Métaheuristiques pour la synthèse de haut niveau

Kods Trabelsi

THESE / UNIVERSITE DE BRETAGNE-SUD
sous le sceau de l'Université européenne de Bretagne
pour obtenir le titre de
DOCTEUR DE L'UNIVERSITE DE BRETAGNE-SUD
Mention : STIC
Ecole doctorale SICMA

présentée par

Kods TRABELSI

Laboratoire des sciences et technologies de l'information, de la communication et de la connaissance (Lab-STICC) CNRS UMR 6285

Métaheuristiques pour la synthèse de haut niveau

Thèse soutenue le 1er décembre 2009

devant le jury composé de :

Mme Alix MUNIER KORDON
Professeur, Université Paris 6 / Rapporteur

M. Emmanuel CASSEAU
Professeur, ENSSAT, Université de Rennes 1 / Rapporteur

M. Kenneth SÖRENSEN
Professeur, Université d'Anvers, Belgique / Président du jury

M. André ROSSI
Dr. Université de Bretagne-Sud / Examinateur

M. Marc SEVAUX
Professeur, Université de Bretagne-Sud / Directeur de thèse

M. Philippe COUSSY
Dr. Université de Bretagne-Sud / Directeur de thèse

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Contexte

Sommaire

Dans ce premier chapitre, nous introduisons les deux domaines à la croisée desquels se situent ces travaux de thèse. D'un coté, nous présentons le contexte de la conception des circuits électroniques avec l'évolution des applications implémentées, l'évolution des technologies de gravure et les solutions dédiées pour accompagner ces évolutions. De l'autre nous adressons le domaine de l'optimisation combinatoire et ce qu'elle peut offrir aux problèmes de grande complexité. Enfin, dans la dernière section de ce chapitre, nous situons les travaux de cette thèse.

1.1 Introduction

Ces travaux de thèse se situent à la croisée de deux domaines, la conception des circuits électroniques et l'aide à la décision. Dans ce chapitre nous allons préciser les contextes des deux domaines, chacun de son coté pour pouvoir situer nos travaux de recherche. Une première section présentera donc l'évolution des applications implémentés sur les circuits numériques, l'évolution technologique et une énumération non exhaustive des nouvelles méthodologies de conception. Une deuxième section s'intéressera au domaine de l'aide à la décision appelé aussi recherche opérationnelle, en introduisant les deux grandes familles de méthodes qu'offre l'aide à la décision pour résoudre les problèmes d'optimisation combinatoire. Enfin, la dernière section définira l'objectif de ces travaux.

1.2 Conception des circuits intégrés

1.2.1 Évolution des besoins et des applications

Grâce à l'évolution du marché de la technologie numérique, de la téléphonie mobile et du multimédia, les industriels proposent aujourd'hui dans un même produit toujours plus compact et plus autonome, de plus en plus de fonctionnalités. Les produits doivent être conçus et fabriqués dans des délais de plus en plus court, afin d'imposer les biens et de résister à la compétitivité. L'écho de la commercialisation des derniers smart phones, téléphones mobiles couplés à des PDA témoigne de la rude concurrence dans ce domaine.

Plus de fonctionnalités dans un produit plus compact et plus autonome se traduit concrètement par des applications hétérogènes et très complexes implémentées dans des circuits intégrés de tailles réduites et basse consommation. En effet, le concepteur doit gérer, d'un coté la complexité des applications qui ne cesse d'augmenter, mais aussi l'hétérogénéité des diverses applications à implémenter dans un même circuit. Par exemple, dans les nouveaux téléphones portables, un même circuit doit pouvoir gérer plusieurs protocoles de communication (Wifi, Edge, WCDMA, 3G), plusieurs normes d'encodage/décodage audio et vidéo (MPEG, H264, JPEG2000, OGG, MP4, AAC, etc).

Ainsi la conception des circuits intégrés doit prendre en compte :
1) l'accroissement continu de la complexité des applications,
2) l'hétérogénéité des applications,
3) la question de l'implémentation matérielle ou logicielle des applications,
4) la taille et la consommation du circuit,
5) et l'accès rapide au marché.

De ce fait, les technologies des semi-conducteurs ainsi que les méthodologies de conception sont amenées à évoluer.

1.2.2 Évolution technologique et architecturale

Parallèlement à la demande accrue en puissance, en miniaturisation et en autonomie, les innovations technologiques en matière de semi-conducteurs n'ont cessé de proposer des solutions à ces exigences. L'évolution technologique repose sur deux aspects : la diminution conséquente des dimensions des structures en silicium réalisables sur un circuit intégré et l'offre de diverses solutions d'implémentation qui s'adaptent aux caractéristiques de la classe d'applications considérée.

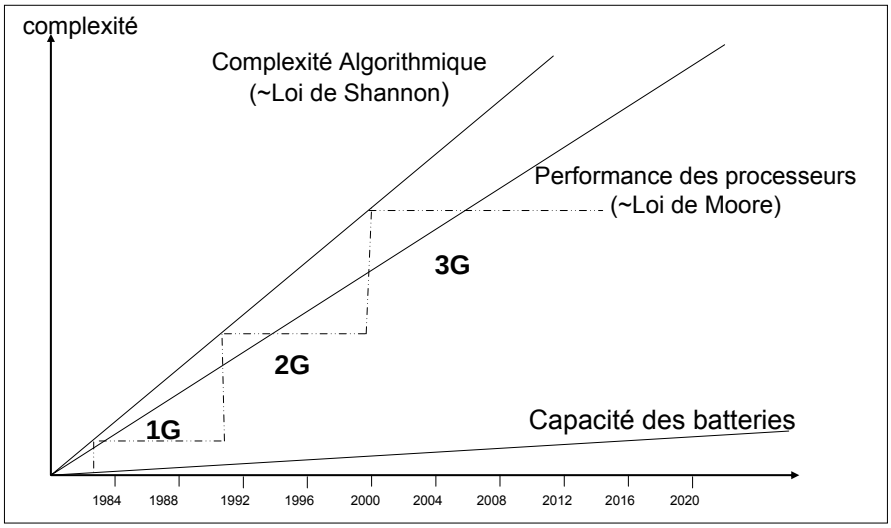


FIGURE 1.1 – Complexité algorithmique *vs.* performance des processeurs

1.2.2.1 La finesse de gravure

L'échelle d'intégration, appelée aussi finesse de gravure qui trduit le nombre de portes logiques par mm^2, est passée de douze pour la première génération de circuits intégrés (1960) à plusieurs dizaines de millions de portes pour la dernière génération. En effet, le nombre de transistors sur une puce a régulièrement doublé tous les dix-huit mois, depuis l'invention du premier circuit intégré. Cette évolution a été prédite par la loi de Moore (Figure 1.1), une règle empirique publiée par Electronics Magazine en 1965 [Moore 1998], qui ne s'était jamais démentie. Cette avancée technologique a déclenché l'invention d'une panoplie de solutions architecturales.

1.2.2.2 Taxonomie des circuits

Plusieurs solutions architecturales ont été proposées pour l'implémentation d'une application donnée (Figure 1.2), les deux extrêmes : une solution purement logicielle qui s'exécute sur

un microprocesseur et une solution purement matérielle qui se traduit par un circuit dédié à l'exécution de l'application.
L'objectif de cette section est de présenter brièvement les différentes solutions architecturales qui ont été proposées jusque là.

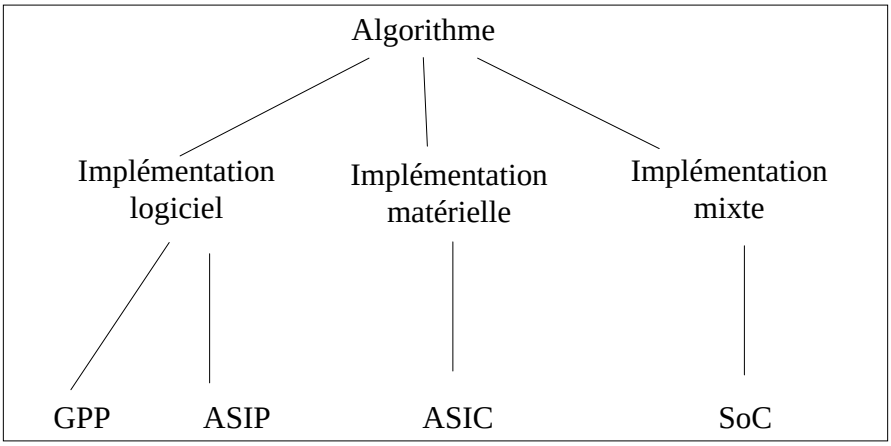


FIGURE 1.2 – Différentes solutions architecturales

L'implémentation logicielle Le microprocesseur généraliste (GPP) offre une plateforme pour l'implémentation logicielle des applications. Les principales caractéristiques d'un microprocesseur sont :

- le nombre de bits que le processeur utilise pour représenter une variable. Les microprocesseurs actuels peuvent traiter des nombres codés sur 64 bits ou plus.
- la vitesse de l'horloge : le rôle de l'horloge est de cadencer le rythme du travail du microprocesseur.
- le jeu d'instructions qu'il peut exécuter : additionner, comparer, multiplier deux nombres, etc.

La combinaison des caractéristiques précédentes détermine la puissance de calcul du microprocesseur.

La figure 1.1 montre l'évolution de la performance des processeurs en suivant la loi de Moore et celle de la complexité algorithmique dans le domaine des télécommunications en suivant la loi de Shannon [Rabaey 2000]. La superposition des courbes révèle la croissance plus rapide de la complexité des algorithmes par rapport à la croissance de la performance des processeurs. Ainsi, la solution purement logicielle n'est pas toujours appropriée.

L'implémentation matérielle Un ASIC est un circuit intégré spécialisé. En général, il regroupe un grand nombre de fonctionnalités uniques et/ou sur mesure. L'intérêt des ASICs est de réduire le coût en silicium et d'augmenter la fiabilité du circuit. Cependant, cette

solution architecturale a un inconvénient majeur qui est le coût de développement élevé voire très élevé, dû au coût de la fabrication des masques de gravure.
Dans cette approche, le coût élevé est dû à l'intervention du fondeur dans la phase de conception. En général, celui-ci ne souhaite pas intervenir dans la phase de conception ; sa tâche étant de réaliser le composant à partir des masques. Ainsi pour réduire les surcoûts dus aux modifications, il s'avère nécessaire d'être rigoureux lors de la phase de développement de telle sorte que le circuit prototype fonctionne dès les premiers essais, ce qui n'est pas du tout trivial.
De ce fait, la production d'ASIC est généralement dédiée pour des gros volumes (>100 000 pièces par an), sauf lorsque le sur-mesure est indispensable. Dans ce cas, le fabricant rattrape le coût de son investissement avec un prix de vente plus élevé.
L'implémentation sur processeur généraliste et celle sur circuit dédié des applications ne sont pas toujours des solutions réalisables vu les limites calculatoires des processeurs et le coût très élevé des ASICs. Ainsi des solutions architecturales plus adaptées ont été proposées.

La solution de compromis Un ASIP [Henkel 2007, Stojcev 2005, Jain 2001] est un processeur conçu pour une classe d'applications particulière. Ainsi l'ASIP est un compromis entre le processeur généraliste et l'ASIC. En effet, des instructions inutiles peuvent être supprimées ou au contraire le jeu d'instructions peut être enrichi d'instructions spécifiques à l'application, par exemple, une opération (mac R1, R2, R3) qui multiplie et accumule (R1 += R2*R3). Du fait de sa spécialisation , un ASIP peut ainsi exploiter les caractéristiques spécifiques d'un ensemble d'applications pour atteindre la performance, le coût architectural et la consommation de puissance requis.

L'implémentation mixte Pour l'implémentation d'applications hétérogènes sur une même puce il y'a ce que l'on appelle les systèmes sur puce [D'silva 2005] ou encore systèmes mono-puce ou SoC (Figure 1.3). En effet, l'évolution phénoménale de la densité d'intégration a permis aussi la réalisation de systèmes complexes sur une seule puce. Un SoC désigne un système complet embarqué sur une puce, pouvant comprendre plusieurs blocs hétérogènes, entre autres, de la mémoire (data/code), un microprocesseur, des accélérateurs matériels, des périphériques d'interface, ou tout autre composant nécessaire à la réalisation de la fonction attendue. Un SoC pouvant contenir un ou plusieurs blocs reconfigurables s'appellera RSoC, s'il peut aussi contenir plusieurs microprocesseurs dans ce cas on l'appellera MPSoc.

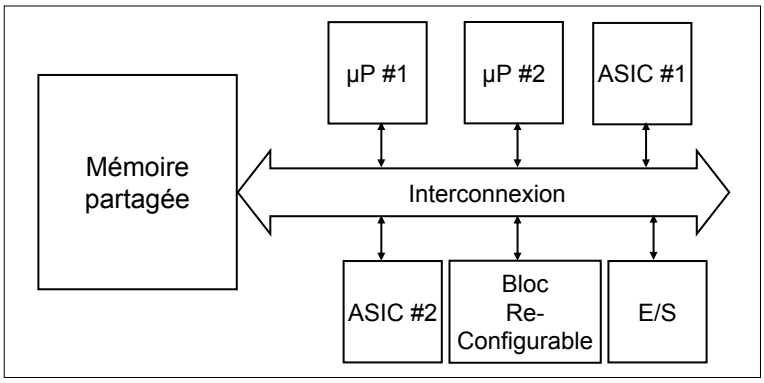


FIGURE 1.3 – Schéma d'un système mono-puce

L'ASSP est un circuit électronique intégré regroupant un grand nombre de fonctionnalités pour satisfaire une application généralement standardisée : par exemple l'ASSP pour GSM (OMAP processeur mobile ARM fabriqué par Texas Instruments), issu d'un fabricant unique, est utilisé comme circuit de base par les différents fabricants de téléphones portables. Ainsi, les ASSPs sont généralement des SoCs.

