLD, permettant l'intégration efficace de décodeurs ou de machines d'états.

Pour réaliser les entrées / sorties, Altera propose des cellules complexes permettant de couvrir tous le besoins des utilisateurs tant au niveau fonctionnel (contrôle de la sortie direct ou séquentiel, retard programmable, ...) qu'au niveau de l'interface électrique (LVTTL, LVCMOS, SSTTL-2, SSTTL-3, AGP, CTT, ...). Ceci étant rendu nécessaire par les vitesses atteintes par les systèmes numériques considérés par ce composant.

2.4.3 Bilan des architectures

Ces deux composants ont été choisis pour notre étude car ils illustrent deux types d'architectures différentes : le modèle ilôt de calcul (Virtex) et le modèle hiérarchique (Apex). Le tableau 2.2 résume les caractéristiques de ces deux familles. Ces deux architectures possèdent une structure de base très différente (matrice de cellules contre organisation hiérarchique). Toutefois, un certain nombre d'éléments sont communs aux deux familles et peuvent être utilisés pour une modélisation qui soit générique.

Les éléments communs que l'on retrouve quelque soient les familles sont les cellules logiques qui permettent la programmation des fonctions définies par l'utilisateur (*Logic Elements* pour Apex et *slices*pour Virtex) et les blocs d'entrées / sorties pour la communication avec le monde extérieur. Les com-

posants peuvent en outre disposer de ressources dédiées (ESB pour Apex et BRAM pour Virtex) permettant l'intégration efficace de certaines fonctionalités, en particulier les mémoires. L'estimation doit prendre en compte

	Apex		*Virtex*	
	$EP20K100$	$EP20K200$	$XCV400$	$XCV1000$
Nombre de portes max	263000	526000	468252	27648
Cellules logiques	4160 lgc elt	8320 lgc elt	4800 slices	12288 slices
Cellules dédiées	26 ESB	52 ESB	40 BRAM	64 BRAM
max RAM bits	53248	106496	163840	262144
Nb max d'entrées / sorties	252	382	404	512

TAB. 2.2 – Caractéristiques Apex et Virtex

ces ressources afin de pouvoir s'appliquer à tous les types de FPGAs. La mesure de l'occupation pour chaque type de ressource (cellules logiques, ressources dédiées, entrées / sorties) permet de donner une appréciation très satisfaisante de la faisabilité de l'intégration.

2.5 Conclusion

La complexité du cycle de conception, en particulier celui du codesign, nécessite le développement de nouvelles méthodologies afin de mieux exploiter les avancées technologiques et de réduire les temps de conception. La solution passe par l'augmentation du niveau d'abstraction où les compromis et fonctionalités du système sont plus faciles à discerner et le potentiel d'optimisation plus important. L'utilisation d'estimateurs permet l'évaluation rapide de différents choix et fournit au concepteur les informations nécessaires pour le guider dans l'espace de conception. La méthodologie de conception développée au LESTER repose sur l'utilisation d'estimateurs, d'abord au niveau système puis au niveau architectural, dont l'objectif est de fournir une estimation des caractéristiques du système. Le travail présenté ici consiste à caractériser le coût de l'implantation des fonctions candidates à une intégration matérielle sur des composants FPGAs. Cette caractérisation passe par l'estimation du coût (taux d'occupation du composant) et des performances. Afin d'obtenir des valeurs d'estimation réalistes, la méthode doit prendre en compte les différentes unités de l'architecture et permettre l'exploration de plusieurs solutions architecturales.

La littérature sur le sujet des estimations est importante et les modèles d'estimation proposés sont ciblés, en général pour une technologie (ASIC, FPGA, ...), un type d'unité (traitement, mémoire, contrôle, communications) ou un type de contrainte (temps, surface, consommation) et à plusieurs niveaux d'abstraction (système jusqu'au plan de masse). Afin de définir la méthode d'estimation architecturale présentée dans ce mémoire, nous nous sommes appuyés sur les techniques d'estimation existantes. L'approche que nous avons développé vise l'exploration et l'estimation des performances et du coût au niveau comportemental, sur plusieurs types de FPGAs. Le chapitre suivant présente le flot d'estimation architecturale que nous avons défini afin d'y parvenir.

Chapitre 3

L'Exploration Architecturale

Ce chapitre présente une technique d'exploration architecturale à partir d'une spécification comportementale. Cette exploration a deux objectifs : il s'agit tout d'abord d'évaluer un certain nombre de solutions architecturales caractérisées en terme de performances et d'occupation sur le composant reconfigurable candidat, puis de fournir au concepteur toutes les informations nécessaires à la conception de la solution retenue. Pour parvenir à ces objectifs, le problème est décomposé en deux étapes qui sont la définition des solutions puis l'estimation de l'occupation et des performances. Nous proposons ici une présentation générale du flot et des différentes hypothèses de travail (modèle de spécification, modèles architecturaux, courbes d'exploration). Nous présentons le détail des deux étapes du flot dans les deux chapitres suivants.

Le plan de ce chapitre est le suivant. Dans un premier temps, nous définissons les étapes du flot. Puis nous introduisons les bases du modèle de représentation d'entrée nécessaires pour la compréhension de la méthode. Nous abordons ensuite le problème de la caractérisation des solutions architecturales à chaque stade de l'exploration, et enfin, nous détaillons les modèles architecturaux sur lesquels s'appuie l'exploration architecturale.

3.1 Approche de la méthode

3.1.1 Introduction

La conception d'un système électronique nécessite la description de celui-ci dans trois domaines qui sont la spécification comportementale, suivie d'une traduction en description structurelle, puis physique. La figure 3.1 illustre ces trois domaines. Dans le domaine comportemental, on s'intéresse à ce que

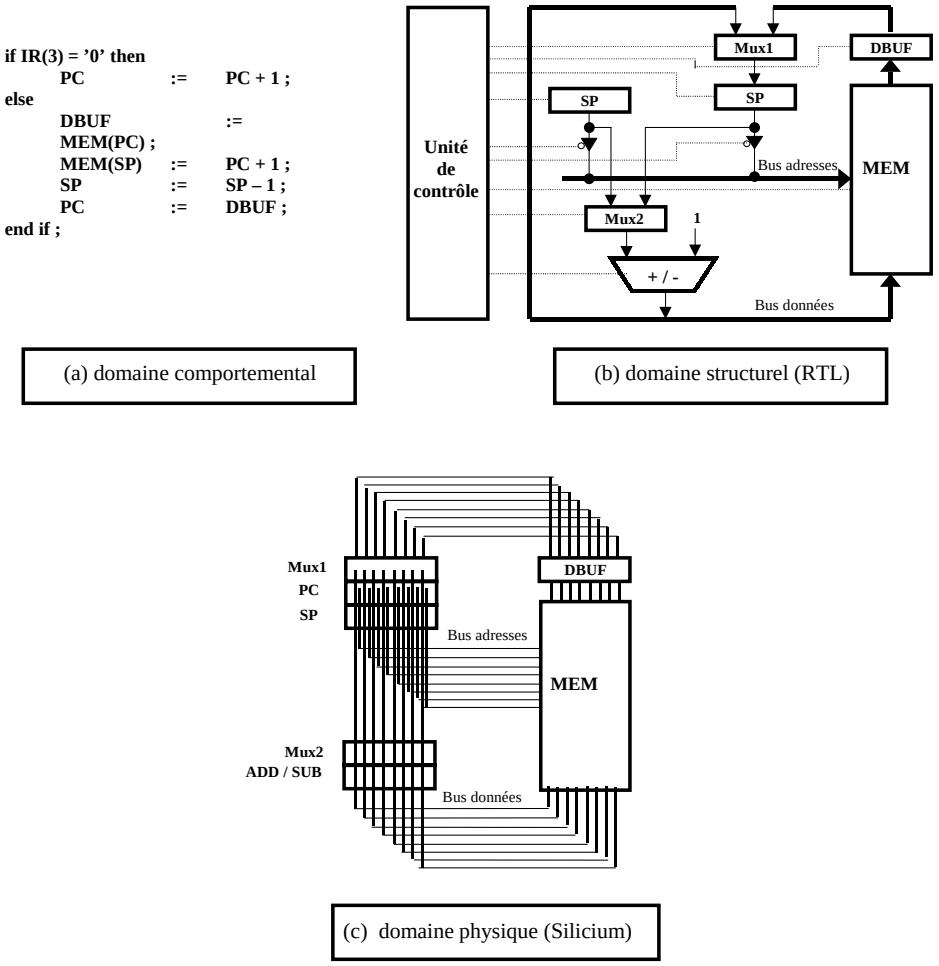


FIG. 3.1 – Trois domaines de description : (a) comportemental, (b) structurel, (c) physique

le système fait, et non à la manière dont on va le construire. À ce niveau, on utilise les variables et les opérateurs du langage de spécification pour

décrire les fonctionalités de l'application. Variables et structures de données ne sont pas encore associées aux mémoires et aux registres de l'architecture, et les opérations ne sont pas reliées aux unités fonctionnelles ni aux états de contrôle. À ce niveau de description, la notion de temps se limite à l'ordre d'exécution des affectations des variables.

La représentation structurelle fait le lien entre le niveau comportemental et le niveau physique. Elle consiste en un ensemble de composants et de connexions et est définie sous des contraintes telles que le coût, la surface, les performances. UALs, multiplieurs, registres, RAMs et ROMs en constituent les composants de base, au niveau RTL. Le domaine physique traduit quand à lui la structure du système sur du silicium.

L'exploration architecturale doit permettre l'évaluation de plusieurs solutions d'implantation à partir d'une description comportementale et sur une technologie FPGA donnée. Le recours aux estimations a été choisi pour permettre l'exploration d'un vaste espace de solutions. Comme pour la conception, l'estimation d'un système décrit au niveau comportemental passe par une étape d'analyse au niveau structurel. Une fois définie l'architecture et ses différentes unités, on peut alors effectuer l'estimation des caractéristiques temps / surface pour un composant d'implantation.

Dans notre cas, la description au niveau structurel peut se limiter à un dénombrement des ressources (unités fonctionnelles, mémoires) et non à une vision complète où les ressources sont interconnectées de façon précise (ce qui serait le cas en utilisant un outil de synthèse architectural). La traduction du niveau structurel vers le niveau physique peut alors être effectuée à partir de la connaissance des caractéristiques temps / surface de ces ressources sur le composant candidat à l'implantation. La réduction de complexité obtenue par cette approche (dénombrement + projection) permet alors d'explorer un plus grand nombre de solutions architecturales.

3.1.2 Le flot d'estimation

L'exploration architecturale se déroule donc en deux principales étapes qui sont l'estimation au niveau structurel et l'estimation au niveau physique (figure 3.2).

Une première phase de pré-estimation permet de vérifier le bon dimen-

sionnement du composant candidat à l'implantation. Elle se base d'une part sur le nombre de broches d'entrées / sorties qui est comparé au nombre de données d'entrées / sorties de la spécification, et d'autre part sur le nombre de ressources de mémorisation dont dispose le FPGA. Le processus d'explo-

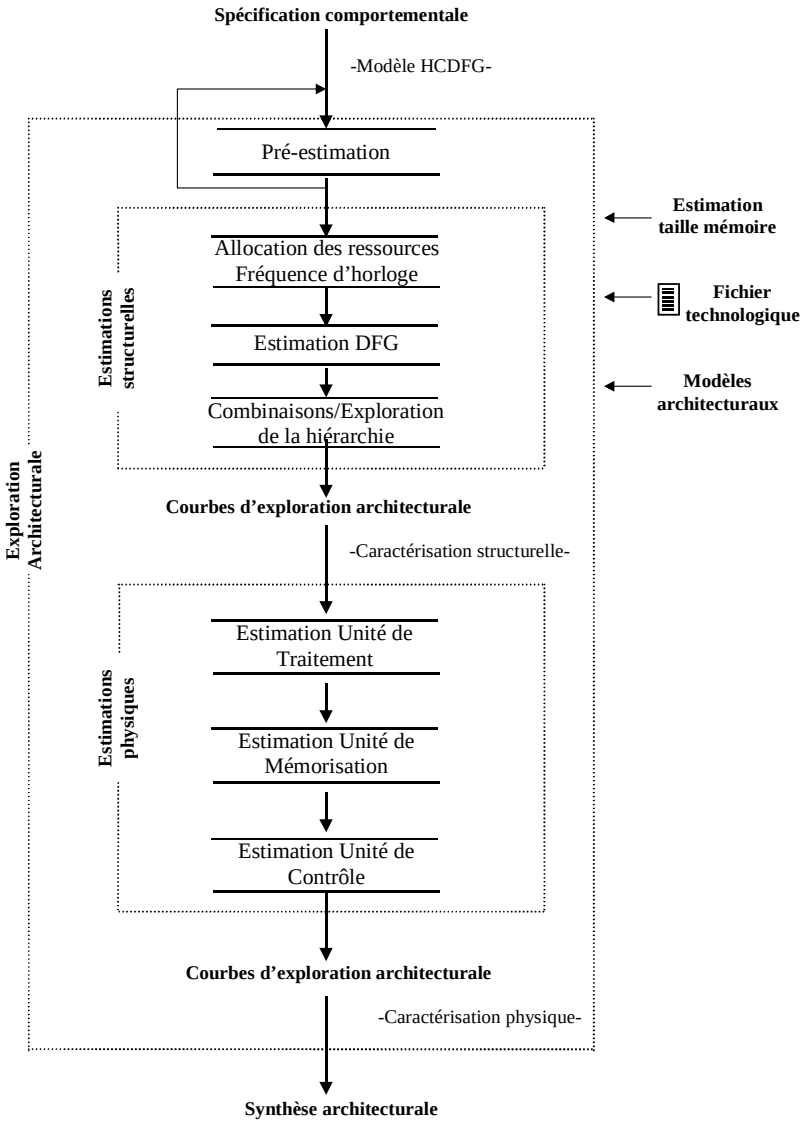


FIG. 3.2 – Flot d'Exploration Architecturale

ration / estimation ne débute qu'à l'issu de cette étape.

Les estimations structurelles ont ensuite pour but de fournir plusieurs solutions architecturales pour la spécification en cours d'analyse. Elles se

basent tout d'abord sur une étape d'allocation de ressources et de calcul de la fréquence d'horloge, puis un ordonnancement partiel est réalisé. Cet ordonnancement ne concerne que les blocs de base de la spécification (les *DFGs*) et est effectué pour plusieurs contraintes de temps, permettant ainsi la définition de plusieurs solutions. L'estimation globale du système est obtenue par l'application de combinaisons qui dépendent du type de contrôle (structures conditionnelles / itératives) et d'exécution (séquentielle / concurrente). Par exemple, on obtient l'estimation d'une structure conditionnelle à partir des résultats d'estimation de la condition et des branches. L'estimation de l'exécution séquentielle de deux graphes est obtenue à partir des résultats d'estimation de ces deux graphes. L'exploration hiérarchique permet ainsi de combiner les résultats jusqu'à atteindre le plus haut niveau de hiérarchie, qui correspond à l'application toute entière. À la fin de cette étape, on dispose de courbes de caractérisation temps / coût de toute l'application.

Les estimations au niveau physique traduisent ensuite la courbe temps / coût de caractérisation structurelle (où le temps est exprimé en nombre de cycles) en courbe de caractérisation physique (exprimée en taux d'occupation des ressources du FPGA / unité de temps physique). Une analyse des unités de traitement, contrôle et mémorisation est effectuée pour une caractérisation complète de l'application. Cette étape nécessite la connaissance précise des caractéristiques de la technologie cible. C'est le rôle du fichier technologique qui contient les paramètres physiques tels que les temps de traversée des opérateurs, les temps d'accès aux mémoires, ainsi que leur coût en surface. À l'issu de l'étape des estimations physiques, on dispose des valeurs physiques de compromis temps / surface pour différentes solutions architecturales. Ces valeurs permettent de vérifier la faisabilité (taux d'occupation inférieur à 100%) ou le respect des contraintes (vitesse d'exécution) du système intégré sur le composant candidat.

3.2 La spécification d'entrée : mode d'utilisation

Après avoir présenté le principe de notre approche d'estimation, nous allons maintenant apporter quelques précisions sur le modèle de représentation utilisé pour la spécification du système et présenté au paragraphe 2.2.2.

Il existe de nombreuses façons de spécifier une application. Celle-ci passe généralement par l'utilisation d'un langage de haut niveau. Le langage C par exemple reste aujourd'hui très utilisé par les organismes de normalisation des applications de télécommunication et du multimédia. Cependant, d'autres langages tels que VHDL, C++, Java ou certains plus récents comme System C, Superlog sont également des langages très utilisés. Ces langages de spécification permettent de valider le fonctionnement de l'application. Ils sont ensuite traduits en une représentation intermédiaire plus facilement manipulable par les outils d'aide à la conception. Cette représentation est très souvent exprimée sous forme de graphes qui peuvent se décliner sous plusieurs formes. Le modèle SPF correspond à une représentation intermédiaire qui peut être accessible à partir de plusieurs langages. Actuellement, le langage de spécification considéré est le langage C.

Par rapport au modèle décrit dans [29] utilisé par l'outil développé au LESTER, l'estimation architecturale se base uniquement sur la vue *fonction* qui correspond à un modèle du type *HCDFG* (*Hierarchical Control and Data Flow Graph*) et permet la modélisation des différentes fonctions de l'application. L'intérêt d'utiliser une forme hiérarchique est de pouvoir manipuler facilement des applications complexes.

La figure 3.3 présente un exemple de graphe modélisé sous la forme *HCDFG* qui illustre les différents types de noeuds existant. Le niveau le plus élevé de la hiérarchie est composé de sept *CDFGs* qui intègrent des dépendances d'exécution séquentielles (HCDFG1 et HCDFG2 par exemple) et parallèles (HCDFG3 et HCDFG4 avec HCDFG6). Le noeud FOR#0 est un noeud de type contrôle décrivant une structure itérative. Comme tous les noeuds contrôle, c'est un noeud hiérarchique dont ici le sous graphe DFG1 est un noeud de type *DFG* et représente le coeur de boucle. Celui-ci est composé de noeuds de type données, qui peuvent être des scalaires ou des tableaux, et de noeuds de type traitement qui représentent les opérations effectuées.

Certains noeuds disposent d'attributs spécifiques comme les noeuds "if" auxquels sont associés des probabilités de branchement permettant le calcul d'un temps moyen d'exécution.

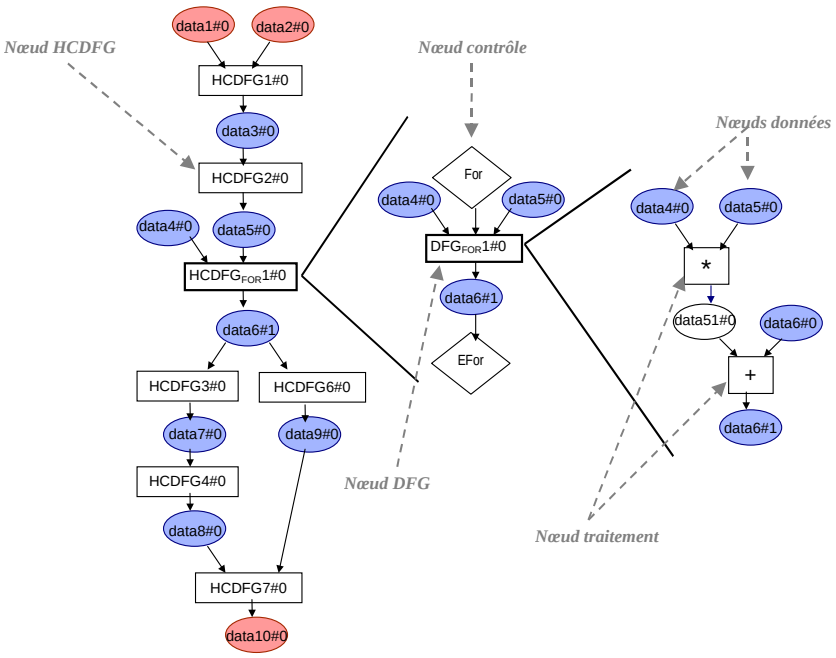


FIG. 3.3 – Modèle de graphe HCDFG

Pour illustrer la modélisation d'une application, un exemple de traduction d'une spécification en langage C vers le modèle de spécification interne *HCDFG* est présenté à la figure 3.4. Cet exemple met en relief les différents niveaux de hiérarchie par rapport à la spécification originale en C. Le graphe est constitué de quatre *HCDFGs* dont deux correspondent à des structures conditionnelles. Les structures de contrôle (IF#0 et IF#1) et les sous graphes (DFG#0 et DFG#1) sont encapsulées dans des noeuds composites à l'intérieur desquels on peut trouver d'autres noeuds composites. Les derniers niveaux de hiérarchie sont constitués par les graphes pour lesquels il n'y a plus de décomposition et correspondent aux blocs de base (ou *DFGs*, sous-graphes grisés sur la figure).

La représentation sous cette forme permet une manipulation beaucoup plus facile par rapport à la spécification utilisant un langage de haut niveau (par exemple pour effectuer une liste des opérations). De plus un grand

nombre de travaux tels que l'estimation ou l'ordonnancement sont directemnent applicables. La hiérarchie du modèle permet en outre la définition de contraintes locales très utiles pour la synthèse du système après estimation, comme nous allons le voir dans le paragraphe suivant.

```
void uppol2 (short AH1, short AH2, short
PH, short PH1, short PH2, short *APH2) {

     short const_128     = 128;
     short const_m128    = -128;
     short const_35512   = 35512;
     short const_12288   = 12288;
     short const_m12288  = -12288;
     short tmp1;
     short tmp2;
     short WD1;
     short WD2;
     short WD3;
     short WD4;
     short WD5;

     tmp1 = AH1 + AH1;
     WD1 = tmp1 + tmp1;
     if ( (PH>>15) == (PH1>>15) )
          WD2 = 0 - WD1;
     else
          WD2 = WD1;
     if ( (PH>>15) == (PH2>>15) )
          WD3 = const_128;
     else
          WD3 = const_m128;
     tmp2 = WD2>>7;
     WD4 = tmp2 + WD3;
     WD5 = AH2 * const_35512;
     *APH2 = WD4 + WD5;
}
```

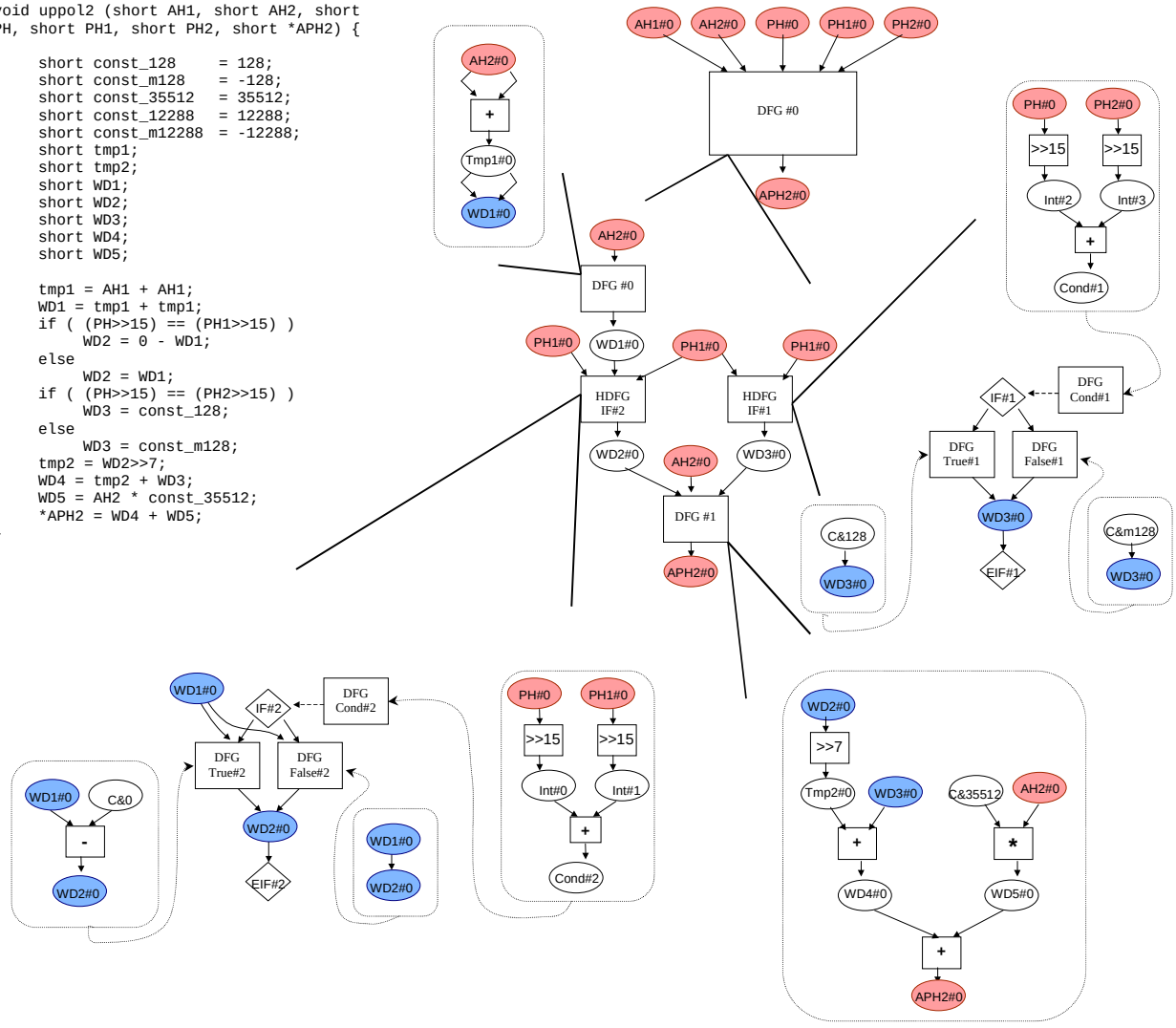


FIG. 3.4 – Exemple de traduction du C vers HCDFG

3.3 Les courbes d'exploration architecturale

3.3.1 Introduction

Pour une application donnée, il existe un grand nombre de solutions d'intégration possibles. Une évaluation réaliste de la faisabilité du système passe par l'évaluation de plusieurs solutions architecturales. La figure 3.5 présente les résultats de l'implantation d'une DCT-2D sur un FPGA de chez Xilinx (XC4000), en nombre de CLBs pour plusieurs contraintes de temps [12]. L'étude d'une telle courbe permet au concepteur de mieux évaluer plusieurs

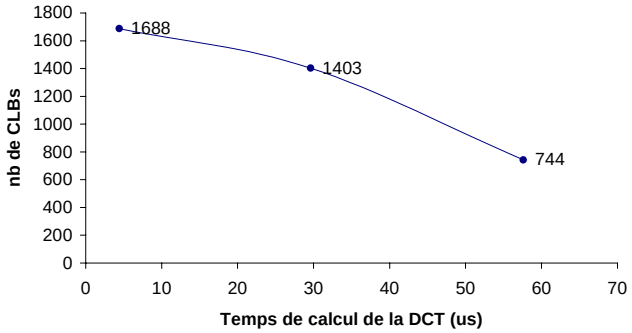


FIG. 3.5 – Surface de la DCT en fonction du temps de calcul

configurations et les temps d'exécutions correspondants. Ainsi, on pourrait par exemple choisir d'implanter la DCT pour un temps d'exécution de $4.4\mu s$ ce qui correspond à une occupation de 1688 CLBs. Ou encore, si on souhaite plutôt minimiser la surface, on pourrait choisir, $T = 57.6\mu s$ ($S = 744$ CLBs). On dispose ainsi d'un ensemble d'informations à partir duquel il est plus facile de converger vers une solution adaptée aux contraintes du concepteur.

L'estimation architecturale que nous avons définie repose sur l'utilisation de courbes similaires dites "courbes d'exploration architecturale" (figure 3.6). À chaque point de l'axe des abcisses correspond une contrainte de temps et une solution architecturale associée. Il existe une courbe de caractérisation structurelle pour laquelle le temps est exprimé en nombre de cycles d'horloge et une courbe de caractérisation physique où l'unité temporelle correspond au temps physique.

3.3.2 Caractérisation structurelle

L'estimation au niveau structurel s'attache à caractériser plusieurs solutions architecturales. La définition des solutions se déroule en deux étapes qui sont l'ordonnancement des blocs de base pour plusieurs contraintes de temps, puis la combinaison des résultats en fonction de la hiérarchie et du contrôle. Par exemple, le nombre de ressources nécessaires pour l'exécution

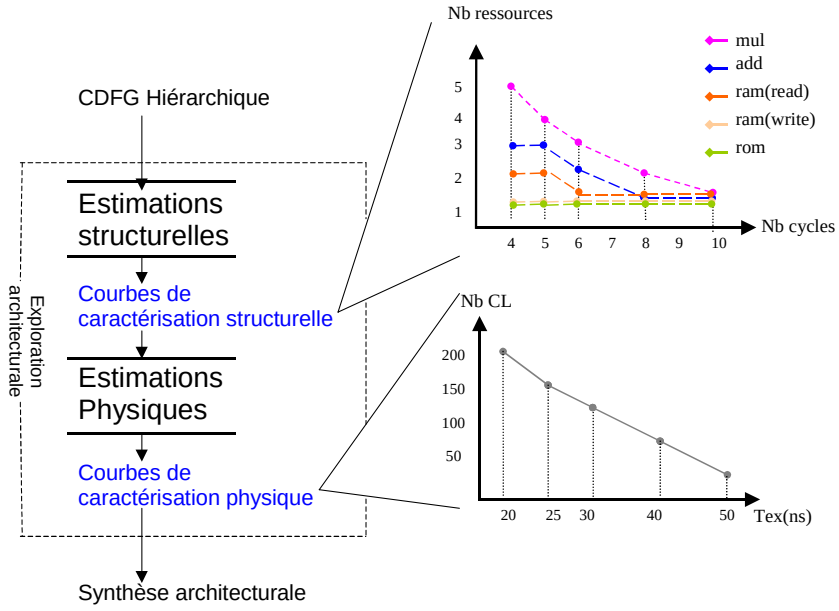
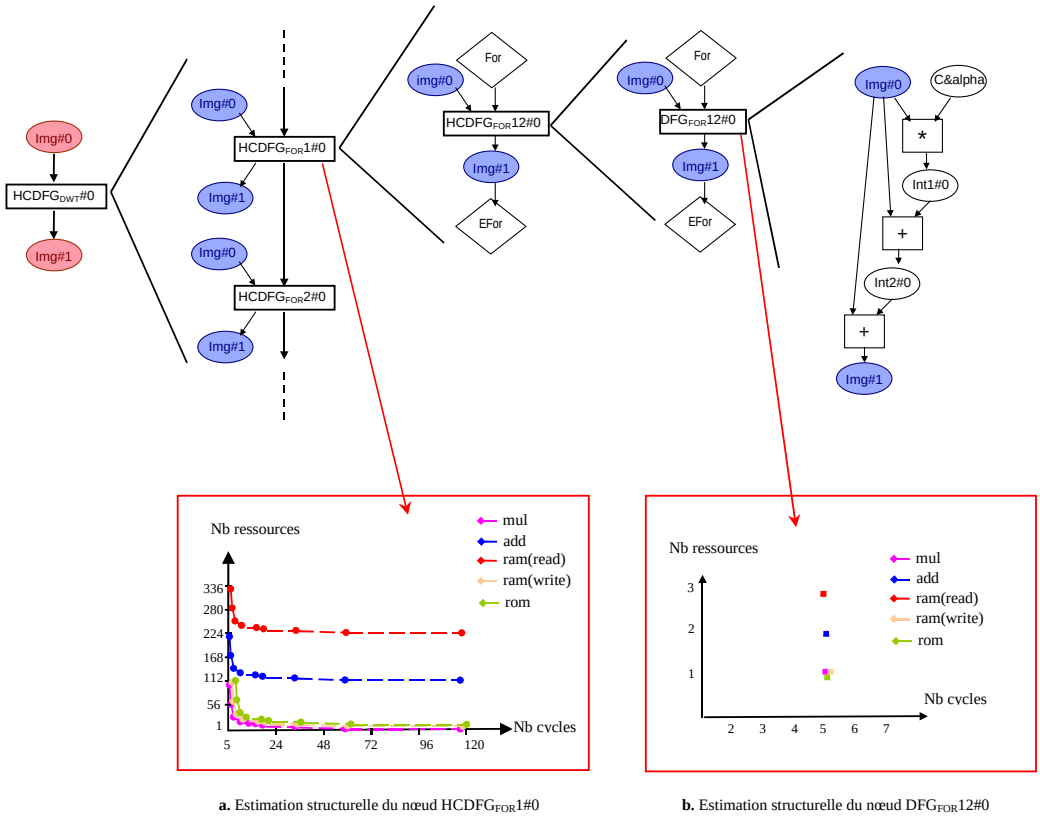


FIG. 3.6 – Courbes d'exploration architecturale

d'une structure itérative et la contrainte de temps correspondante peuvent être estimés à partir de la connaissance des caractéristiques du coeur de boucle (nombre de ressources, contrainte de temps) et du nombre d'itérations. La figure 3.7 illustre ce cas : l'estimation débute par le coeur de boucle $DFG_{FOR}12\#0$ (figure 3.7.b.) par une technique d'ordonnancement (que l'on peut appliquer car il s'agit d'un DFG). Les résultats obtenus permettent alors l'estimation de la boucle $HCDFG_{FOR}12\#0$ qui permettent à leur tour l'estimation de la boucle $HCDFG_{FOR}1\#0$ (figure 3.7.a.). On procède de la même façon pour la boucle $HCDFG_{FOR}2\#0$. L'estimation du noeud $HCDFG_{DWT}$, qui représente le plus haut niveau de la hiérarchie, est ensuite obtenu par une combinaison des résultats d'estimation des noeuds $HCDFG_{FOR}1\#0$ et $HCDFG_{FOR}2\#0$ pour une exécution séquentielle.