Les circuits logiques reprogrammables Le est un circuit logique reprogrammable, est une plateforme pour l'implémentation matérielle, ou logicielle ou mixte. Cette solution architecturale est devenue la plateforme matérielle la plus courante pour l'implémentation des algorithmes à calcul intensif [Brown 1996]. Il est largement utilisé pour des applications (TDSI) .

Les deux intérêts majeures des FPGAs sont la souplesse de programmation et le coût de conception réduit. En effet, les FPGAs sont composés de nombreuses cellules logiques élémentaires librement assemblables dont les fonctionnalités seront déterminées par les configurations qui leurs sont appliquées. Le coût réduit de la conception des FPGAs, permet de multiplier les essais et d'optimiser l'architecture développée. C'est effectivement, pour cette raison, la testabilité, que les FPGAs sont largement utilisés. Ainsi, bien qu'ils n'atteignent pas la performance des circuits dédiés, les FPGAs sont capables d'implémenter des algorithmes de calcul intensif tout en ayant un coût de conception raisonnable.

Bilan La figure 1.4 [Rabaey 2000] illustre le compromis performance/flexibilité de toute les solutions architecturales définies précédemment. Cette illustration simplifie le choix de la plateforme d'implémentation en fonction des besoins en performance et en flexibilité.

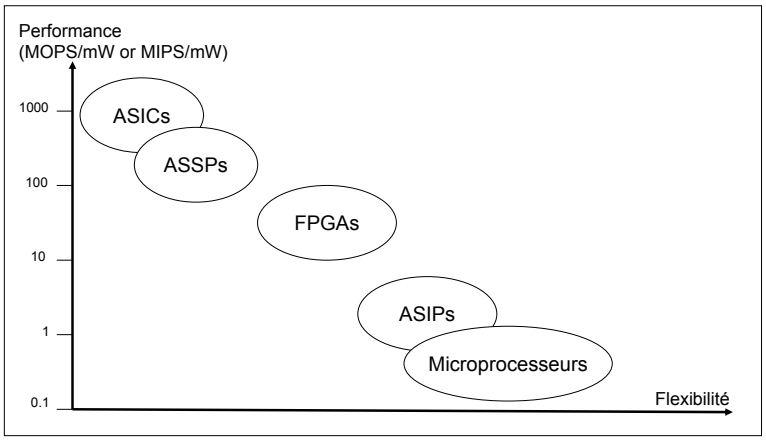


FIGURE 1.4 – Performance *vs.* Fléxibilité

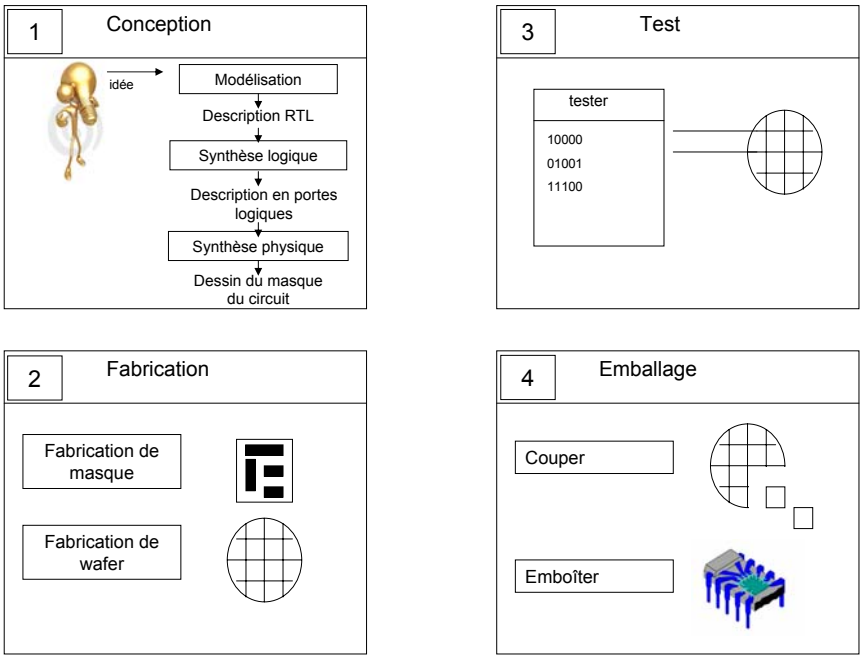


FIGURE 1.5 – Cycle de développement d'un circuit intégré

1.2.3 Évolution des méthodologies de conception

Avec la maturité des technologies VLSI , l'industrie a commencé à s'intéresser au cycle de développement du produit (Figure 1.5) [Micheli 1994] afin de gagner en productivité et en compétitivité. En effet, le processus technologique autorise un accroissement de la complexité de 59% par an, alors que l'efficacité des concepteurs n'augmente que de 25% par an (Figure 1.6) [itr 2007]. Comme le montre le tableau, tiré de [Urard 2007] , avec une technologie de 32 nm

et un taux de 200.000 portes par concepteur, il faut compter 300 HommeAn pour developper un circuit intégrant 60 millions de portes logiques. Ce besoin exponentiellement croissant en nombre de concepteurs coûte excessivement cher à l'industrie de la microélectronique. De ce fait, l'automatisation du processus de conception et de la spécification du silicium était devenu nécessaire pour rattraper l'écart imposé par l'évolution technologique.

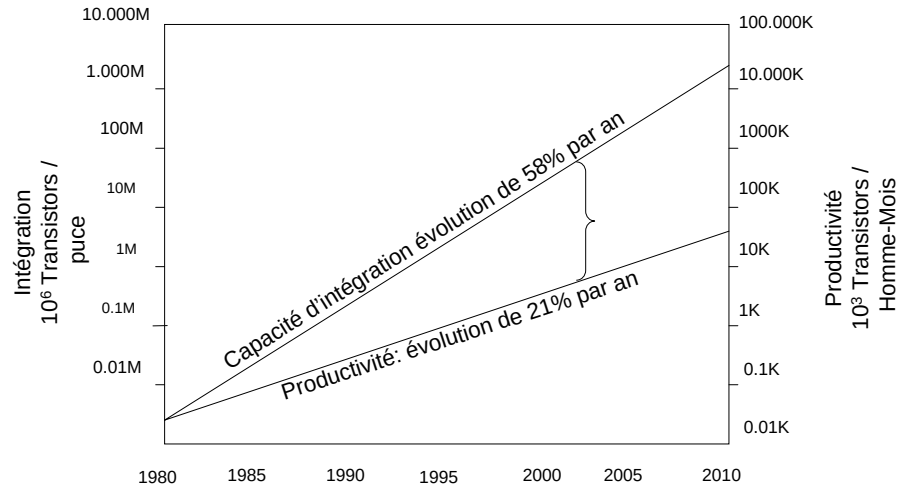


FIGURE 1.6 – Productivité *vs.* Capacité d'intégration

La première tâche du concepteur consiste à trouver un modèle qui va représenter fidèlement le comportement de l'application. Cette représentation se fait à l'aide d'un langage de description matériel HDL tel que *SystemC*, *Verylog* et *VHDL* ou d'un langage de programmation simple tel que *C*.

L'étape suivante est le passage du niveau algorithmique au niveau transfert de registres RTL. La description RTL profile le circuit en composants (opérateurs, registres et multiplexeurs) interconnectés. On trouve, par la suite, l'étape de synthèse logique qui consiste à raffiner le modèle RTL en une représentation plus détaillée en portes logiques. Cette phase est assez longue, et peut durer des heures en fonction de la complexité des applications. Enfin, la phase de synthèse physique traduira à son tour le modèle logique en un modèle plus raffiné à base de transistors afin d'obtenir le dessin du masque du circuit.

Les outils commerciaux d'aide à la conception de circuits ont essayé de suivre l'évolution technologique en gagnant, tout les dix ans à peu prés, un niveau d'abstraction supplémentaire. Ainsi ont été commercialisé successivement, des outils de dessin de masques, puis des outils de synthèse logique (niveau portes) et enfin, à l'heure actuelle, des outils de synthèse travaillant sur le passage du niveau algorithmique au niveau transfert de registres.

TABLE 1.1 – Évolution du nombre de concepteurs

1992	1994	1996	1998	2000	2002	2004	2006	2008	2010
Finesse de gravure(nm)									
0,6	0,5	0,35	0,25	0,18	0,13	90	65	45	32
# de portes / mm^2									
1k	5k	15k	30k	45k	80k	150k	300k	600k	1,2M
# de portes / die($50mm^2$)									
50k	250k	750k	1,5M	2,25M	4M	7,5M	15M	30M	60M
# de portes / concepteur / an									
4k	6k	9k	40k	56k	91k	125k	200k	200k	200k
Homme.Année / die($50mm^2$)									
~10	~40	~80	~40	~40	~43	~60	~75	~150	~300

1.2.3.1 La synthèse de haut niveau

Les outils de synthèse de haut niveau automatisent la transformation d'une description algorithmique en *Langage C* par exemple, en une description RTL structurelle [Gajski 1992]. Jusque là, cette étape se faisait à la main. Cependant, récemment les industriels commencent à faire confiance aux outils de synthèse de haut niveau. Cette automatisation permettra la réduction du cycle de conception, l'exploration d'un espace de solutions plus large et enfin pour des circuits de grande complexité de surpasser les concepteurs humains. Plus de détails sur la synthèse de haut niveau seront donnés dans le chapitre II.

1.2.3.2 La conception matérielle/logicielle

La conception matérielle logicielle appelée aussi conception conjointe ou encore Hw/Sw Co-Design est le développement conjoint de la partie matérielle et de la partie logicielle d'une application [Micheli 2001, Ernst 1998, Wolf 2003].

La figure 1.7 illustre le flot de la phase de conception conjointe. Comme le montre cette figure, les outils de Co-Design permettent une exploration rapide pour chacune des fonctions composants l'application considérée de plusieurs solutions architecturales (FPGA, ASIC, ASSP, etc.). En effet, ces outils partent d'une spécification d'une application composée de plusieurs fonctions, et s'appuient sur des métriques tels que le parallélisme moyen, le taux d'accès à la mémoire, les contraintes de communication, etc. pour choisir une solution architecturale pour chacune des fonctions de l'application. Pour finir, les outils de Co-Design synthétisent des interfaces de communication entre les architectures.

1.2.3.3 Les composants virtuels

Les composants virtuels appelés aussi IP permettent la réutilisation des composants préconçus. Ce concept bien connu dans le domaine de la conception logiciel, s'applique depuis quelques

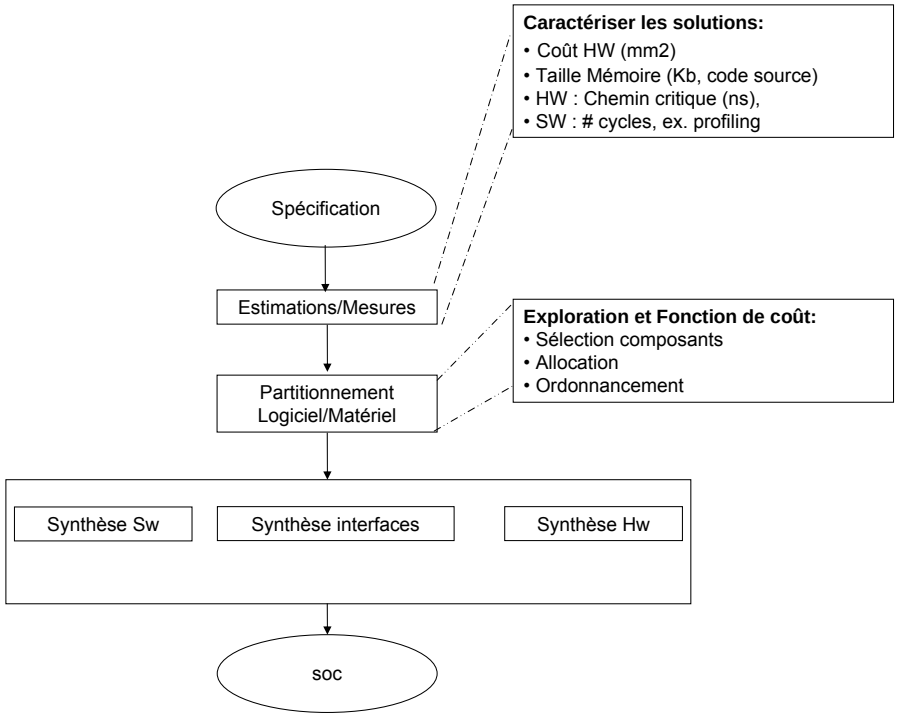


FIGURE 1.7 – Le flot de la conception matérielle logicielle

années à la conception matérielle. Il s'agit de construire une bibliothèque qui contiendra des blocs susceptibles d'être réutilisés.

L'IP peut prendre plusieurs formes : algorithmique [Coussy 2003] , logiciel, firm [Nekoogar 2003] ou encore matérielle. Ces formes ont des niveaux de réutilisabilité différents. En effet, moins l'IP contient de détails architecturaux plus il est flexible et a un niveau de réutilisation élevé. La conception d'IPs peut se faire manuellement ou en utilisant les outils de synthèse de haut niveau.