Cette façon de procéder implique qu'il existe un résultat d'estimation pour chaque noeud hiérarchique du graphe. Ceux-ci sont mémorisés de façon à fournir un ensemble d'informations utiles pour la synthèse du système après estimation. D'autre part, la caractérisation est définie pour plusieurs contraintes de temps, ce qui permet une prise en compte du parallélisme inhérent au système. À chaque contrainte de temps N_{cycles} correspond



a. Estimation structurelle du nœud HCDFG$_{FOR}$1#0 **b.** Estimation structurelle du nœud DFG$_{FOR}$12#0

FIG. 3.7 – Exemple de caractérisation structurelle (dwt)

un nombre d'unités fonctionnelles de chaque type ($N_{op_k}(N_{cycles})$). Il s'agit des opérateurs de base tels que les opérateurs arithmétiques (additionneurs, multiplieurs, comparateurs), logiques (and, or, not, ...), les registres et les multiplexeurs. Les accès mémoires sont aussi pris en compte. On dénombre ainsi le nombre d'accès simultanés en lecture / écriture à la mémoire RAM ($N_{ram\_lec}(N_{cycles})$ et $N_{ram\_ecr}(N_{cycles})$) et le nombre d'accès simultanés à la ROM ($N_{rom}(N_{cycles})$). On evalue aussi le nombre d'états ($N_{états}(N_{cycles})$) qui permet ensuite l'estimation de la surface de l'unité de contrôle. Une solution

architecturale du système est donc caractérisée au niveau structurel pour la contrainte de temps N_{cycles} par les paramètres suivants :

- $\{N_{op_k}(N_{cycles}),\ k \in [1;l]\}$ où l représente le nombre d'opérateurs de types différents alloués
- $N_{ram\_lec}(N_{cycles})$
- $N_{ram\_ecr}(N_{cycles})$
- $N_{rom}(N_{cycles})$
- $N_{états}(N_{cycles})$

3.3.3 Caractérisation physique

Une fois définies les différentes solutions architecturales du système, l'étape suivante consiste à estimer l'occupation des ressources du composant en fonction d'une valeur physique de contrainte de temps (ns, μs). Les paramètres pris en compte pour la caractérisation physique du système sont :

- le nombre de cellules logiques du FPGA utilisées (*slices*, *logic elements*).
- le nombre de cellules dédiées ($BRAM$ ou ESB).
- le nombre d'entrées / sorties.
- le nombre de buffers trois états le cas échéant.

Contrairement aux estimations structurelles, la caractérisation physique n'est réalisée que pour le plus haut niveau de la hiérarchie, qui correspond à l'application toute entière. Elle est déduite de la caractérisation structurelle du niveau système (toute l'application). On estime les valeurs physiques de temps et de surface occupée pour chaque unité de l'architecture (traitement, mémoire et contrôle).

La surface de l'unité de traitement se base sur une technique de projection. Le principe est le suivant : la contribution en surface d'un opérateur est obtenue en multipliant la surface de l'opérateur par le nombre d'instances de cet opérateur. Par exemple, si les besoins en ressources sont de k additionneurs et l multiplieurs, la surface de l'UT sera $(k * S_{add}) + (l * S_{mul})$ où S_{add} et S_{mul} sont les surfaces respectives d'un additionneur et d'un multiplieur. Pour l'unité de mémorisation, l'allocation et la définition des mémoires est déduite du nombre d'accès simultanés et de l'estimation de la taille mémoire totale nécessaire. Enfin, l'unité de contrôle est estimée à partir de la connaissance du nombre d'états et du nombre total de signaux de contrôle nécessaires pour

le séquencement des unités de traitement et de mémorisation. Cette estimation au niveau physique, appelée aussi projection technologique, prend ainsi en compte chaque unité de l'architecture pour lesquels ont été définis des modèles architecturaux.

3.4 Les modèles architecturaux pour l'estimation

Pour être précise, l'estimation architecturale doit reposer sur un modèle d'exécution précis qui correspond à l'architecture du système estimé. Il existe donc un modèle pour chaque unité (traitement, contrôle, mémoire). Leur définition est basée sur les spécificités architecturales des FPGAs. Ainsi, par exemple, le modèle pour l'unité de contrôle est une ROM associée à un registre d'état ce qui permet une intégration efficace (possibilité d'intégrer la ROM dans les cellules dédiées). Ces modèles interviennent à la fois au niveau structurel et au niveau physique. Nous proposons ici une étude des différents modèles possibles, et les choix effectués pour chaque type d'unité constituant l'architecture.

3.4.1 Unité de traitement

Les modèles d'architectures pour l'unité de traitement ont en commun une structure de base constituée d'un réseau de registres et d'unités fonctionnelles connectées. Les unités fonctionnelles exécutent les traitements de données et les registres sont utilisés pour la mémorisation temporaire des données et également pour la synchronisation des transferts. Les composants d'interconnexion correspondent à des multiplexeurs, des démultiplexeurs et / ou des tristates. On distingue deux grandes classes d'architecture : l'une à base de registres généraux et l'autre à base de bus généraux (figure 3.8). Dans le premier cas, on privilégie l'optimisation du nombre de registres, au détriment du nombre de bus. En effet, ceux-ci sont occupés pendant la durée totale de traversée d'une unité fonctionnelle. Ce modèle introduit une certaine séquentialité dans l'exécution. L'autre modèle permet quant à lui de minimiser les chemins de communication, par contre, le nombre de registres devient rapidement élevé. Il facilite en outre l'exécution d'opérations

en parallèle.

Le choix du modèle d'architecture pour l'unité de traitement est motivé par les observations suivantes : tout d'abord, les cellules logiques des FPGAs disposent d'éléments séquentiels (registres) de sorte que l'utilisation d'un registre en sortie d'une unité fonctionnelle n'entraîne aucun surcoût en surface. L'optimisation du nombre de registres n'est donc pas un problème critique

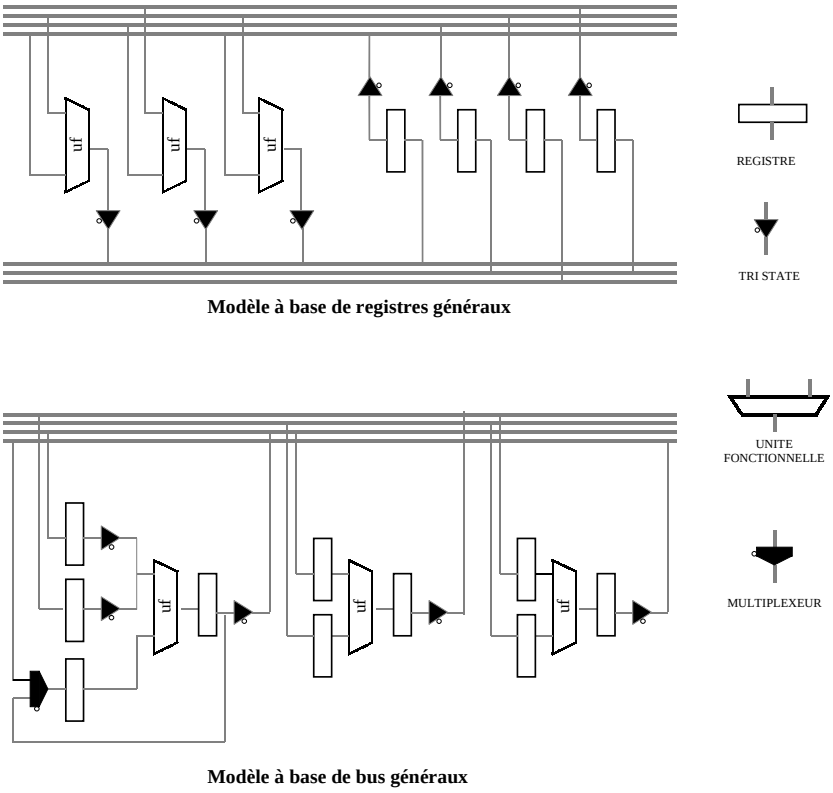


FIG. 3.8 – Modèles architecturaux pour l'UT

dans ce cas. De plus, il nous parait important de privilégier un modèle qui minimise le nombre de bus. En effet, la complexité du réseau de routage des FPGAs ne permet pas de garantir l'homogénéité du routage d'un bus. Les délais de propagation peuvent s'en ressentir d'autant plus fortement que le nombre de bus est important et que l'on se rapproche du taux d'occupation maximal. Pour éviter d'avoir des écarts importants au niveau des valeurs temporelles d'estimation par rapport aux performances réelles, il vaut mieux privilégier le modèle à base de bus généraux.

D'autre part, les familles actuelles de FPGAs disposent d'un certain nombre de ressources spécifiques dont il faut favoriser l'utilisation si l'on souhaite obtenir des valeurs d'estimation réalistes. C'est la raison pour la-

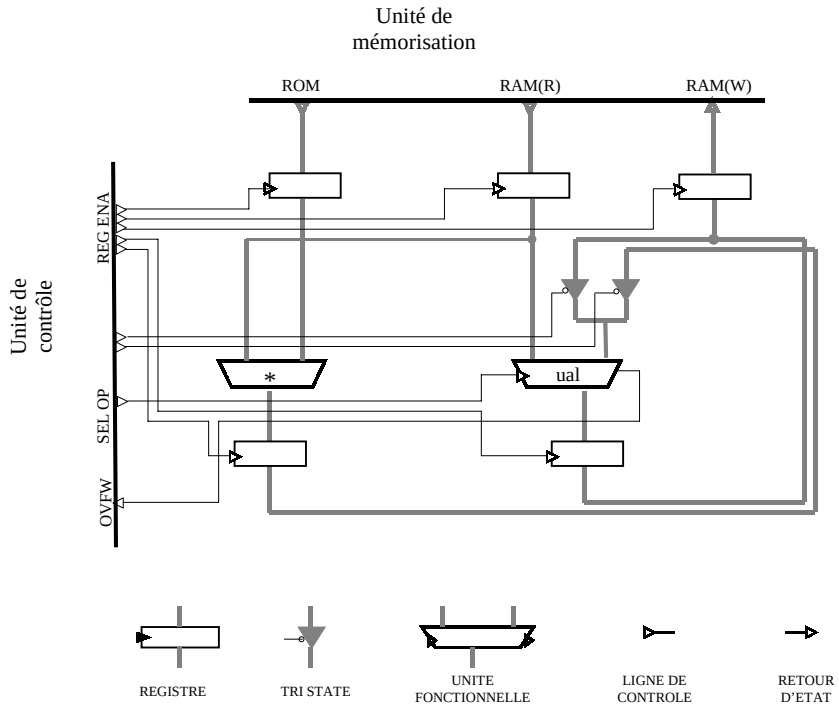


FIG. 3.9 – Modèle d'Architecture pour l'Unité de Traitement

quelle l'estimation tient compte de l'utilisation des buffers trois etats, précaractérisés existant dans l'architecture de certains composants (Virtex). Leur emploi est privilégié à celui de multiplexeurs car ceux-ci entrainent un surcoût dû à l'utilisation de cellules logiques pour leur intégration. L'unité de traitement est donc constituée d'unités fonctionnelles, de registres, et de multiplexeurs / buffers trois états. À chaque unité fonctionnelle est associé un registre en sortie pour la synchronisation des données. Un exemple d'architecture pouvant être générée est donné à la figure 3.9.

3.4.2 Unité de mémorisation

L'unité de mémorisation est constituée d'une ou de plusieurs ROM(s) et / ou RAM(s). En ce qui concerne les mémoires RAMs, nous nous limitons à

l'utilisation de mémoires double port dans un souci de simplification (figure 3.10). Des optimisations sont donc possibles à ce niveau lors de l'intégration

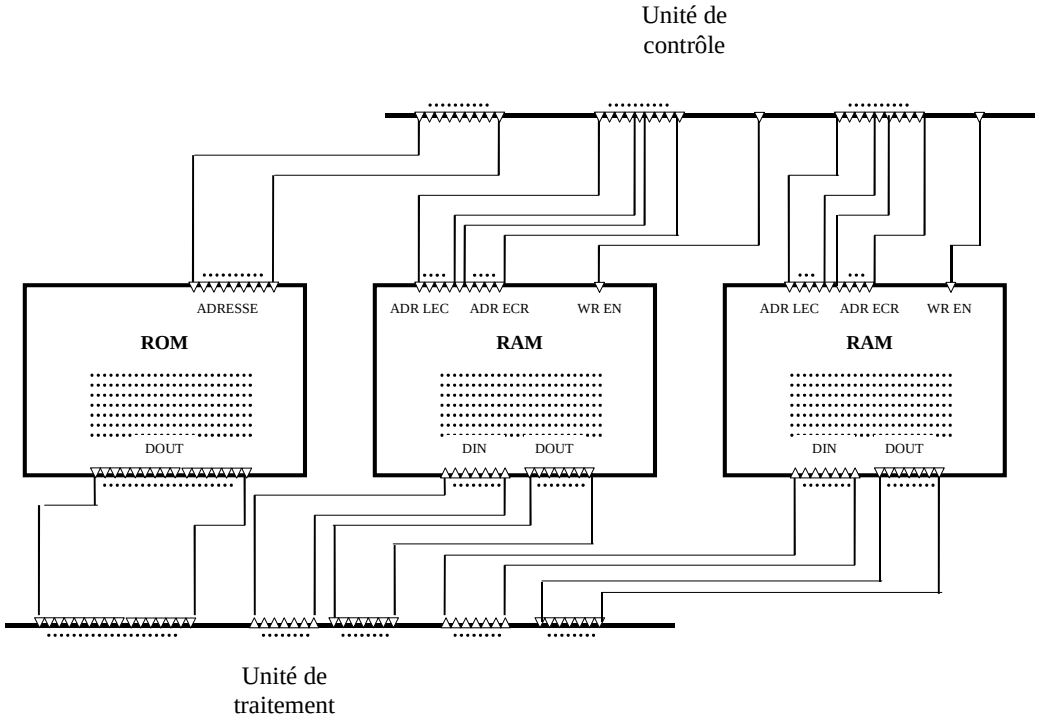


FIG. 3.10 – Modèle d'Architecture pour l'Unité de Mémorisation

du système (par l'emploi de mémoires simple port dans certains cas) et sont laissées au soin du concepteur. Les mémoires peuvent être implantées en utilisant les cellules logiques des FPGAs (mémoires distribuées) ou utiliser les cellules dédiées (mémoires dédiées), possibilité offerte par les dernières familles de composants.

Les mémoires distribuées conviennent bien pour l'intégration de mémoires de petite taille, de l'ordre du kilobyte. Elles se servent de la mémoire de configuration des cellules logiques et peuvent être utilisées pour le stockage des coefficients d'un filtre par exemple. Les mémoires dédiées permettent l'intégration efficace de mémoires performantes de l'ordre de la centaine de kilobytes. Le choix du type de mémoire dépend de la taille mémoire totale nécessaire.

À chaque port de lecture / écriture de chaque mémoire correspond un registre dans l'unité de traitement pour la synchronisation des données (figure

3.9). La génération de l'adresse est confiée à l'unité de contrôle, mais son coût n'est pas pris en compte pour l'estimation.

3.4.3 Unité de contrôle

L'unité de contrôle peut être vue comme un ensemble d'états dont l'enchaînement dépend de l'état courant et des signaux de retour de l'unité de traitement. Elle spécifie la valeur de l'état suivant et des signaux de contrôle nécessaires pour le séquencement du système. Elle est constituée d'un registre d'état, qui a la charge de mémoriser l'état courant, et de la logique de contrôle qui positionne les valeurs des signaux de contrôle et de l'état suivant (figure 3.11).

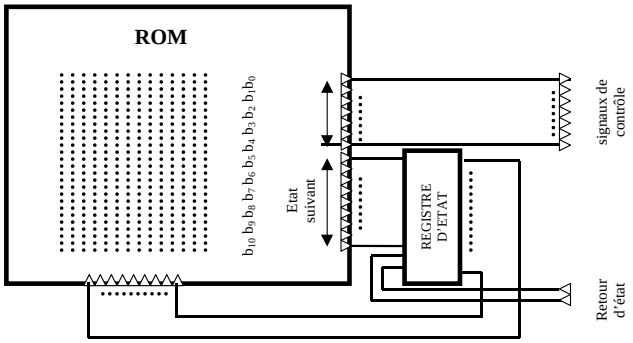


FIG. 3.11 – Modèle d'Architecture pour l'Unité de Contrôle

Les signaux de contrôle pris en compte sont les suivants :

- Les signaux des registres de sortie des opérateurs (figure 3.9).
- Les signaux des registres associés aux bus de données mémoire (figure 3.9).
- Les signaux de sélection d'une opération pour les unités multi-fonctionnelles (figure 3.9).
- Les signaux de l'unité de mémoire : validation d'écriture (*write enable*) et génération des adresses (figure 3.10).

Étant donné le grand nombre de signaux possible, les retours d'états (résultats de comparateurs, bit de retenue d'une UAL, ...) peuvent être négligés.

Une réalisation possible de l'unité de contrôle repose sur la traduction en somme de termes produits pour l'implantation à l'aide de portes logiques.

Mais cette solution n'est intéressante que dans le cas de composants qui disposent de matrices de type PLA, ce qui n'est pas le cas de tous les FPGAs. C'est la raison pour laquelle le modèle de l'unité de contrôle retenu est basé sur l'utilisation d'une ROM (figure 3.11), de façon à permettre l'intégration de la logique de contrôle en utilisant les ressources les plus adaptées du FPGA (ESB dans le cas des Apex, Slices dans le cas des Virtex).

3.5 Conclusion

L'approche de l'exploration architecturale est basée sur des techniques d'estimation et se compose de deux étapes : les estimations au niveau structurel et les estimations au niveau physique. Le niveau structurel a pour but de définir les solutions architecturales. Une solution est caractérisée en terme de nombre d'états, de nombre de ressources allouées et de nombre d'accès mémoire simultanés pour une contrainte de temps. L'estimation structurelle fournit une vision aussi bien locale (i.e pour chaque niveau de hiérarchie) que globale (i.e pour l'application dans son ensemble). Puis, lors de la caractérisation physique, les valeurs d'occupation et de performances sont calculées de façon à permettre au concepteur de faire le choix de la meilleure solution d'implantation. Les informations fournies par l'exploration architecturale (contraintes de temps locales, nombre d'états, allocation, ...) permettent ensuite de guider la synthèse de cette solution, synthèse qui peut être manuelle en s'appuyant sur les modèles architecturaux de l'estimateur ou automatique à partir d'un outil de synthèse d'architecture.

Chapitre 4

Les Estimations au Niveau Structurel

Le chapitre précédent introduit deux niveaux d'estimation : l'un au niveau structurel et l'autre au niveau physique. Après avoir présenté l'approche générale et les différentes hypothèses de travail telles que la spécification d'entrée, la caractérisation du coût et les modèles architecturaux, nous allons maintenant détailler le principe des estimations structurelles. Celles-ci ont plusieurs objectifs : elles doivent permettre de vérifier à priori le bon dimensionnement du composant candidat à l'implantation, d'estimer la taille mémoire et le nombre total de ressources nécessaires. L'exploration de l'espace de conception permet d'évaluer plusieurs solutions architecturales pour différentes allocations possibles, plusieurs fréquences d'horloge et plusieurs degrés de parallélisme. La complexité de l'algorithme d'estimation doit être suffisament faible pour permettre l'exploration d'un grand nombre de solutions. Ainsi, seules les structures de base du graphe de spécification sont "réellement" estimées. Cela signifie que ces résultats sont obtenus par l'ordonnancement des blocs flot de données purs (qui ne contiennent ni contrôle ni hiérarchie). Puis, par un jeu de combinaisons, on remonte progressivement les niveaux de hiérarchie jusqu'à obtenir l'estimation de toute l'application. Cette approche permet de réduire considérablement la complexité de l'estimateur. Les combinaisons se basent sur des modèles d'ordonnancement précis ce qui permet de faciliter la synthèse de la solution architecturale retenue. Les différentes solutions obtenues sont ensuite analysées au niveau physique afin de déterminer les valeurs de performances et d'occupation de l'application sur le composant cible.

4.1 Introduction

Avant de lancer le processus d'estimation, il faut choisir un ou plusieurs composants pour l'évaluation. Une vérification préalable de la faisabilité du système sur le ou les composant(s) candidat(s) est effectuée pendant l'étape de pré-estimation. Ensuite commence véritablement le processus d'estima-

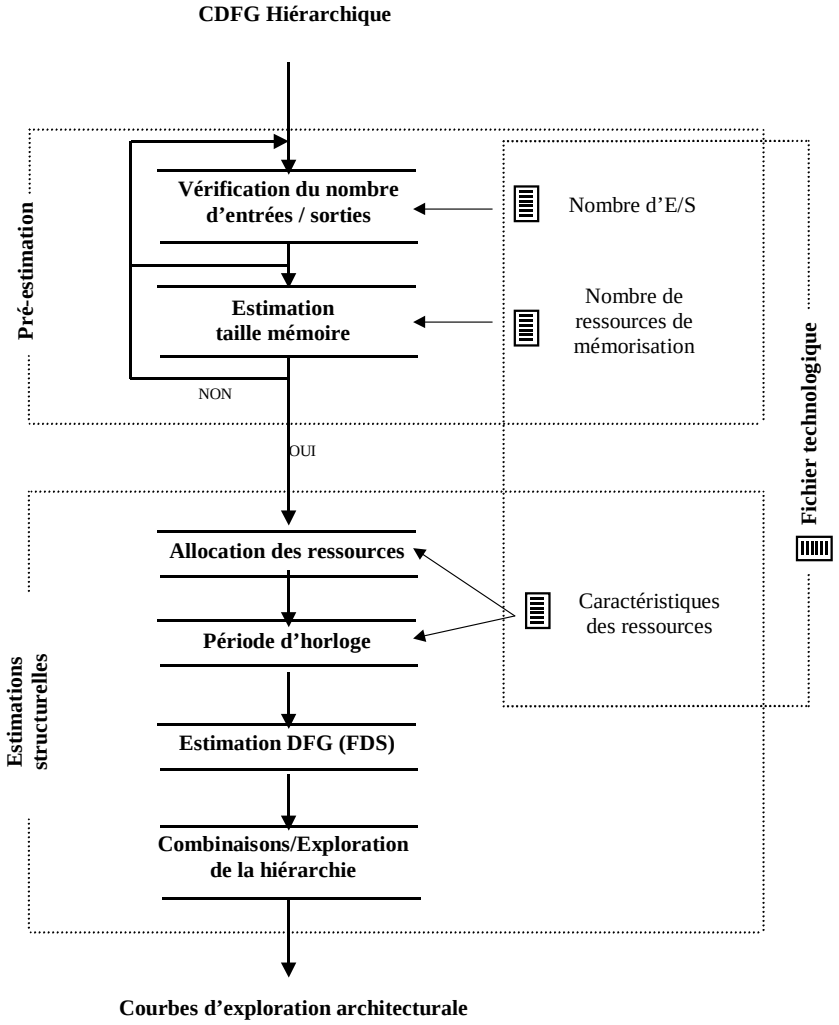


FIG. 4.1 – Flot d'estimations structurelles

tions structurelles. L'exploration de l'espace de conception se base sur la définition de plusieurs allocations de ressources et fréquences d'horloges qui sont chacune évaluées par la suite. À partir d'une allocation et d'une fré-

quence d'horloge, différentes solutions architecturales sont définies. Les solutions sont obtenues à partir d'un ordonnancement partiel de la spécification. Seuls les blocs de base sont traités : les *DFGs* sont ordonnancés pour plusieurs contraintes de temps allant du maximum (chemin critique) au minimum de parallélisme. Puis, l'exploration de la hérarchie permet de combiner les résultats en fonction du type de contrôle (structures conditionnelles ou itératives) ou du type d'exécution (parallèle / séquentielle). De cette façon, on remonte progressivement les niveaux de hiérarchie jusqu'à obtenir l'estimation globale de la spécification.

4.2 Pré-estimation

4.2.1 Choix d'un composant d'implantation

À partir de la spécification comportementale et d'un composant candidat à l'implantation, on effectue tout d'abord une vérification portant sur le nombre d'entrées / sorties : on effectue pour cela un comptage des variables d'entrées / sorties au plus haut niveau de la hiérarchie (noeud composite contenant toute l'application). Connaissant le format de ces variables, on peut calculer le nombre de broches d'entrées / sorties nécessaires pour le routage des signaux vers l'extérieur. Si le FPGA candidat ne dispose pas du nombre de broches nécessaire, il doit être rejeté.

Ensuite, il faut vérifier que le composant dispose d'un nombre suffisant de ressources de mémorisation. Une estimation de la taille mémoire est donc nécessaire. L'estimation de la taille mémoire ROM (TM_{ROM}) est effectuée en comptant les constantes de la spécification. La méthode utilisée pour l'estimation de la taille mémoire RAM (TM_{RAM}) est présentée au paragraphe suivant.

La capacité maximale d'intégration des mémoires distribuées est limitée à 1024 bits et celle des mémoires dédiées est contenue dans le fichier technologique (voir 5.1.2). La capacité totale d'intégration mémoire correspond alors à la somme de la capacité maximale d'intégration dans les cellules logiques ($TM_{CL} = 1024$) et de celle des cellules dédiées (TM_{CD}). Si la taille mémoire totale nécessaire dépasse la capacité d'intégration du composant candidat,

c'est à dire si

$$TM_{RAM} + TM_{ROM} > TM_{CL} + TM_{CD}$$

alors celui-ci doit être rejeté. Cette vérification permet d'éviter l'évaluation de solutions infaisables et de sélectionner un ensemble de composants candidats.

4.2.2 Estimation de la taille mémoire

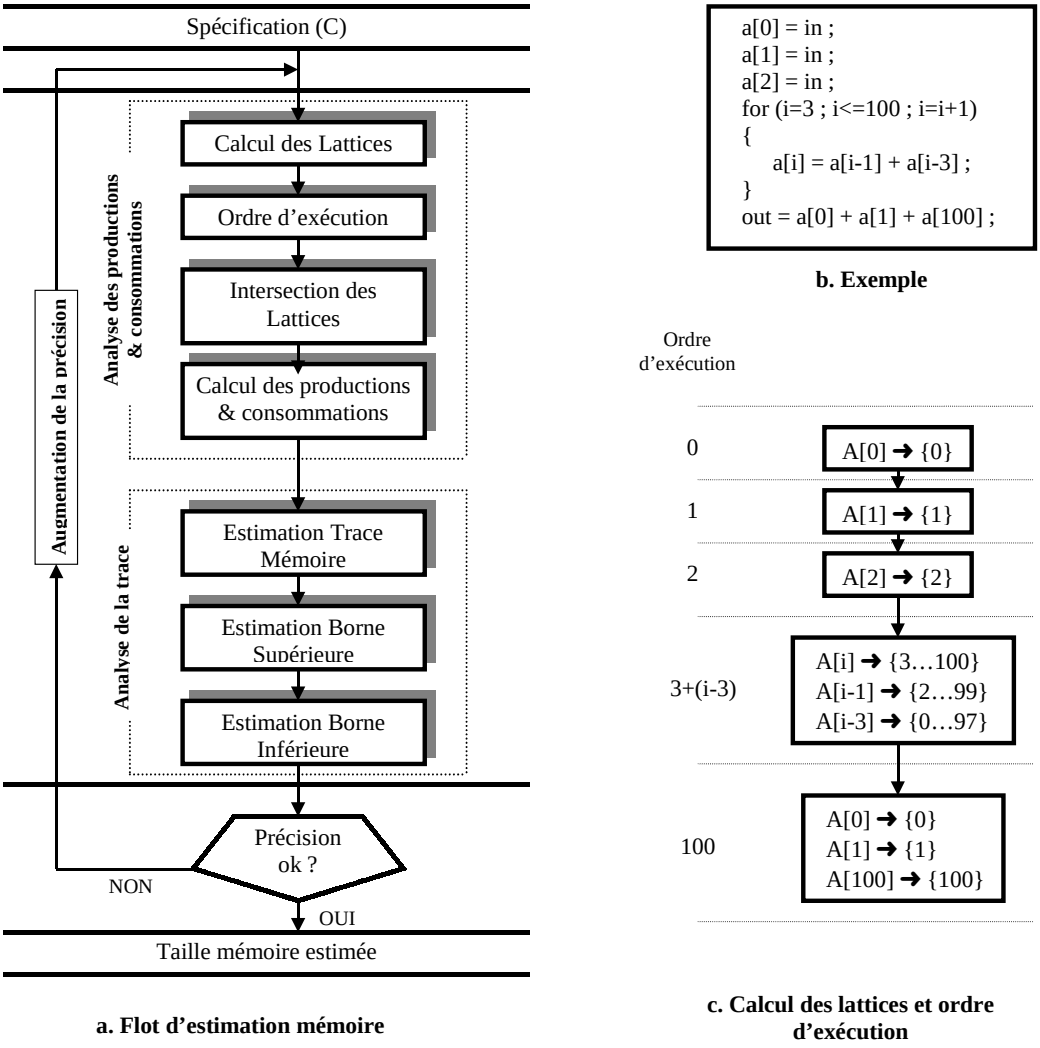
Une caractérisation réaliste du coût de l'implantation doit prendre en compte l'influence de la mémorisation des données. Dans le but de limiter la complexité de l'estimateur, l'estimation de la taille mémoire ne prend en compte que les variables multi-dimensionnelles. Cette estimation permet d'une part de vérifier que le composant candidat à l'implantation possède suffisamment de ressources de mémorisation (étape de pré-estimation) et d'autre part de déterminer la surface occupée par l'unité de mémoire (étape de projection technologique).

Nous décrivons ici une méthode d'estimation de la taille mémoire à un niveau comportemental. Cette méthode décrite dans [28] permet d'estimer le nombre de points mémoire nécessaires à l'exécution d'une spécification en C et s'étend au cas des spécifications concurrentes. Elle se base sur le principe suivant : sachant que tous les éléments d'un tableau ne sont pas utiles au même moment, il est possible de partager les points mémoire entre les différents éléments du tableau. Lors d'une affectation, les productions (terme de gauche de l'affectation) génèrent un ensemble de données appelé *domaine de définition* et les consommations (terme(s) à droite d'une affectation) génèrent le *domaine des opérandes*. Le nombre de données à mémoriser est obtenu à partir du calcul du nombre de scalaires produits et du nombre de scalaires consommés à chaque pas de l'exécution. La différence entre les productions et les consommations permet d'établir la valeur de la trace mémoire pour le pas d'exécution courant. Le maximum atteint par cette trace sur toute la durée de l'exécution correspond à la taille mémoire nécessaire.

L'intéret de cette méthode réside dans le fait qu'elle se situe au niveau comportemental (spécification en C) et que sa faible complexité la destine parfaitement à l'estimation de systèmes dont les descriptions comportent souvent un grand nombre de signaux multi-dimensionnels et de boucles imbriquées. De plus, son application s'étend au cas des spécifications concurrentes

et la précision peut être ajustée en fonction de la complexité de la spécification. Elle atteint 1.5 % d'erreur pour 5.1 secondes sur l'exemple d'une transformée en ondelettes 2D.

La stratégie d'estimation est présentée à la figure 4.2.a. La première étape



a. Flot d'estimation mémoire

b. Exemple

c. Calcul des lattices et ordre d'exécution

FIG. 4.2 – Estimation mémoire : principe et exemple

consiste à calculer pour chaque accès en lecture (référence à un tableau situé à droite d'une affectation), le nombre d'éléments consommés (nombre d'éléments qui sont lus pour la dernière fois) et pour chaque accès en écriture, le nombre d'éléments produits (nombre d'éléments qui sont produits pour la première fois). Pour chaque accès à un tableau, on calcule une lattice qui

représente les éléments du tableau accédés. Pour l'exemple de la figure 4.2.b, dans l'affectation $a[i] = a[i-1]+a[i-3]$ où i est l'indice de boucle variant de 3 à 100, les deux accès en lecture au tableau a génèrent les lattices $\{2..99\}$ et $\{0..97\}$ respectivement. Ensuite, celles-ci sont ordonnées de façon à déterminer les consommations en fonction de l'ordre d'exécution des instructions du code. L'ordre d'exécution ne correspond pas au moment exact de l'exécution d'une instruction, mais traduit plutôt l'ordre relatif d'exécution des affectations (figure 4.2.c). La connaissance de cet ordre est indispensable car on a besoin de connaître le nombre d'éléments produits et consommés à chaque stade de l'exécution.