Un composant virtuel réalise, en général, une fonction non triviale, mais à usage suffisamment générique pour être réinvesti dans un grand nombre d'applications. L'IP doit pouvoir être aisément réplicable à l'aide des outils d'intégration du commerce, en d'autres termes, il doit pouvoir être intégré à un système sans la présence de son concepteur et surtout avoir un coût de réutilisation nettement inférieur à celui de sa re-conception.

1.3 La recherche opérationnelle

La recherche opérationnelle (RO), aussi appelée aide à la décision, propose des modèles conceptuels pour analyser des situations complexes et permet aux décideurs de faire les choix les plus efficaces. Ces situations dites complexes peuvent être formulées en problèmes d'optimisation combinatoire. Un problème d'optimisation combinatoire est défini par un ensemble

d'instances. A chaque instance du problème est associé un ensemble discret de solutions incluant un sous-ensemble représentant les solutions admissibles (réalisables) et une fonction de coût qui associe à chaque solution réalisable, un nombre réel évaluatif.

Résoudre un tel problème consiste à trouver une solution optimisant la valeur de la fonction de coût. Une telle solution s'appelle une solution optimale ou un optimum global. Plusieurs méthodes de résolution des problèmes d'optimisation combinatoire existent. Ces méthodes de résolution suivent quatre approches différentes pour la recherche d'une solution : l'approche de construction, l'approche de relaxation, l'approche de voisinage et l'approche d'évolution [Hao 1999].

- L'approche de construction consiste à construire pas à pas une solution. Partant d'une solution partielle initialement vide, elle cherche à étendre à chaque étape la solution partielle de l'étape précédente.
- L'approche de relaxation repose sur l'affranchissement de quelques contraintes et la résolution du problème relaxé.
- L'approche de voisinage, appelée aussi recherche locale ou la descente est un processus fondé sur deux éléments essentiels : un voisinage et une procédure exploitant le voisinage. Il s'agit de 1) débuter avec une configuration admissible quelconque, et 2) choisir un voisin tel que le coût du voisin soit inférieur à celui de la configuration de départ et considérer la nouvelle configuration comme solution de départ et répéter jusqu'à l'arrivée à la solution dont le coût est un inférieur à celui de tout ses voisins.
- L'approche d'évolution est basé sur un des principes du processus d'évolution naturelle. Les méthodes évolutives doivent leur nom à l'analogie avec les mécanismes d'évolution des espèces vivantes. Un algorithme évolutif typique est composé de trois éléments essentiels : 1) une population constituée de plusieurs individus représentant des solutions potentielles du problème donné ; 2) un mécanisme d'évaluation de l'adaptation de chaque individu de la population à l'égard de son environnement extérieur ; 3) un mécanisme d'évolution composé d'opérateurs permettant d'éliminer certains individus et de produire de nouveaux individus à partir des individus sélectionnés.

Les méthodes de résolution quelque soit l'approche qu'elles suivent font partie de deux groupes de nature différente : les méthodes exactes et les méthodes approchées.

1.3.1 Les méthodes exactes

Les méthodes exactes ou complètes garantissent la complétude de la résolution. Ce sont des méthodes d'énumération implicite : toutes les solutions possibles du problème peuvent être énumérées. En revanche, l'approche effective du problème se confronte à deux difficultés. La première est qu'il n'existe pas forcément un algorithme simple pour énumérer les configurations possibles. La seconde est que le nombre de solutions réalisables est très grand, ce qui signifie que

le temps d'énumération de toutes les solutions est prohibitif. La complexité algorithmique est en général exponentielle. Pour diminuer le temps de recherche, différentes techniques existent pour calculer des bornes permettant d'élaguer le plus tôt possible des branches conduisant à un échec. Ainsi, les méthodes exactes permettent de résoudre des problèmes de manière optimale, mais souvent pour des instances de petite taille. En même temps, pour les problèmes NP-difficiles ces méthodes ne sont pas applicables. Les méthodes approchées appelées aussi heuristiques ont vu le jour pour palier à cette carence et proposer des méthodes qui effectuent un compromis entre le temps de résolution et la qualité de la solution.

1.3.2 Les méthodes approchées

Les méthodes approchées tentent de trouver une solution de bonne qualité en un temps de calcul raisonnable sans garantir l'optimalité de la solution obtenu. Les méthodes approchées s'appuient sur un équilibre entre l'intensification de la recherche et la diversification de celle-ci. L'intensification permet de rechercher des solutions de plus grande qualité en s'appuyant sur les solutions déjà trouvées. La diversification met en place des stratégies qui permettent d'explorer un plus grand espace de solutions et d'échapper aux minima locaux. En préservant cet équilibre, les heuristiques évitent de converger trop vite vers des minima locaux (manque de diversification) oude faire une exploration trop longue (manque d'intensification).

1.4 Contexte de ces travaux

Le processus de conception d'un circuit numérique devient de plus en plus complexe étant donnée l'évolution des applications d'une part et l'accroissement de l'écart entre la capacité d'intégration des circuits et la capacité de conception d'autre part. En même temps, les méthodes d'optimisation exactes et approchées se sont développées pour proposer des solutions de meilleures qualités aux problèmes combinatoires de grande complexité dans tous les domaines. Ainsi, les méthodes de recherche opérationnelle peuvent fournir des solutions aux problèmes combinatoires du flot de conception des circuits numériques.

Ces travaux de thèse se situent entre le domaine de l'optimisation combinatoire et le domaine de la conception des circuits intégrés. Nous nous sommes intéressés à l'implémentation des applications de traitement de signal et de l'image sur les FPGAs. La contrainte principale de cette famille d'applications étant temporelle (le débit), nous sous sommes concentrés dans un premier temps sur la minimisation de la surface occupée une fois l'application implémentée sur un FPGA.

1.4.1 Le challenge de la réduction de la surface

La réduction de la surface occupée par une application donnée a toujours était parmi les principaux objectifs des concepteurs. Plusieurs approches ont été proposées pour la réduction de la surface.
Une application se traduit par un ensemble d'opérations (addition, soustraction, comparaison, etc.) [Gajski 1992]. La surface occupée lors de l'implémentation de celle-ci sur un FPGA est comparable au :

- nombre d'unités reconfigurables occupées par l'implémentation des opérateurs qui vont exécuter les opérations de l'application considérée
- en plus de celles utilisées pour les unités de stockage appelées aussi registres responsables de la sauvegarde des données sur les quelles les opérateurs vont opérer et enfin les unités nécessaires pour l'implémentation des unités responsables du cheminement des données en cas de réutilisation des opérateurs ou des registres.

Ces travaux de thèse s'inscrivent dans la phase d'exploration de l'espace de solutions, dans un flot de synthèse de haut niveau dédié aux applications de traitement du signal et de l'image sur FPGAs.

1.4.2 Plan de la thèse

Ce manuscrit est composé de cinq chapitres. Le chapitre II présentera le processus de la synthèse de haut niveau et un état de l'art non exhaustif sur les méthodes d'optimisation et leurs applications aux différents sous problèmes de ce processus. Pour chacune des méthodes présentées nous en donnerons les points forts et les insuffisances. Une première approche de résolution du problème de la minimisation de la surface au cours de la synthèse de haut niveau avec les résultats et bilan seront présentées dans le chapitre III. Le chapitre IV présentera des algorithmes plus avancées ainsi que leurs résultats respectifs. Enfin, le dernier chapitre permettra de conclure cette étude et de proposer des perspectives de recherche.

Chapitre 2

Etat de l'Art

Sommaire

La première partie de ce chapitre présentera les grandes étapes qui constituent un flot de synthèse de haut niveau. La deuxième partie, présentera un état de l'art des principales méthodes d'optimisation et de leurs applications au problème de la synthèse de haut niveau. La dernière section de ce chapitre souligne les insuffisances des approches existantes et définit les objectifs de nos travaux.

2.1 La synthèse de haut niveau

2.1.1 L'élévation du niveau d'abstraction

Les outils de synthèse logique ont permis de combler en partie le fossé entre la modélisation d'une architecture et la réalisation proprement dite du circuit VLSI , en utilisant des langages de description matériel. Ces outils automatiques de synthèse logique ont atteint, depuis déjà une dizaine d'années, un degré de maturité tel que leur utilisation dans le domaine de conception des circuits VLSI est devenue courante.

La spécification au niveau transfert de registres, proposée à l'entrée des outils de synthèse logique est aujourd'hui principalement écrite manuellement par les concepteurs. C'est une description structurelle de l'architecture constituée de composants combinatoires (opérateurs et multiplexeurs) et des composants séquentiels (registres, RAM, etc.). Cette description s'appuie sur un ensemble bien défini et standardisé de constructions syntaxiques. Cependant avec la croissance rapide de la complexité des algorithmes et des applications à implémenter, écrire une spécification au niveau RTL devient une tâche laborieuse. Elle rallonge la durée et augmente le coût de la phase de conception. De ce fait, le recours à une automatisation de cette étape s'impose. Cette automatisation requière une élévation du niveau d'abstraction pour la spécification des systèmes.

L'élévation du niveau d'abstraction de la spécification simplifie d'un coté la tâche du concepteur mais ouvre en plus la voie à un vaste champ de recherche visant à automatiser l'exploration architecturale (voir figure 2.1). Ainsi l'avènement des outils de synthèse de haut niveau HLS , appelée également synthèse architecturale ou aussi synthèse comportementale s'inscrit dans cette optique. La synthèse de haut niveau est un module complémentaire à ceux de la synthèse logique et de la synthèse physique dans une perspective d'automatisation du processus de raffinement d'un composant matériel.

Les outils commerciaux de synthèse de haut niveau sont passés par plusieurs phases depuis leur apparition en 1980. Les outils de HLS de la première et la deuxième génération souffraient d'un manque de maturité [Martin 2009]. Toutefois aujourd'hui, le besoin pressant en nouvelles techniques de synthèse permettant de passer plus rapidement et surtout à moindre coût du niveau système au silicium est en train de susciter l'intérêt des chercheurs pour le développement de ces outils et des concepteurs pour leur utilisation.

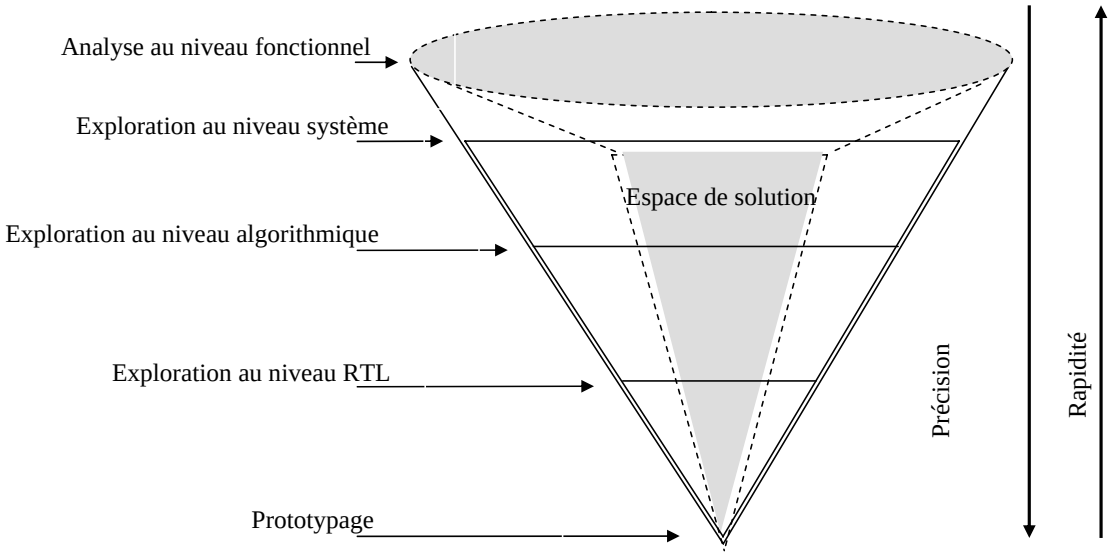


FIGURE 2.1 – Exploration de l'espace des solutions aux différents niveaux d'abstraction

2.1.2 Le flot de synthèse

La synthèse de haut niveau peut être comparée à la compilation dans le domaine logiciel. Dans les deux processus, il s'agit de passer d'un modèle à un haut niveau d'abstraction à un modèle plus détaillé. Les modèles de départ se veulent d'une part intuitifs, lisibles et concis, et d'autre part aussi indépendants que possible des cibles chargées de l'exécution . Les outils de synthèse de haut niveau prennent en entrée une description algorithmique, les contraintes prédéterminées (débit, ressources, etc.), les objectifs attendus (minimisation de la surface, minimisation de la consommation, etc.) et génèrent automatiquement la description RTL du circuit. Le processus de la synthèse de haut niveau (Figure 2.2) passe par la compilation, la sélection, l'allocation, l'ordonnancement, l'assignation et la génération de la description RTL de l'architecture. L'ordre de ces étapes n'est pas figé et varie d'un outil à un autre selon les objectifs et les contraintes supportés par ce dernier. Chacune des étapes de la HLS a un rôle bien précis. Ainsi :

La compilation réalise la vérification syntaxique et sémantique de la description initiale pour la traduire ensuite en une représentation intermédiaire propre à l'environnement de synthèse. Cette phase se charge des optimisations telles que l'élimination du code mort, la propagation des constantes, la transformation des boucles [Bacon 1994], etc.
La représentation interne, résultat de la compilation peut prendre plusieurs formats. Deux modèles sont généralement utilisés : Les graphes flot de données DFG [Gajski 1992] et les graphes flot de données et de contrôle CDFG [Orailoglu 1986]. Ces deux modèles seront