Les éléments d'un tableau sont consommés par un accès en lecture si ces éléments ne sont pas relus plus tard dans le code. Donc pour connaître le nombre d'éléments consommés, il faut faire l'intersection de la lattice correspondant à cette lecture avec toutes les lattices de consommation qui se produisent plus tard. Ces intersections correspondent aux éléments qui ne sont pas consommés. Les consommations de ce domaine sont donc les éléments de la lattice qui n'appartiennent à aucune intersection. De la même façon, les éléments d'un tableau qui sont produits par un accès en écriture sont ceux qui n'ont pas déjà été produits lors d'une précédente affectation. Les éléments produits correspondent dans ce cas au complément de l'intersection de la lattice de production avec toutes les lattices de production précédentes.

Le tableau 4.1 présente les résultats du calcul des productions et des consommations pour l'exemple de la figure 4.2.b : Le calcul des consom-

Domaine	a[0]	a[1]	a[2]	a[i]	
Productions	1	1	1	98	
Domaine	a[i-1]	a[i-3]	a[0]	a[1]	a[100]
Consommations	2	96	1	1	1

TAB. 4.1 – Calcul des productions et des consommations

mations du domaine $a[i-3]$ est réalisé de la manière suivante : la lattice correspondante est $\{0...97\}$ et les lattices correspondant à des lectures suivantes sont $\{0\}$, $\{1\}$ et $\{100\}$. Les éléments accédés et non relus par $a[i-3]$ sont donc repésentés par la lattice $\{2...97\}$ qui contient 96 éléments. En ce qui concerne les productions, l'analyse est simple dans ce cas puisque l'intersection des domaines est vide. Le nombre de données produites par un

domaine est egal au nombre d'éléments de la lattice.

Une estimation de la trace mémoire peut ensuite être calculée à partir des productions et des consommations de chaque domaine, ainsi qu'une borne supérieure et une borne inférieure. À chaque pas d'exécution du code, la taille mémoire augmente ou diminue en fonction des éléments produits ou consommés. La trace mémoire traduit le nombre de points mémoire nécessaires au cours de l'exécution de l'algorithme. La plus grande valeur atteinte par cette trace donne la taille mémoire nécessaire. Si la trace est connue pour chaque pas d'exécution de l'algorithme, on peut déterminer la valeur exacte du nombre d'emplacements mémoires. Sinon, on calcule une borne supérieure et une borne inférieure. C'est ce qui se produit dans le cas des boucles : la trace peut être calculée de façon exacte à l'extérieur des boucles en ajoutant à la taille mémoire courante le nombre d'éléments produits en en soustrayant le nombre d'éléments consommés. À l'intérieur des boucles, les ordonnancements ASAP (figure 4.3.a) et ALAP (figure 4.3.b) des productions et consommations permettent de calculer deux traces mémoires fournissant ainsi une borne supérieure et une borne inférieure de la taille mémoire.

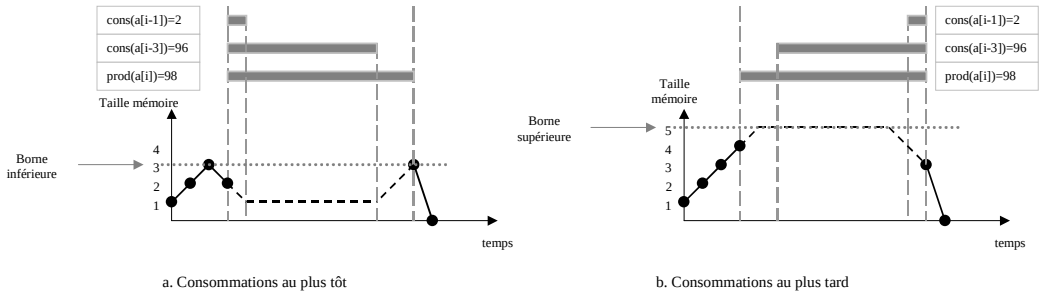


FIG. 4.3 – Estimation des bornes supérieure et inférieure

Puisque l'ordre d'exécution des consommations est quelque part entre la solution ASAP et la solution ALAP, la taille mémoire exacte se trouve quelque part entre la borne inférieure et la borne supérieure. Cette valeur pourrait être déterminée à partir de la connaissance de l'instant de consommation de chaque élément du tableau à l'intérieur des boucles, mais cela conduirait à une trop grande complexité de la méthode. Pour résoudre ce problème, il suffit de décomposer la boucle externe en parties égales et de recalculer les bornes (figure 4.4). Par exemple, si les indices de la boucle

varient de 1 à 100, on la décompose en deux boucles variant de 1 à 50 et de 51 à 100. Pour chaque boucle, on recalcule le nombre de consommations et de productions. Comme les écarts entre les moments où se produisent les consommations sont plus petits, les tailles mémoires dans le cas ASAP et dans le cas ALAP seront plus proches. Ainsi, en recalculant les bornes, on atteint une estimation plus précise. Cette précision converge vers la valeur exacte de la taille mémoire si on répéte plusieurs fois le processus de dichotomie.

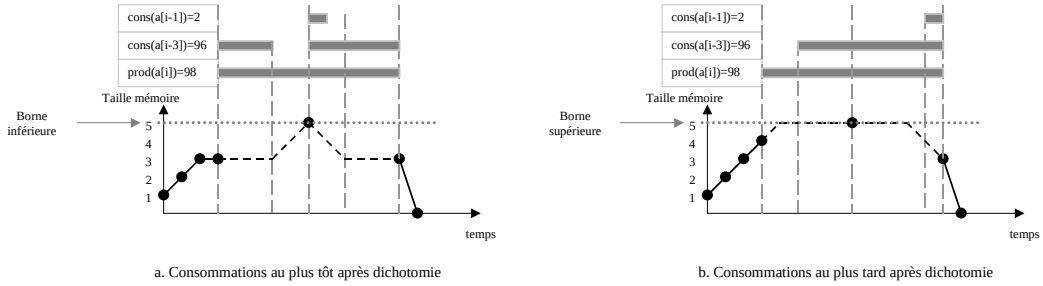


FIG. 4.4 – Amélioration de la précision : méthode dichotomique

Au final, on obtient une estimation de la taille mémoire nécessaire pour chaque tableau de la spécification. Leur somme fournit la taille mémoire RAM totale nécessaire.

4.3 Exploration à l'étape de sélection / allocation

L'exploration à l'étape de sélection / allocation porte sur l'analyse de différentes allocations et fréquences d'horloge. Elle est abordée selon deux approches : elle peut être automatique et exhaustive (évaluation du maximum de solutions possibles) ou guidée par le concepteur. Dans le premier cas, il faut construire toutes les solutions d'allocation possibles en fonction des ressources de la librairie d'opérateurs ainsi que toutes les solutions possibles pour la fréquence d'horloge et les évaluer. L'autre approche consiste à choisir une solution parmi l'ensemble d'allocation et de fréquences d'horloge possibles, de l'évaluer, puis éventuellement d'en évaluer une ou plusieurs autres en fonction des résultats de l'estimation.

4.3.1 Allocation des ressources

Un parcours du graphe de spécification permet de déterminer la liste des opérations. À chaque opération correspond un ou plusieurs opérateurs dans le fichier technologique, ce qui résulte en un ensemble d'allocations candidates. La figure 4.5 présente un exemple de détermination des ensembles d'allocations possibles : à partir de la liste des opérations du graphe ($*$, $+$, $-$, $<$) et de celle des opérateurs de la librairie réalisant ces opérations (*mul*, *add*, *sub*, *addsub*, *cmp*), on détermine quatre allocations possibles. On peut

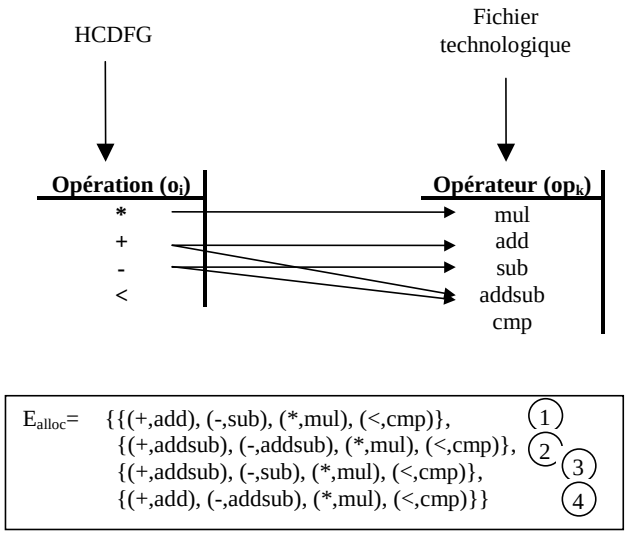


FIG. 4.5 – Les ensembles d'allocation opération - opérateur possibles

choisir d'effectuer une exploration exhaustive de toutes les possibilités, solution rendue possible par le fait que le nombre d'unités fonctionnelles décrites dans la librairie est limité ce qui réduit l'espace de recherche. On peut aussi ne sélectionner que quelques allocations et laisser le concepteur guider l'exploration. Dans notre exemple, on pourrait par exemple choisir d'éliminer les solutions 3 et 4, afin de n'étudier que l'effet de l'allocation d'un additionneur / soustracteur au lieu de deux opérateurs distincts.

4.3.2 Détermination de la fréquence d'horloge

Une fois l'allocation effectuée (qu'elle soit automatique ou guidée), il faut déterminer la fréquence d'horloge. Cela permettra par la suite d'associer un

nombre de cycles à chaque noeud du graphe et d'effectuer l'ordonnancement des *DFGs*. Un ensemble initial de valeurs candidates est déterminé. Il est constitué de toutes les valeurs entières contenues dans l'intervalle $[T_{min}, T_{max}]$ où T_{min} représente le temps de traversée de l'opérateur le plus rapide et T_{max} celui de l'opérateur le plus lent. Cet ensemble peut être simplifié en tenant compte du fait que certaines solutions sont sous optimales. Pour chaque solution, on calcule la distribution du nombre de cycles induit pour chaque type d'opération. Pour une distribution donnée du nombre de cycles, on ne garde que celle pour laquelle la valeur de la période d'horloge est la plus faible afin d'optimiser le temps d'exécution ($N_{cycles} * T_h$).

Prenons un exemple simple où la spécification est constituée de deux types d'opérations (addition et multiplication) pour lesquelles l'allocation est constituée d'un additionneur (5 ns) et d'un multiplieur (13 ns). Le tableau 4.2 présente les périodes d'horloge candidates. Selon le principe énoncé précédemment, on peut éliminer les distributions 2, 4, 5, 6, 7 et 8. Les solutions $T_h = 5$ et $T_h = 6$ possèdent la même distribution du nombre de cycles (3 pour la multiplication et 1 pour l'addition). Dans ces deux cas, l'estimation structurelle va donner exactement les mêmes résultats. En particulier, les contraintes de temps N_{cycles} seront les mêmes. La solution $T_h = 5$ est donc forcément plus performante que la solution $T_h = 6$ ($T_{ex} = N_{cycles} * 5$ contre $T_{ex} = N_{cycles} * 6$). Au final, il ne reste que trois valeurs candidates pour la période d'horloge. Comme dans le cas de l'allocation, on peut choisir d'évaluer toutes les solutions ou de sélectionner les plus pertinentes aux yeux du concepteur et d'affiner ainsi progressivement l'exploration.

distribution	T_h	N_{cycles}*(add)*	N_{cycles}*(mul)*
1	5	1	3
2	6	1	3
3	7	1	2
4	8	1	2
5	9	1	2
6	10	1	2
7	11	1	2
8	12	1	2
9	13	1	1

a. périodes d'horloge candidates

distribution	T_h	N_{cycles}*(add)*	N_{cycles}*(mul)*
1	5	1	3
3	7	1	2
9	13	1	1

b. périodes d'horloge possibles

TAB. 4.2 – Détermination d'un ensemble de périodes d'horloge

4.4 Estimation des DFGs

Les DFGs constituent les blocs de base de la spécification. Leur estimation peut être réalisée au moyen d'une méthode d'estimation du nombre de ressources ou d'une technique d'ordonnancement. Quelque soit la méthode choisie, la description ne doit contenir aucune structure de contrôle et on a besoin de connaître les dates d'ordonnancement au plus tôt et au plus tard (figure 4.6). Le choix de la technique utilisée dépend du rapport com-

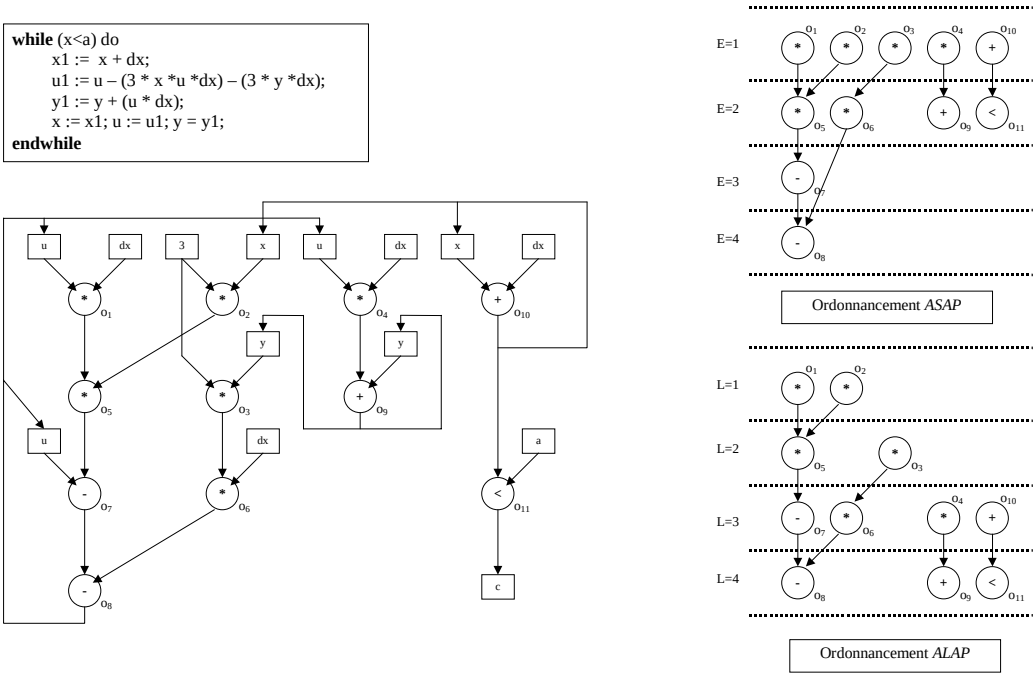


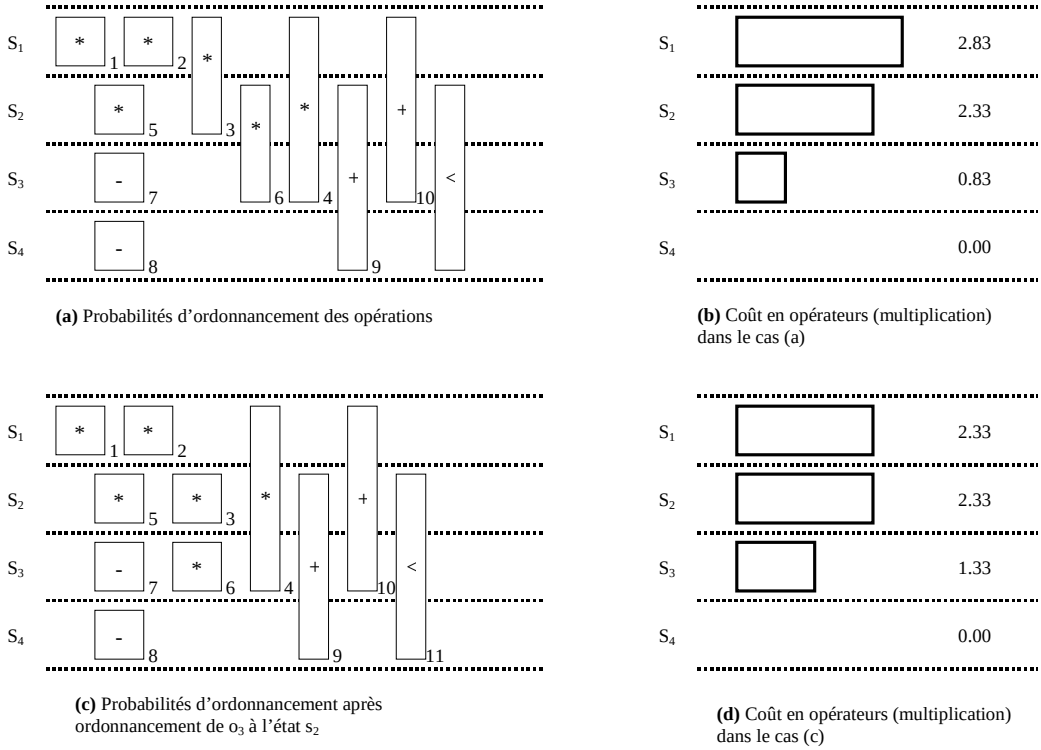
FIG. 4.6 – Ordonnancements *ASAP* et *ALAP*

plexité / précision souhaité. En effet, les techniques d'ordonnancement sont plus complexes (typiquement $O(n^2)$) que les techniques d'estimation ($O(n)$), mais elles garantissent la faisabilité de la solution trouvée.

L'algorithme utilisé pour l'évaluation de la méthode est le *Force Directed Scheduling* (FDS) décrit dans [1], qui présente une complexité importante et permettra ainsi d'évaluer le temps de calcul de l'algorithme dans le pire cas. D'autres techniques d'estimation ou d'ordonnancement (voir 2.3.2) sous contrainte de temps peuvent être utilisées en fonction de la complexité de la spécification et du nombre de solutions à explorer.

La technique du FDS repose sur une distribution uniforme des opérations de même type sur l'ensemble des pas de contrôle disponibles ($mrange(o_i)$). L'algorithme se base sur le calcul des dates $ASAP$ et $ALAP$ pour déterminer les bornes sur les dates d'ordonnancement de chaque opération. On suppose que chaque opération o_i possède une probabilité d'ordonnancement uniforme à l'intérieur de l'encadrement calculé précédemment et une probabilité nulle en dehors. Ainsi pour un pas de contrôle s_j tel que $E_i \leq j \leq L_i$ (où E_i et L_i représentent les dates au plus tôt et au plus tard), la probabilité que l' opération o_i soit ordonnancée dans l'état s_j vaut $p_j(o_i) = 1/(L_i - E_i + 1)$.

Ces calculs de probabilité sont illustrés à la figure 4.7.a pour l'exemple de la figure 4.6. Les opérations o_1, o_2, o_5, o_7 et o_8 ont une probabilité de 1



(a) Probabilités d'ordonnancement des opérations

(b) Coût en opérateurs (multiplication) dans le cas (a)

(c) Probabilités d'ordonnancement après ordonnancement de o_3 à l'état s_2

(d) Coût en opérateurs (multiplication) dans le cas (c)

FIG. 4.7 – Exemple d'application de la technique FDS

d'être ordonnancées dans les états s_1, s_2, s_3 et s_4 respectivement car les dates $ASAP$ et $ALAP$ sont égales pour ces opérations. La largeur d'un rectangle de la figure 4.7 traduit la probabilité $(1/L_i - E_i + 1)$ pour une opération d'être ordonnancée dans cet état. Par exemple, l'opération o_3 possède une

probabilté de 0.5 d'être ordonnancée à l'état s_1 ou s_2 ($p_1(o_3) = p_2(o_3) = 0.5$).

Un ensemble de distributions de probabilités est construit à partir des probabilités d'ordonnancement de chaque opération, sous forme de graphique à barres. À chaque opération correspond une barre qui représente le coût en opérateurs (EOC) pour chaque état. Le coût attendu pour l'état s_j de l'opération de type k est donné par $EOC_{j,k} = c_k * \sum_{i,s_j \in mrange(o_i)} p_j(o_i)$, où o_i est une opération de type k et c_k le coût de l'unité fonctionnelle réalisant l'opération de type k. Par exemple, on calcule la valeur de $EOC_{1,multiplication}$ de la manière suivante : $c_{multiplication} * (p_1(o_1) + p_1(o_2) + p_1(o_3) + p_1(o_4))$ ce qui donne $c_k * (1.0 + 1.0 + 0.5 + 0.33) = 2.83 * c_k$. L'exemple de la figure 4.7.b montre le coût en opérateurs pour l'opération de multiplication dans les quatre états de contrôle, respectivement 2.83, 2.33, 0.83 et 0.00. Comme les unités fonctionnelles peuvent être partagées entre les états, la valeur maximale du coût en opérateurs (EOC) atteinte sur tous les états donne une mesure du coût d'implantation pour les opérations de ce type. Ce calcul du coût est répété pour tous les autres types d'opérations du graphe.

Le principe du FDS repose sur une répartition équilibrée des unités fonctionnelles entre tous les états basée sur l'uniformisation du coût en opérateurs. L'algorithme 4.1 présente une façon d'effectuer cette uniformisation de l'EOC [1]. La variable $S_{current}$ représente l'ordonnancement partiel le plus récent et S_{work} est une copie sur laquelle un certain nombre d'affectations temporaires sont effectuées. À chaque itération, les variables $BestOp$ et $BestStep$ mémorisent la meilleure opération candidate et l'état le plus intéressant pour l'ordonnancement. Une fois ces variables déterminées pour une itération donnée, l'ordonnancement $S_{current}$ est modifié par l'appel à la fonction $SCHEDULE\_OP(S_{current}, o_i, s_j)$ qui renvoie un nouvel ordonnancement après placement de l'opération o_i dans l'état s_j. L'ordonnancement d'une opération dans un état donné implique une modification des probabilités des autres opérations. La fonction $ADJUST\_DISTRIBUTION$ ajuste les distributions de probabilités des noeuds prédécesseurs et successeurs du graphe.

La fonction $COST(S)$ permet l'évaluation du coût d'implantation de l'ordonnancement partiel S. Elle peut être calculée en réalisant la somme des

valeurs de coût EOC pour chaque type d'opération :

$$COST(S) = \sum_{1 \leq k \leq m} max_{1 \leq j \leq s} EOC_{j,k}$$

Le calcul de ce coût utilise les valeurs des dates $ASAP$ / $ALAP$ des noeuds du graphe.

```
ForceDirectedScheduling()
DÉBUT
    ASAP();
    ALAP();
    TANT QUE il existe o_i tel que E_i ≠ Li FAIRE
        MaxGain = −∞;
        /* On essaie d'ordonnancer les opérations non ordonnancées */
        /* dans chaque état possible */
        POUR CHAQUE o_i, E_i ≠ L_i  FAIRE
            POUR CHAQUE j, E_i ≤ j ≤ L_i  FAIRE
                S_work = SCHEDULE_OP(S_current, o_i, s_j);
                ADJUST_DISTRIBUTION(S_work, o_i, s_j);
                SI COST(S_current) − COST(S_work) > MaxGain  ALORS
                    MaxGain = COST(S_current) − COST(S_work);
                    BestOp = O_i; BestStep = s_j;
                FIN SI
            FIN POUR
        FIN POUR
        S_current = SCHEDULE_OP(S_currentBestOp, BestStep);
        ADJUST_DISTRIBUTION(S_current, BestOp, BestStep);
    FIN TANT QUE
FIN
```

ALG. 4.1 - Algorithme Force-Directed Scheduling

À chaque itération, le coût des différents ordonnancements est calculé en utilisant la variable S_{work}. Celui qui minimise la fonction de coût précédente est validé et l'ordonnancement courant $S_{current}$ est mis à jour. Ceci implique qu'à chaque itération, une opération o_i est ordonnancée à l'état s_k où $E_i \leq k \leq L_i$. La distribution de probabilité pour l'opération o_i s'en trouve donc modifiée et vaut $p_k(o_i) = 1$ et $p_j(o_i) = 0$ pour $j \neq k$. L'opération o_i est ainsi fixée pour tout le reste de l'exécution de l'algorithme. Dans l'exemple précédent, la distribution de probabilité initiale des multiplications est présentée à la figure 4.7.b. où les coûts d'affectation de chaque opération aux différents états sont calculés. L'affectation de l'opération o_3 à l'état s_2 minimise le coût OEC de la multiplication car la valeur $\max(p_j)$ passe de 2.83

à 2.33. Cette affectation est donc retenue et les valeurs de probabilités s'en trouvent modifiées (figure 4.7.c).

Pour obtenir les courbes de caractérisation, on applique l'ordonnancement pour plusieurs contraintes de temps N_{cycles}. Celles-ci varient du chemin critique (date *ASAP*) au minimum de parallélisme qui correspond à la contrainte de temps pour laquelle le nombre d'unité fonctionnelles de chaque type et d'accès mémoires simultanés valent 1. Le nombre d'états est aussi estimé et correspond au nombre de cycles (dans ce cas précis de l'estimation des *DFGs*).

Une extension de cette technique est réalisée afin d'obtenir une estimation du nombre d'accès mémoires simultanés. Il suffit pour cela de traiter les noeuds mémoires de la même façon que les noeuds traitement et de leur attribuer un nombre de cycles à partir duquel on pourra calculer les dates *ASAP* / *ALAP*. Les noeuds pris en compte correspondent à des lectures / écritures de données multi-dimensionnelles (contenues dans des RAMs) ainsi que les données de type constante (contenues dans des ROMs). On obtient à l'issu de l'ordonnancement le nombre d'accès RAM simultanés en lecture, le nombre d'accès RAM simultanés en écriture et nombre d'accès simultanés à la ROM. L'analyse des valeurs obtenues permet par la suite d'effectuer un certain nombre d'hypothèses sur l'unité de mémorisation du système.

4.5 Les différents types de combinaison

Une fois estimés les blocs de base, il faut combiner les résultats d'estimation en fonction des structures de contrôle et des dépendances d'exécution. La figure 4.8 présente les différents cas de combinaison qui peuvent se présenter. À partir des quatre types de combinaison (itérative, conditionnelle, concurrente et séquentielle), il est possible de remonter progressivement chaque niveau de la hiérarchie. On arrive ainsi à estimer toute l'application en partant de l'estimation des blocs de base. Les algorithmes de combinaison se basent dans chaque cas sur des schémas d'exécution bien définis, ce qui permet de faciliter l'intégration du système après estimation. Par exemple, l'estimation d'une structure itérative est effectuée à partir de celle du coeur de la boucle et d'un modèle d'ordonnancement précis. La connaissance des caractéristiques du coeur (allocation, nombre d'états, contrainte de temps) et de l'ordonnan-

cement des itérations (séquentialisation des itérations ou pipeline) permet alors la synthèse.

Cette approche peut introduire une erreur car elle ne tient pas compte de l'ordonnancement global mais se base plutôt sur l'agencement d'ordonnancements partiels de *DFGs*, ce qui peut laisser échapper un certain nombre d'optimisations. Toutefois, à l'issue de cette étape, le concepteur dispose d'un

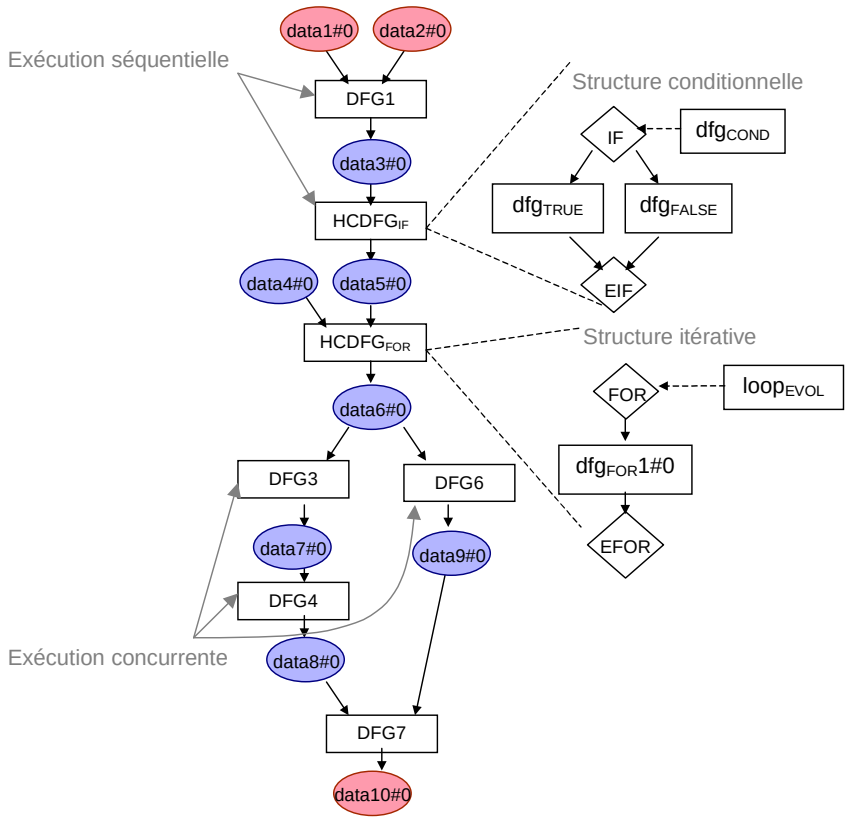


FIG. 4.8 – Les quatre types de combinaison hiérarchique

ensemble de solutions architecturales pour lesquelles des valeurs fiables d'estimation temps / surface lui sont fournies. Nous allons maintenant passer en revue les différents types de combinaisons et le détail des modèles d'exécution et d'estimation dans chaque cas.

4.5.1 Structures conditionnelles

Une structure conditionnelle est l'équivalent d'une instruction "if" ou "case" dans un langage de programmation. Elle est constituée de deux branches mutuellement exclusives qui s'exécutent en fonction du résultat de l'évaluation d'une condition. L'exécution commence par l'évaluation de la condition, puis un saut est effectué à l'adresse du premier état de la branche à exécuter. Dans le cas du "if", l'estimation dépend des résultats d'estimation des trois graphes composant la structure. La figure 4.9 présente un exemple de spécification d'une structure conditionnelle sur laquelle nous allons illustrer l'estimation. L'extension de la méthode dans le cas du "case" n'est pas présenté dans un souci de clarté. Cette extension est triviale et n'apporte pas de complexité supplémentaire. Dans la suite de ce paragraphe, les notations

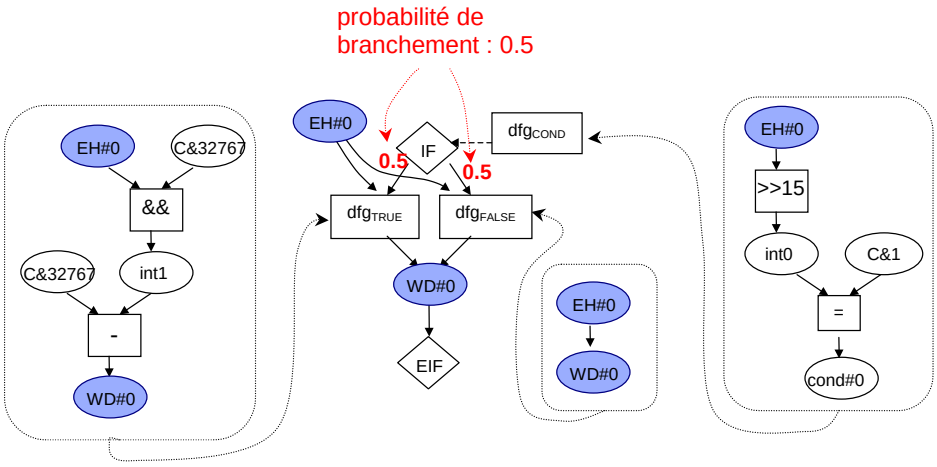


FIG. 4.9 – Graphe de spécification d'une structure conditionnelle

suivantes sont adoptées : l'indice 0 est relatif au graphe de condition, l'indice 1 au graphe exécuté si la condition est vraie et l'indice 2 au graphe exécuté si la condition est fausse.

Pour obtenir le temps d'exécution moyen N'_{cycles}, on utilise des probabilités de branchement P_{b_1} et P_{b_2} associées à chaque arc de transition [3] :

$$N'_{cycles} = N_{cycles_0} + [P_{b_1} * N_{cycles_1}] + [P_{b_2} * N_{cycles_2}] + 1$$

Celles-ci peuvent être définies de trois manières : équiprobabilités (probabilité 1/2 de transition pour chaque branche), probabilités définies par l'utilisateur

ou probabilités issues de la simulation (*profiling*). Le terme +1 correspond au saut vers le premier état de la branche sélectionnée. Le nombre d'états est la somme des états nécessaires pour l'évaluation de la condition et l'exécution de chaque sous-branche, auquel on rajoute un état pour le branchement :

$$N'_{états}(N'_{cycles}) = N_{états_0}(N_{cycles_0}) + N_{états_1}(N_{cycles_1}) + N_{états_2}(N_{cycles_2}) + 1$$

Le nombre de ressources est obtenu en faisant l'hypothèse du maximum de réutilisation. Dans ce cas, l'ordonnancement consiste à partager les ressources entre les branches et la condition. Si on suppose que le nombre de ressources de type k est connu pour la condition ainsi que pour chaque branche, et valent respectivement $N_{op_{k_0}}(N_{cycles_0})$, $N_{op_{k_1}}(N_{cycles_1})$ et $N_{op_{k_2}}(N_{cycles_2})$, le nombre total de ressources de type k est calculé de la manière suivante :

$$N'_{op_k}(N'_{cycles}) = MAX[N_{op_{k_0}}(N_{cycles_0}), N_{op_{k_1}}(N_{cycles_1}), N_{op_{k_2}}(N_{cycles_2})]$$

D'autre part, un surcoût dû au branchement doit être rajouté. Un ou plusieurs multiplexeurs deux entrées sont chargés d'effectuer l'aiguillage des données en fonction des résultats de l'évaluation de la condition. Le nombre de multiplexeurs est égal au nombre de variables de sorties de la structure conditionnelle et le format des données est celui des variables de sorties. De plus, il faut rajouter à ce coût le nombre de multiplexeurs induits par le partage de ressources possible entre les différentes branches. L'analyse de la réutilisation est traité au chapitre 4.6.3.

Dans le cas de l'exemple de la figure 4.9, si on suppose que la période d'horloge vaut la latence de l'opérateur le plus lent (comparateur d'égalité dans ce cas, soit 6.338 ns), chaque opération s'exécute en 1 cycle. L'estimation

graphe	and	sub	shr	eq	mux	ROM	Nb états	Nb cycles
cond	0	0	1	1	0	1	2	2
b_1	1	1	0	0	0	1	3	3
b_2	0	0	0	0	0	0	0	0
IF	1	1	1	1	1	1	6	4.5

TAB. 4.3 – Exemple d'estimation d'une structure conditionnelle

de chaque graphe, puis de la structure dans son ensemble donne les résultats du tableau 4.3. La condition s'exécute en 2 cycles, la branche *true* en 3 cycles et la branche *false* en 0 cycle. Le temps d'exécution moyen est de 4.5 cycles

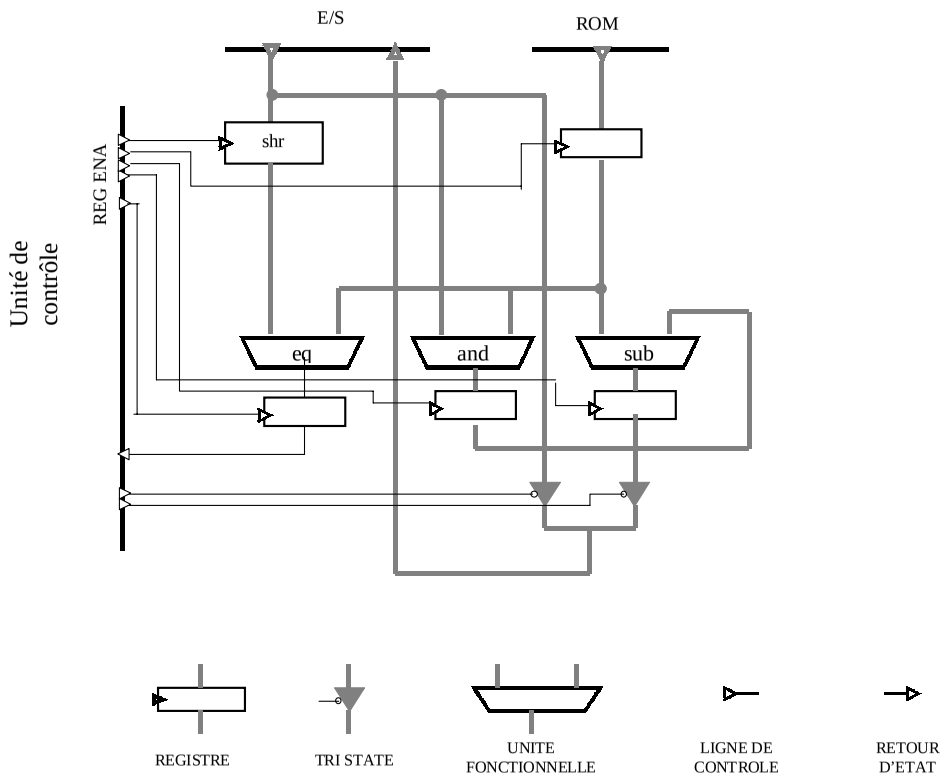


FIG. 4.10 – Implantation d'une structure conditionnelle

$(2+(0.5*3)+(0.5*0)+1)$ et nécessite le codage de 6 états. Notons l'estimation d'un multiplexeur 16 bits pour l'affectation correcte de la variable de sortie. L'architecture de l'unité de traitement correspondante est donnée à la figure 4.10.

4.5.2 Structures itératives

Une description comportementale contient fréquemment des boucles ou structures itératives. Ce type de structure est récurrent dans les applications de traitement du signal et sert souvent à décrire des opérations de filtrage. Elle permet une simplification de la spécification lorsqu'un ensemble de traitements répétitifs est appliqué à un ensemble de données. Leur estimation est un problème délicat car il faut prendre en compte le parallélisme potentiel entre les différentes itérations. On peut en effet obtenir une exécution performante en exécutant plusieurs itérations de la même boucle de

façon concurrente. Il existe donc plusieurs solutions en fonction du nombre d'itérations parallélisées. L'approche que nous proposons consiste à déduire l'estimation de la structure itérative à partir des résultats d'estimation du coeur de boucle. On évite ainsi de dérouler la spécification, c'est à dire, de transformer le *CDFG* en *DFG*, description à partir de laquelle il est possible d'appliquer l'algorithme d'ordonnancement des blocs de base, ce qui permet d'obtenir des valeurs d'estimation plus précises. Cette façon de procéder est en effet trop complexe car elle implique une augmentation sensible du nombre de noeuds à ordonnancer, et donc des temps d'estimation.

Plusieurs techniques sont disponibles en fonction du type d'ordonnancement. Nous présentons ici les différents types d'ordonnancement utilisés comme modèles pour l'estimation et les régles de combinaison correspondantes. Puis nous analysons les différents types de boucles afin de déterminer dans quel cas de figure chaque modèle doit être utilisé. Dans la suite, nous utilisons les notations N_{cycles}, $N_{états}$ et N_{op_k} pour exprimer le nombre de cycles, le nombre d'états et le nombre d'opérateurs de type k du coeur de la boucle et N'_{cycles}, $N'_{états}$ et N'_{op_k} pour exprimer le nombre de cycles, le nombre d'états et le nombre d'opérateurs de type k de l'ensemble de la boucle.

La façon la plus simple pour ordonnancer une boucle consiste à juxtaposer l'exécution de chaque itération jusqu'à la validation d'une condition d'arrêt (figure 4.11). Si on prend l'exemple d'une boucle de N_{iter} itérations dont

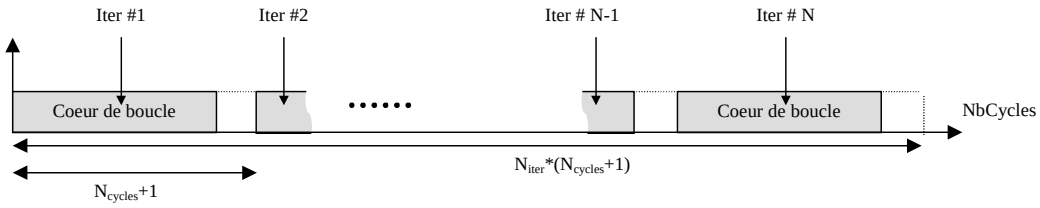


FIG. 4.11 – Ordonnancement des boucles : séquentiel

le coeur s'exécute en N_{cycles} cycles, le nombre de cycles nécessaires pour l'exécution séquentielle des itérations est :

$$N'_{cycles} = N_{iter} * (N_{cycles} + 1)$$

Le nombre d'états de contrôle est le même que pour le séquencement du coeur de boucle, un état supplémentaire permet d'effectuer le saut conditionnel du

dernier au premier état.

$$N'_{états}(N'_{cycles}) = N_{états}(N_{cycles}) + 1$$

Le nombre de ressources quand à lui est égal au nombre de ressources pour l'exécution du coeur :

$$N'_{op_k}(N'_{cycles}) = N_{op_k}(N_{cycles})$$

Cette approche simple est applicable pour tous les types de boucle.

Si le nombre d'itération est inconnu, N_{iter} peut être déterminé, soit par l'utilisateur, soit par *profiling* comme dans le cas des probabilités de branchement pour les structures conditionnelles. Dans le cas des boucles déterministes (boucles dont le nombre d'itération est connu et qui ne comporte aucune structure conditionnelle), cette solution ne permet pas d'exploiter le parallélisme potentiel.

La deuxième technique d'ordonnancement appelée *déroulage partiel* consiste à exécuter plusieurs itérations en parallèle. On définit pour cela un facteur de parallélisme f_p qui traduit le degré de déroulement de la boucle. La figure

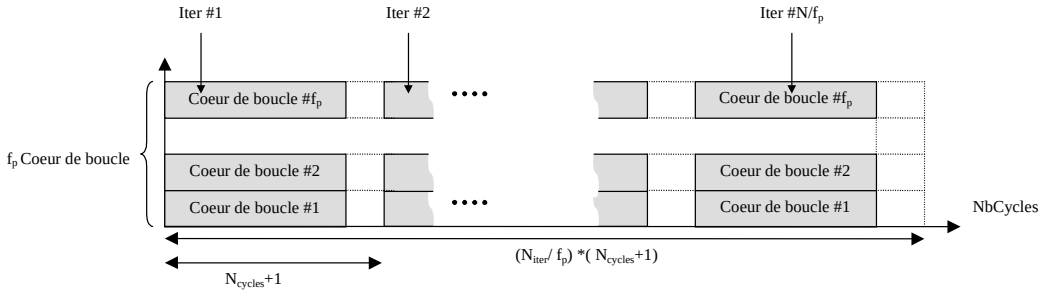


FIG. 4.12 – Ordonnancement des boucles : déroulage

4.12 illustre ce cas : les N_{iter} itérations de la boucle sont exécutées sur les f_p instances du coeur de boucle, le nombre de cycles nécessaires pour l'exécution vaut alors :

$$N'_{cycles} = (N_{iter}/f_p) * (N_{cycles} + 1)$$

Le nombre d'états est le même que celui du coeur plus un état pour le saut à l'état initial :

$$N'_{états}(N'_{cycles}) = N_{états}(N_{cycles}) + 1$$

À partir du nombre de ressources de type k nécessaires à l'exécution du coeur et f_p le facteur de parallélisme, on déduit le nombre de ressources nécessaires à l'exécution de la boucle pour cette configuration :

$$N'_{op_k}(N'_{cycles}) = N_{op_k}(N_{cycles}) * f_p$$

Le calcul des f_p résulte de la décomposition en facteurs premiers du nombre d'itérations. Par exemple, si le nombre d'itérations est de 12, les f_p correspondants sont 1, 2, 3, 4, 6, 12 que l'on peut retrouver à partir de la décomposition en facteurs premiers (12 = 2*2*3).

Une autre possibilité d'ordonnancement repose sur le recouvrement des exécutions de chaque itération (exécution *pipeline*). Elle exploite le parallélisme intra boucle en recouvrant l'exécution des itérations sucessives de manière pipeline (figure 4.13). Ce type d'ordonnancement ne peut être appliqué

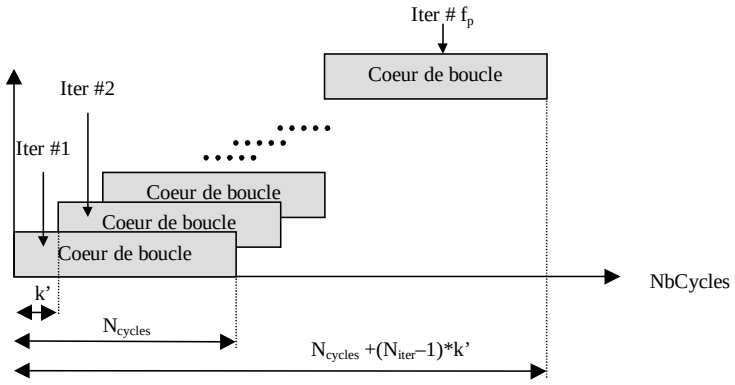


FIG. 4.13 – Ordonnancement des boucles : recouvrement (*pipeline*)

qu'à certaines conditions. L'exemple de la figure 4.14 permet d'illustrer ces restrictions. Si on suppose que l'addition et la multiplication s'exécutent en un cycle, l'ordonnancement proposé n'est réalisable que si l'on alloue deux additionneurs, car les opérations d'addition se recouvrent au temps de cycle trois. De manière générale, ce type d'ordonnancement n'est possible que si l'on alloue autant d'opérateurs qu'il y a d'opérations dans le coeur de boucle. Les caractéristiques du coeur ne sont plus issues de l'algorithme d'ordonnancement, comme c'est le cas pour tous les autres *DFGs* : le nombre de cycles nécessaires (N_{cycles}) est fournit par l'ordonnancement *ASAP* et correspond au chemin critique du graphe. Pour les opérations multicycles, le nombre de

cycles attribués pour chaque opération est étendu à celui de l'opérateur le plus lent (k' cycles par la suite). Le nombre d'états et le nombre de ressources sont calculés de la façon suivante :

$$N_{états}(N_{cycles}) = N_{cycles}$$

$$N_{op_k}(N_{cycles}) = N_{O_k}$$

où N_{O_k} représente le nombre d'opérations de type k dans le graphe. Dans ces

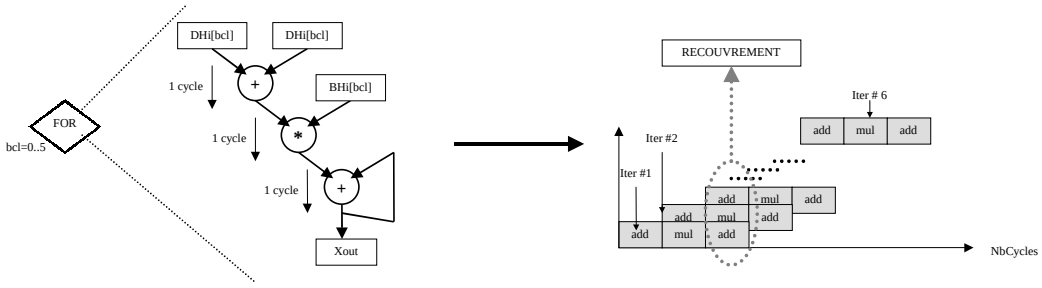


FIG. 4.14 – Exemple de recouvrement

conditions, le nombre de cycles nécessaires à l'exécution de la boucle vaut :

$$N'_{cycles} = N_{cycles} + (N_{iter} - 1) * k'$$

Le coeur de boucle est exécuté N_{iter} fois, mais l'exécution de la deuxième itération commence k' cycles après la première, d'où le nombre d'états :

$$N'_{états}(N'_{cycles}) = N_{états}(N_{cycles}) + (N_{iter} - 1) * k'$$

Le nombre d'opérateurs pour la boucle reste quand à lui le même que pour le coeur :

$$N'_{op_k}(N'_{cycles}) = N_{op_k}(N_{cycles})$$

Il n'y a qu'une seule solution architecturale définie dans ce cas, comme dans le cas de la séquentialisation des itérations.

Il existe une quatrième possibilité basée sur la combinaison des deux précédentes, à savoir le recouvrement de plusieurs déroulages de la boucle. Elle consiste à la fois à paralléliser l'exécution de f_p itérations et à pipeliner les N_{iter}/f_p itérations restantes (figure 4.15). Le coeur de la boucle est estimé

comme dans le cas du pipeline présenté précédemment. La combinaison des résultats dépend du facteur de parallélisme (nombre d'instances de boucles mises en parallèle). L'estimation est réalisée de la façon suivante :

$$N'_{cycles} = N_{cycles} + (N_{iter}/f_p - 1) * k'$$

$$N'_{états}(N'_{cycles}) = N_{états}(N_{cycles}) + (N_{iter}/f_p - 1) * k'$$

$$N'_{op_k}(N'_{cycles}) = N_{op_k}(N_{cycles}) * f_p$$

On obtient alors autant de solutions architecturales qu'il y a de facteurs de parallélisme f_p.

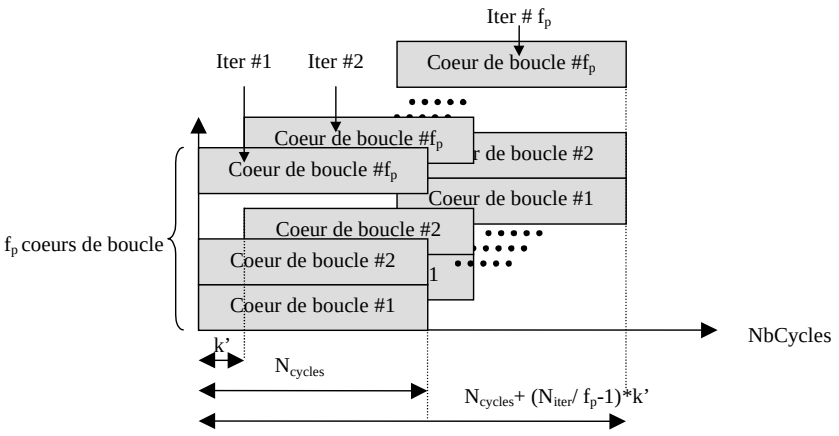


FIG. 4.15 – Ordonnancement des boucles : déroulage et recouvrement

Cette dernière solution d'ordonnancement est la plus performante mais n'est pas toujours applicable. La figure 4.16 montre un exemple de *DFG* qui comporte des dépendances inter-itérations et l'ordonnancement correspondant pour trois unités fonctionnelles allouées. La dépendance entre les noeuds R et E par exemple implique que l'exécution d'une deuxième itération ne peut se faire avant l'état S_3. Il n'est pas possible dans ce cas d'appliquer les ordonnancements de type pipeline et déroulage tels qu'ils ont été définis précédemment.

À partir des quatre possibilités d'ordonnancement présentées (séquentiel, déroulage partiel, pipeline, déroulage et recouvrement), on peut estimer tous les types de boucles. Le choix de l'algorithme d'estimation à appliquer dépend du déterminisme de l'exécution et de l'existence ou non de dépendances

inter-itérations. Une boucle dont le nombre d'itérations est inconnu ou dont le corps comporte des structures de contrôles non déterministes (structures conditionnelles ou autre boucle non déterministe) est dite non déterministe. Le seul ordonnancement possible dans ce cas est la séquentialisation des ité-

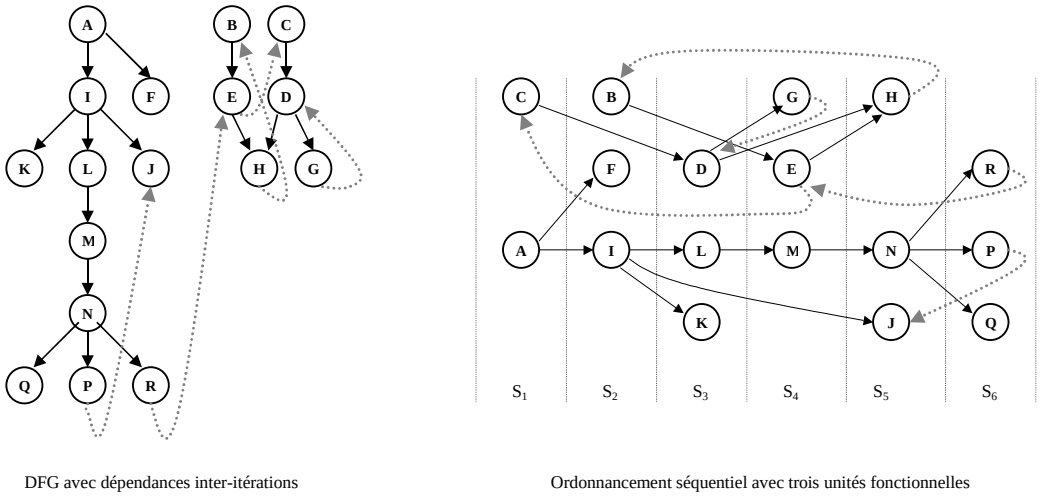


FIG. 4.16 – Exemple de dépendances inter-itérations

rations. Pour les boucles déterministes, on utilise l'algorithme d'ordonnancement par déroulage et recouvrement (le plus performant), sauf dans le cas où il existe des dépendances inter-itérations. L'ordonnancement appliqué est alors celui de la séquentialisation des itérations et il n'y a pas d'exploration du parallélisme dans ce cas.

4.5.3 Exécution séquentielle de deux graphes

L'estimation de deux graphes qui s'exécutent de façon séquentielle est obtenue à partir des résultats d'estimation de ces deux graphes en faisant l'hypothèse du maximum de réutilisation des ressources. Le nombre de ressources de type k est estimé en prenant le maximum du nombre de ressources de type k pour chaque graphe. Les temps d'exécution et le nombre d'états s'ajoutent :

$$N'_{cycles} = N_{cycles_1} + N_{cycles_2}$$

$$N'_{états}(N'_{cycles}) = N_{états_1}(N_{cycles_1}) + N_{états_2}(N_{cycles_2})$$

$$N'_{op_k}(N'_{cycles}) = MAX(N_{op_{k_1}}(N_{cycles_1}), N_{op_{k_2}}(N_{cycles_2}))$$

La figure 4.17 illustre l'estimation de l'exécution séquentielle de deux graphes sur un exemple tiré de la transformée en ondelettes 2D. Il s'agit

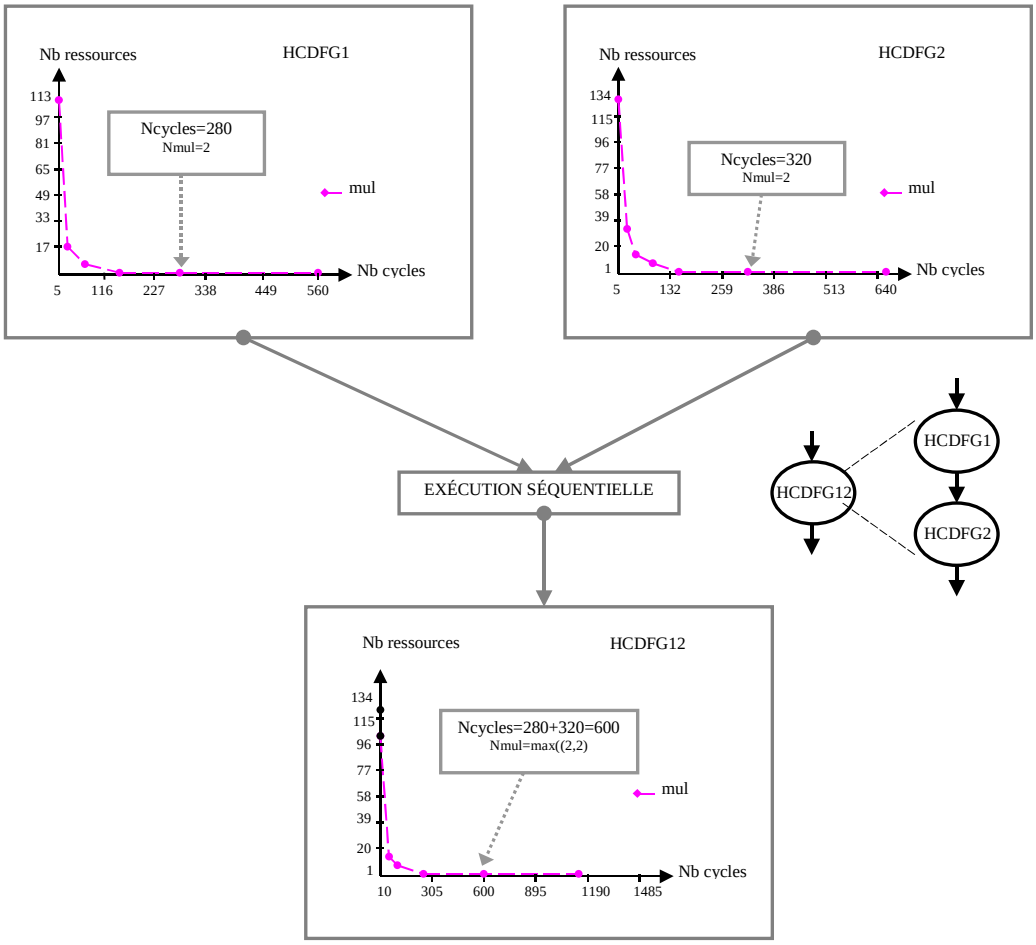


FIG. 4.17 – Estimation de l'exécution séquentielle de deux graphes

de deux opérations de filtrage successives décrites chacune par deux boucles imbriquées. Les résultats de l'estimation de la première et de la deuxième boucle correspondent respectivement au HCDFG1 et HCDFG2 et ceux de la combinaison, au HCDFG12 (seuls les résultats d'estimation du nombre de multiplieurs sont affichés, pour plus de clarté). À chaque solution estimée pour le HCDFG12 correspond une solution d'estimation de chaque graphe. Par exemple, la solution correspondant à la contrainte de temps de 600 cycles

résulte de la combinaison des solutions $N_{cycles_1} = 280$ et $N_{cycles_2} = 320$. Le nombre de multiplieurs nécessaires est le maximum du nombre de multiplieurs pour N_{cycles_1} et N_{cycles_2} soit $MAX(2,2) = 2$. Le nombre de ressources pour les autres types d'opérations, le nombre de lectures, écritures RAM et lectures ROM sont obtenus de la même manière. Toutes les combinaisons de points 2 à 2 sont effectuées.

Certaines simplifications peuvent intervenir à ce niveau. Considérons par exemple la solution $N_{cycles_1} = 280$ et $N_{cycles_2} = 640$. La solution correspondante pour le graphe HCDFG12 a une contrainte de temps de 280+640 = 920 cycles et le nombre de ressources est de 2 (MAX(2,1)) pour chaque type. Si on compare cette solution à la précédente (600 cycles, 2 ressources de chaque type), on constate que les deux solutions occupent la même surface. La deuxième solution peut donc être écartée puisqu'elle est moins performante.

4.5.4 Exécution concurrente de deux graphes

Etant donnés deux graphes dont on connait les résultats d'estimation, l'estimation de l'exécution concurrente de ces deux graphes est réalisée comme suit :

$$N'_{cycles} = MAX(N_{cycles_1}, N_{cycles_2})$$

$$N'_{états}(N'_{cycles}) = N_{états_1}(N_{cycles_1}) + N_{états_2}(N_{cycles_2})$$

$$N'_{op_k}(N'_{cycles}) = N_{op_{k_1}}(N_{cycles_1}) + N_{op_{k_2}}(N_{cycles_2})$$

Sur le plan structurel, cela revient à instancier les unités de traitement des deux fonctions décrites par les graphes et implique l'utilisation de machines d'états concurrentes. Le temps d'exécution est celui de la fonction la plus lente. La réutilisation n'est pas prise en compte dans ce cas (le nombre d'opérateurs est estimé en faisant une somme), car elle nécessite la mémorisation des disponibilités de chaque unité fonctionelle, ce qui conduirait à une complexité trop importante .

La figure 4.18 illustre un exemple de combinaison de deux graphes selon une exécution concurrente (dans le cas du multipieur). La combinaison des solutions $N_{cycles_1} = 4$ et $N_{cycles_2} = 9$ conduit à la solution $N'_{cycles} = MAX(4,9) = 9$. Notons qu'il existe en réalité deux solutions pour cette

contrainte de temps. En effet, la combinaisons des solutions ($N_{cycles_1} = 3$ et $N_{cycles_2} = 9$) aboutit à une solution différente ($N_{mul} = 4$ et $N_{add} = 4$ au lieu de $N_{mul} = 5$ et $N_{add} = 5$) qui n'est pas représentée par souci de clarté. La

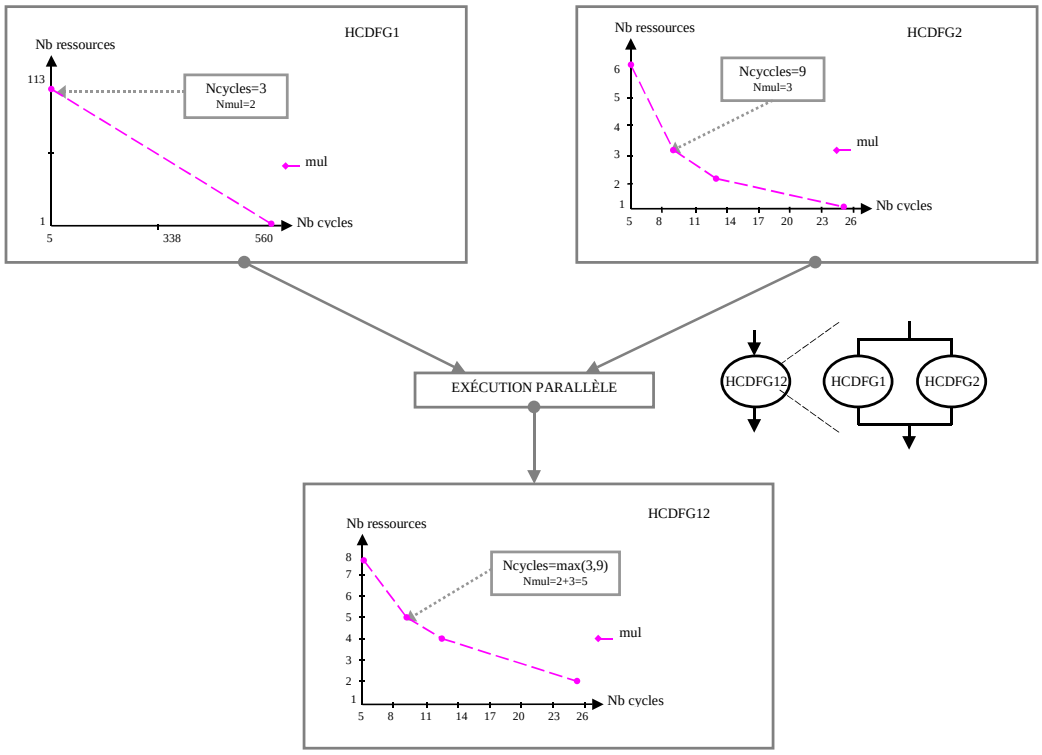


FIG. 4.18 – Estimation de l'exécution concurrente de deux graphes

sélection de la solution est effectuée à l'étape de projection technologique en ne retenant que la solution la moins coûteuse en surface.

4.6 Caractérisation structurelle globale

À partir des quatre types de combinaison possibles (conditionnelles, itératives, séquentielles et concurrentes) et des résultats d'estimation des *DFGs*, on peut obtenir l'estimation globale de tout le système. L'application des règles de combinaison est liée à la hiérarchie du système. Nous allons maintenant expliquer le principe de parcours du graphe et d'application des combinaisons.

4.6.1 Exploration de la hiérarchie

Le parcours de la hiérarchie se base sur une approche récursive ascendante. On commence par estimer les graphes feuilles constitués uniquement de *DFGs* à l'aide d'une technique d'ordonnancement (chapitre 4.4), puis les niveaux de hiérarchie supérieurs sont progressivement pris en compte. À chaque fois qu'un noeud hiérarchique est estimé, une structure particulière appelée *noeud résultat* qui contient ses résultats d'estimation structurelle lui est associée. L'algorithme de création des noeuds résultats (algorithme 4.2.) parcourt tous les noeuds hiérarchiques de façon récursive.

```
Explorer(graph Gcourant)
DÉBUT
    NbComposite = 0
    POUR CHAQUE noeud de Gcourant FAIRE
        SI le noeud courant est un composite ALORS
            NbComposite = NbComposite + 1
            SI le composite courant n'est pas un Noeud Résultat ALORS
                Gtemp = le sous-graphe du composite
                Explorer(Gtemp)
            FIN SI
        FIN SI
    FIN POUR
    TypePère = le type du père de Gcourant
    SI TypePère = IF ALORS
        Appliquer combinaisonIF sur Gcourant
    SINON SI TypePère = FOR ALORS
        Appliquer combinaisonFOR sur Gcourant
    SINON SI NbComposite > 0 ALORS
        Appliquer combinaisonsÉlémentaires sur Gcourant
    FIN SI
FIN
```

ALG. 4.2 - Algorithme de parcours de la hiérarchie

Si un noeud composite (i.e. hiérarchique) n'est composé que de noeuds résultats, on peut alors réaliser l'estimation du noeud composite à partir des algorithmes de combinaison définis précédemment. La condition d'arrêt de parcours du graphe est validée lorsque chaque noeud composite possède un noeud résultat. Il existe alors un résultat d'estimation pour chaque noeud hiérarchique, y compris celui qui contient toute l'application.