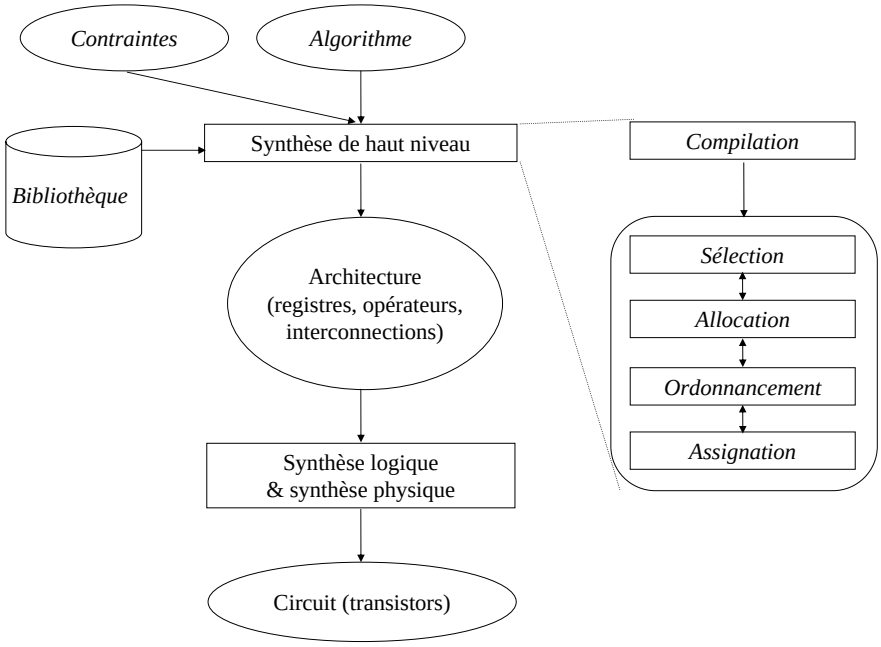


FIGURE 2.2 – Flot de synthèse de haut niveau

détaillés dans la section 2.1.3.

La sélection consiste à choisir pour chaque classe d'opération le type de l'opérateur qui se chargera de leur exécution parmi ceux présents dans la bibliothèque de composants matérielles. Ce choix se base sur l'évaluation de critères tels que la surface, la vitesse et la consommation de chaque composant de la bibliothèque [Bakshi 1996].

L'allocation fixe pour chaque type d'opérateur sélectionné le nombre d'instances à utiliser dans l'architecture finale [Gutberlet 1992]. Celle-ci peut se faire, par exemple, manuellement (par l'utilisateur) si l'application traitée est sous contraintes de ressources ou automatiquement si l'application est sous contraintes de temps.

L'ordonnancement affecte une date de début d'exécution aux opérations du graphe. L'ordonnancement doit respecter les contraintes imposées par le concepteur ainsi que les dépendances de données exprimées dans le graphe produit par l'étape de compilation. Les contraintes et les objectifs de cette étape varient selon les caractéristiques de la classe d'applications ciblées [Walker 1995, Micheli 1994, Rau 1994]. On distingue deux catégories d'ordonnancement : l'ordonnancement sous contraintes de ressources et l'ordonnancement sous contraintes de temps. Dans la première catégorie, le nombre maximal

de ressources matérielles (opérateurs et/ou registres..) à utiliser dans le circuits est fixé à l'avance, et l'objectif consiste à minimiser la durée d'ordonnancement. Dans la deuxième catégorie, la durée totale de l'ordonnancement est fixé, et l'objectif est de minimiser le nombre de ressources matérielles.

L'assignation des opérations aux opérateurs associe une instance d'opérateur à chaque opération. Deux opérations, qui d'une part peuvent être exécutées par le même type d'opérateur et d'autre part ont des durées d'exécution qui ne se chevauchent pas, peuvent partager le même opérateur. L'assignation des opérations aux opérateurs peut se faire avant ou après l'ordonnancement.

L'assignation des variables aux registres associe un registre à chaque variable. Une variable intermédiaire (donnée) produite par une opération du graphe doit être stockée dans un registre tant que toutes les opérations utilisant cette donnée n'ont pas été exécutées. La durée de vie d'une variable est l'intervalle défini par la date de sa création et la date de sa dernière utilisation. Les variables qui ont des périodes de vie disjointes peuvent partager le même registre. L'un des principaux critères d'optimisation dans cette tâche est la minimisation du nombre de registres [Paulin 1989b] ainsi que les multiplexeurs [Cong 2008].

2.1.3 Les modèles formels

Pour explorer efficacement l'espace des solutions architecturales, les étapes de la synthèse de haut niveau opèrent sur des modèles formels. Chaque modèle formel traduit une partie des informations permettant d'aboutir à une description structurelle de l'architecture.
Ainsi la première étape de la synthèse de haut niveau consiste à traduire la spécification algorithmique (Figure 2.4-a) en une représentation interne, par exemple le DFG (Figure 2.4-b). Ce modèle formel est par la suite exploité par les étapes de sélection, d'allocation et d'ordonnancement afin de se rapprocher encore plus de la description structurelle de l'architecture. Le DFG permet de mettre en évidence, par analyse des dépendances, les opérations pouvant être exécutées en parallèle.

Ainsi un **Graphe Flot de Données** ou DFG, $G(V_1, V_2, E)$ est un graphe bipartite orienté et acyclique avec V_1 l'ensemble des noeuds représentant les opérations atomiques que le circuit doit réaliser, V_2 l'ensemble des noeuds représentant les variables. E représente l'ensemble des arcs qui traduisent les dépendances de données entre opérations et variables. Généralement, les noeuds variables sont omis et remplacés par des étiquettes aux arcs qui dénotent les variables en entrées et en sortie des opérations (Figure 2.3).

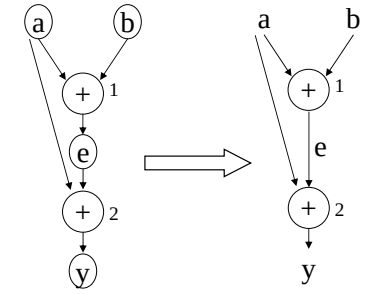


FIGURE 2.3 – Graphe Flot de Données

Une fois la représentation fournie, les étapes d'allocation et d'ordonnancement définissent le nombre nécessaire d'opérateurs ainsi que les dates de début des opérations représentées par les noeuds du DFG.

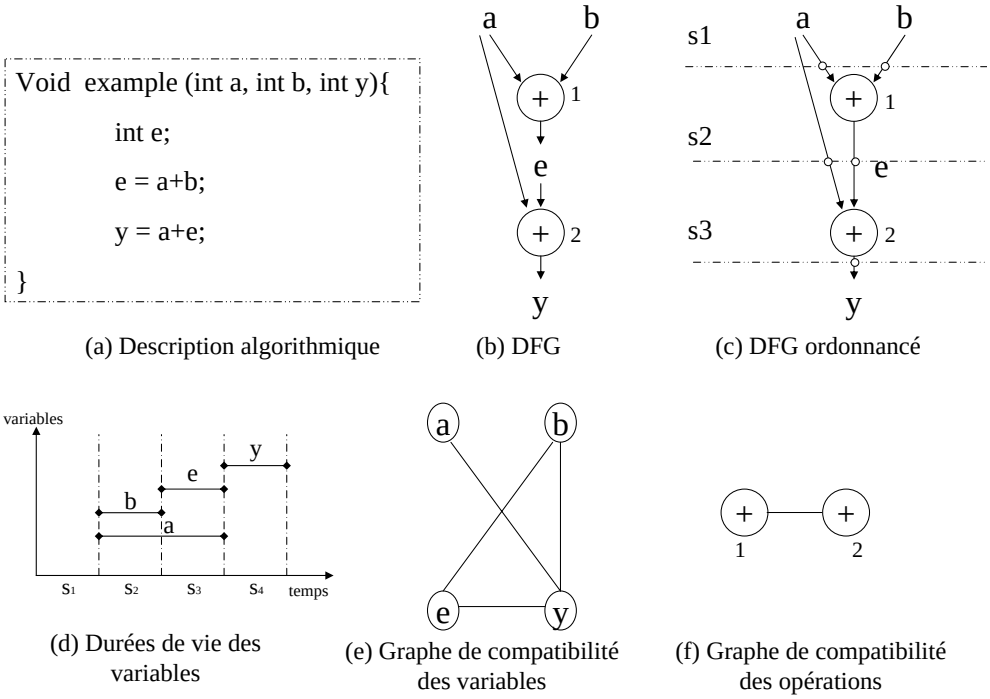


FIGURE 2.4 – Exemple

Dans l'exemple de la figure 2.4 nous allons supposer que l'étape de l'ordonnancement se fera avant l'assignation des opérations et des variables. L'opération d'addition 1 a été, ainsi, ordonnancée dans le cycle S_2 et l'opération d'addition 2 a été ordonnancée dans le cycle S_3.

Un seul opérateur d'addition suffira : il exécutera l'opération 1 et une fois libéré sera réutilisé pour l'exécution de l'opération 2. À l'issu de l'étape d'ordonnancement, les dates de début et de fin de chaque opération sont connues ainsi que les intervalles durant lesquels les valeurs des données sont actives $[d_c, d_u]$, avec d_c la date de création de la variable et d_u la date de sa dernière utilisation. Ces informations génèrent des contraintes pour les étapes d'assignation des opérations et l'assignation des variables. Ces contraintes peuvent être représentées sous forme de graphes de compatibilité.

Graphe de compatibilité des variables : un graphe de compatibilité des variables G_{dc} défini par $G_{dc} = (V_{dc}, A_{dc})$, avec V_{dc} l'ensemble des noeuds représentant les variables. Une arête $a_{dc} = (vd_i, vd_j) \in A_{dc}$ relie deux sommets si et seulement si leurs durées de vie respectives ne se chevauchent pas. Dans ce cas, les variables vd_i et vd_j sont dites compatibles l'une avec l'autre et peuvent être affectées au même registre. La figure 2.4-d est le graphe de compatibilité des variables déduit après l'ordonnancement figure 2.4-c. À partir de ce graphe, on peut déduire qu'il nous faut au moins deux registres, comme il existe au moins un cycle (S2 par exemple) qui a deux variables en vie simultanément. En effet, le nombre de registres minimum est égal au nombre maximum de variables qui sont en vie en même temps. Ainsi, la variable a est compatible avec y : un registre peut contenir ces deux variables. Un deuxième registre sera nécessaire pour stocker les variables b et e. Une autre affectation intuitive des variables consiste à créer un registre pour chaque variable, par conséquent quatre registres seront utilisés.

Graphe de compatibilité des opérations : un graphe de compatibilité des opérations G_{oc} défini par $G_{oc} = (V_{oc}, A_{oc})$, avec V_{oc} l'ensemble des noeuds représentant les opérations. Une arête $a_{oc} = (vo_i, vo_j)$ relie deux sommets si et seulement si les opérations correspondantes peuvent être exécutées par le même type d'opérateur, et l'opération vo_i s'achève avant le début de vo_j. Dans ce cas, les opérations vo_i et vo_j sont dites compatibles et peuvent ainsi être affectées à un même opérateur.
La figure 2.4-e montre que les opérations 1 et 2 sont compatibles : un additionneur suffirait pour les exécuter.

2.1.4 Complexité de la synthèse de haut niveau

La figure 2.5 montre deux solutions possibles. Une solution correspond à une architecture composée d'opérateurs, de registres et leurs interconnexions. La différence entre les deux solutions réside au niveau de l'assignation des variables.

Dans la première solution figure 2.5-a, chaque variable a été assignée à un registre. Quatre registres, un opérateur et un multiplexeur ont été utilisés dans cette solution. Le multiplexeur

a été crée pour aiguiller les variables qui précédent l'une des entrées de l'opérateur. Celle-ci reçoit une fois la variable b et une fois la variable e, toutes deux stockées dans deux registres différents. Dans la deuxième solution figure 2.5-b les variables ont été regroupées : les variables a et y dans un registre et les variables e et b dans un deuxième registre. Chacune de ces solutions a un coût qui correspond à la somme des coûts des composants de l'architecture. La solution figure 2.5-b a un coût inférieur à celui de la figure 2.5-a comme celle-ci utilise deux registres et un multiplexeur de plus que la solution figure 2.5-b.

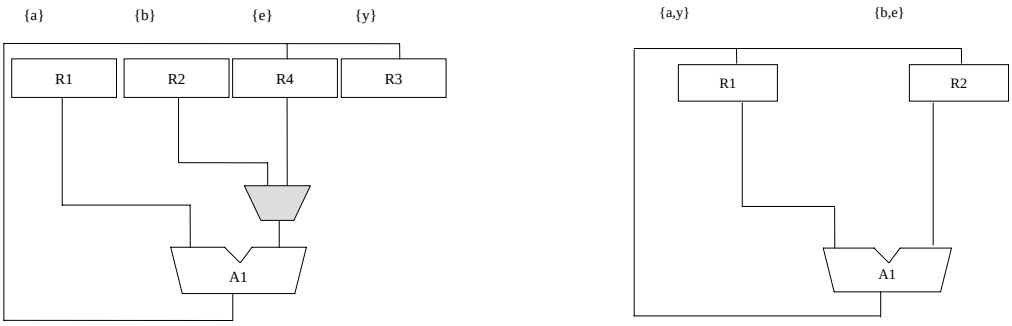


FIGURE 2.5 – Deux solutions d'un même problème

L'un des objectifs principales des outils de synthèse de haut niveau est de trouver une architecture qui a un coût minimal. Trouver les techniques d'ordonnancement, d'allocation, d'assignation des variables et des opérations qui mènent systématiquement à la solution qui coûte le moins est quasiment impossible. Ceci est dû à la complexité et l'interdépendance entre ces sous problèmes.

Parmi les stratégies utilisées dans les outils de synthèse, on trouve le choix de résolution de l'allocation avant l'ordonnancement[Coussy 2008]. Dans ce cas, le nombre d'opérateurs défini à la fin de l'étape d'allocation servira de guide pour l'ordonnancement. Dans certains cas, ce nombre n'est pas définitif et pourra être mis en cause par le résultat de l'ordonnancement. L'ordonnancement se fait ainsi sous contraintes de temps (le débit) et de ressources aussi mais cette dernière contrainte étant flexible peut être revue si son respect s'oppose à celui du respect de la contrainte de temps. Dans le cas ou l'allocation se fait après l'ordonnancement le nombre de ressources allouées sera supérieur ou égal au nombre maximum de ressources utilisées en même temps. Ainsi, le choix de l'ordre d'exécution de ces deux phases influera la solution finale.

Une forte interdépendance existe aussi entre les étapes d'ordonnancement et d'assignation. Ces deux problèmes sont NP-complets [Garey 1979, Pangrle 1991]. En outre, la résolution optimale de chacun de ces sous-problèmes ne mènent pas forcément à la solution optimale au

problème global [Cong 2008]. Ainsi, le choix de l'ordre d'exécution des étapes de la synthèse, définira l'espace de solutions explorable. Par exemple, si l'on décide de résoudre les problèmes d'ordonnancement et d'assignation des opérations en premier, le résultat obtenu limitera les possibilités d'assignations des variables.

Dans la plupart des contributions précédentes qui adressent la génération automatique de la description structurelle, un ordre d'exécution des étapes est prédéfini afin de résoudre le problème étape par étape. Dans cet optique, une étape est entièrement résolue avant de passer aux suivantes. Cette approche sacrifie la bonne résolution d'une étape en faveur des précédentes et ne conduisent pas, par conséquent, à une bonne solution au problème global.

Ainsi les travaux existants qui traitent le problème de la minimisation des ressources dans le flot de la HLS abordent un seul sous problème à la fois ou alors plusieurs de ces sous problèmes dans un ordre prédéfini.

2.2 Minimisation de la surface du circuit lors de la synthèse de haut niveau

Depuis une vingtaine d'années, plusieurs méthodes ont été proposées pour améliorer le résultat de la synthèse de haut niveau en terme de surface. Très peu sont les travaux qui ont tenté de résoudre tous les sous problèmes de la synthèse de haut niveau à la fois. Dans un premier temps, les chercheurs se sont focalisés sur la minimisation du nombre d'unités fonctionnelles [Gajski 1992, Paulin 1989a, Rau 1994, Cong 2006] ou des registres [Hashimoto 1971] qui semblaient être, les facteurs qui avaient le plus d'impact sur la surface occupée par une architecture.

La minimisation des opérateurs se fait lors des phases d'allocation et d'ordonnancement. Le problème d'ordonnancement est défini principalement à partir des données qui lui sont proposées en entrée. Deux cas de figures existent :

- L'allocation se fait avant l'ordonnancement. On doit résoudre alors un problème d'ordonnancement sous contraintes de ressources [Gajski 1992].
- En revanche dans le cas contraire ou l'allocation se fait en même temps ou après l'ordonnancement. Ce problème se formule en ordonnancement sous contrainte de temps [Paulin 1989a].

La minimisation des registres, quant à elle, se fait au niveau de l'assignation des variables aux registres [Hashimoto 1971].

Dans un deuxième temps, des études relativement récentes [Chen 2004, Cong 2008] ont montré que le coût des interconnexions doit être pris en compte. En effet, la surface d'un composant d'interconnexion (multiplexeur) peut dépasser celle d'un opérateur ou d'un registre. Par conséquent, des algorithmes de réduction de registres qui tentent de créer un minimum d'interconnexions ont été proposés.

Dans la partie qui suit, nous présenterons les méthodes exactes proposées pour la synthèse de haut niveau (présentation de la méthode - application à la HLS), ensuite nous présenterons les heuristiques et métaheuristiques qui ont été proposées pour la même problématique. Les méthodes exactes seront illustrées au travers de la programmation linéaire en nombre entier et la relaxation continue de cette formulation appliquées principalement aux problèmes d'ordonnancement et d'assignation des opérations. Les méthodes approchées, quant à elles, seront illustrées au travers d'heuristiques appliquées aux problèmes d'allocation, d'ordonnancement et d'assignation des variables et de métaheuristique appliquée à la totalité du flot de synthèse.

2.2.1 Les méthodes exactes pour la HLS

Dans la première partie de cette section nous présentons le principe général de la formulation en programme linéaire en nombre entier. Dans la deuxième partie nous montrons comment cette approche aide à résoudre le problème d'ordonnancement et d'assignation des opérations, ainsi que la façon dont les contraintes d'intégrité de cette formulation peuvent être relaxées pour pouvoir résoudre le problème d'ordonnancement avec la programmation linéaire.

2.2.1.1 La formulation en problème de programmation linéaire en nombre entier

Parmi les premières propositions pour la résolution du problème de la synthèse de haut niveau nous trouvons la formulation des sous problèmes d'ordonnancement et d'assignation en problèmes de programmation linéaire en nombres entiers (PLNE). En mathématique, les problèmes de programmation linéaire (PL) [Guéret 2000] sont des problèmes d'optimisation où la fonction objectif et les contraintes sont toutes linéaires.

Considérons le programme linéaire suivant : A est une matrice $m \times n$ tel que ses coefficients a_{ij} sont entiers, c est un vecteur de longueur n tel que les c_j sont entiers et b est un vecteur de longueur m tel que les b_i sont entiers.

$$PL = \begin{cases} \text{Minimiser } z = c.x \\ \text{Sous les contraintes :} \\ \text{A.x=b} \\ \text{x}\geq 0 \end{cases}$$

Si l'on résout ce PL tel quel, nous allons trouver généralement une solution optimale à composante non entière. Mais, il arrive qu'une solution à composante entières soit nécessaire pour une classe de problèmes particuliers. Ainsi de nouvelles contraintes dites d'intégrités sont nécessaires. On obtient alors le Programme linéaire en Nombres Entiers suivant :

$$PLNE = \begin{cases} \text{Minimiser } z = c.x \\ \text{Sous les contraintes :} \\ \text{A.x=b} \\ x \geq 0 \ x_j \text{ entier } (j = 1..n) \end{cases}$$

Nombreux sont les problèmes NP-complets qui peuvent être exprimés comme des PLNE, les programmes linéaires en nombres entiers sont ainsi des problèmes NP-complet. La relaxation continue d'un PLNE (c'est le PLNE sans les contraintes d'intégrité) est un PL qui peut être résolu efficacement et fournir ainsi une borne inférieure.

Plusieurs approches existent pour résoudre un PLNE, on en citera deux :
- L'algorithme par séparation et évaluation qui se base sur une relaxation continue pour diminuer au maximum l'énumération des solutions (Branch and Bound).
- Les méthodes de coupes qui s'appuient sur la résolution du PL continu dans un premier temps, puis dans un deuxième temps sur l'ajout progressif de contraintes qui vont éliminer les solutions extrêmes continues sans éliminer les solutions entières (c'est ce qu'on appelle une coupe) et s'arrêter dès que notre PL nous donne une solution extrême entière.

2.2.1.2 Formulations de problèmes de la HLS en PLNE

Cette sou-section présente :
- A- une formulation du problème d'ordonnancement en PLNE,
- B- une extension de cette formulation pour résoudre le problème d'assignation des opérations en même temps que l'ordonnancement,
- C- une relaxation continue du PLNE pour l'ordonnancement.

A- PLNE pour l'ordonnancement

La résolution du PLNE du problème d'ordonnancement sous contraintes de temps pour la synthèse de haut niveau mène à un ordonnancement optimal en terme de nombre d'opérateurs. La résolution de la formulation PLNE de ce problème peut se faire à l'aide de la méthode d'évaluation et de séparation, branch-and-bound.
Soient les notations du tableau 2.2.1.2. La fonction objectif 2.1 assure la minimisation du coût des opérateurs. La contrainte 2.2 veille à ce que chaque opération o_i soit placée dans une seule étape s_j qui appartient à la fenêtre de temps $[Ei, Li]$. Ei est la date d'ordonnancement de l'opé-

ration o_i au plus tôt et Li sa date d'ordonnancement au plus tard obtenues respectivement suite aux ordonnancements ASAP et l'ordonnancement ALAP . La contrainte 2.3 oblige le nombre de d'opérateurs de type k alloués, à être supérieur ou égal au nombre maximal d'opérateurs sollicité dans les différentes étapes j.

$OP = \{o_i / i = 1..n\}$:	l'ensemble des opérations et n le nombre d'opérations
$T = \{t_k / k = 1..m\}$:	l'ensemble des différents types d'opérateurs et m la taille de cet ensemble
OP_{t_k} :	l'ensemble des opérations de type t_k
$INDEX_{t_k} = \{i / o_i \in OP_{t_k}\}$:	l'ensemble des indices des opérations de type t_k
N_{t_k} :	le nombre d'opérateurs de type t_k
C_{t_k} :	le coût d'un opérateur de type t_k
$S = \{s_j / j = 1..r\}$:	le temps étant discrétisé, S est l'ensemble des étapes et r le nombre d'étape

TABLE 2.1 – Notations

$$minimiser \sum_{k=1}^{m} C_{t_k} \times N_{t_k} \tag{2.1}$$

$$\forall i, 1 \leqslant i \leqslant n, (\sum_{E_i \leqslant j \leqslant L_i} x_{i,j} = 1) \tag{2.2}$$

$$\forall j, 1 \leqslant j \leqslant r, \forall k, 1 \leqslant k \leqslant m, (\sum_{i \in INDEX_{t_k}} x_{i,j} \leqslant N_{t_k}) \tag{2.3}$$

$$\forall i, j, o_i \in pred_{o_j} (\sum_{E_i \leqslant k \leqslant L_i} (k \times x_{i,k}) - \sum_{E_i \leqslant l \leqslant L_i} (l \times x_{j,l}) \leqslant -1) \tag{2.4}$$

$$x_{ij} \in \{0, 1\} \forall i, \forall j. \tag{2.5}$$

La contrainte 2.4 garantit pour une opération donnée o_j, que tous ses prédécesseurs (i.e. $pred_{o_j}$) achèvent leurs exécutions plus tôt que sa date de début. En d'autres termes que si $x_{i,k} = x_{j,l} = 1$ alors $k < l$, avec k et l sont respectivement les dates de début éventuelles des

opérations i et j. Enfin, la contraintes 2.5 fixe le domaine de définition des variables de décision x_{ij}. Les variables de décision sont spécifiées par :

$$x_{ij} = \begin{cases} 1 & \text{si l'opération } i \text{ est placée à l'étape } j \\ 0 & \text{sinon} \end{cases}$$

Une fois définies, les valeurs des variables de décision x_{ij} construisent une solution pour le problème d'ordonnancement des opérations d'un graphe donné.

Exemple *Prenons l'exemple [Gajski 1992] d'une application qui peut utiliser quatre types d'opérateurs (multiplieur, additionneur, soustracteur, comparateur). C_m et N_m sont respectivement le coût et le nombre de multiplieur ; C_a et N_a sont respectivement le coût et le nombre d'additionneur ; C_s et N_s sont respectivement le coût et le nombre de soustracteur ; C_c et N_c sont respectivement le coût et le nombre de comparateur.*
La fonction objectif est : Minimiser $Cm * N_m + C_a * N_a + C_s * N_s + C_c * N_c$

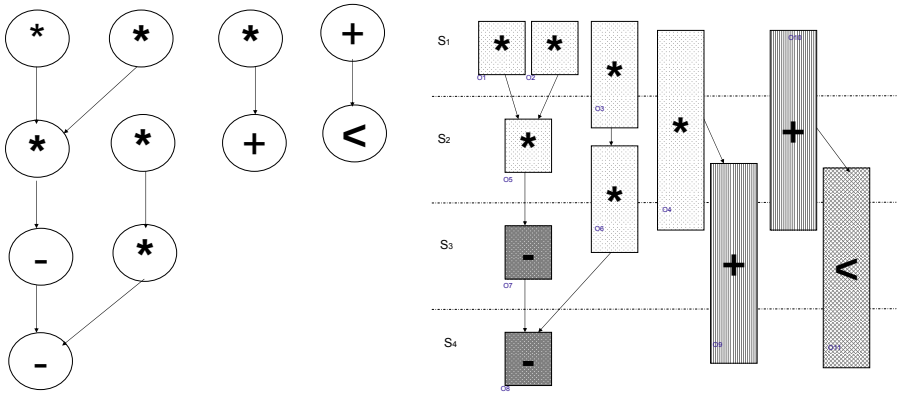


FIGURE 2.6 – (a) DFG (b) Expression des contraintes

Ainsi, pour l'exemple de la figure 2.6, les contraintes définissant une fenêtre de temps pour chaque opération deviennent :

$$\begin{aligned} &x_{1,1} = 1, && x_{2,1} = 1 \\ &x_{5,2} = 1, && x_{7,3} = 1 \\ &x_{7,3} = 1, && x_{8,4} = 1 \\ &x_{3,1} + x_{3,2} = 1, && x_{4,1} + x_{4,2} + x_{4,3} = 1 \\ &x_{6,2} + x_{6,3} = 1, && x_{9,2} + x_{9,3} + x_{9,4} = 1 \\ &x_{10,1} + x_{10,2} + x_{10,3} = 1, && x_{11,2} + x_{11,3} + x_{11,4} = 1 \end{aligned}$$

les contraintes permettant de respecter les contraintes de précédences :

$$1x_{4,1} + 2x_{4,2} + 3x_{4,3} - 2x_{9,2} - 3x_{9,3} - 4x_{9,4} = -1$$
$$1x_{3,1} + 2x_{3,2} - 2x_{6,2} - 3x_{6,3} = -1$$
$$1x_{10,1} + 2x_{10,2} + 3x_{10,3} - 2x_{11,2} - 3x_{11,3} - 4x_{11,4} = -1$$

et les contraintes définissant pour chaque type le nombre d'opérateur utilisé dans un cycle donné :

$$x_{1,1} + x_{2,1} + x_{3,1} + x_{4,1} = N_m, \qquad x_{9,3} + x_{10,3} = N_a$$
$$x_{3,2} + x_{4,2} + x_{5,2} + x_{6,2} = N_m, \qquad x_{9,4} = N_a$$
$$x_{4,3} + x_{6,3} = N_m, \qquad x_{11,2} = N_c$$
$$x_{7,3} = N_s, \qquad x_{11,3} = N_c$$
$$x_{8,4} = N_s, \qquad x_{11,4} = N_c$$
$$x_{10,1} = N_a,$$
$$x_{9,2} + x_{10,2} = N_a$$