Plusieurs cas de figure peuvent se présenter lorsqu'on traite un noeud composite pour lequel tous les sous-graphes on été estimés :

- Le noeud composite est une structure de contrôle (itérative ou conditionnelle) : dans ce cas, on applique l'algorithme de combinaison adéquat (voire 4.5.1 et 4.5.2).
- Le noeud composite est constitué d'un ensemble de sous graphes qui disposent tous de noeuds résultats (figure 4.19). Il faut alors combiner les résultats d'estimation en fonction des dépendances d'exécution séquentielles / parallèles (*combinaisonsÉlémentaires* dans l'algorithme). Celles-ci se basent sur les algorithmes de combinaison deux à deux définis en 4.5.3 et 4.5.4. L'application des algorithmes de combinaison dans ce cas est expliqué dans le paragraphe suivant.

L'analyse récursive ascendante et les combinaisons de noeuds résultats permettent de traiter tous les cas de figure possibles quand à la hiérarchie. Au final, on aboutit à l'estimation de toute l'application et on dispose en outre des résultats d'estimation structurelle de chaque noeud composite.

4.6.2 Cas des dépendances d'exécution

Dans le cas où un noeud composite est constitué de sous-graphes déjà estimés (*DFGs* ordonnancés ou noeuds composites dont les résultats sont issus de combinaisons précédentes), les noeuds résultats doivent être combinés en fonction des dépendances d'exécution entre sous-graphes. Celles-ci se basent sur des combinaisons deux à deux. Dans l'exemple de la figure 4.19, les noeuds HCDFG2 et HCDFG3 sont combinés de façon parallèle pour former le noeud résultat HCDFG23, qui est ensuite combiné avec le noeud HCDFG4 pour former le noeud HCDFG234. À l'étape suivante, ce sont les noeuds HCDFG234, HCDFG6 et HCDFG1, HCDFG5 qui sont traités. Une analyse des dépendances entre graphes est donc nécessaire afin de déterminer le bon enchaînement des combinaisons.

L'algorithme 4.3 présente le principe d'application des combinaisons. L'idée consiste à combiner tous les prédécesseurs immédiats d'un noeud courant (noeuds pour lesquels il existe un arc aboutissant au noeud courant) de façon parallèle, puis de combiner séquentiellement le résultat avec le noeud courant. Certaines précautions doivent être prises lorsqu'on construit la liste des

noeuds courants. Dans le cas de la figure 4.19.c par exemple, les noeuds successeurs des noeuds HCDFG5, HCDFG6 sont les noeuds HCDFG8 et HCDFG7. Or il existe une dépendance entre ces deux noeuds qui introduit une erreur

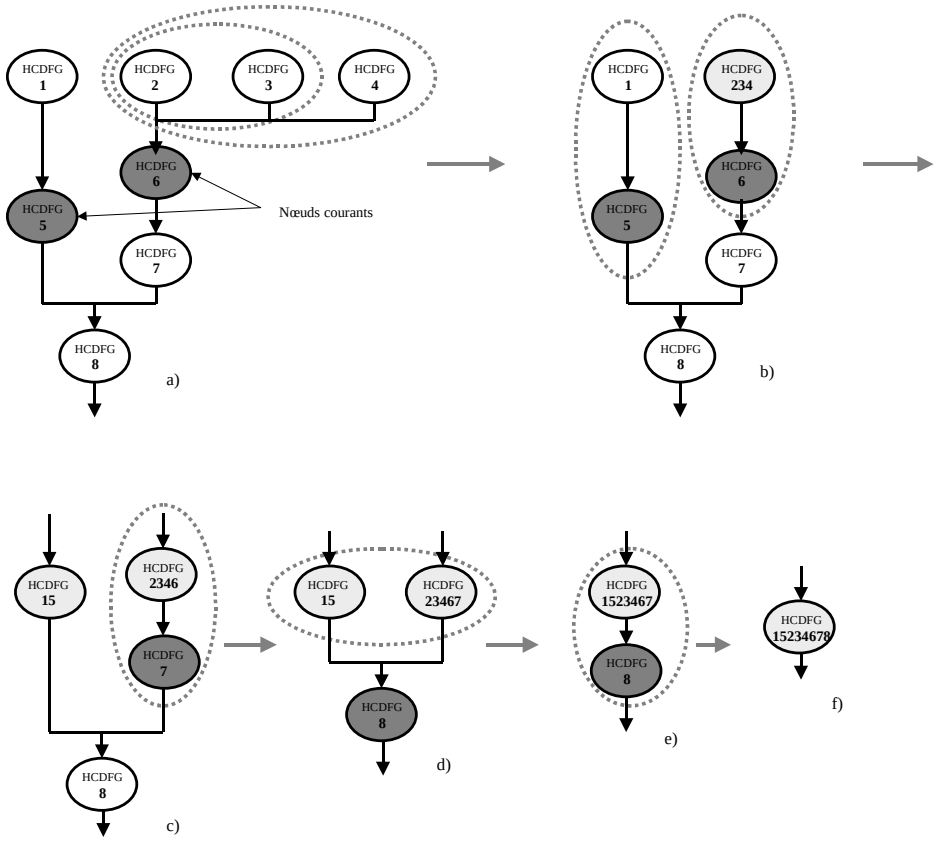


FIG. 4.19 – Combinaison des résultats

car il faut d'abord effectuer la combinaison séquentielle de HCDFG2346 et de HCDFG7. Le problème est résolu grâce à la fonction *Successeurs* qui renvoie la liste des noeuds constituant les successeurs immédiats de la liste des noeuds courants (noeuds pour lesquels il existe un arc issu de chaque noeud courant). S'il existe un arc reliant deux noeuds de la liste des successeurs, le noeud auquel aboutit cet arc est supprimé de la liste des successeurs.

```
combinaisonsÉlémentaires(graph Gcourant)
DÉBUT
    Initialiser la liste des noeuds courants (LNC) avec les noeuds racines
    TANT QUE LNC ≠ ∅
        LNC = Successeurs(LNC)
        POUR CHAQUE noeud ∈ LNC  FAIRE
            combinaisonParallèle des prédécesseurs 2 à 2
            combinaisonSéquentielle du noeud courant et du noeud précédent
        FIN POUR
    FIN TANT QUE
FIN
```

ALG. 4.3 - Algorithme de combinaisons

4.6.3 Analyse de la réutilisation

Une fois réalisées toutes les combinaisons, on dispose des résultats d'estimation structurelle de toute l'application. Avant de passer à l'étape de projection technologique qui permet de connaître l'occupation du FPGA et les performances du système, on analyse l'effet du partage des unités fonctionnelles.

La réutilisation introduit un coût en multiplexeurs nécessaire pour l'aiguillage des données vers l'unité fonctionnelle chargée d'effectuer le traitement. L'estimation du nombre de multiplexeurs est effectuée à partir du calcul de l'utilisation moyenne des opérateurs [3] : si N_{O_k} représente le nombre d'opérations de type k à réaliser dans le graphe et N_{op_k}, le nombre d'opérateurs alloués, l'utilisation moyenne des opérateurs de type k vaut $U_{op_k} = \lceil \frac{N_{O_k}}{N_{op_k}} \rceil$. En supposant que les opérateurs disposent de deux opérandes d'entrées, le nombre de multiplexeurs nécessaires vaut $N_{mux} = 2 * N_{op_k}$. Le nombre d'entrées des multiplexeurs est égal au nombre d'opérations que doit réaliser chaque unité fonctionnelle, soit en moyenne U_{op_k}. La figure 4.20 illustre le cas où l'on doit effectuer six opérations à l'aide de deux opérateurs. Le nombre de multiplexeurs nécessaire vaut 4 (1 multiplexeur par opérande d'entrée). Chaque opérateur est utilisé en moyenne trois fois. Il faut donc quatre multiplexeurs à trois entrées dans ce cas.

Cette façon de procéder peut introduire une surestimation importante dans certain cas. En effet, l'utilisation d'un multiplexeur par opérande d'entrée représente le pire cas possible. En particulier, leur emploi n'est pas néces-

saire lorsque les données proviennent de la mémoire. Aussi, lors du décompte du nombre d'opérations de type k, on ne prend pas en compte celles dont les opérandes correspondent à des variables multi-dimensionnelles ou à des constantes puisque ces données sont supposées être contenues en mémoire. C'est aussi la raison pour laquelle la ré-utilisation intra-boucle n'est pas prise en compte. Dans ce cas, l'hypothèse est faite que les opérandes sont des tableaux ou constantes contenus en mémoire et accédés par une fonction du ou des indice(s) de boucle(s), ce qui ne nécessite pas l'emploi de multiplexeurs. Au final, l'analyse de la réutilisation n'est appliquée que dans le cas des *DFGs*, des structures conditionnelles et des graphes pour lesquels on effectue une combinaison séquentielle.

L'intégration des multiplexeurs dans les composants FPGA peut se faire de deux façons : soit ils utilisent les cellules logiques du composant, soit ils utilisent des buffers trois états (figure 4.20) si le composant cible en dispose, comme c'est le cas par exemple pour les composants de la famille Virtex. Dans ce dernier cas, l'utilisation de ces ressources spécifiques permet de réaliser une économie en cellules logiques. Dans les deux cas, le principe d'estimation du nombre de multiplexeurs est le même. Seule change la façon de calculer l'occupation des ressources. Dans l'exemple de la figure 4.20, le coût de la réutilisation est de 4 multiplexeurs à 3 entrées. Le coût pour

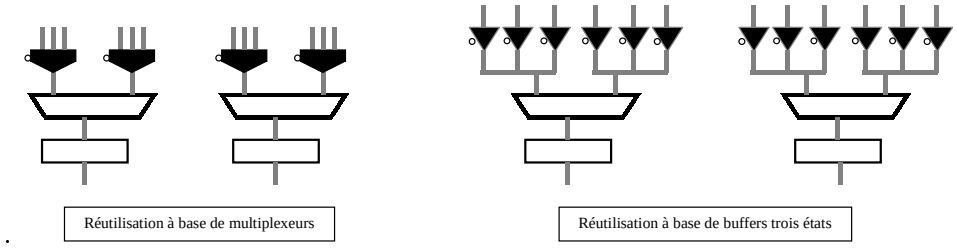


FIG. 4.20 – Réutilisation des unités fonctionnelles

une intégration à l'aide de buffers trois états vaut : $12 * L_{opérande}$ ($L_{opérande}$ représente la largeur des opérandes). Le calcul du coût pour une intégration dans les cellules logiques dépend lui de la caractérisation technologique des multiplexeurs (chapitre 5.2.2).

4.7 Conclusion

À l'issue de l'étape d'estimation structurelle, on obtient un ensemble de solutions architecturales définies chacune par une contrainte de temps, un nombre d'états et un nombre de ressources (unités fonctionnelles, accès RAM / ROM simultanés, multiplexeurs). L'estimation de ces valeurs s'effectue à partir de l'ordonnancement des blocs de base, puis on remonte progressivement les niveaux de hiérarchie par combinaison des résultats des graphes estimés. Les algorithmes de combinaison sont définis à partir de modèles d'ordonnancement de manière à garantir la faisabilité de la solution architecturale. Le principe de parcours de la spécification fournit en outre un ensemble d'informations locales qui permet une analyse fine de l'application à chaque niveau de la hiérarchie. Ces informations peuvent servir à guider la synthèse du système ou à repérer les points critiques succeptibles d'être optimisés.

À partir des résultats d'estimation structurelle de l'application toute entière, on peut obtenir une estimation de l'occupation et de la vitesse d'exécution du système sur le composant candidat. Ce calcul permet de fournir les valeurs physiques de compromis temps / surface des différentes solutions architecturales à partir desquelles le concepteur pourra faire un choix. Il est réalisé au cours de l'étape des estimations physiques (ou projection technologique), que nous allons maintenant présenter.

Chapitre 5

Les Estimations au Niveau Physique

L'exploration architecturale se déroule en deux étapes qui sont l'analyse structurelle du système (chapitre 4) et les estimations physiques. Si l'analyse structurelle permet de définir les solutions architecturales (en terme d'allocation, de nombre d'états, d'accès mémoire en fonction du nombre de cycles d'horloge), les estimations au niveau physique fournissent au concepteur les valeurs de performances et d'occupation des ressources d'un FPGA donné pour ces différentes solutions. Ce passage nécessite la connaissance des caractéristiques du composant candidat à l'implantation telles que les temps de traversée des opérateurs, le nombre de cellules logiques nécessaires, ... C'est le rôle du fichier technologique dont l'élaboration est expliquée en détail dans ce chapitre. Les techniques d'estimation des différentes unités de l'architecture (traitement, mémoire et contrôle), puis de l'application dans son ensemble sont ensuite présentées.

5.1 Le fichier technologique

Le fichier technologique permet de décrire le composant candidat à l'implantation. On peut le décomposer en trois parties en fonction des différentes caractéristiques du composant à prendre en compte :

- La caractérisation temps / surface des ressources.
- La caractérisation temps / surface des mémoires.
- La caractérisation de l'architecture du FPGA.

La caractérisation des ressources se présente sous la forme d'une bibliothèque qui contient les caractéristiques des unités fonctionnelles telles que leurs temps de traversée, le nombre et le type de cellules (logiques ou dédiées) utilisées et les opérations qu'elles réalisent. Il s'agit ici des opérateurs de base tels que les opérateurs arithmétiques, logiques et les ressources de connexion (registres, multiplexeurs). Les ressources de mémorisation sont caractérisées en temps et en surface dans un deuxième temps. Puis, des informations concernant l'architecture du composant comme le nombre de cellules logiques disponibles par exemple, sont ensuite décrites. Nous allons tout d'abord détailler la constitution de la librairie des opérateurs de base, puis celle des mémoires et enfin nous étudierons les paramètres architecturaux du composant à prendre en compte.

5.1.1 La bibliothèque de ressources

La bibliothèque de ressources contient les caractéristiques des unités fonctionnelles de base telles que leurs temps de traversée et les ressources du FPGA nécessaires pour leur implantation. La caractérisation des unités fonctionnelles pour un composant donné est effectuée à l'aide de l'outil de synthèse logique (*Foundation* pour Xilinx et *Quartus* pour Altera). La synthèse s'effectue à partir des descriptions que fournissent les fabricants afin de garantir une implantation efficace et les meilleures performances. Leur caractérisation consiste à relever les temps de traversée et les ressources du FPGA utilisées pour différents formats de données (figure 5.1).

Étant donné le nombre d'opérateurs et de formats possibles pour les opérandes, des simplifications sont apportées afin de limiter la complexité de

la caractérisation. Les opérandes sont limités à quatre formats (8, 16, 32 et 64 bits). Dans le cas de notre outil, cette limitation se justifie aussi par la spécification en C dont est issue le graphe de représentation interne qui implique l'utilisation des types fournis par ce langage (int : 16 bits, long : 32 bits). Cette limitation peut être à l'origine d'une importante surestimation

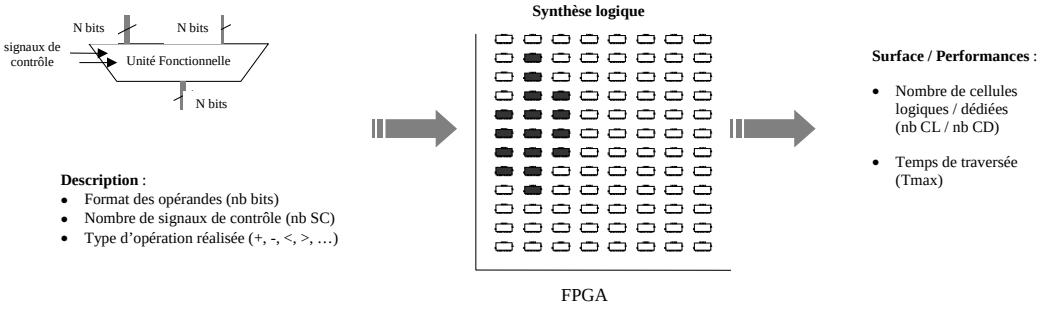


FIG. 5.1 – Principe de caractérisation des opérateurs

dans certains cas. Par exemple, si on souhaite effectuer une multiplication d'opérandes 16 bits avec un résultat tronqué sur 28 bits, on devra utiliser un multiplieur 32 bits. Cette approche permet toutefois d'obtenir une première estimation sans connaître avec précision les formats de données (ce qui est le cas au niveau comportemental). Un raffinement des résultats basé sur l'utilisation de formats de données particuliers peut être effectué en enrichissant la bibliothèque de ressources pour des formats de données spécifiques.

Des simplifications peuvent être apportées à l'élaboration de cette caractérisation en constatant une forte similitude entre certains opérateurs. Le tableau 5.1 présente un exemple de caractérisation des ressources pour des opérandes entiers sur 32 bits et pour deux types de technologies (Virtex et Apex). Ces résultats montrent qu'une économie du nombre de caractérisations peut être réalisée en considérant que certaines unités fonctionnelles possèdent des caractéristiques très proches. C'est le cas des opérateurs logiques par exemple car leur implémentation repose sur des tables mémorisées dans des LUTs. Ainsi, les opérateurs *and*, *or*, *nand*, *nor*, *xor* possèdent les mêmes caractéristiques et sont regroupés sous le terme *lgc* dans la suite. De la même façon, les opérations d'addition et de soustraction sont équivalentes en terme de surface et de performances car leur réalisation repose sur les mêmes principes. On considére donc par la suite que les unités fonctionnelles de type

additionneur, soustracteur, additionneur / soustracteur et additionneur / accumulateur ont les mêmes caractéristiques qu'un additionneur. Il en est de

	V400EPQ240-7		*EP20K200RC208-1*	
Opérateur	Nb CL	Latence (ns)	Nb CL	Latence (ns)
add	16	8.4	34	15.2
sub	16	8.4	35	15.2
mul	549	25.9	1617	34.6
lt	18	7.1	45	22.7
le	18	7.1	45	22.7
gt	18	7.1	45	22.7
ge	18	7.1	45	22.7
eq	17	8.3	18	10.9
neq	17	9.3	21	12.0
and	16	5.4	32	6.2
or	16	5.4	32	6.2
nand	16	5.4	32	6.2
nor	16	5.4	32	6.2
xor	16	5.4	32	6.2
not	16	4.5	32	5.4

TAB. 5.1 – Caractérisation des opérateurs de base

même pour les ressources de comparaison ($<$, $>$, $\leq$, $\geq$) qui sont regroupées sous le terme *cmp* par la suite (tableau 5.2).

Ces différentes remarques permettent de réduire la complexité de réalisation de la bibliothèque de ressources. Compte tenu de ce qui a été dit précédemment, cette caractérisation est effectuée pour des opérandes 8, 16, 32 et 64 bits. On y trouve donc les caractérisations des opérateurs d'additions (qui est la même que celle d'une soustraction), de multiplication, de comparaison ($<$, $>$, $\leq$, $\geq$), de test d'égalité et de non égalité, les opérateurs logiques (*and*, *or*, *nand*, *nor*, *xor*) et de négation. Un exemple de caractérisation est fourni au tableau 5.2. Les unités fonctionnelles qui y figurent correspondent au minimum nécessaire pour caractériser entièrement les opérateurs de base et les ressources de connexion (36 caractérisations au total). La liste des ressources présentée dans l'exemple du tableau 5.2 constitue un ensemble de base qui peut être enrichi. D'autres ressources comme les unités multifonctionnelles peuvent être rajoutées à la bibliothèque (UAL, additionneur / soustracteur, ...). Celles-ci doivent être caractérisées en temps et en surface de la même façon que les autres unités fonctionnelles.

Les ressources de connexion sont elles aussi caractérisées. Il s'agit des registres et des multiplexeurs. La caractérisation des registres en nombre de cellules logiques suit une loi linéaire liée au nombre d'éléments séquentiels dans une cellule logique (2 par *slice* pour un composant virtex et 1 par *logic element* pour un composant apex). En ce qui concerne les multiplexeurs, on

Virtex V400EPQ240-7					
Opérateur	Nb Bits	Latence (ns)	Nb Cell	Type Cell	Nb SC
add	8	4.9	4	slice	1
add	16	5.7	8	slice	1
add	32	8.4	16	slice	1
add	64	9.1	32	slice	1
mul	8	12.3	36	slice	1
mul	16	19.1	140	slice	1
mul	32	25.9	549	slice	1
mul	64	38.5	2135	slice	1
cmp	8	4.9	6	slice	1
cmp	16	5.8	10	slice	1
cmp	32	7.1	18	slice	1
cmp	64	8.7	34	slice	1
eq	8	5.1	4	slice	1
eq	16	6.3	9	slice	1
eq	32	8.3	17	slice	1
eq	64	11.1	34	slice	1
neq	8	4.8	4	slice	1
neq	16	7.3	9	slice	1
neq	32	9.3	17	slice	1
neq	64	13.3	34	slice	1
lgc	8	3.7	4	slice	1
lgc	16	4.5	8	slice	1
lgc	32	5.4	16	slice	1
lgc	64	7.6	32	slice	1
not	8	3.4	4	slice	1
not	16	3.9	8	slice	1
not	32	4.5	16	slice	1
not	64	9.0	32	slice	1
Ressources de connexion					
Ressource	8 bits	16 bits	32 bits	64 bits	Type cell
reg	4	8	16	32	slice
mux	–	–	–	–	3state

TAB. 5.2 – Exemple de caractérisation des ressources

distingue deux cas en fonction des ressources utilisées pour leur implantation : les multiplexeurs implantés dans les cellules logiques (MUX_{cl}) et les

multiplexeurs à base de buffers trois états (MUX_{3state}). Le tableau 5.3 présente les résultats de la synthèse de multiplexeurs sur les composants virtex (qui dispose de buffers trois états) et apex (qui n'en dispose pas). L'emploi

	V400EPQ240-7	*EP20K200RC208-1*
MUX_{3state}	Nb 3state	Nb 3state
2x1	16	–
3x1	24	–
4x1	32	–
8x1	64	–
MUX_{cl}	Nb slices	Nb logic elements
2x1	4	8
3x1	8	16
4x1	8	24
8x1	16	88

TAB. 5.3 – Caractérisation des mutiplexeurs (opérandes 8 bits)

des tristates est favorisé dans la mesure où ils permettent de réaliser l'économie des cellules logiques. Si le composant candidat permet l'utilisation de buffers trois états, il n'est pas nécessaire de caractériser les multiplexeurs. L'occupation des buffers dépend directement du nombre d'entrées et de la largeur des opérandes : $N_{tristate} = N_{entrées} * L_{opérandes}$.

En ce qui concerne les multiplexeurs "classiques", il faut pouvoir calculer leur coût en cellules logiques. La caractérisation peut être modélisée par une loi linéaire en fonction du nombre d'entrées, comme le montrent les résultats du tableau 5.3 : $N_{CL} = \alpha * (N - 1)$ où α représente la surface d'un multiplexeur 2x1 et N le nombre d'entrées. En réalité, la loi n'est pas strictement linéaire (un multiplexeur 8x1 aurait alors une surface de $8 * 7 = 56$ au lieu de 88). Une erreur croissante avec le nombre d'entrées est introduite. Elle résulte de la logique de sélection qui devient plus complexe à mesure que le nombre d'entrées augmente. L'erreur d'estimation est donc d'autant plus importante que le potentiel de réutilisation est grand. Toutefois, des optimisations sont souvent possibles au niveau de la réutilisation (aboutissant à la diminution du nombre d'entrées des multiplexeurs) et introduisent une nouvelle erreur qui compense la précédente. Le choix est fait ici de ne pas analyser plus finement la réutilisation ainsi que la modélisation des multiplexeurs afin de ne pas trop augmenter la complexité algorithmique.

La caractérisation en surface des unités fonctionnelles permet de calculer

la surface de l'unité de traitement. La connaissance des temps de traversée permet, elle, le calcul de la fréquence d'horloge et l'attribution d'un nombre de cycles aux noeuds du graphe pour l'étape d'ordonnancement des *DFGs*. Comme nous l'avons vu au chapitre 4.4, les noeuds mémoires sont aussi pris en compte pour cet ordonnancement. La détermination des temps d'accès aux mémoires est donc nécessaire afin de d'attribuer un nombre de cycles pour ce type de noeud. Cette caractérisation fait l'objet de la partie suivante.

5.1.2 Caractérisation des mémoires

L'intégration des mémoires sur les composants programmables peut se faire de deux façons : soit elles utilisent la mémoire de configuration des cellules logiques (mémoires "distribuées"), soit elles utilisent des ressources spécifiques. La première possibilité se limite au cas de petites mémoires car l'implantation se fait au détriment des ressources disponibles pour la programmation de fonctions logiques par l'utilisateur. Au regard des performances, cette solution convient bien pour l'intégration de mémoires de l'ordre du kilobyte, typiquement pour le stockage des coefficients d'un filtre. Dans le second cas, d'autres ressources que les cellules logiques sont utilisées. Cette particularité offerte par les nouvelles familles de composants doit être prise en compte pour l'estimation. Ces ressources permettent l'intégration de mémoires performantes jusqu'à des capacités de l'ordre de la centaine de kilobytes.

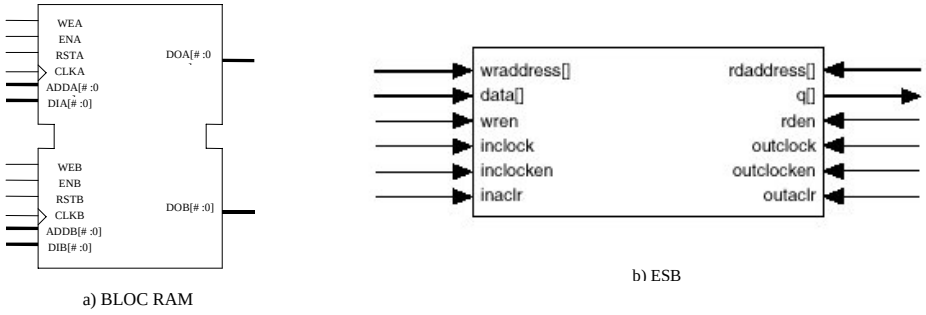


FIG. 5.2 – Mémoires dédiées : Virtex vs. Apex

Chez Xilinx, ces ressources dites "dédiées" se présentent sous la forme de modules mémoire RAM synchrone double port d'une capacité de 4096 bits

dénommés *block select RAM* (BRAM par la suite). Chaque BRAM possède deux ports distincts (A et B) configurables indépendamment (figure 5.2.a). Chez Altera, l'utilisation de ressources dédiées est apparue sur la famille FLEX10k avec les *EABs* (Embedded Array Blocks), et a été reprise sur la famille Apex sous le nom *ESB* (Embedded System Blocks). Elles permettent la synthèse efficace de mémoires (ROM, RAM, CAM) et de termes produits. Chaque ESB est constitué d'un bloc de 2048 bits de mémoire RAM (figure 5.2.b) utilisable de manière synchrone et asynchrone, en simple ou double port et dispose de facilités de combinaison pour réaliser des mémoires plus larges et plus profondes.

La surface des mémoires distribuées peut être caractérisée par le nombre de bits par cellule. Par exemple, pour la famille Virtex, une cellule logique (*slice*) permet l'intégration d'une mémoire RAM double port 16x1 bits. La caractérisation en surface vaut 16 bits / slice dans ce cas. En ce qui concerne les mémoires dédiées, on peut se servir du même principe. Par exemple, un BRAM permet l'intégration d'une RAM double port de 4096 bits au maximum. Il est alors facile de déduire le coût en cellules logiques : par exemple, une RAM double port de 16 mots de 16 bits nécessite l'utilisation de 16 *slices* pour une mémoire distribuée, et de 256 bits d'un BRAM pour une mémoire dédiée.

La caractérisation temporelle est rendue difficile par le grand nombre de configurations possibles (taille / largeur des mots). Le tableau 5.4 présente les temps d'accès de différentes mémoires et pour les deux technologies Virtex et Apex. Dans le cas de l'Apex, les valeurs mesurées restent du même ordre de

Synthèse RAM double-port	Taille mémoire (nb bits)	Occupation ress. dédiées	Temps d'accès
EP20K200RC208-1	65536	61.4%	< 4 ns
	98304	92.3%	< 4 ns
V400EPQ240-7	4096	2.5%	4.7 ns
	65536	50%	8.4 ns
	163840	100%	11.8 ns

TAB. 5.4 – Performances des mémoires dédiées

grandeur, même dans le cas des grosses mémoires. Ceci s'explique par le fait que ce composant dispose de ressources facilitant la mise en cascade des blocs mémoire. Ce n'est pas le cas dans le Virtex, ce qui explique les différences de

temps d'acès. Toutefois, les performances restent très correctes (de l'ordre de la dizaine de nanosecondes pour les plus grosses mémoires). Le choix est fait de caractériser les temps d'accès aux mémoires dans le pire cas (mémoire la plus lente, qui correspond à la plus grosse mémoire intégrable dans les cellules dédiées). Ceci est motivé par le fait qu'on a besoin d'associer un nombre de cycles aux noeuds mémoire du graphe afin d'effectuer l'ordonnancement des *DFGs*. Si ce nombre de cycles est sous-estimé, l'ordonnancement calculé est faux ce qui rend plus difficile l'intégration du système après estimation.

La même approche est utilisée pour la caractérisation des mémoires distribuées. Le tableau 5.5 présente les résultats de la synthèse de mémoires ROM pour plusieurs tailles. Les fabricants de FPGAs préconisent l'emploi

Synthèse ROM	Taille mémoire (nb bits)	Nb cellules logiques	Temps d'accès
EP20K200RC208-1	2048	450 le	11.8 ns
	1024	198 le	11.2 ns
	512	64 le	6.8 ns
V400EPQ240-7	2048	128 sl	13.8 ns
	1024	64 sl	13.4 ns
	512	32 sl	7.2 ns

TAB. 5.5 – Performances des mémoires distribuées

des mémoires distribuées pour des tailles de l'ordre du kilobyte. Les résultats présentés au tableau 5.5 montrent que cette taille mémoire permet de rester dans l'ordre de grandeur des performances des mémoires dédiées. Pour des tailles supérieures, nous considérons que ces mémoires ne sont pas assez performantes pour êre utilisées. D'autre part, n'oublions pas que l'implantation de ces mémoires se fait au détriment des cellules logiques permettant la définition de fonctions par l'utilisateur. Il vaut mieux alors se rabattre sur le choix d'un composant qui dispose de ressources de mémorisation plus efficaces. C'est la raison pour laquelle les temps d'accès des mémoires distribuées sont caractérisées pour des mémoires de 1024 bits.

La connaissance du nombre de ressources de mémorisation disponibles permet de guider le choix d'un composant d'implantation : si la taille mémoire totale estimée (paragraphe 4.2) est de l'ordre du kilobyte, alors on peut se contenter d'utiliser un composant classique (XC4000, Flex8k). Dans le cas où elle est supérieure, on utilisera un composant disposant de res-

sources de mémorisation dédiées (Virtex, Apex ou Flex10k). Un exemple de caractérisation des ressources de mémorisation est donné au tableau 5.6. On y trouve deux caractérisations possibles pour la ROM dans le cas de l'Apex qui permet l'intégration dans les *logic elements* (ROM distribuée) et dans les *ESB* (ROM dédiée). Le choix du type de mémoire se fait à l'étape d'al-

Virtex V400EPQ240-7			
Mémoires distribuées	Nb Bits/cell	Tps d'accès	Type cell
ROM	32	13.4 ns	slice
Mémoires dédiées	Nb Bits/cell	Tps d'accès	Type cell
RAM DP	4096	11.8 ns	bram
Apex EP20K200RC208-1			
Mémoires distribuées	Nb Bits/cell	Tps d'accès	Type cell
ROM	16	11.2 ns	lgc elt
Mémoires dédiées	Nb Bits/cell	Tps d'accès	Type cell
ROM	2048	4 ns	esb
RAM DP	2048	4 ns	esb

TAB. 5.6 – Exemple de caractérisation des mémoires

location. Pour chaque mémoire décrite dans le fichier technologique, il faut préciser le type de ressource qu'elle utilise (cellules logiques ou dédiées). Afin de pouvoir calculer le taux d'occupation correspondant, il est nécessaire de connaître le nombre de cellules de chaque type disponibles dans le composant cible, c'est l'objet de la caractérisation architecturale du composant, décrite ci-après.

5.1.3 Caractérisation de l'architecture du composant

La caractérisation architecturale d'un composant porte sur le nombre et le type de ressources dont il dispose. La figure 5.3 présente une vue simplifiée de l'architecture faisant apparaître les différentes ressources prises en compte quelque soit le type de composant. Il s'agit des cellules logiques, (slice, logic element), des cellules dédiées (esb, bram), des buffers trois états et des entrées / sorties. Ces ressources sont caractérisées en nombre de façon à pouvoir vérifier que l'on ne dépasse pas les capacités d'intégration du composant cible. Deux exemples de caractérisation (Virtex et Apex) sont fournis au tableau 5.7. Le champs *Type cell* permet de faire le lien avec le champs *Type cell* de la caractérisation de bibliothèque de ressources et des mémoires. On

	Virtex V400EPQ240-7		*Apex EP20K200RC208-1*	
ressources FPGA	Nombre	Type cell	Nombre	Type cell
Cellules logiques	4000	slice	8320	lgc elt
Entrées / sorties	158	iob	208	ioe
Buffers trois états	4960	3state	0	–
Cellules dédiées	40	bram	52	esb

TAB. 5.7 – Exemple de caractérisation d'une architecture

peut ainsi caractériser les unités fonctionnelles, les ressources de connexion et les mémoires pour différents types d'implantation possibles (cellules logiques ou dédiées, buffers trois états pour les multiplexeurs, ...).