Ainsi, en supposant que $C_m = 2$ et $C_a = C_b = C_a = 1$ et $N_m = 2$, $N_a = N_s = N_c = 1$, la fonction de coût est minimisée et toutes les contraintes sont respectées si les variables décision ont les valeurs suivantes : $x_{1,1} = x_{1,2} = x_{1,1} = x_{3,2} = x_{4,3} = x_{5,2} = x_{6,3} = x_{7,3} = x_{8,4} = x_{9,4} = x_{10,2} = x_{11,4} = 1$ et les autres variables de décision $x_{i,j} = 0$.
Cette solution peut être obtenue à l'aide d'une procédure de séparation et d'évaluation (PSE).

Cette procédure consiste à construire un arbre de recherche et séparation. Un arbre de recherche est un arbre (souvent binaire) dont chaque sommet représente une solution partielle réalisable. Celle-ci est retrouvée en prenant les valeurs données aux variables de décision, partant de la racine jusqu'au noeud considéré. En arrivant aux feuilles de l'arbre, nous obtenons des solutions réalisables complètes. À chaque noeud est associé une valeur qui correspond au coût de la solution partielle représentée par le noeud.

Dans la figure 2.7, le noeud racine à un coût nul. Ce noeud a un seul fils comme la variable $x_{1,1}$ ne peut prendre qu'une seule valeur selon les contraintes définissant les fenêtres de temps. Le deuxième noeud, partant du haut dans le même arbre, correspond ainsi à la solution partielle qui affecte l'opération o_1 à l'étape s_1. Cette affectation à un coût égal à deux. Ce coût correspond à l'utilisation d'un multiplieur.

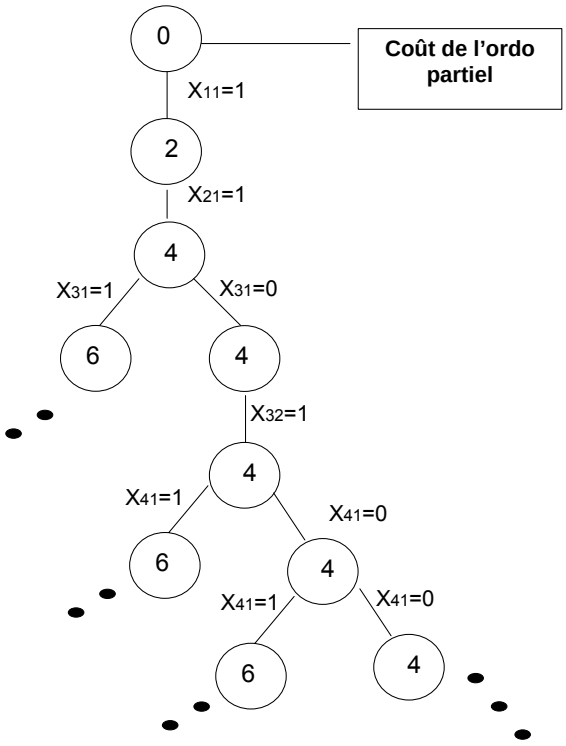


FIGURE 2.7 – Arbre de recherche et séparation

B- PLNE pour l'ordonnancement et l'assignation

Pour résoudre le problème de l'assignation en même temps que l'ordonnancement il suffit d'ajouter une variable de décision qui associe une opération à un opérateur et des contraintes qui garantissent la compatibilité des opérations assignées à un même opérateur [Wilson 1994].

- Si deux opérations sont assignées à un même opérateur, cela signifie que l'une doit précéder l'autre.
- Si une opération précède une autre et sont toutes les deux assignées à un même opérateur , la date de début de la deuxième opération est bornée par la date de fin de la première opération.

Critique Comme dit précédemment un problème linéaire en nombre entier est un prblème NP-complet. Ainsi, cette méthode se limite aux instances ayant un nombre de noeuds et de contraintes très petit. C'est pour cette raison que cette piste n'a pas était developpée depuis. Cependant, une étude assez récente, propose une relaxation continue de la formulation PLNE du problème d'ordonnancement.

C- Relaxation continue de la formulation PLNE du problème d'ordonnancement

Dans les travaux de [Cong 2006] le problème d'ordonnancement pour la HLS a été modélisé par un programme linéaire. En effet, seules les solutions entières du programme linéaire correspondent à des solutions au problème. La difficulté réside donc à prouver qu'il existe toujours une solution entière au programme linéaire. Cette méthode ne garantit pas l'optimalité de la solution mais assure une solution optimisé en un temps polynomiale.

Les étapes de cette méthode sont les suivantes :
- Formuler les contraintes en un système de contraintes différenciées SDC .
- Vérifier que le SDC est faisable.
- Représenter l'objectif de l'ordonnancement en une fonction objective linéaire.
- Résoudre la relaxation linéaire et choisir la solution entière qui optimise la fonction objectif.

La première étape consiste donc à traduire les contraintes du problème en un système de contraintes différenciées .

Définition 1 *Un système de contraintes différencier SDC est un système* (X, C) *tel que* X *est l'ensemble de variables et* C *l'ensemble des inéquations linéaires de la forme* $x_j - x_i \leq b_k$*, ou* $1 \leq i, j \leq n$ *et* $1 \leq k \leq m$*, avec* n *le nombre de variables d'ordonnancement qui indique le cycle de début d'exécution d'une opération donnée et* m *le nombre de contraintes.*

Cette forme restrictive de contraintes linéaires peut être représentée par un graphe appelé graphe de contraintes. Celui-ci peut être construit en modélisant les variables par des sommets et une contraintes $x - y \leq b$ par un arc dirigé allant de y à x et ayant un poids b. Un SDC est cohérent si et seulement si son graphe de contraintes ne contient aucun cycle négatif.

Définition 2 *Dans un graphe annoté, un cycle négatif est un cycle dont la somme des poids des arcs est négative.*

Pour détecter l'existence de cycle négatif dans le graphe on peut résoudre le problème du plus court chemin dans le graphe de contraintes. En utilisant l'algorithme de Bellman-Ford, la complexité est de $O(mn)$ avec n le nombre de variables et m le nombre de contraintes. Une fois le système de contraintes SDC construit et sa faisabilité prouvée, reste à démontrer que le programme linéaire donne toujours des solutions entières au problème. En effet, la matrice sous-jacente à un système de contraintes SDC est une matrice totalement unimodulaire. Cette propriété garantie que la relaxation du problème d'ordonnancement en un programme linéaire

donne des optimums entiers en un temps polynomial.

Définition 3 *Une matrice est totalement unimodulaire, si toute sous-matrice carrée a un déterminant de -1,0 ou +1.*

Théorème 1 *Un programme linéaire (min cT x tel que $x \geq 0$, $Ax \geq b$) a un optimum entier pour tout vecteur c, b (avec b entier) si et seulement si A est totalement unimodulaire.*

Cette méthode peut modéliser les contraintes de ressources, la fréquence de l'horloge, la contrainte de latence et les contraintes de temps relatives. Elle peut optimiser ou bien la latence du chemin critique, ou la latence totale. La modélisation peut représenter les opérations multi-cycles et les opérateurs pipelines.

Les inconvénients majeurs de cette méthode sont : (1) La modélisation des contraintes de ressources se basent sur une heuristique qui supposera que certaines opérations sont plus prioritaires que d'autres. Ainsi, l'espace de solution sera partiellement exploré. (2) La modélisation d'une fonction objectif minimisant les ressources n'a pas était proposée.

2.2.2 Les méthodes approchées pour la HLS

Les méthodes exactes étant applicables uniquement aux petites instances, diverses heuristiques ont été proposées pour la HLS. Une heuristique est un algorithme qui fournit rapidement (en temps polynomial) une solution réalisable, pas nécessairement optimale, pour un problème d'optimisation. Généralement, une heuristique est conçue pour un problème particulier, en s'appuyant sur sa structure propre, mais les approches peuvent contenir des principes plus généraux. On parle de métaheuristiques pour les méthodes approximatives générales, pouvant s'appliquer à différents problèmes (comme le recuit simulé par exemple).

La qualité d'une heuristique peut s'évaluer selon deux critères scientifiques :
- Critère pratique, ou empirique : on implémente l'algorithme approximatif et on évalue la qualité de ses solutions par-rapport aux solutions optimales (ou aux meilleures solutions connues). Ceci passe par la mise en place d'un banc d'essai (en anglais benchmark, ensemble d'instances d'un même problème accessible à tous).
- Critère mathématique : il faut démontrer que l'heuristique garantit des performances. La garantie la plus solide est celle des algorithmes approchés, sinon il est intéressant de démontrer une garantie probabiliste, lorsque l'heuristique fournit souvent, mais pas toujours, de bonnes solutions.

Différentes heuristiques adaptées aux sous problèmes de la HLS ont été proposées. Pour l'étape d'ordonnancement, nous citons l'ordonnancement dirigé par les forces (FDS) [Paulin 1989b] et la méthode du raffinement itératif pour l'ordonnancement [Park 1991], proposées pour le cas ou l'allocation se fait après l'ordonnancement et l'ordonnancement par liste [Gajski 1992] et de l'ordonnancement modulo [Rau 1994] proposées pour le cas ou l'allocation se fait avant l'ordonnancement.

Plusieurs heuristiques ont été proposées pour l'étape d'assignation des variables. Nous citons le "Left-Edge" [Hashimoto 1971, Kurdahi 1987], la version modifiée de cet algorithme [Andriamisaina 2008], et l'heuristique qui s'appuie sur la méthode du MWBM [Galil 1986, Papadimitriou 1998]. Cette dernière peut être utilisée pour résoudre l'assignation des opérations aussi. Des études relativement récentes [Cong 2008] ont proposé une heuristique qui assigne simultanément les opérateurs et les registres.

Dans ce manuscrit nous avons choisi de présenter les algorithmes du "Left-Edge", la version modifiée de celui-ci et l'heuristique qui assigne simultanément les opérateurs et les registres. Enfin, nous présentons une méthaheuristique qui s'appuie sur une approche évolutionniste pour l'optimisation de la surface et du temps [Ferrandi 2007]. Cette approche touche à toutes les

étapes de la synthèse de haut niveau.

Algorithme "Left-Edge" L'objectif de cette méthode est d'aboutir à une solution qui utilise un minimum de registres [Hashimoto 1971, Kurdahi 1987]. Différentes architectures utilisant le nombre minimum de registres existent. Ceci dépend des assignations des variables à ces registres. L'inconvénient majeur de cet algorithme est qu'il ne se soucie pas de la création éventuelle de multiplexeurs. Ainsi la réduction du nombre de registres peut conduire à la création de plusieurs multiplexeurs. Par conséquent, la surface occupée lors de la création des multiplexeurs peut être supérieure aux gains à cause de la réduction du nombre de registres.

La première étape de cet algorithme consiste à trier les variables par dates de production. Ensuite, un premier ensemble de registres est crée pour les variables les plus à gauche de la liste triée, dont les durées de vie se chevauchent. Enfin, pour chacune des variables restantes, une recherche de registre compatible entre ceux qui existent déjà est faite. Cette recherche se fait en partant du registre le plus à gauche et en testant si la variable est compatible avec les variables déjà assignées au registre. Si aucune possibilité d'assignation de la variable n'est trouvée alors un nouveau registre est créé pour la stocker.

Algorithme "Left-Edge" modifié L'algorithme LEA trouve le nombre optimal de registres, cependant, il ne se soucie pas du surcoût engendré par la création de multiplexeurs. Cet algorithme a été modifié par [Andriamisaina 2008] pour prendre en compte le surcoût des multiplexeurs. Cette fois, le choix du registre se fait après une phase d'évaluation de toutes les solutions partielles possibles.