Avec le fichier technologique, on dispose d'une caractérisation complète du composant candidat à l'implantation. Les données qu'il contient permettent

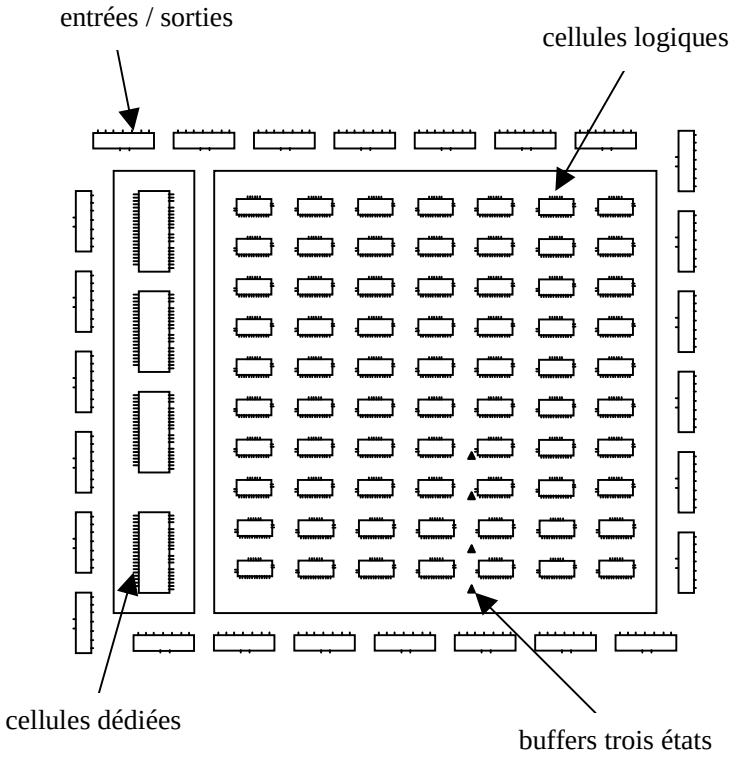


FIG. 5.3 – Caractérisation de l'architecture du composant

de calculer précisément la surface occupée par les différentes unités du système et la vitesse d'exécution. Ce calcul est détaillé dans la partie suivante.

5.2 La projection technologique

La projection technologique permet d'établir les valeurs physiques d'occupation des ressources du FPGA en fonction de la vitesse d'exécution des différentes solutions. La figure 5.4 montre le principe de calcul du coût et des

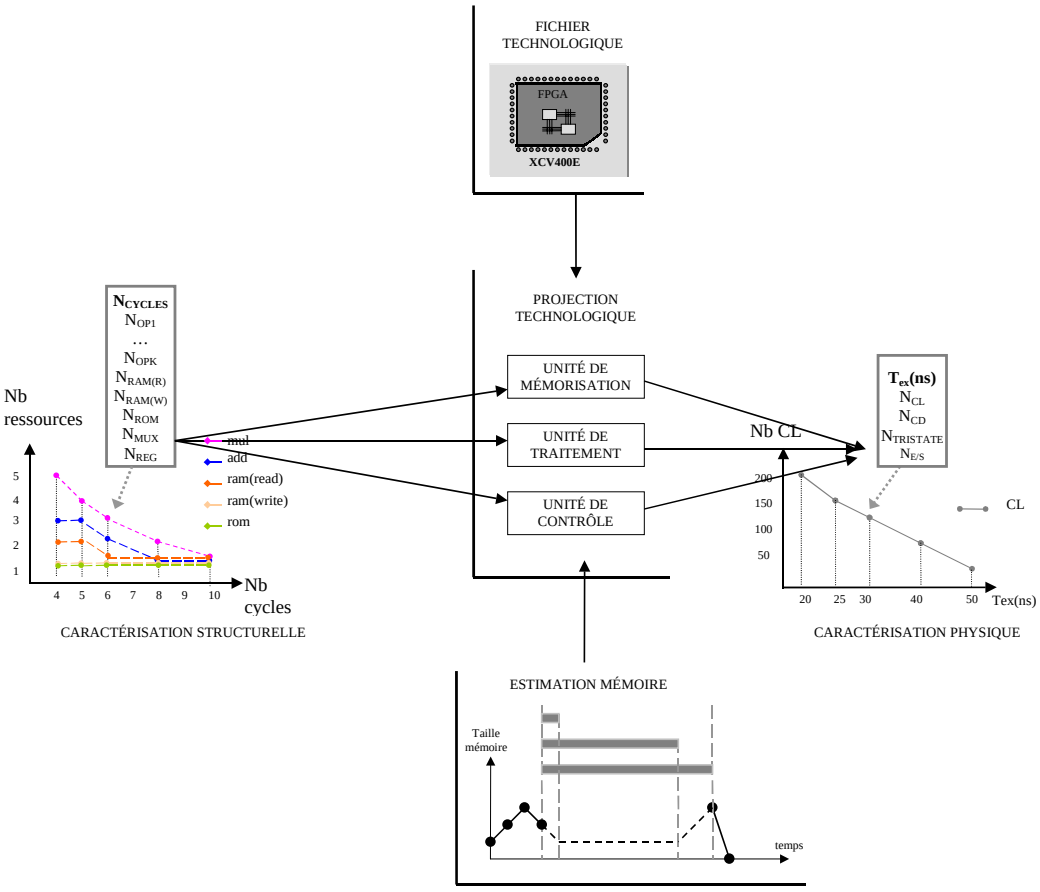


FIG. 5.4 – Principe de la projection technologique

performances. Celui-ci se base sur les résultats de l'estimation structurelle du plus haut niveau de la hiérarchie (qui correspond à l'application toute entière). Pour une solution architecturale donnée, il correspond une contrainte de temps, un nombre d'unités fonctionnelles de chaque type, un nombre d'accès mémoire (RAM / ROM) simultanés, ainsi qu'un coût en multiplexeurs. Ceci induit un coût en surface pour les différentes unités du système (mémoire, traitement, contrôle) qui est calculé à partir des caractéristiques du

composant décrites dans le fichier technologique et de l'estimation de la taille mémoire. Ce coût correspond à un taux d'occupation des ressources du FPGA (cellules logiques, cellules dédiées, entrées / sorties, buffers trois états) pour une contrainte de temps exprimée ici en unité de temps physique (μs, ns), et fournit la caractérisation physique de la solution en cours d'analyse. Le processus est ensuite ré-itéré pour chaque solution architecturale de la caractérisation structurelle.

Nous allons maintenant détailler le calcul du coût de chacune des unités, puis de l'architecture dans son ensemble. Dans la suite, nous ne représenterons plus le terme temporel dans un souci de clarté. Toutefois, les valeurs estimées dans la suite correspondent toutes à une contrainte de temps donnée.

5.2.1 Unité de mémorisation

L'étape d'analyse de l'unité de mémorisation permet de déterminer d'une part la surface occupée par les différentes mémoires, et d'autre part de déterminer les paramètres nécessaires pour l'estimation de l'unité de contrôle.

La distinction des accès en lecture / écriture au niveau des estimations structurelles implique l'utilisation de mémoires double port qui permettent la lecture et l'écriture simultanée de données. À partir du nombre d'accès simultanés, on déduit un nombre minimum de mémoires nécessaires. Par exemple, si les résultats des estimations structurelles font apparaître que pour une certaine contrainte de temps, 3 accès en lecture et 1 accès en écriture sont nécessaires, on pourra allouer trois mémoires double port.

Connaissant la taille mémoire totale nécessaire et le nombre de bits par cellule (logique ou dédiée) , on peut calculer la surface totale occupée par l'unité de mémoire. Ce calcul dépend du type de cellule utilisée pour l'implantation :

- Implantation dans les cellules logiques : on effectue une projection du nombre de cellules logiques nécessaires pour implanter la taille mémoire estimée.

$$N_{cl}^{ram} = \lceil (TM_{RAM} * L_{mots}^{ram} / N_{bits/cl}^{ram})$$

$$N_{cl}^{rom} = \lceil (TM_{ROM} * L_{mots}^{rom} / N_{bits/cl}^{rom})$$

  où TM_{RAM} et TM_{ROM} représentent les tailles mémoires de la RAM et de la ROM respectivement, $N_{bits/cl}^{ram}$ et $N_{bits/cl}^{rom}$ représentent le nombre de

bits par cellule logique, L^{ram}_{mots} et L^{rom}_{mots} représentent la largeur des mots.

– Implantation dans les cellules dédiées : dans ce cas, il faut tenir compte du fait qu'il n'est possible d'intégrer qu'une seule mémoire par cellule dédiée (par exemple, dans un Virtex, on ne peut intégrer qu'une mémoire double port dans un BRAM). Pour illustrer ce cas, supposons que le nombre d'accès simultanés vaut 4, et que la taille mémoire estimée nécessite l'utilisation de 3 cellules dédiées. Il faut alors allouer 4 cellules dédiées afin de rendre possible les 4 accès simultanés à la mémoire (on ne peut intégrer qu'une mémoire double port par cellule dédiée). Le nombre de mémoires est alors pris comme étant le maximum entre le nombre de cellules dédiées nécessaires pour la mémorisation de toutes les données, et le nombre d'accès simultanés (qui représente le nombre minimum de mémoires à allouer).

$$N^{ram}_{cd} = MAX[\lceil (TM_{RAM} * L^{ram}_{mots}/N^{ram}_{bits/cd}), N_{ram\_lec}, N_{ram\_ecr}]$$

$$N^{rom}_{cd} = MAX[\lceil (TM_{ROM} * L^{rom}_{mots}/N^{rom}_{bits/cd}), N_{rom}]$$

Ensuite, on calcule le coût en signaux de contrôle d'après le modèle d'architecture pour l'unité de mémoire (paragraphe 3.4.2) :

$$N^{ram}_{sc} = (2 * L^{ram}_{adr} + 1) * MAX(N_{ram\_lec}, N_{ram\_ecr})$$

$$N^{rom}_{sc} = L^{rom}_{adr} * N_{rom}$$

Les signaux de contrôle proviennent des bus d'adresse et des signaux de validation d'écriture (d'où le terme +1 pour chaque RAM). Le nombre de bus d'adresse est de deux pour la RAM et un pour la ROM. La largeur des bus d'adresse est calculée dans chaque cas en prenant le logarithme à base 2 du nombre de mots de chaque mémoire, soit :

$$L^{ram}_{adr} = \lceil log_2(TM_{RAM}/MAX(N_{ram\_lec}, N_{ram\_ecr}))$$

$$L^{rom}_{adr} = \lceil log_2(TM_{ROM}/N_{rom})$$

Le coût total en signaux de contrôle pour l'unité de mémoire vaut :

$$N^{um}_{sc} = N^{ram}_{sc} + N^{rom}_{sc}$$

Comme nous l'avons vu au chapitre 3.4.1, le modèle architectural de l'unité de traitement fait l'hypothèse de l'existence d'un registre par bus de

données pour la synchronisation et implique un surcoût qui n'est pas toujours négligeable. Le nombre de registres induit est déduit du nombre minimum de mémoires à allouer et du nombre de ports, soit :

$$N_{reg}^{ram} = 2 * MAX(N_{ram\_lec}, N_{ram\_ecr})$$

puisqu'il s'agit de mémoires double port et,

$$N_{reg}^{rom} = N_{rom}$$

pour la ROM. Le coût en surface et en signaux de contrôle de ces registres est pris en compte à l'étape de projection de l'unité de traitement.

5.2.2 Unité de traitement

La surface de l'unité de traitement est obtenue en effectuant une projection de la surface de chaque unité fonctionnelle utilisée (N_{op_k} est le nombre d'unités fonctionnelles de type k et S_{op_k} leur surface) :

$$S_{UT} = \sum_{op_k} N_{op_k} * S_{op_k}$$

où op_k comprend les opérateurs de base (arithmétiques, logiques, ...), mais aussi les multiplexeurs et les registres additionnels (associés aux bus de données et calculés précédemment). Comme dans le cas de l'unité de mémorisation, on a besoin de connaître le nombre de signaux de contrôle nécessaires pour le séquencement de l'unité de traitement. On distingue quatre types de signaux :

- Les signaux de validation d'horloge des registres en sortie des unités fonctionnelles : N_{sc}^{op}. Chaque unité fonctionnelle dispose d'un registre en sortie. Le nombre de signaux est donc égal au nombre d'unités fonctionnelles.
- Les signaux de sélection pour les unités multi-fonctionnelles : $N_{sc}^{multi\_op}$. Ils dépendent du nombre d'unités multi-fonctionnelles et du nombre de signaux de sélection nécessaires.
- Les signaux issus des registres associés aux ports de lecture / écriture des mémoires : $N_{sc}^{reg} = N_{reg}^{ram} + N_{reg}^{rom}$. On fait l'hypothèse qu'il y a un signal par registre (validation d'horloge par exemple).

– Les signaux de contrôle des multiplexeurs / buffers trois états : N_{sc}^{mux}. Le nombre de signaux dépend du type de cellules utilisées pour l'implantation. Pour une implantation à base de cellules logiques, le nombre de signaux de contrôle vaut $\lceil log_2(N_{entrées})$. Pour une implantation à base de buffers trois-états, le nombre de signaux de contrôle est égal au nombre d'entrées.

Le coût total en signaux de contrôle pour l'unité de traitement vaut donc :

$$N_{sc}^{ut} = N_{sc}^{op} + N_{sc}^{multi\_op} + N_{sc}^{reg} + N_{sc}^{mux}$$

D'où le nombre total de signaux de contrôle nécessaires pour le séquencement des unités de traitement et de mémorisation :

$$N_{sc} = N_{sc}^{ut} + N_{sc}^{um}$$

Connaissant le nombre total de signaux de contrôle et le nombre d'états, on peut alors estimer la surface occupée par l'unité de contrôle.

5.2.3 Unité de contrôle

D'après les hypothèses effectuées au chapitre 3.4.3, l'unité de contrôle est composée d'un registre d'état et de la logique de contrôle, qui utilise une mémoire ROM. L'estimation de la surface de l'unité de contrôle est basée sur l'estimation en surface de cette ROM et dépend donc du nombre et de la taille des mots qu'elle contient.

Le registre d'état mémorise l'adresse de l'état courant. Sa largeur doit permettre de stocker tous les états du système soit :

$$N_{bits\_reg\_état} = log_2(N_{état})$$

Les mots de la ROM doivent mémoriser l'adresse de l'état suivant ainsi que la valeur des signaux de contrôle. La largeur des mots vaut donc :

$$N_{bits\_rom} = N_{bits\_reg\_état} + N_{sc}$$

et le nombre total de mots lui est égal au nombre d'états. Il faut donc une ROM de $N_{états}$xN_{bits_rom} pour réaliser la logique de contrôle. Le coût en surface est déduit de la surface de la ROM qui peut être calculée comme présenté en 5.1.2.

5.2.4 Caractérisation physique globale

La surface totale est exprimée en terme de taux d'occupation des ressources du FPGA. Les ressources prises en compte sont les cellules logiques (slice, clb, logic elements), les cellules dédiées (bram, esb, eab), les buffers trois états et les entrées / sorties. On dispose à ce stade de la projection technologique des valeurs d'occupation de différentes ressources du FPGA pour chaque unité de l'architecture (hors entrées / sorties qui ne rentrent pas en compte dans le processus de projection) :

- N_{cl}^{um}, N_{cl}^{ut} et N_{cl}^{uc}
- N_{cd}^{um}, N_{cd}^{ut} et N_{cd}^{uc}
- N_{3state}^{ut}

Pour chacune de ces ressources, l'occupation est calculée en faisant la somme de la contribution de chaque unité de l'architecture.

- $N_{cl} = N_{cl}^{um} + N_{cl}^{ut} + N_{cl}^{uc}$
- $N_{cd} = N_{cd}^{um} + N_{cd}^{ut} + N_{cd}^{uc}$
- $N_{3state} = N_{3state}^{ut}$

La valeur physique de la contrainte de temps N_{cycles} de la solution en cours de projection est déduite de la valeur de la période d'horloge définie à l'étape d'allocation (chapitre 4.3) :

$$T = N_{cycles} * T_{horloge}$$

On obtient alors le coût exprimé en terme d'occupation des ressources du FPGA, pour une vitesse d'exécution exprimée en temps physique (ns, μs). Le processus de projection est ré-téré pour chaque solution architecturale définie par les estimations structurelles. Au final, on obtient un ensemble de solutions caractérisées en temps / surface.

Pour une solution {T, $N_{cl}(T)$, $N_{cd}(T)$, $N_{3state}(T)$} donnée, un certain nombre de simplifications basées sur l'élimination de solutions infaisables ou non optimales peuvent alors être effectuées :

- Élimination des solutions pour lesquelles on dépasse la capacité d'intégration, c'est à dire si :
  - $N_{cl}(T) > N_{cl}^{fpga}$ ou
  - $N_{cd}(T) > N_{cd}^{fpga}$ ou
  - $N_{3state}(T) > N_{3state}^{fpga}$

où N_{cl}^{fpga}, N_{cd}^{fpga} et N_{3state}^{fpga} sont respectivement le nombre de cellules logiques, le nombre de cellules dédiées et le nombre de buffer trois états du composant cible.

– Élimination des solutions non optimales : Si on suppose que l'espace de conception est caractérisé par l'ensemble des points $\{T, f(T)\}$, où $f(T)$ est la surface du système pour la contrainte de temps T, la simplification repose sur le principe qu'une solution $\{T, f(T)\}$ est sous-optimale s'il existe une solution $\{T', f(T')\}$ telle que $T' \leq T$ et $f(T') \leq f(T)$. La solution $\{T', f(T')\}$ est plus performante puisqu'elle est à la fois plus rapide et moins coûteuse en surface. On élimine donc la solution $\{T, f(T)\}$. On obtient finalement un nouvel ensemble de solutions $\{T^*, f(T^*)\}$ appelées points Pareto [31][32][30].

  Dans notre cas, la surface est caractérisée par plusieurs courbes $N_{cl}(T)$, $N_{cd}(T)$ et $N_{3state}(T)$. La simplification des solutions sous-optimales s'exprime alors de la façon suivante : pour une solution donnée {T, $N_{cl}(T), N_{cd}(T), N_{3state}(T)\}$, s'il existe une solution {T', $N_{cl}(T'), N_{cd}(T')$, $N_{3state}(T')\}$ telle que :

  – $T' \leq T$ et
  – $N_{cl}(T) \leq N_{cl}(T')$ et
  – $N_{cd}(T) \leq N_{cd}(T')$ et
  – $N_{3state}(T) \leq N_{3state}(T')$

  Alors, cette solution peut être éliminée.

Cette étape termine le cycle d'exploration / estimation. Le concepteur possède alors des informations sur la vitesse d'exécution et l'occupation des différentes solutions architecturales définies par l'outil d'exploration.

5.3 Conclusion

Ce chapitre présente comment passer de la caractérisation structurelle de l'application qui fournit un coût au niveau RTL, aux valeurs d'occupation des ressources du FPGA en fonction de la vitesse. Les caractéristiques du composant cible sont décrites dans le fichier technologique. Celui-ci contient des informations sur les temps de traversée et la surface des unités fonctionnelles de base (opérateurs, ressources de connexion et mémoires) en fonction des ressources du FPGA qu'elles utilisent. Ces informations permettent le calcul

de l'occupation du composant et des performances des différentes solutions architecturales. Ce calcul prend en compte le coût de chaque unité de l'architecture, ce qui permet d'obtenir une caractérisation complète (traitement, mémoire, contrôle). Les résultats obtenus fournissent ainsi au concepteur des valeurs d'estimation complètes et réalistes pour lui permettre d'effectuer ses choix.

Nous allons voir dans le chapitre suivant sur quelques exemples, d'une part, comment synthétiser la solution retenue, et d'autre part quelle est la précision des valeurs fournies par l'estimation physique par rapport à celles du système une fois implanté. Cette partie aura deux objectifs : chiffrer et discuter du rapport précision complexité de la méthode et démontrer l'apport d'une exploration rapide dans le cycle de conception.

Chapitre 6

Applications

Les chapitres précédents ont montré comment, à partir d'une spécification comportementale, on définit et estime les caractéristiques temps / coût de plusieurs solutions architecturales d'un système. Dans ce chapitre, nous effectuons une présentation de l'outil Design Trotter *et de son module d'exploration architecturale. Puis, nous proposons une évaluation de la précision des valeurs d'estimation par rapport aux valeurs obtenues après synthèse. Les applications choisies sont représentatives de deux domaines d'application : codage audio (G722) et traitement d'image (transformée en ondellettes).*

6.1 Présentation de l'outil d'exploration architecturale

L'outil *Design Trotter* actuellement développé au laboratoire LESTER permet l'application de la méthodologie d'exploration architecturale sur des exemples complexes. La figure 6.1 présente une vue de l'interface graphique. Celle-ci est constituée :

- d'un onglet pour la traduction du langage C vers *HCDFG*,
- d'un onglet pour la compilation de graphes,
- d'un onglet pour la compilation d'un fichier technologique,
- d'un onglet pour la visualisation de graphes,
- d'un onglet pour l'estimation système,
- d'un onglet pour l'estimation architecturale.

La figure 6.1 montre un exemple d'exploration réalisée sur une sous-fonction de la transformée en ondelettes sur un composant Virtex. Il s'agit d'une boucle imbriquée d'ordre deux réalisant un filtrage sur un tableau à deux dimensions. On dispose de cinq solutions pour lesquelles on connait le nombre de *slices* et de *brams* en fonction de la vitesse d'exécution. Si on choisit par exemple d'examiner plus précisément la solution 2 (930 slices, 612 ns), l'outil fournit les informations sur le nombre d'états et le nombre de ressources nécessaires. Le détail de l'allocation permet alors la synthèse des unités de traitement et de mémoire. L'unité de traitement est constituée de 4 multiplieurs, 8 additionneurs et de 28 registres. L'unité de mémoire comprend quant à elle 12 mémoires RAM et 4 mémoires ROMs. La fenêtre *Result hierarchy* fournit en outre d'autres informations relatives à chaque niveau de hiérarchie (il s'agit des résultats d'estimation structurelle de chaque noeud composite du graphe), qui sont utiles pour permettre la réalisation de l'unité de contrôle. Dans le cas de notre exemple, à partir du détail des caractéristiques du coeur de boucle (1 multiplieur, 2 additionneurs, 5 états) et de l'algorithme de combinaison utilisé dans le cas des boucles déterministes (voir 4.5.2), on peut déduire l'ordonnancement de la solution retenue.

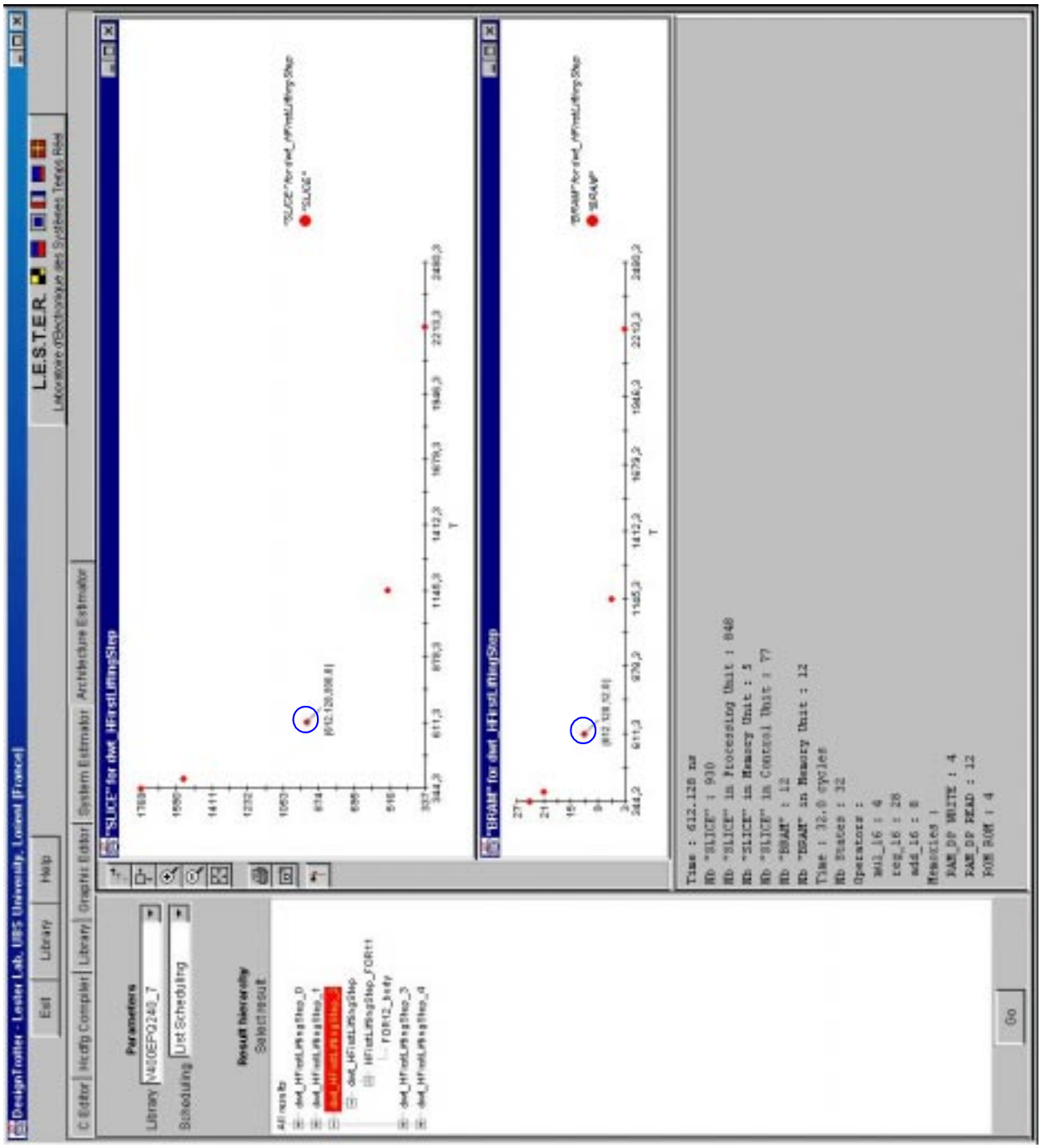


FIG. 6.1 – Exemple d'exploration architecturale

Celle-ci est constituée de quatre coeurs de boucle qui s'exécutent de façon pipeline. L'ordonnancement correspondant est donné à la figure 6.2. Il nécessite le codage de 32 états.

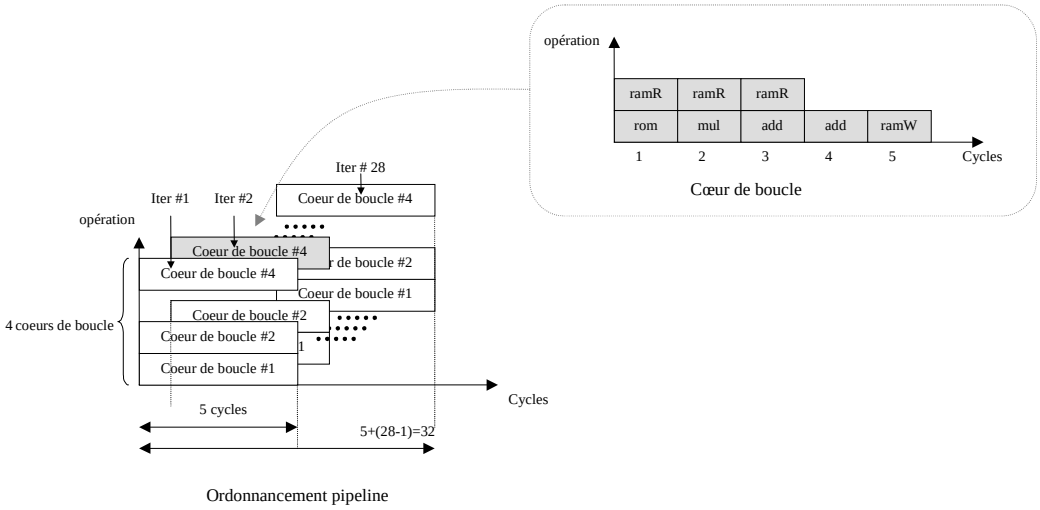


FIG. 6.2 – Ordonnancement pour l'exemple de la figure 6.1

La connaissance précise des caractéristiques des unités de traitement, mémoire et contrôle définit l'architecture de la solution. La synthèse peut alors être effectuée et donne les résultats suivants : 926 slices, 12 brams et 571 ns pour une estimation de 930 slices, 12 brams, 612 ns. Pour cet exemple, le temps nécessaire pour l'exploration des cinq solutions est de l'ordre de la milliseconde. Le temps nécessaire à la synthèse logique de la solution retenue est de l'ordre de la dizaine de minutes.

Nous allons maintenant étudier la précision des estimations sur des applications plus complexes représentatives du domaine du traitement numérique du signal.

6.2 Résultats d'estimation vs synthèse : Introduction

L'objectif de cette partie est d'évaluer la précision des valeurs d'estimation des solutions architecturales explorées. Nous considérons pour cela deux applications de traitement numérique du signal : une application de codage

audiofréquence (norme G722) et du traitement d'image avec la transformée en ondelettes. L'application de la méthode à la norme de codage audiofréquence G722 permet de juger de l'efficacité de l'estimation dans le cas des systèmes ayant une exécution non déterministe (systèmes qui comportent des boucles non déterministes et des structures conditionnelles). Le cas des systèmes orientés vers le traitement et la mémorisation intensive de données multidimensionnelles est abordé avec la transformée en ondelettes 2D.

Pour chacune de ces applications, une solution est sélectionnée à partir des résultats de l'exploration architecturale. Elle est ensuite synthétisée en se basant sur les informations telles que les contraintes de temps locales, le nombre d'états et l'allocation correspondante, fournies par l'outil d'exploration. Étant donné le temps nécessaire à la conception d'une architecture, en particulier pour des spécifications complexes comme dans le cas de la transformée en ondelettes, une seule solution est synthétisée pour chaque application et comparée avec les valeurs d'estimation temps / surface correspondantes. La solution architecturale synthétisée est définie d'après la solution architecturale issue de l'exploration : elle possède le même ordonnancement et le même nombre d'états. C'est donc l'erreur sur la fréquence d'horloge qui va déterminer la précision des estimations temporelles. Ainsi, seules les vitesses d'exécution sont présentées dans les résultats car elles dépendent de la fréquence d'horloge qui est calculée par l'outil de synthèse logique ($T = N_{cycles} * T_{horloge}$).

En ce qui concerne les comparaisons temps d'exploration / estimation vs temps de synthèse, notons qu'il s'agit de la synthèse logique. Le temps de conception de l'architecture n'est pas pris en compte car il est difficilement estimable. Celui-ci dépend en effet de facteurs tels que l'expérience du concepteur (pour du code écrit à la main) ou des performances de l'outil de synthèse architectural (pour du code généré automatiquement), le degré de vérification (simulation fonctionnelle, temporelle), ... La comparaison des temps d'exploration / estimation par rapport au temps de synthèse logique (hors synthèse architecturale) donne en outre une sous-estimation déjà significative de la faible complexité de la méthode. Notons enfin que ces comparaisons temporelles ont été mesurées sur un PC de 1GHz de fréquence d'horloge avec 512 Mo de mémoire RAM.

6.3 codage de la parole : G722

6.3.1 Présentation de la norme G722

La recommandation G722 de l'UIT-T est l'une des composantes audio de la norme H320, utilisée pour des applications de visioconférence. Elle décrit les caractéristiques d'un système de codage audiofréquence (de 50 à 7000 Hz) qui peut être utilisé pour une grande variété d'applications de codage de signaux vocaux de haute qualité. Le système de codage fait appel à la modulation par impulsions et codage différentiel adaptatif à sous-bandes (SB-MICDA). La technique SB-MICDA utilisée consiste à diviser la bande de fréquences en deux sous-bandes (la sous-bande supérieure et la sous-bande inférieure), et à coder les signaux de chaque sous-bande selon la technique MICDA [46]. Le traitement des deux sous-bandes étant similaire, nous limitons l'étude au codeur de la sous-bande supérieure dont le schéma de principe est donné à la figure 6.3.