Cet algorithme commence par trouver tous les registres capable de stocker la variable. Ensuite, de calculer pour chacun de ces registres, le nombre de sources $Src(v_j, DFG)$ et de destinations $Des(v_j, DFG)$ du registre suite à l'assignation éventuelle de la variable. Enfin, le choix du registre auquel la variable va être affectée se fait sur celui qui est précédé et succédé par les plus petits multiplexeurs. Ces informations sont déduites à partir des variables $Src(v_j, DFG)$ et $Des(v_j, DFG)$. Ainsi, la considération des sources et des destinations des variables lors de l'assignation permet de réduire le nombre et la taille des multiplexeurs à insérer respectivement à l'entrée des registres partagés et à l'entrée des opérateurs partagés.

Définition 4 *La source d'une variable v_j dans un graphe DFG, notée $Src(v_j, DFG)$, correspond à l'opérateur producteur de la variable v_j dans le DFG.*

Définition 5 *La destination d'une variable v_j dans un graphe DFG, notée $Des(v_j, DFG)$, correspond à l'opérateur consommateur de la variable v_j dans le DFG.*

Algorithme 1 : Left-Edge Modifié

Entrée :
Liste des variables à assigner ;
Sortie :
Liste des registres ;
Début
Trier les variables par date de production
Assigner les variables les plus à gauches à des registres distincts $r_1, r_2, ..r_d$
Supprimer les variables déjà assignées de la liste triée ;
Pour chaque variable v_j de la liste triée
Pour tous les registres r_i, allant de 1 à d
Si v_j **et** les variables dans r_i sont compatibles **alors**
Calculer le poids $W(v_j, r_i)$;
Si $W(v_j, r_i)$ est le poids de plus élevé **alors**
Choisir r_i comme le registre sur lequel sera assigné v_j ;
Fin Si
Fin Si
Fin pour
S'il existe un registre r_i sur lequel peut être assigné v_j **alors**
Assigner v_j au registre r_i ;
Sinon
Incrémenter le nombre de registres $(d = d + 1)$;
Assigner v_j un nouveau registre r_d ;
Fin si
Supprimer v_j de la liste
Aller à la prochaine variable
Fin Pour
Fin

Définition 6 *Deux variables sont dites de sources communes si elles ont le même opérateur producteur.*

Définition 7 *Deux variables sont dites de destinations communes si elles ont le même opérateur consommateur.*

Définition 8 *Le nombre de sources communes noté $NbSrc_c(v_j, r_i)$ entre une variable v_j et un registre r_i correspond au nombre de variables assignées dans r_i et de sources communes avec v_j.*

Définition 9 *Le nombre de destinations communes noté* $NbDes_c(v_j, r_i)$ *entre une variable* v_j *et un registre* r_i *correspond au nombre de variables assignées dans* r_i *et de destinations communes avec* v_j.

Définition 10 $W(v_j, r_i) = NbSrc_c(v_j, r_i) + NbDes_c(v_j, r_i)$. $W(v_j, r_i)$ *détermine le poids de l'assignation de* v_j *à* r_i. *Ce poids représente l'intérêt d'une assignation de* v_j *sur* r_i.

Exemple *Partant d'un graphe ordonnancé, dont les opérations sont assignées (figure 2.8)nous allons montrer la différence entre le MLEA et LEA. L'opération* o_1 *est assigné à l'opérateur* A_1 *et les opérations* o_2, o_3 *et* o_4 *sont assigné à l'opérateur* A_2. *La figure 2.9 assigne les variables a, b et c en utilisant le LEA. Les variables a et b ne sont pas compatibles. Deux registres* R_1 *et* R_2 *sont créés pour stocker les deux variables. La variables c est compatible avec le registre le plus à gauche* R_1, *elle lui est assignée. Ce choix d'assignation des variables à créé un multiplexeur avant le registre* R_1, *comme deux opérateurs différents* A_1 *et* A_2 *écrivent dedans. La figure 2.10 assigne les variables a, b et c en utilisant le MLEA. Deux registres* R_1 *et* R_2 *sont créés pour stocker les variables a et b. Lors de l'assignation de la variable c, des poids* $W(v_j, r_i)$ *sont calculés. On obtient* $W(c, R_1) = 0$ *et* $W(c, R_2) = 2$ *comme* R_1 *à un opérateur consommateur et un opérateur producteur en commun avec la variable c.*

Bilan Ces méthodes offrent un solution partielle au problème de la synthèse de haut niveau. Elle suppose un ordre d'exécution des étapes de la synthèse. Ainsi l'espace de solution finale d'implémentation d'une application donnée n'est que partiellement exploré. Des méthodes plus sophistiquées ont été proposés pour résoudre plus d'une étape à la fois. L'approche de l'assignation simultanée des opérateurs et des registres , comme son nom l'indique propose un algorithme qui essaye de gérer l'inter-dépendance entre les deux étapes d'assignation. Une approche évolutionniste pour l'optimisation du la surface et du temps est encore plus générale, elle essaye de diminuer l'interdépendance entre toutes les étapes de la synthèse de haut niveau. Ces deux approches seront détaillées dans les deux sous sections suivantes.

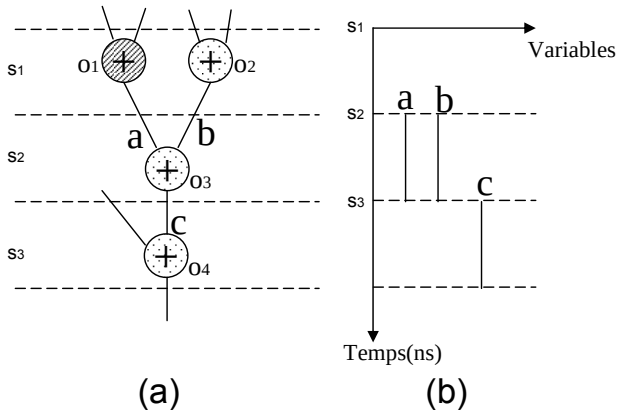


FIGURE 2.8 – Données d'entrée à l'assignation des registres

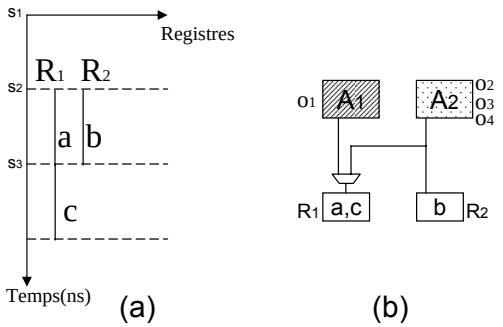


FIGURE 2.9 – Assignation des registres en utilisant le LEA

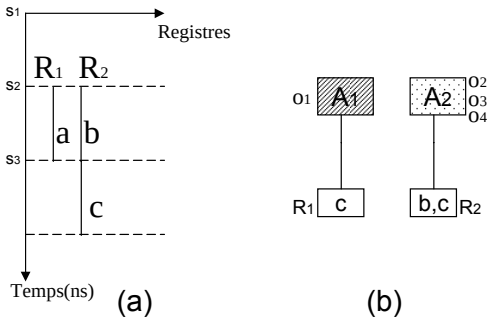


FIGURE 2.10 – Assignation des registres en utilisant le MLEA

2.2.2.1 Assignation simultanée des opérateurs et des registres

L'idée de base de l'assignation simultanée des opérateurs et des registres est de ne pas effectuer une assignation complète des opérateurs suivie d'une assignation complète des registres ou le contraire [Cong 2008]. Il s'agit d'exécuter en parallèle les deux phases et de les faire interagir.

Algorithme 2 : Assignation simultanée des opérateurs et des registres

Entrée :
DFG ordonnancé ;
Contraintes de ressources ;
Initialisation :
b_{fu} = nombre total d'opérations ;
b_{reg} = nombre total de variables ;
Début
Tant que les contraintes de ressources ne sont pas respectées
Diminuer b_{fu} et b_{reg} de $r\%$
Définir le nombre de FU de type f,$b_{fu}(f)$
En se basant sur l'assignation partielle des FUs et des REGs des itérations précédentes :
Assigner les opérations de type f aux $b_{fu}(f)$ FUs
En se basant sur l'assignation partielle des REGs de l'itération précédente et l'assignation courante des opérations de type f aux $b_{fu}(f)$ FUs : Assigner les variables aux b_{reg}
Déduire les MUX nécessaires
Fin Tant que
Fin

L'algorithme 2 présente la technique de l'assignation simultanée des opérateurs et des registres. Les entrées de cette méthodologie sont (1) un graphe flot de donnée ordonnancé et (2) contrairement aux méthodes détaillées précédemment, un nombre prédéfini de ressources (opérateurs et registres). Le but de cette méthode est d'assigner les opérations et les variables respectivement aux opérateurs et registres donnés tel que le coût des multiplexeurs utilisés soit réduit.
Les variables b_{reg} et b_{fu} sont respectivement le nombre de registres et le nombre total d'unités fonctionnelles utilisés dans l'itération courante. Initialement, b_{fu} et b_{reg} sont initialisés respectivement par le nombre total d'opérations et par le nombre total de variables dans le graphe.
Coeur de boucle

- À chaque itération b_{fu} et b_{reg} seront diminués de $r\%$, avec $r\%$ une valeur pour contrôler la vitesse à laquelle le ratio de partage des ressources augmente. Le ratio de partage des opérateurs est défini par l'équation $1-|F|/|O|$, avec $|F|$ le nombre d'unités fonctionnelles et $|O|$ le nombre total d'opérations. Le ratio de partage des registres est défini par l'équation $1-|R|/|V|$, avec $|R|$ le nombre de registres et $|V|$ le nombre total de variables. Le calcul de $b_{fu}(f)$ le nombre d'opérateurs de type f est l'arrondi de $b_{fu} * |F_f|/|F|$, avec F_f l'ensemble des unités fonctionnelles de type f défini par les contraintes de ressources, et F est l'ensemble total des unités fonctionnelles défini par les contraintes de ressources. Cela signifie que chaque type d'opérations occupera un nombre proportionnel à celui décidé par les contraintes de ressources.
- L'allocation des ressources achevée, l'assignation des opérateurs et des registres peut se faire en se basant sur les assignations intermédiaires des itérations précédentes ayant un ratio de partage de ressources plus grand. Les deux phases d'assignation des opérateurs et celle d'assignation des registres peuvent se faire dans n'importe quel ordre. Dans l'algorithme 2, apparaît le cas où l'assignation des unités fonctionnelles est faite en premier. La phase d'assignation des opérateurs de l'algorithme 2 se base sur les informations récoltées à l'itération précédente.
- Une fois la phase d'assignation des unités fonctionnelles terminée, l'assignation des registres se fait en se basant sur l'assignation des registres de l'itération précédente et de l'assignation des opérateurs de l'itération courante.
- Enfin, les multiplexeurs qui précédent les registres et les unités fonctionnelles, sont calculés. Les informations récoltées lors de cette itération (assignation des ressources, tailles des MUX et chemin critique), seront passées à l'itération suivante.

Condition d'arrêt

L'algorithme d'assignation simultanée des opérateurs et des registres s'arrête dès que b_{fu} et b_{reg} atteignent les nombres d'opérateurs et de registres préfixés.

Dans cette méthode l'inter-dépendance a beaucoup moins d'influence sur la solution finale, étant donné que ces deux tâches peuvent interagir l'une avec l'autre au lieu d'être complètement séparées.

Exemple *La figure 2.11 illustre un exemple qui met en avant l'avantage de la méthode d'assignation SFR par rapport aux algorithmes classiques. La figure 2 (a) montre six opérations du même type o_i avec $(i \in [1,6])$. Les flèches représentent la durée de vie des variables de sortie, v_i avec $(i \in [1,6])$. Pour des raisons de simplicité, seuls les multiplexeurs précédant les registres sont considérés dans l'exemple. Toutes les paires d'opérations sont compatibles entre elles et peuvent partager la même unité fonctionnelle. Les paires de variables sont aussi compatibles*

entre elles deux à deux à l'exception des variables v_1 et v_2. Supposons que les ressources disponibles comprennent deux unités fonctionnelles, et deux registres.

Pour la première itération b_{fu} et b_{reg} sont initialisés à 3. Supposons que la solution initiale résultante de la première itération est (o_1, o_2), (o_3, o_4), (o_5, o_6) pour les opérations, et (v_1, v_3), (v_2, v_4), (v_5, v_6) pour les variables.
Les opérations o_1 et o_2 sont assignées au même opérateur puisque elles sont compatibles alors que leurs produits respectifs sont assignés à deux registres différents du moment leurs durées de vie présentent un conflit. La solution intermédiaire est présentée dans la figure 2.11-b. Deux multiplexeurs à deux entrées et une sortie ont été utilisés avant les registres $R1$ et $R2$, et chacun reçoit deux variables venant chacune d'un opérateur différent. Ainsi l'assignation des opérations a conduit à la création de deux multiplexeurs. Dans la deuxième itération le nombre d'opérateurs passe à 2 ainsi que le nombre de registres. Les informations récoltées de l'itération précédente permettront de procéder à un ajustement et d'assigner les opérations o_1 et o_2 à deux opérateurs différents.

Le résultat de la seconde itération est (o_1, o_3, o_4), (o_2, o_5, o_6) pour les opérations et (v_1, v_3, v_4), (v_2, v_5, v_6) pour les registres. Ce résultat est illustrée par la figure 2.11-c. Les multiplexeurs redondant ont été éliminés suite à l'assignation progressive et l'interaction des phases d'assignation des opérations et celle d'assignation des variables. Un algorithme traditionnel, aurait obtenu les regroupements (o_1, o_2, o_3), (o_4, o_5, o_6) pour les opérations et (v_1, v_3, v_4), (v_2, v_5, v_6) pour les variables. La solution est illustrée à la figure 2.11-d. Deux multiplexeurs sont utilisés avant les registres.