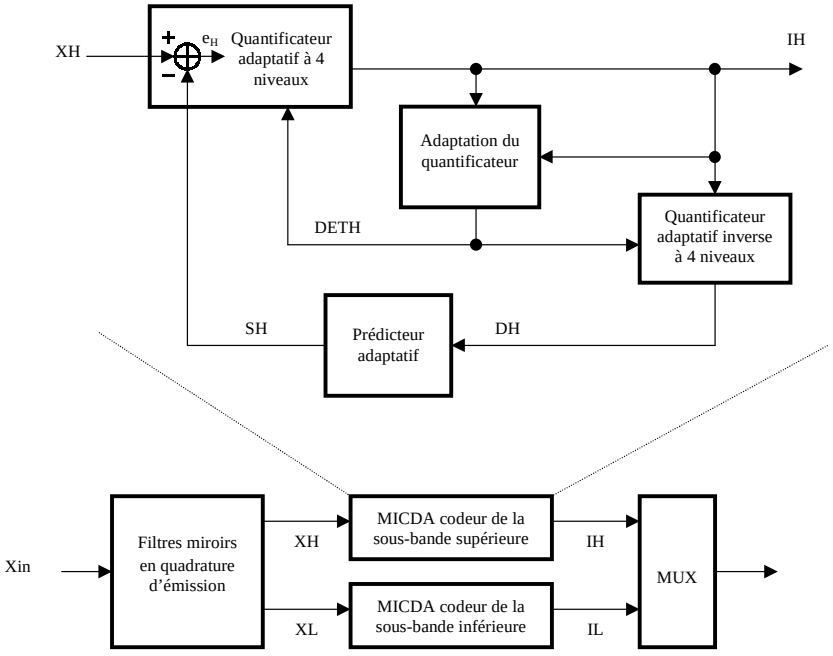


FIG. 6.3 – Schéma de principe du codeur SB-MICDA

Le signal d'entrée de la sous-bande supérieure XH, dont on soustrait l'estimation SH du signal d'entrée donne le signal de différence EH. Un quantifica-

teur adaptatif non-linéaire à quatre niveaux attribue deux éléments binaires à la valeur du signal de différence afin de produire le signal IH. Le quantificateur adaptatif inverse produit un signal de différence quantifié DH à partir de ces deux éléments binaires. On ajoute l'estimation SH à ce signal de différence quantifié pour obtenir une version reconstituée du signal d'entrée. Un prédicteur adaptatif, qui utilise à la fois le signal reconstitué et le signal de différence quantifié, fournit une estimation SH du signal d'entrée, ce qui ferme la boucle de contre-réaction.

Le prédicteur est le bloc le plus complexe du codeur. Pour donner un ordre d'idée, la spécification représente 250 lignes de code C (300 lignes en grammaire hcdfg). C'est sur cette fonction que nous allons utiliser l'outil d'exploration. Une description du prédicteur est donnée à la figure 6.4. Il

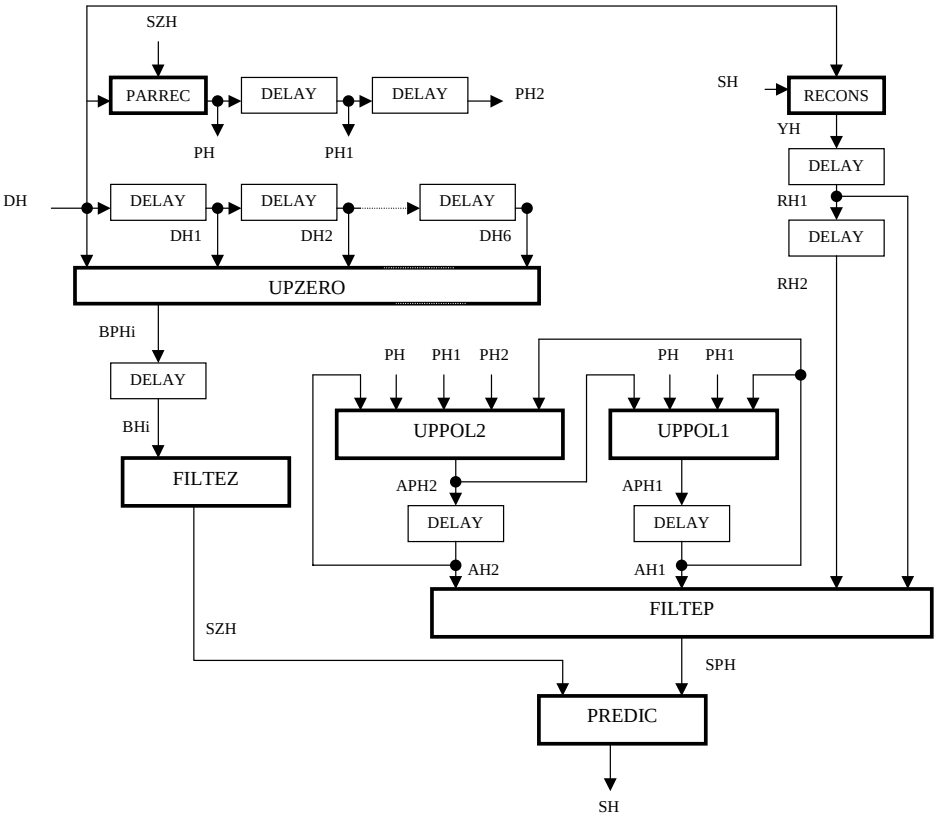


FIG. 6.4 – Prédicteur adaptatif de la sous-bande supérieure

est constitué de huit sous-fonctions qui s'exécutent de façon concurrente.

Chacune de ces fonctions a été synthétisée séparément de façon a obtenir un plus grand nombre de points de comparaison estimation / synthèse.

Les résultats de l'exploration architecturale du prédicteur pour deux composants d'implantation (Virtex et Apex) sont présentés respectivement figure 6.5 et 6.6. Pour cet exemple, l'outil a exploré 16 solutions architecturales en 1.1 secondes. La solution sélectionnée pour la synthèse correspond à celle qui minimise la surface (solution No 15). Les résultats fournissent le détail de l'allocation, des accès mémoire, le nombre d'états et la contrainte de temps pour le prédicteur ainsi que pour les sous-fonctions qui le composent. La génération de l'architecture est réalisée à partir de la connaissance de ces informations. Celles-ci peuvent en outre renseigner le concepteur sur le potentiel d'optimisation du système. Par exemple, Si on analyse les noeuds composites de la spécification, on peut voir que c'est la fonction *upzero* qui impose la vitesse d'exécution du système (tableau 6.1). Cette fonction constitue donc un point critique dont l'optimisation permettrait d'améliorer le temps d'exécution.

La solution une fois synthétisée correspond à une occupation de 30% et de 38% pour le Virtex et l'Apex et à un temps d'exécution de $1.22\mu s$ et $1.50\mu s$ respectivement. Ces résultats sont comparés avec ceux de la synthèse logique au paragraphe suivant.

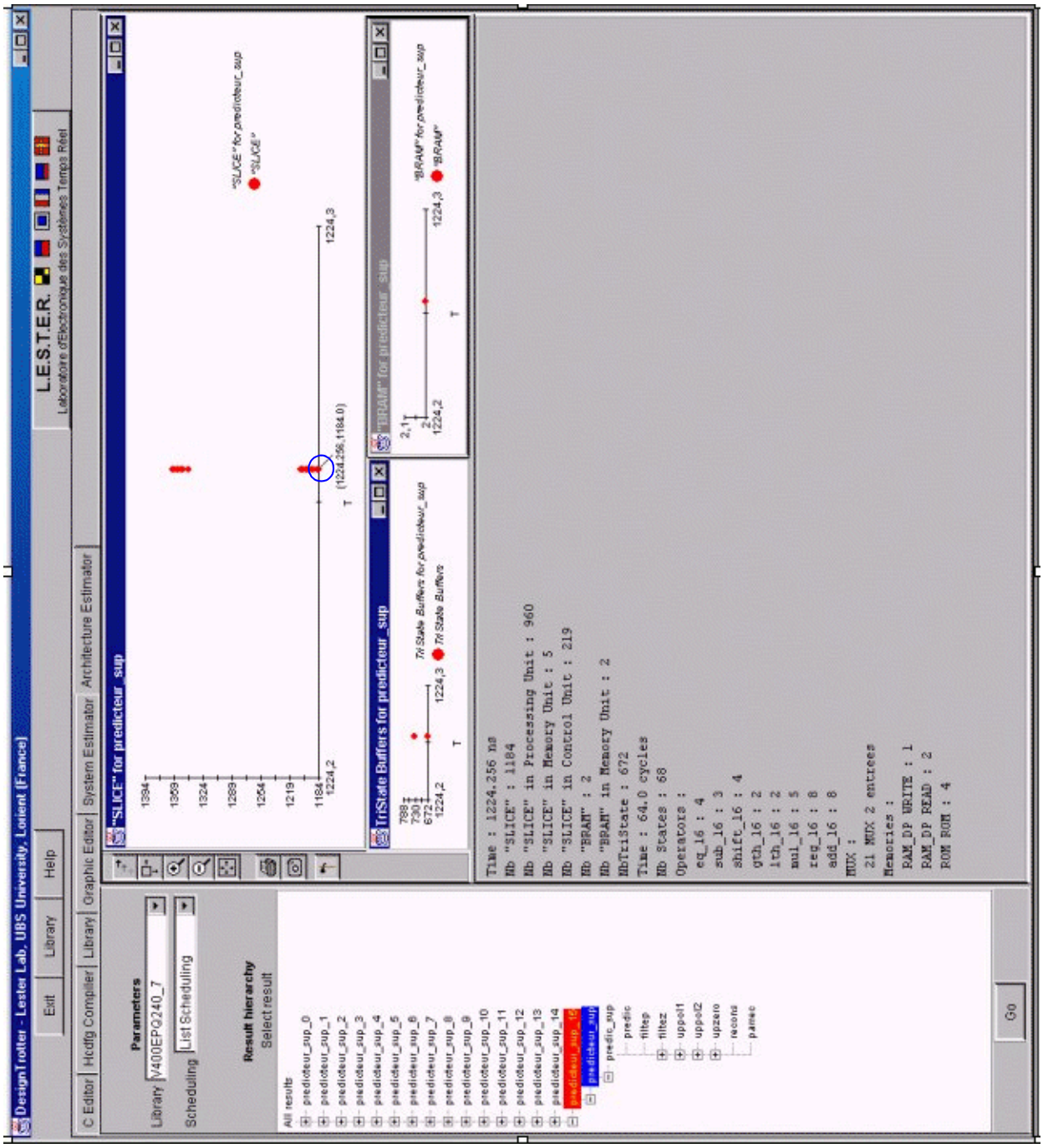


FIG. 6.5 – Résultats d'estimation du prédicteur (Virtex)

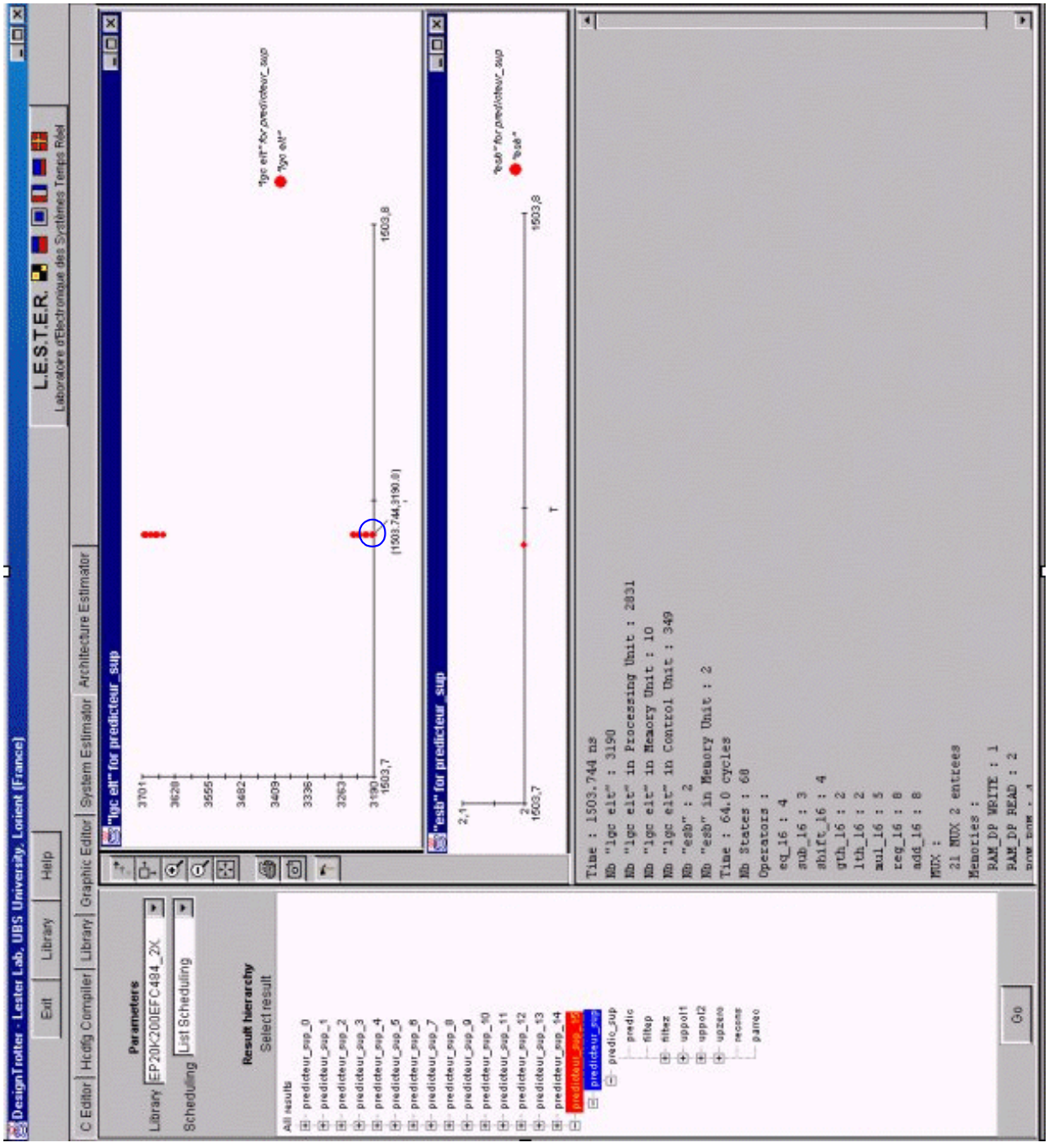


FIG. 6.6 – Résultats d'estimation du prédicteur (Apex)

6.3.2 Synthèse du prédicteur

Le tableau 6.1 présente les valeurs d'occupation des cellules logiques et de temps d'exécution estimés par rapport aux résultats de la synthèse. Les écarts correspondants sont founis au tableau 6.2.

	Virtex V400EPQ240-7				*Apex EP20K200EFC484-2X*			
	Estimation		*Synthèse*		*Estimation*		*Synthèse*	
	Slices	$T_{ex}(ns)$	Slices	$T_{ex}(ns)$	lgc elt	$T_{ex}(ns)$	lgc elt	$T_{ex}(ns)$
parrec	9	6	10	6	17	9	19	9
recons	9	6	10	6	17	9	19	9
upzero	217	1224	255	1171	608	1504	400	1272
uppol2	257	292	303	300	857	358	718	302
uppol1	216	230	275	254	589	282	612	236
filtep	150	77	163	88	518	94	648	103
filtez	181	593	177	511	484	728	497	515
predic	9	6	10	6	17	9	19	9
predicteur	1166	1224	1263	1317	3132	1504	3027	1318

TAB. 6.1 – Prédicteur : Estimation vs Synthèse

L'écart est de 7% et 14.1% en ce qui concerne les vitesses d'exécution et de 7.7% et 3.4% en ce qui concerne l'estimation du nombre de cellules logiques (respectivement pour Virtex et Apex).

On constate un écart important dans l'estimation des cellules logiques de la sous-fonction *upzero*. Ceci provient du fait que cette fonction utilise un

	Virtex V400EPQ240-7				*Apex EP20K200EFC484-2X*			
	Ecarts (%)		$T_{expl/synth}$ (s)		*Ecarts (%)*		$T_{expl/synth}$ *(s)*	
	Slices	T_{ex}	T_{expl}	T_{synth}	lgc elt	T_{ex}	T_{expl}	T_{synth}
parrec	-10	+1.4	0.05 s	1 min	-10.5	+4.9	0.05 s	1 min
recons	-10	+1.1	0.05 s	1 min	-10.5	+4.9	0.05 s	1 min
upzero	-14.9	+4.5	0.22 s	5 min	+52	+18.3	0.06s	5 min
uppol2	-15.2	-2.7	0.11 s	5 min	+19.4	+18.6	0.11 s	5min
uppol1	-21.5	-9.8	0.11 s	5 min	-3.8	+19.6	0.11 s	5 min
filtep	-8	-13.1	0.05 s	1 min	-20.1	-8.4	0.05 s	1 min
filtez	+2.2	+16.1	0.05 s	2 min	-2.6	+41.4	0.05 s	2 min
predic	-10	-2.2	0.05 s	1 min	-10.5	+4.9	0.06 s	1 min
predicteur	-7.7	-7	0.9 s	15 min	+3.4	+14.1	0.4 s	10 min

TAB. 6.2 – Prédicteur : Écarts

multiplieur câblé avec un opérande constant, ce qui simplifie le multiplieur

utilisé et donc diminue sa surface ainsi que son temps de traversée (qui conditionne la période d'horloge dans ce cas, et donc aussi la vitesse). Toutefois, cette erreur sur la fréquence d'horloge n'est pas visible dans l'estimation du prédicteur dans son ensemble, car d'autres multiplieurs complets sont utilisés par les autres sous-fonctions. Dans le cas de la fonction *upzero*, l'écart est beaucoup plus important sur la cible Apex, ce qui laisse supposer que l'outil Quartus est plus performant que Foundation. De manière générale, l'estimation locale des sous-fonctions est biaisée par le fait que l'outil de synthèse logique effectue des simplifications qui n'ont pas lieu d'être lorsque l'on considère la spécification dans son ensemble. On remarque d'ailleurs que ces simplifications ne sont pas systématiques en fonction de l'outil de synthèse utilisé. D'autre part, ceci implique que le calcul de l'écart effectué ici est lié aux outils de synthèse logique utilisés et montre les limites d'une telle comparaison.

Toutes les sous-fonctions du prédicteur ont été synthétisées afin d'établir une valeur moyenne des écarts. L'erreur moyenne sur le nombre de cellules logiques est de 12.9%. Pour les valeurs temporelles, elle est de 10.7%.

6.3.3 Conclusion

L'erreur moyenne est de l'ordre de 12% sur cet exemple. L'outil a permis l'exploration de 16 solutions pour des temps de l'ordre de la seconde. La synthèse logique de la solution architecturale retenue est de l'ordre de l'heure.

6.4 Transformée en ondelettes 2D

6.4.1 Présentation de la transformée en ondelettes

La transformée en ondelettes est utilisée dans le cadre de la compression d'images (*JPEG*, *MPEG*). Elle est issue de la théorie de la multirésolution qui consiste à étudier l'image à différentes résolutions et à la remplacer à une échelle donnée par son approximation la plus proche. L'idée de base est de représenter l'information comme une superposition d'ondelettes en introduisant une notion d'échelle ou de résolution par contraction ou dilatation de l'échelle des temps [47][48]. Les différences entre deux niveaux consécutifs représentent les détails qui sont relevés ou atténués lorsque la résolution

augmente ou diminue [49].

La figure 6.7 présente le principe de décomposition d'une image par la transformée en ondelettes bidimensionnelle à base de bancs de filtres. L'espace d'origine est décomposé en quatre espaces par une suite de fil-

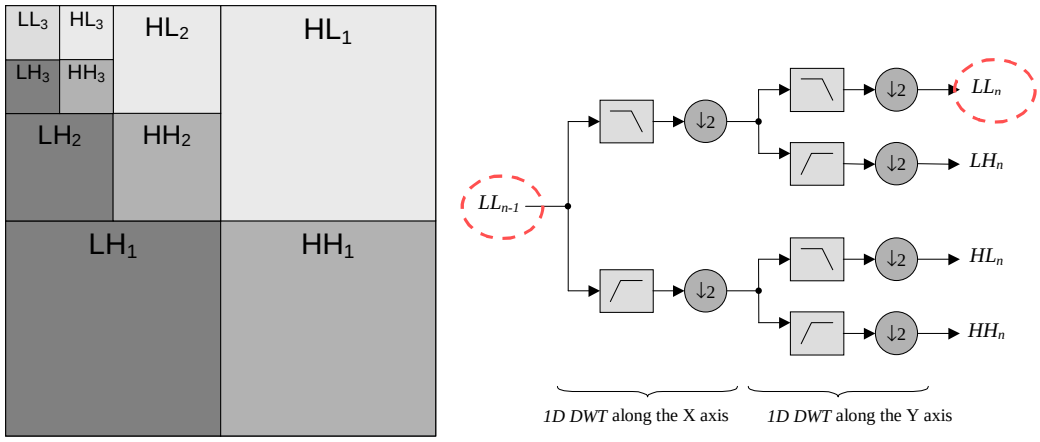


FIG. 6.7 – Décomposition d'une image par la transformée en ondelettes sur 3 niveaux

trages suivis de décimations d'ordre 2, appliqués d'une part aux lignes puis aux colonnes. Une sous-image de texture et trois sous-images de détails sont ainsi obtenues. Les sous-images de détail comportent peu d'informations et peuvent donc donner lieu à un fort taux de compression. Chacun de ces espaces occupe alors un quart de l'espace d'origine. La transformée inverse s'obtient de manière similaire à l'aide des filtres conjugués.

L'implémentation de la transformée en ondelettes à base de bancs de filtres étant coûteuse en terme de surface, de nouveaux algorithmes comme le *lifting scheme* ont été développés. Cette technique permet de réduire le nombre d'opérations nécessaires pour une qualité de compression équivalente. De plus, les opérations peuvent être effectuées à la suite, les valeurs calculées remplaçant celles d'origine ce qui permet une économie en nombre de points mémoire. Enfin, la transformée inverse s'obtient facilement à partir de la transformée directe en inversant l'ordre des opérations et le signe des opérateurs.

La figure 6.8 présente le principe de la transformation utilisée par le *lifting scheme*. Pour fixer un ordre d'idée de la complexité, la spécification

représente 230 lignes de code C (500 lignes en grammaire hcdfg). C'est sur cet algorithme que nous allons effectuer l'exploration. À partir de l'image originelle, on applique un ensemble de six traitements successifs horizontalement, puis verticalement. Ceux-ci consistent en quatre *lifting sheme*, la multiplication par un facteur d'échelle, puis le ré-agencement de l'image en deux sous-images, l'une contenant les pixels d'indice pair, l'autre ceux d'indice impair. Ces traitements sont décrits à l'aide de boucles imbriquées d'ordre

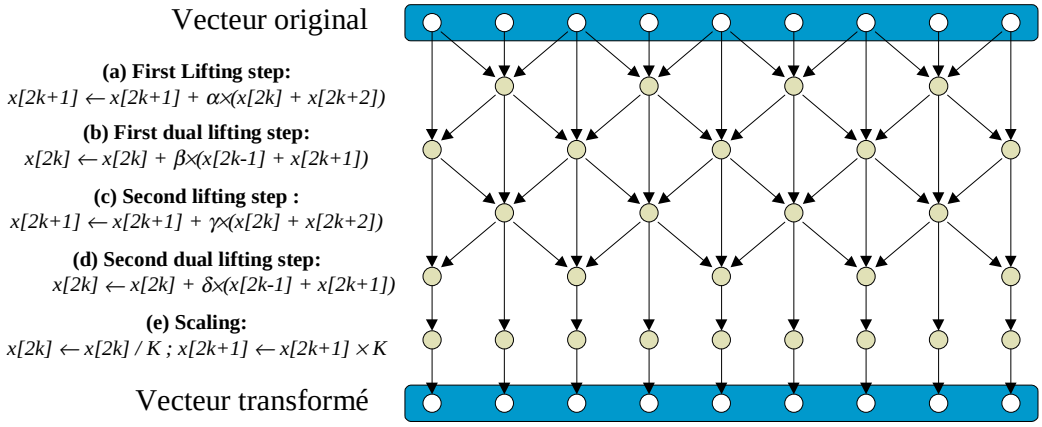


FIG. 6.8 – Décomposition de l'image par l'utilisation du Lifting Scheme

deux qui permettent ainsi de manipuler facilement les différents éléments de l'image qui sont contenus dans un tableau à deux dimensions.

Les résultats fournis par l'outil d'exploration architecturale sur cet exemple sont présentés aux figures 6.9 et 6.10. La solution sélectionnée pour la synthèse est celle qui correspond au point dont les caractéristiques temps - surface sont mises en relief. L'unité de traitement correspondante est constituée de quatre multiplieurs, huit additionneurs et 28 registres. L'unité de mémoire comprend 12 RAMs et 4 ROMs. Pour diminuer la complexité de réalisation (en particulier au niveau de la machine d'état) mais aussi du fait de la limitation en taille des composants, la transformée codée s'applique à des images 16x16. L'estimation de la taille mémoire dans ce cas est de 512 mots de 16 bits. L'intégration des mémoires nécessite alors l'utilisation de 12 ressources dédiés, ce qui correspond à une occupation de 30% des *brams* pour le Virtex et de 23% des *esb* pour l'Apex. L'occupation des cellules logiques vaut respectivement 53% et 62% pour des temps d'exécution de 8.53μs et 10.5μs. Le

séquencement nécessite le codage de 446 états. Dans cet exemple, l'unité de contrôle est la plus coûteuse en surface.

La spécification est constituée de six boucles d'ordre 2 pour l'application de la transformée horizontale et de six autres pour la transformée verticale. Au total, 12 boucles d'ordre deux sont appliquées séquentiellement. Les résultats partiels d'estimation de chaque boucle puis de la transformée dans son ensemble sont comparés aux résultats de la synthèse au paragraphe suivant.

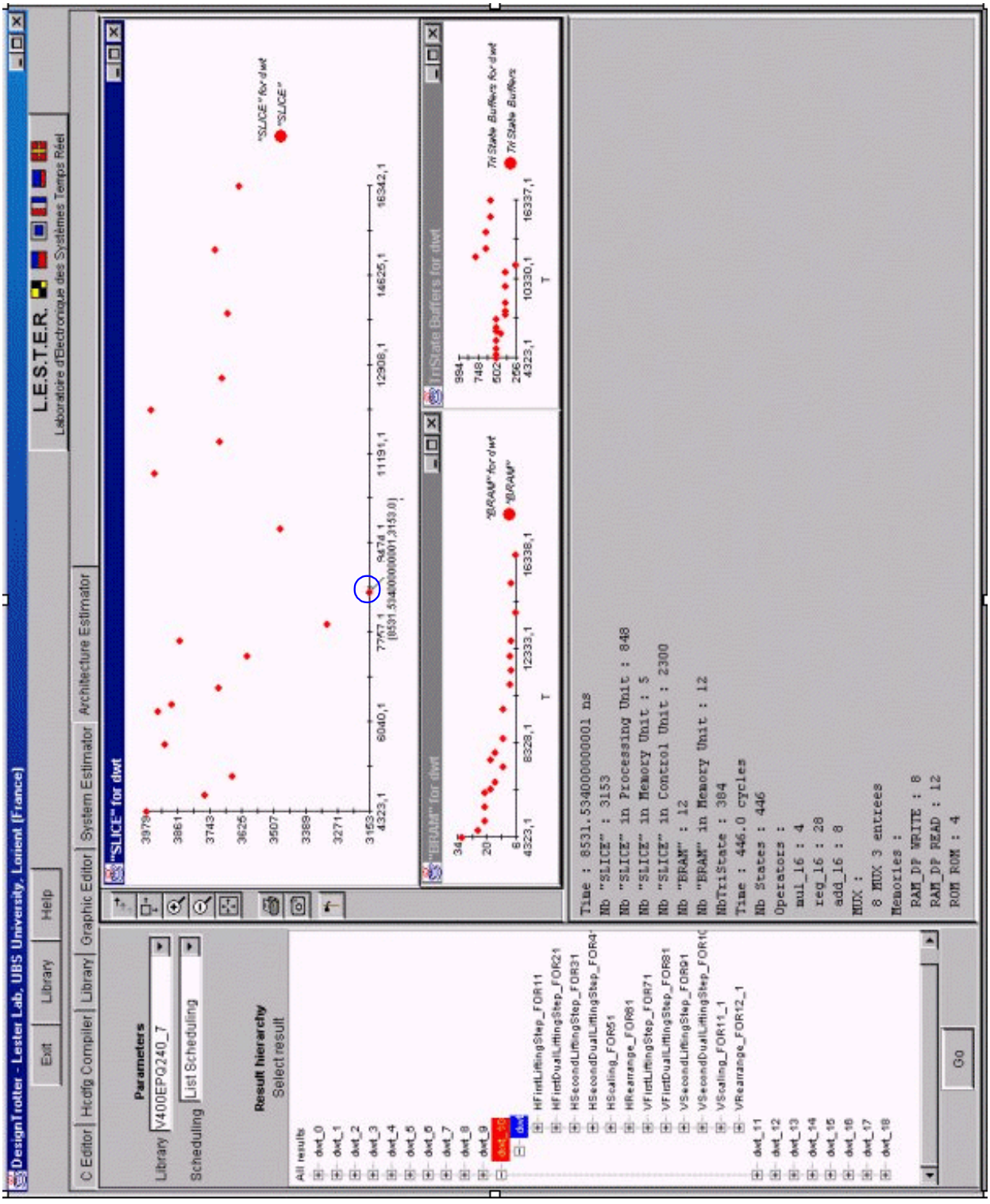


FIG. 6.9 – Résultats d'estimation de la transformée en ondelettes (Virtex)

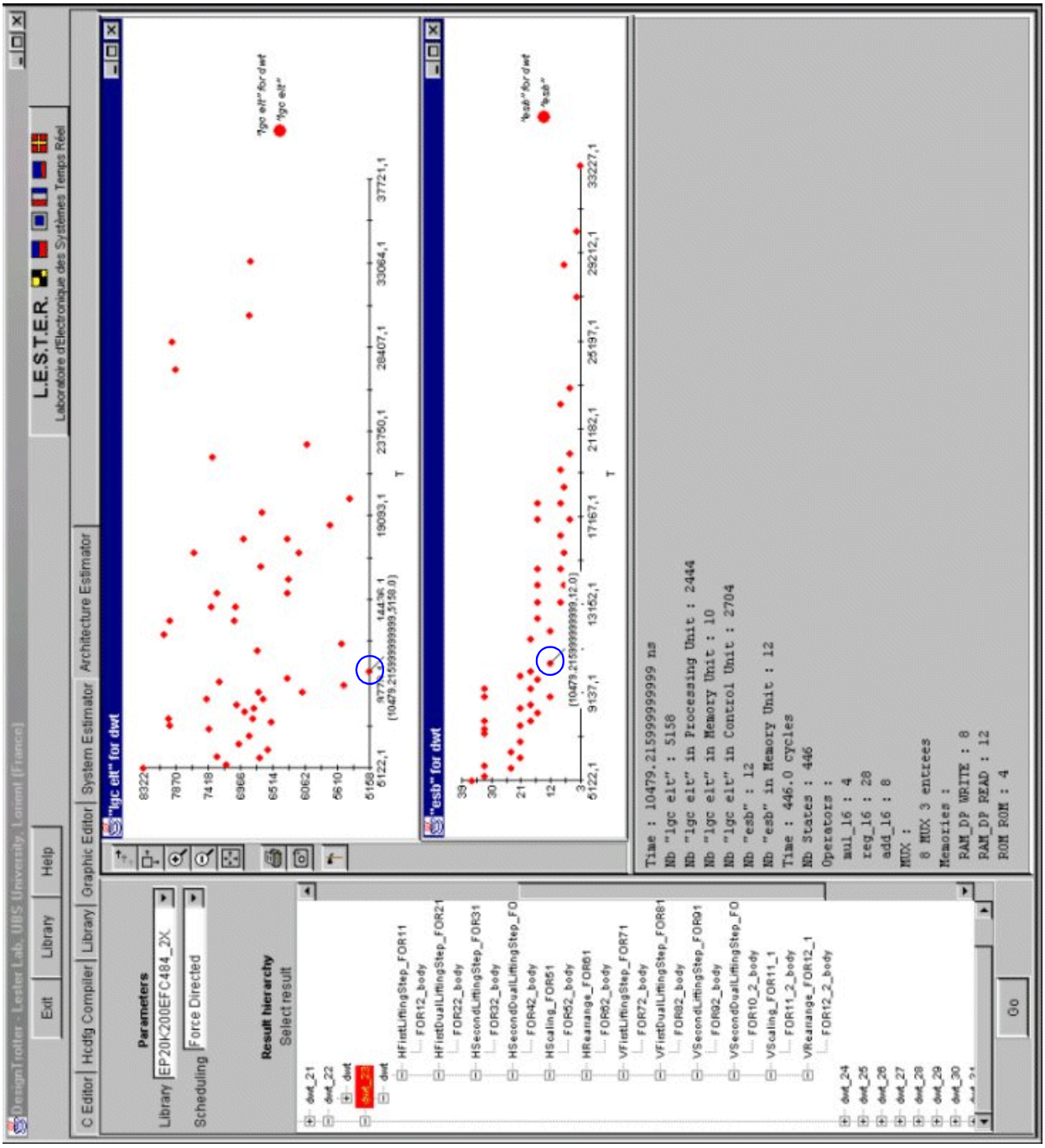


FIG. 6.10 – Résultats d'estimation de la transformée en ondelettes (Apex)

6.4.2 Synthèse de la transformée en ondelettes

Le tableau 6.3 présente les résultats d'estimation vs synthèse pour la solution sélectionnée. Les écarts correspondants sont fournis au tableau 6.4, ainsi que les temps nécessaires pour l'exploration de toutes les solutions qui sont comparés au temps de synthèse logique de la solution retenue.

Les résultats montrent un écart faible entre l'estimation et la synthèse partielle des fonctions de filtrage du *lifting scheme* (9.6% en surface et 8.7% en temps). Par contre, une erreur plus importante est à signaler concer-

	Virtex V400EPQ240-7				*Apex EP20K200EFC484-2X*			
	Estimation		*Synthèse*		*Estimation*		*Synthèse*	
	Slices	$T_{ex}(ns)$	Slices	$T_{ex}(ns)$	lgc elt	$T_{ex}(ns)$	lgc elt	$T_{ex}(ns)$
1stHLiftStep	990	612	926	571	2472	752	2438	699
1stHDualLiftStep	990	612	952	608	2472	752	2410	738
2ndHLiftStep	990	612	942	537	2472	752	2404	688
2ndHDualLiftStep	973	536	949	488	2438	658	2443	647
HScaling	791	1263	770	1218	2074	1551	1978	1496
HRearrange	160	896	109	1194	319	773	191	709
1stVLiftStep	990	612	924	488	2472	752	2476	730
1stVDualLiftStep	990	612	943	524	2472	752	2488	715
2ndVLiftStep	990	612	942	517	2472	752	2444	730
2ndVDualLiftStep	973	536	941	453	2438	658	2504	611
VScaling	791	1263	765	1197	2074	1551	2009	1493
VRearrange	160	89	106	1384	319	773	198	676
DWT 2D	3153	8532	2320	7220	6831	10479	4983	10162

TAB. 6.3 – Transformée en ondelettes : Estimation vs Synthèse

nant l'estimation globale de la transformée (36.5% en surface et 10.7% en temps). Pour comprendre d'où provient l'erreur, nous avons synthétisé une demi transformée (transformée selon une seule dimension). L'estimation de la transformée 1D fait apparaître un écart de 16.4% en surface par rapport à la synthèse alors qu'il est de 35.9% pour la transformée 2D. Les unités de traitement et de mémorisation étant strictement identiques, l'erreur provient de l'unité de contrôle et croit en fonction du nombre d'états. Cette erreur s'explique par le fait qu'un certain nombre de simplifications ont été effectuées par les outils de synthèse logique (plus le nombre d'état est important, plus le potentiel de simplification est important). Comme nous l'avons vu au paragraphe 6.3.2, un écart lié aux optimisations effectuées par les outils de synthèse logique est introduit (utilisation d'opérateurs câblés, optimisation

de la logique de contrôle, ...).

D'autre part, les optimisations effectuées et les écarts qui en résultent sont dûs au modèle utilisé pour l'unité de contrôle. En effet, celle-ci à la charge de la génération des adresses. Or, la solution architecturale synthétisée

	Virtex V400EPQ240-7				*Apex EP20K200EFC484-2X*			
	Écarts (%)		*$T_{expl/synth}$ (s)*		*Écarts (%)*		*$T_{expl/synth}$ (s)*	
	Slices	T_{ex}	T_{expl}	T_{synth}	lgc elt	T_{ex}	T_{expl}	T_{synth}
1stHLiftStep	+6.9	+7.1	0.1 s	5 min	+1.4	+7.6	0.05 s	8 min
1stHDualLiftStep	+4	+0.6	0.05 s	5 min	+2.6	+1.9	0.06 s	8 min
2ndHLiftStep	+5.1	+13.9	0.06 s	5 min	+2.8	+9.3	0.05 s	8 min
2ndHDualLiftStep	+2.5	+9.8	0.06 s	5 min	-0.2	+1.7	0.05 s	8 min
HScaling	+2.7	+3.6	0.1 s	5 min	+4.9	+3.6	0.1 s	8 min
HRearrange	+46.8	-25	0.06 s	5 min	+67	+9.1	0.05 s	8 min
1stVLiftStep	+7.1	+25.5	0.1 s	5 min	-0.2	+2.9	0.05 s	8 min
1stVDualLiftStep	+5	+16.9	0.05 s	5 min	-0.6	+5.1	0.06 s	8 min
2ndVLiftStep	+5.1	+18.5	0.06 s	5 min	+1.1	+2.9	0.05 s	8 min
2ndVDualLiftStep	+3.4	+18.3	0.06 s	5 min	-2.6	+7.7	0.05 s	8 min
VScaling	+3.4	+5.5	0.1 s	5 min	+3.2	+3.8	0.1 s	8 min
VRearrange	+50.9	-5.5	0.06 s	5 min	+61	+3.8	0.05 s	8 min
DWT 2D	+35.9	+18.2	5 min	1.5 jours	+37	+3.1	5 min	2 jours

TAB. 6.4 – Transformée en ondelettes : Écarts

pour notre transformée en ondelettes est composée de 12 mémoires RAM et 4 ROMs, ce qui représente une charge de calcul très importante pour la génération des adresses. L'utilisation d'un modèle plus fin utilisant des unités de calcul d'adresse pourrait résoudre ce problème et permettre la définition d'une unité de contrôle plus optimale.

6.4.3 Conclusion

L'exploration a fourni 47 solutions architecturales en 5 minutes. En réalité, l'algorithme génère environ 350 solutions, mais elles ne sont pas toutes retenues à l'étape de projection technologique car un grand nombre d'entre elles dépasse les capacités d'intégration des FPGAs ciblés. L'erreur moyenne totale sur cet exemple est de 23.5% en surface et de 9.7% en temps. La synthèse logique de la solution retenue a duré 2 jours.

6.5 Conclusions générales sur les résultats

L'application de la méthodologie d'exploration sur des exemples démontre la faisabilité des solutions architecturales définies (grâce à une caractérisation complète traitement mémoire contrôle et au principe d'ordonnancement / combinaison) et la précision des valeurs d'estimation fournies *pour ces solutions* (de l'ordre de 10% pour les valeurs temporelles et 18% en surface).

Ce chapitre démontre d'autre part la possibilité d'explorer un très vaste espace de conception puisque sur un exemple, environ 350 solutions sont explorées pour un temps de traitement de l'ordre de 5 minutes (paragraphe 6.4.3). L'objectif de départ qui était l'évaluation rapide de plusieurs compromis performances / occupation est donc atteint. De plus, toutes les solutions définies sont synthétisables, ce qui permet de garantir la faisabilité du système. La synthèse s'appuie sur les informations locales (résultats d'estimation structurelles de chaque noeud composite) qui permettent en outre l'analyse du système en vue d'une éventuelle optimisation. Enfin, la généricité de la méthode (appliquée ici dans le cas du Virtex et de l'Apex) a été démontrée, ce qui permet ainsi la comparaison relative de plusieurs composants d'implantation.

La validation montre aussi un certain nombre de limitations. Premièrement la précision est très dépendante des optimisations au niveau logique. La précision moyenne mesurée sur les exemples (environ 15%) doit donc être pondérée par le fait que cette précision peut varier de façon importante (jusqu'à 50% sur nos exemples). Cette erreur est principalement due à la simplification des équations logiques et à l'emploi d'opérateurs câblés qui en résulte. Ce type d'erreur est donc sensible dans le cas des unités fonctionnelles reliées à des opérandes constants. Les imprécisions qui en résultent sont inévitables et leur valeur est difficilement prédictible à partir du moment où les performances des outils de synthèse commerciaux interviennent dans cette comparaison des résultats de l'estimation par rapport à ceux de la synthèse.

D'autre part, les solutions architecturales sont sous optimales sur certains plans (partage des ressources et fusion de machines d'état pour du parallélisme inter HCDFG, ordonnancement des boucles, génération des adresses). Ceci provient de la simplicité des estimations structurelles qui ne prennent

volontairement pas en compte toutes les optimisations possibles, l'objectif de départ étant la définition d'une méthode de complexité la plus réduite possible pour permettre l'exploration d'un vaste espace de recherche (et qui se justifie par le domaine d'application à savoir le codesign).

Un certain nombre d'améliorations sont possibles à ce niveau comme par exemple l'utilisation d'un modèle séparé pour la génération des adresses ce qui permettrait d'estimer plus finement l'unité de contrôle (qui pour le moment intègre la génération d'adresse). Une autre amélioration possible repose sur la redéfinition des algorithmes d'estimation structurelle en se rapprochant le plus possible du mode de fonctionnement d'un outil de synthèse d'architecture (mais en se limitant à un dénombrement de ressources, sans aller jusqu'à la description des connexions entre unités). Les solutions définies correspondraient alors à des solutions automatiquement synthétisables et plus optimales. Toutefois, la complexité en serait forcément affectée ce qui réduirait l'espace de recherche exploré. Le rapport complexité / précision / pertinence des solutions dépend donc du degré d'exploration que l'on cherche à obtenir.

Au stade de développement actuel, l'outil permet la compilation d'une spécification décrite dans le modèle de représentation et la compilation d'un fichier technologique. L'estimation système et le parser C vers HCDFG sont en cours de développement. L'outil est développé en langage Java et met en oeuvre la plupart des algorithmes définis pour l'estimation architecturale. Dans sa version actuelle, on peut estimer un *HCDFG* sur une technologie cible donnée. La période d'horloge est actuellement définie comme étant égale à la latence de l'opérateur le plus lent. Toutefois, on peut choisir une fréquence d'horloge quelconque car les algorithmes d'ordonnancement utilisés (*Force Directed Scheduling* et *List Scheduling*) supportent un ordonnancement multi-cycle. L'allocation de ressources dans cette version associe à une opération donnée la première unité fonctionnelle qui réalise l'opération dans le fichier technologique (l'allocation peut donc être modifiée manuellement). Il reste à intégrer l'estimation mémoire à la structure de données de l'outil et à écrire les algorithmes d'exploration de plusieurs allocations et fréquences d'horloge.

Chapitre 7

Conclusions et perspectives

Aujourd'hui, la maturité des technologies programmables offre une réelle alternative à l'utilisation des ASICs. En enffet, pour des systèmes intégrant plus de 100.000 portes, un concepteur sur trois vise l'utilisation de composants programmables comme cible finale. Les performances offertes par ces circuits (10 millions de portes équivalentes, horloge interne à 300 MHz, entrées / sorties à des vitesses de l'ordre du Gbits / s, technologie 0.15 μm avec huit niveaux de métallisation) et les perspectives d'évolution attendues conduisent à la définition des SORCs (*System on Reconfigurable Chip*). Ces circuits émergeant intègrent un ou plusieurs coeurs de processeurs (ARM9 pour Excalibur d'Altera, 4 power PC pour Virtex II Pro de Xilinx), des mémoires et des unités matérielles spécifiques (configurées sur la matrice programmable). Cependant, ces évolutions technologiques sont récentes et n'ont pas été suivies par la définition de méthodologies de conception et par des outils associés. Aussi, pour appréhender efficacement la conception de tels systèmes, où le concepteur est confronté à un espace de conception très vaste au sein duquel il est difficile de converger vers les solutions les plus adaptées, il est nécessaire d'imaginer de nouvelles approches de conception. Face à ce problème, nous proposons une méthode d'exploration des solutions architecturales matérielles qui se basent sur l'utilisation de techniques d'estimation.

Objectifs

Le travail présenté dans ce mémoire s'intéresse donc à l'estimation des performances et de la surface pour des applications décrites à un niveau comportemental (donc dès les phases de spécification) et implantées sur des composants reconfigurables (les FPGAs). Pour apporter une réponse intéres-

sante à ce problème complexe, nous nous sommes fixé les objectifs suivants :

1. Proposer une méthode considérant une application pouvant être décrite de façon hiérarchique, contenant des structures de contrôle, intégrant du parallélisme potentiel et pouvant manipuler des données multidimensionnelles. Ces caractéristiques sont nécessaires pour décrire les applications de traitement du signal et des images.
2. Proposer une méthode qui prenne en compte les unités de traitement, de contrôle et de mémorisation pour une caractérisation complète du système.
3. Proposer une méthode qui permette d'explorer *automatiquement* plusieurs solutions architecturales afin d'offrir au concepteur différentes alternatives d'implantation.
4. Proposer une méthode qui puisse s'étendre à plusieurs composants programmables et qui ne soit pas spécifique à une seule famille.
5. Proposer une méthode de complexité réduite par rapport aux méthodes de synthèse architecturale afin de permettre l'évaluation de plusieurs architectures et de plusieurs composants.
6. Proposer une méthode qui puisse fonctionner de façon indépendante mais qui puisse également s'inscrire dans un flot de conception conjointe plus global, et notamment, celui développé au LESTER.

Ces objectifs ont été atteints et un outil logiciel automatisant le processus d'exploration a été développé au cours de ces travaux.

BILAN

La méthode proposée s'articule autour de deux points principaux : les estimations structurelles et les estimations physiques. Les estimations structurelles ont pour rôle de définir les solutions architecturales. Elles se basent sur l'ordonnancement des blocs de base de la spécification (*DFGs*). Les niveaux de hiérarchie sont progressivement analysés par des combinaisons qui dépendent du type de contrôle (structures conditionnelles, structures itératives) et du type de dépendance d'exécution (exécution séquentielle, exécution concurrente). Au final, on obtient l'estimation locale (des sous graphes) et globale (du système) du nombre de ressources, d'accès mémoire et d'états en fonction du nombre de cycles d'horloge alloués. La deuxième étape, dite

d'estimation physique, fournit alors l'estimation des performances et de l'occupation des ressources du FPGA pour les différentes solutions architecturales. Les caractéristiques des unités de mémoire, de traitement et de contrôle sont calculées à partir de la description du composant cible contenue dans le fichier technologique. À l'issu de ces deux étapes, on obtient plusieurs solutions architecturales (caractérisées en terme d'allocation de ressources, de nombre d'états, ...) et une estimation du taux d'occupation du FPGA et de la vitesse d'exécution exprimée en secondes. La validation de cette méthode a été réalisée sur deux des composants les plus utilisés actuellement dans le domaine des FPGAs (Virtex et Apex).

Un outil a été développé au cours de la thèse afin d'appliquer la méthode sur des exemples complexes. L'application de la méthode sur des exemples révèlent deux grandes constatations :

- D'abord, la complexité volontairement réduite du processus d'estimation par rapport au processus réel de synthèse conduit à certaines limites sur la pertinence des solutions architecturales définies. Celles-ci ne sont donc pas toujours optimales (partage de ressource pour de la concurrence inter HCDFG, génération des adresses, ...) du fait de la simplicité du principe des estimations structurelles (ordonnancement des blocs de base + combinaisons). Notre objectif principal était de permettre l'exploration d'un très vaste espace de recherche, ce qui explique que l'accent ait été mis sur la réduction de la complexité, au détriment de la définition de solutions plus approfondies. Toutefois, les valeurs fournies correspondent toujours à des solutions intégrables, qui fournissent alors un ordre d'idée sur le degré de parallélisme (et le compromis performance / occupation correspondant) à partir duquel une solution de départ peut être affinée, en utilisant un outil de synthèse d'architecture par exemple.
- Un certain nombre d'améliorations sont possibles, en particulier au niveau de la définition des solutions architecturales. La contrepartie de ces améliorations est qu'elle réduit le champs d'exploration à cause de l'augmentation de la complexité. Tout dépend du degré d'exploration souhaité : plus il est réduit, plus les solutions peuvent être pertinentes. Ainsi, une perspective intéressante pourrait se baser sur la définition de solutions architecturales plus fines, en se rapprochant de la façon de

procéder des outils de synthèse d'architecture. Les solutions se limiteraient toujours à un dénombrement des ressources, ce qui permettrait de garder une complexité nettement plus faible et donc d'explorer plusieurs solutions. L'utilisation conjointe d'un outil de synthèse de haut niveau en aval permettrait alors l'exploration et la synthèse très rapide du système. Ceci laisse entrevoir la possibilité d'explorer plus ou moins finement en fonction du degré d'abstraction auquel on se situe (système, RTL).

Notre choix s'est porté sur l'aspect exploration plus que sur l'aspect optimisation / pertinence des solutions architecturales, en particulier à cause du domaine d'application (le codesign). L'objectif ici n'était clairement pas de faire de la synthèse d'architecture. Au regard des résultats, les temps d'analyse (exploration + estimation) sont significatifs étant donné le nombre de solutions traitées, et les valeurs d'estimation cohérentes, ce qui prouve l'intérêt d'une telle méthode, malgré le niveau d'abstraction élevé.

Synthèse vs Estimation

Toutefois, la méthode proposée à l'issue de cette thèse peut être vue comme un outil de synthèse d'architecture simplifié. En effet, une phase d'allocation est réalisée, ainsi qu'un ordonnancement. La différence par rapport à un outil de synthèse de haut niveau se situe dans le fait qu'on s'arrête aux étapes d'allocation et d'ordonnancement et qu'on ne réalise pas la description détaillée des connexions entre les différentes unités du système. Par ailleurs, l'étape d'ordonnancement effectuée est simplifiée puisqu'elle ne concerne que les DFGs de la spécification. Le gain en complexité permet l'exploration automatique d'un vaste espace de conception. Sur un exemple complexe tel que celui de la transformée en ondelettes, l'outil explore automatiquement 341 solutions en 5 minutes , et fournit de plus les estimations de coût et de performances là où un outil de synthèse d'architecture ne génère qu'une description au niveau RTL et pour une seule solution architecturale. D'autre part, l'approche de type projection employée pour obtenir les caractéristiques physiques des solutions architecturales permet une réduction très sensible des temps d'analyse (il faut deux jours pour effectuer la synthèse logique de la transformée en ondelettes contre cinq minutes pour l'exploration / estimation des 47 solutions architecturales retenues). Le gain en complexité permet alors l'exploration du parallélisme, de tester l'effet de plusieurs allocations

de ressources et de fréquences d'horloges, ainsi que l'évaluation de plusieurs composants d'implantation pour des temps de calculs raisonnables.

À partir des valeurs d'estimation de coût et de performances, le concepteur peut effectuer le choix d'une solution architecturale adaptée à ses contraintes. Les informations fournies par l'outil d'exploration architecturale peuvent alors permettre de synthétiser la solution à l'aide d'un outil de synthèse de haut niveau. Ceci montre finalement la complémentarité de l'estimation par rapport à la synthèse : l'estimation permet l'exploration d'un très grand nombre de solutions sur plusieurs composants cibles, et la synthèse de haut niveau sa conception automatique. La conception y gagne alors à la fois en temps et en sûreté.

PERSPECTIVES

Ces travaux s'inscrivent dans la définition d'un environnement de codesign pour les SORCs. Un tel environnement vise à apporter une réponse efficace au problème de la conception des systèmes hétérogènes. La méthode développée durant cette thèse constitue un des éléments de cet environnement, qui repose sur les outils suivants :

- L'estimation système
- L'estimation logicielle (ARM9)
- L'estimation matérielle (Apex, Virtex, XC4000, FLEX)
- Le partitionnement (statique, dynamique)

L'objectif est d'obtenir pour chaque solution architecturale une caractérisation complète en temps / surface et consommation. Un certain nombre de travaux sont en cours sur ces différents sujets à l'heure actuelle.

L'application de cette méthode aux ASICs pourrait se baser sur les travaux de Narayan et Gajski [3]. Une première approche pourrait consister à définir un fichier technologique adapté à la description de ce type de composant. On pourrait alors directement appliquer la méthodologie d'exploration et apporter les corrections nécessaires en fonction des résultats d'estimation par rapport à ceux de la synthèse. Des extensions probables sont à prévoir en ce qui concerne la prise en compte des interconnexions à ce niveau.

Une autre perspective intéressante serait d'étendre les travaux aux architectures reconfigurables dynamiquement et ainsi proposer une méthode d'estimation qui permette l'exploration des différentes possibilités de placement des fonctions en tenant compte des temps de reconfiguration et des

contextes de fonctionnement des fonctions du système. Au final, cette méthode complète d'exploration architecturale pour des systèmes candidats à une intégration matérielle permettrait l'évaluation de plusieurs architectures sur des technologies de type ASIC, reconfigurable et reconfigurable dynamiquement.

Il serait également important de pouvoir estimer la consommation qui est un paramètre important pour les applications embarquées. La prise en compte de la consommation pourrait se baser sur la caractérisation de la consommation moyenne par accès des différentes ressources (opérateurs arithmétiques et logiques, mémoires, ...) et du nombre d'opérations à réaliser dans le graphe. Des travaux ont été démarré dans cette voie, mais n'ont malheureusement pas pu être finalisés.

L'outil d'estimation développé pendant la thèse est intégré à l'environnement de conception conjointe développé au laboratoire. Toutefois actuellement, l'estimation s'effectue de façon indépendante. Il est donc important de mieux définir les intéractions entre les différents outils afin d'aboutir à un flot de conception incrémental et cohérent. Au final, ce flot constituerait un outil d'exploration puissant permettant l'exploration d'une part au niveau système (découpage fonctionnel, répartition logiciel / matériel, ...), et d'autre part au niveau architectural (allocation des composants d'implantation, exploration du parallélisme, ...). Le vaste espace de conception ainsi analysé permettrait la définition d'une architecture optimale et son utilisation pour la conception de SOC (*System on Chip*). Ces deux dernières perspectives trouvent leur place dans le cadre du projet RNTL ÉPICURE [50] auquel collabore le LESTER.

Bibliographie

[1] D.D. Galski, N. Dutt, A. Wu and S. Lin, *High-Level Synthesis : introduction to Chip and System Design*, Kluwer Academic Publishers, 1992.

[2] J.P. Diguet, *Estimation de Complexité et Transformations d'Algorithmes de Traitement du Signal pour la Conception de Circuits VLSI*, Thèse, Université de Rennes I, 1996.

[3] S. Narayan and D.D. Gajski, *Area and Performance Estimation from System-Level Specifications*, Technical Report, University of California, 1992.

[4] S. Narayan and D.D. Gajski, *Rapid Performance Estimation For System Design*, EURO-DAC '96.

[5] A. Sharma and R. Jain, *Estimating Architectural Resources and Performance for High-Level Synthesis Applications*, IEEE Transaction on Very Large Scale Integration Systems, Vol 1, No 2, June 1993.

[6] M. Rabaey and M. Potkonjak, *Estimating Implementation Bounds for Real Time DSP Application Specific Circuits*, IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, Vol 13, No 6, June 1994.

[7] M. Rim and R. Jain, *Lower-Bound Performance Estimation for the High-Level Synthesis Scheduling Problem*, IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, Vol 13, No 4, April 1994.

[8] B. M. Pangrle and D. D. Gajski, *Slicer : a state synthetizer for intelligent silicon compilation*, Digest of the IEEE International Conference on Computer Aided Design, 1987.

[9] P. G. Paulin and J. P. Knight, *Force-Directed Scheduling for the behavioral Synthesis of ASIC's*, IEEE Transactions on Computer Aided Design, Vol 8, No 6, June 1989.

[10] F.J. Kurdahi and A. Parker, *Techniques for Area Estimation of VLSI Layouts*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol 8, No 1, January 1989.

[11] , S.Y. Ohm, F.J. Kurdahi, N. Dutt and M. Xu, *A Comprehensive Estimation Technique for High-Level Synthesis*, 1995.

[12] P. Chedmail, *étude sur FPGA et DSP de l'implantation de fonctions d'un décodeur MPEG-4*, Rapport Technique, LESTER, Juin 1999.

[13] V. Betz, J. Rose and A. Marquardt, *Architecture and CAD for Deep-Submicron FPGAs*, Kluwer Academic Publishers, 1999.

[14] M. Xu, *Linking High-Level Synthesis with Physical Design*, PhD, University of Califonia, Irvine, 1997.

[15] M. Xu and F.J. Kurdahi, *Area and Timing Estimation for Lookup Table Based FPGAs*, Proceedings Of ED&TC, March 1996.

[16] M. Xu and F. Kurdahi, *Layout Driven High-Level Synthesis for FPGA based Architectures*, Design and Test in Europe '98.

[17] R. Enzler, T. Jeger, D. Cottet, and G. Tröster, *High-level area and performance estimation of hardware building blocks on FPGAs* in Field-Programmable Logic and Applications (Proc. FPL'00), volume 1896 of Lecture Notes in Computer Science, pages 525-534. Springer, 2000.

[18] W. Miller and K. Owyang, *Designing a high performance FPGA – using the PREP benchmarks*, in Wescon'93 Conf. Record, pages 234-239, 1993.

[19] S. Kliman, *PREP benchmarks reveal performance and capacity tradeoffs of programmable logic devices*, in Proc. IEEE Int. ASIC Conf. and Exhibit (ASIC'94), pages 376-382, 1994.

[20] S. Brown and J. Rose, *Architecture of FPGAs and CPLDs : A Tutorial*, Department of Electrical and Computer Engineering, University of Toronto.

[21] Xilinx, *Virtex-II 1.5V Field-Programmable Gate Arrays*, July 2001.

[22] Altera, *APEX 20K Programmable Logic Device Family*, August 2001.

[23] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele and A. Vandecappelle, *Custom Memory Management Methodology*, Kluwer Academic Publishers, 1998.

[24] M. Balakrishnan, A.K. Majumdar, D.K. Banerji, J.G. Linders, J.C. Majithia, *Allocation of multiport memories in datapath synthesis*, IEEE Transactions on Computer-Aided Design of Integrated Circuits, 7 (4), April 1988.

[25] L. Stok, *Interconnect optimization during datapath allocation*, European Conference on Design Conference, March 1990.

[26] C.J. Tseng and D. Siewiorek, *Automated synthesis of data paths in digital systems*, IEEE Transactions on CAD, Vol 5, No 3, July 1986.

[27] M. Van Swaaij, F. Franssen, F. Catthoor, and H. De Man, *Automating high level control flow transformations for dsp memory management*, IEEE Press VLSI Signal Processing, November 1992.

[28] G. Grun, N. Dutt and F. Balasa, *System Level MemorySize Estimation*, Technical Report, University of California, 1997.

[29] J.P. Diguet, G. Gogniat, P. Danielo, M. Auguin and J.L. Philippe, *The SPF Model*, accepted at FDL, Tübingen, Germany, September 2000.

[30] S. A. Blythe and R. E. Walker, *Efficient Optimal Design Space Characterization Methodologies*, IEEE Transactions on Design Automation of Electronic Systems, Vol 5, No3, Jult 2000.

[31] R. K. Brayton and R. Spencer, *Sensitivity and Optimization*,Computer Aided Design of Electrnic Circuits, Elsevier Science Publishing Co, INC, 52 Vandervilt Avenue, New York, NY 10017, USA, 1984.

[32] G. De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill series in electrical and computer engineering, New York, NY, USA, 1994.

[33] M. Rahmouni and A.A. Jerraya, *Formulation and Evaluation Of Scheduling Techniques For Control Flow Graphs*, Euro DAC 1995.

[34] R. K. Gupta, *Co-Synthesis of Hardware and Software for Digital Embedded Systems*, Kluwer Academic Publishers, 1995.

[35] B. Tabbara, A. Tabbara and A. Sangiovanni-Vincentelli, *Function / Architecture Optimization and Co-design of Embedded Systems*, Kluwer Academic Publishers, 2000.

[36] T-Y Yen and W. Wolf, *Hardware-Software Co-Synthesis of Distributed Embedded Systems*, Kluwer Academic Publishers,1996.

[37] H. Thomas, J.P. Diguet and J.L. Philippe, *A methodology for an Application Profiling at a System Level*, SiPS, Taïwan, October 1999.

[38] F. Vahid and D. D. Gajski, *Closeness metrics for system-level functional partitioning*, EDAC, Sept 95, pp328-333.

[39] R. Camposano and R. Brayton, *Partitioning before logic synthesis*, ICCAD, 1987.

[40] E. Lagnese and D. Thomas, *Architectural partitioning for system level synthesis of integrated cicuits*, IEEE Transactions on CAD, July 1991.

[41] X. Xiong, E. Barros and W. Rosentiel, *A method for partitioning UNITY language in hardware and software*, EuroDAC, 1994.

[42] L. Guerra, M. Potkonjak and J. Rabaey, *System-level design guidance using algorithm properties*, IEEE workshop on VLSI Signal Processing, 1994.

[43] J.P. Diguet, O. Sentieys, J.L Philippe and E. Martin, *Probabilistic Resource Estimation for pipeline architecture*, IEEE workshop on VLSI Signal Processing, October 1995.

[44] J. Madsen, J. Grode, P.V. Knudsen, M.E. Petersen and A. Haxthausen, *LYCOS : the Lyngby Co-Synthesis System*, Design Automation for Embedded Systems, 1997.

[45] J. Henkel and R. Ernst, *A Hardware / Software Partitioner Using a Dynamically Determined Granularity*, 34th DAC, Anaheim, June 1997.

[46] Rec. G722, *Codage audiofréquence à 7KHz à un débit inférieur ou égal à 64kbit/s*, Melbourne, 1988.

[47] I. Daubechies, *The Wavelett Transform, time-frequency localization and signal analysis*, IEEE Transactions on Information Theory, vol. 36, No 5, September 1990.

[48] F. Truchet, *Ondelettes pour le signal numérique*, Ed. Hermes, 1998.

[49] M. Antonini, *Transformée en ondelettes et compression numérique des images*, Thèse de l'université de Nice (Sophia Antipolis), Septembre 1991.

[50] CEA-LETI, LESTER-Université Bretagne Sud, I3S-Université Nice Sophia-Antipolis, THOMSON-CSF Communications, SIMULOG, *Projet EPICURE : Environnement de PartIonnement et de Co-développement pour Utilisation sur architectures REconfigurables*, Réseau National de Technologies Logicielles 20/10/2000.

Publications personnelles :

[51] S. Bilavarn, G. Gogniat and J.L. Philippe, *Estimation d'Architectures Hétérogènes pour la Conception Conjointe Logicielle / Matérielle*, Colloque CAO'99, Aix-en-Provence, Mai 1999.

[52] S. Bilavarn, G. Gogniat and J.L. Philippe, *A Hardware-Software Codesign Methodology for Heterogeneous Architecture Estimation*, ICSPAT'99, Orlando, November 1999.

[53] S. Bilavarn, J.P. Diguet, G. Gogniat, Y. Le Moullec and J.L. Philippe, *Méthode de Conception d'Architectures Hétérogènes pour les Applications de Traitement Numérique du Signal*, JNRDM 2000, Montpellier, Mai 2000.

[54] S. Bilavarn, G. Gogniat and J.L. Philippe, *FPGA Area Time Power Estimation for DSP Applications*, ICSPAT 2000, Dallas, Otober 2000.

[55] S. Bilavarn, G. Gogniat and J.L. Philippe, *Area Time Power Estimation for FPGA Based Designs at a Behavioral Level*, ICECS 2000, Beyrouth, December 2000.

[56] S. Bilavarn, G. Gogniat et J.L. Philippe, *Estimation de performances à un niveau comportemental pour l'implantation sur composants FPGA*, Sympa'7, Paris, Avril 2001.

Résumé : Un facteur important dans l'évolution des systèmes électroniques modernes est l'apparition de nouvelles architectures basées sur la programmation de circuits matériels tels que les composants programmables. Les récentes évolutions des différentes familles autorisent aujourd'hui l'intégration de systèmes de plus en plus complexes avec des contraintes de performances de plus en plus fortes. D'autre part, la flexibilité offerte par ce type de technologie fait des FPGAs (*Field Programmable Gate Arrays*) une cible architecturale promise à un bel avenir.

L'évaluation des performances d'une application sur une technologie reconfigurable est un problème peu étudié à ce jour. Jusqu'à présent, les chercheurs ont principalement porté leurs efforts sur l'amélioration des architectures afin de les rendre plus performantes et ainsi constituer une réelle alternative aux ASICs (*Application Specific Integrated Circuits*). L'objectif du travail présenté dans ce mémoire consiste à proposer des techniques et les outils associés permettant l'évaluation rapide des performances (temps, surface) d'applications sur des architectures programmables. La méthode développée est générique (elle s'applique à plusieurs familles de FPGAs) et se situe au niveau comportemental. Elle permet l'exploration de plusieurs solutions architecturales et s'intègre dans un flot de conception conjointe logiciel / matériel.

Mots clés : Estimation temps / coût, exploration architecturale, FPGAs, conception conjointe logiciel / matériel, synthèse de haut niveau.

Abstract : A significant factor in the evolution of modern electronic systems is the appearance of new architectures based on the programming of hardware components such as Field Programmable Gate Arrays (FPGAs). The introduction of those components as an alternative computation unit and the flexibility they offer increase the interest of such an integration solution. Moreover, the recent evolutions of the different families allow today the implementation of complex systems with higher constraints on performances.

Few works focus on the estimation of an application on a technology of this type. Until now, researchers mainly carried their efforts on the improvement of reconfigurable architectures in order to make them more powerful and thus to constitute a real alternative to ASICs (*Application Specific Integrated Circuits*). The objective of the work presented in this thesis is to propose techniques and tools associated allowing fast evaluation of performances (area vs execution time) of applications on programmable architectures. The developed method is generic (it can be applied to several FPGA families) and is located at the behavioral level. It allows the browsing of several architectural solutions and is integrated in a hardware / software codesign methodology.

Keywords : Performance and cost estimation, architectural exploration, FPGAs, hardware / software codesign, high level synthesis.