2.2.2.2 Une approche évolutionniste pour l'optimisation de la surface et du temps

Cette méthode [Ferrandi 2007] propose l'application du NSGA-II [Deb 2001] pour explorer l'espace de solution et trouver l'architecture qui présente le meilleur compromis entre la surface totale occupée et le temps d'exécution.

Les étapes principales de cette méthode sont les suivantes :

- Coder une solution : un chromosome encode toute les informations nécessaires pour retrouver une solution. Ce chromosome est tout simplement un vecteur, réparti en deux parties. Les gènes de la première partie décrivent l'assignation des opérations aux opérateurs. Ceux de la deuxième partie contiennent les algorithmes d'ordonnancement, d'assignation des registres et l'allocation des interconnexions. Comme ces algorithmes sont déterministes la solution peut être reconstruite en appliquant les algorithmes.

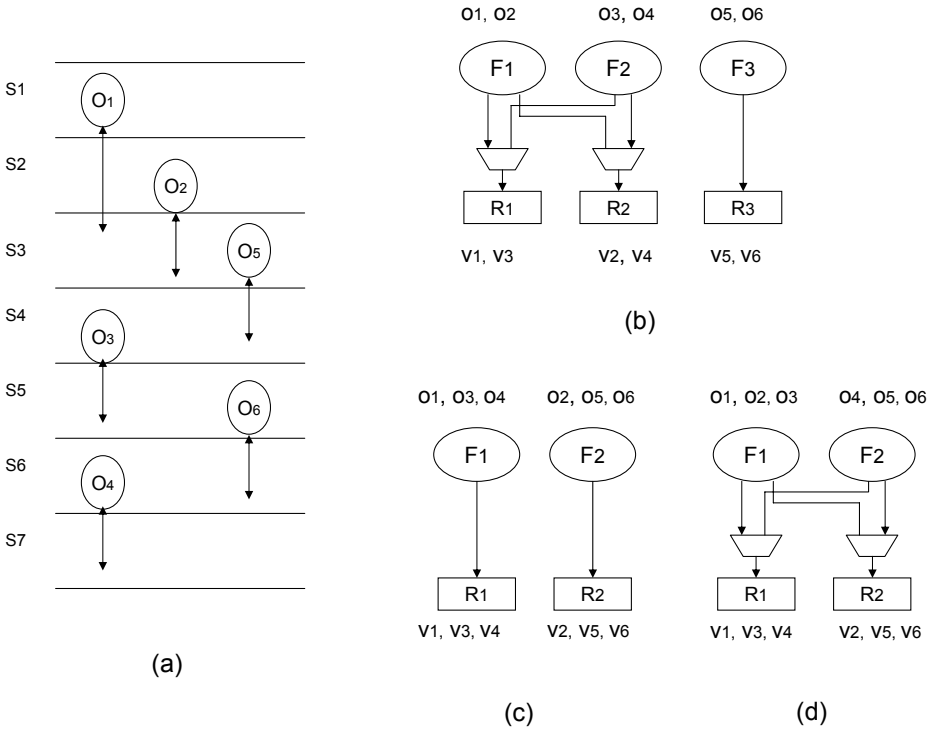


FIGURE 2.11 – Application de l'algorithme de l'assignation simultanée

– Définir la fonction de sélection. Cette fonction est multi-objectif définie comme suit :

$$F(x) = \left\{ \begin{array}{c} f_1(x) \\ f_2(x) \end{array} \right\} = \left\{ \begin{array}{c} Surface(x) \\ Latence(x) \end{array} \right\}.$$

Avec $f_1(x) = Surface(x)$ une estimation de la surface occupée par la solution x et $f_1(x) = Latence(x)$ est une estimation du pire temps d'exécution du chemin critique de la solution x .

– Construire une population initiale. Une population correspond à un ensemble de solution. La population initiale peut être construite aléatoirement en ne gardant que les solutions faisables ou à partir de solution intéressantes (avec le nombre minimum d'opérateurs par exemple).

– Trier les solutions par niveaux selon les valeurs des fonctions de coût. Les solutions non dominées (voir figure 2.12) sont classées au premier niveau. Le reste est ensuite trier en utilisant l'algorithme fast-non-dominated-sort [Deb 2001] du NSGA-II.

– Explorer l'espace de solution en utilisant les opérateurs génétiques comme la mutation et le croisement. La mutation consiste à changer un gène selon ses valeurs admissibles. Le croisement consiste à mixer deux solutions de façon à obtenir deux autres solutions admissibles.

– L'élitisme : pour garantir la diversité des solutions une distance minimum est préservée entre les solutions, en affectant un indicateur à chaque solution. Cet indicateur définits la distance qui la sépare de son plus proche voisin. Les solutions qui ont des voisins très proches seront moins prioritaires.

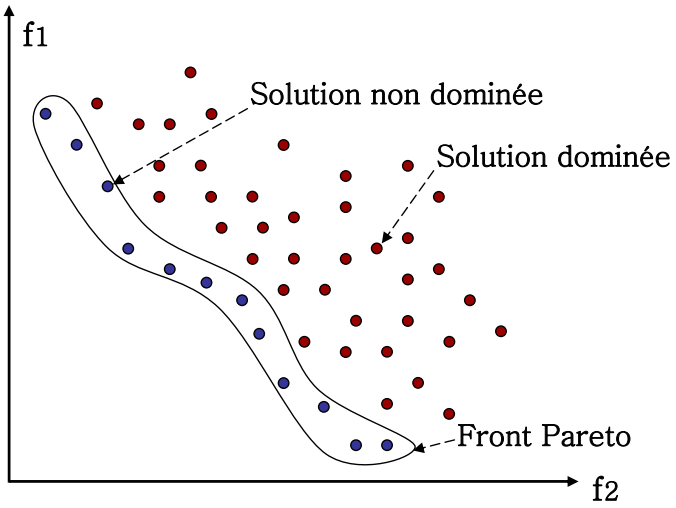


FIGURE 2.12 – Identification des solutions optimales au sens de Pareto optimal en cas de minimisation

2.3 Insuffisances actuelles et proposition

Comme mentionné précédemment, toutes les méthodes existantes à l'exception de l'approche évolutionniste, résolvent le problème de la minimisation de la surface totale d'une architecture générée par la synthèse de haut niveau en solutionnant un à un les sous problèmes de la HLS. Ces méthodes peuvent résoudre efficacement et même de façon optimale les sous problèmes. En revanche cette façon de faire empêche d'avoir une vue globale du processus de la HLS et de trouver ainsi une architecture de bonne qualité en terme de surface.

Par exemple la minimisation des opérateurs ou celle des registres ne conduit pas automatiquement à une bonne solution au problème de minimisation de la surface totale. En effet, la réutilisation d'opérateurs ou des registres peut créer des multiplexeurs qui peuvent avoir un coût supérieur à celui des opérateurs ou des registres comme le montre le tableau 2.2. Dans ce tableau on voit bien qu'un multiplexeur à huit entrées (mux24bit8_to1) occupe plus de place qu'un additionneur (additionneur24bit_cla) et qu'un multiplexeur à 32 entrées (mux24bit32_to1) a presque la même taille qu'un multiplieur.

TABLE 2.2 – Caractérisation d'un additionneur, un multiplieur et des MUX

Unités fonctionnelle et MUX	Implémentation	Surface (CLB)	Latence (ns)	Puissance (W)
additionneur24bit_cla	Carry look-head	26	11.8	0.010
multiplieur18bit_wall	Booth-recorded Wallace	280	14.8	0.308
mux24bit_2tol	Synopsys design	6	0.6	0.002
mux24bit_8tol	Synopsys design	66	4.6	0.023
mux24bit_32tol	Synopsys design	276	10.9	0.240

Ainsi, comme les méthodes exactes se retrouvent incapables de résoudre le problème de la minimisation de la surface du circuit en un temps raisonnable et que les heuristiques proposées sont dédiées aux sous problèmes de la HLS, ne résolvent que des sous parties du problème, dans ces travaux de thèse nous proposons une méthodologie permettant d'explorer un espace de solution plus large, pour choisir à la fin la meilleure solution rencontrée dans l'espace exploré. L'espace de solutions pouvant être colossal pour certaines applications, le recours aux métaheuristiques pour l'obtention du plus grand nombre de solutions. Pour l'évaluation de la qualité de chacune des solutions traduite par la surface de celle-ci, l'utilisation d'un estimateur est nécessaire. Celui ci faisant l'estimation au plus haut niveau doit être bien étudié afin de s'approcher au mieux du résultat obtenu suite à l'implémentation réelle de l'application considérée.

Chapitre 3

Estimation et recherche locale pour la synthèse de haut niveau

Sommaire

L'accroissement rapide de la complexité des applications et la multiplication du nombre de fonctions à implémenter dans un même circuit, rendent indispensable le compactage de la surface occupée par l'implémentation d'une application donnée . Dans ce chapitre, nous présentons la première approche que nous avons proposée pour optimiser la surface du circuit à implémenter. C'est une approche appartenant à la famille des méthodes de recherche locale qui permet l'exploration de l'espace des solutions. Elle repose sur une fonction d'évaluation des solutions. Dans notre cas, l'évaluation se fait en estimant la surface occupée dans le FPGA après implémentation de la solution. Ainsi, nous proposons un estimateur qui nous évitera de réaliser systématiquement la synthèse logique, qui est coûteuse en temps de calcul.

3.1 Introduction

Comme toutes les méthodes de recherche locale ou heuristiques à base de voisinages la méthode de descente s'appuie sur :

- l'obtention d'une solution x_0, considérée comme un point de départ (calculée par exemple par une heuristique constructive).
- le passage d'une solution à une solution voisine par déplacements successifs.

Dans les sections suivantes, nous allons définir ce que nous appelons une solution, comment évaluer cette solution, ce qu'est un voisinage et enfin quelles sont les techniques d'exploration de ce voisinage.

3.2 Représentation d'une architecture

Une solution est une architecture capable d'exécuter l'application considérée. Cette architecture est définie une fois que les dates de début des opérations ainsi que l'assignation des opérations et des variables sont connues. En réalité, pour une application donnée, il peut exister différentes solutions.
Toute solution est évaluée par sa surface, et se caractérise par :

- le nombre et le type d'opérateurs qui vont exécuter les tâches de l'application,
- le nombre et le type de registres qui vont sauvegarder les valeurs des variables,
- le nombre et le type de multiplexeurs qui vont gérer l'acheminement des variables en cas de réutilisation des opérateurs, et/ou des registres.

Reprenons l'exemple de la figure 2.4 présenté dans le chapitre II. La figure 3.2 illustre une solution réalisable s au problème d'implémentation de l'application de la figure 2.4. Dans cette solution quatre registres, un additionneur et un multiplexeur sont utilisés. Ainsi, on peut définir une solution comme un ensemble de composants (opérateurs, registres et de multiplexeurs) et leurs interconnexions. Le nombre d'opérateurs et celui des registres découlent directement des résultats des étapes d'ordonnancement, d'assignation des opérations et des variables.

Un multiplexeur est connecté à l'entrée d'un registre si plus d'un opérateur écrit son résultat dans ce registre. De même pour les opérateurs, un multiplexeur précède une entrée d'un opérateur (un opérateur peut avoir plus d'une entrée) si plus d'un registre envoie une valeur à cette entrée. Ainsi pour estimer la surface totale de l'architecture on a besoin de deux informations :

- le type et le nombre des composants de l'architecture.
- la surface (en portes logiques) de chacun des composants pouvant être utilisés.

3.2.1 Un estimateur de surface d'architecture

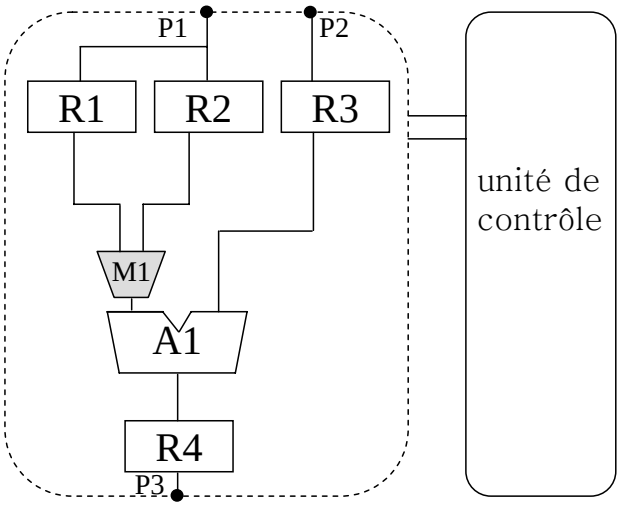


FIGURE 3.1 – Exemple d'un circuit

En réalité un circuit est constitué d'un chemin de données et d'une unité de contrôle (voir figure 3.1). Le chemin de donnée appelé aussi unité de traitement est l'ensemble des opérateurs, des registres et des multiplexeurs. L'unité de contrôle est une machine d'états qui se charge du pilotage de tous les composants du chemin de données. Dans la figure 3.1, dans la partie unité de traitement nous trouvons :

- trois registres (R1, R2 et R3) qui stockent les variables qui arrivent des ports d'entrée P1 et P2,
- un additionneur A1,
- un multiplexeur M1 qui va aiguiller le cheminement des variables stockées dans les registres (R1 et R2) vers l'entrée droite de l'additionneur A1,
- un registre R4 qui va sauvegarder le résultat des additions.

L'unité de contrôle sera responsable de :

- charger une variable dans le registre R1 ,R2, R3 et R4,
- piloter le multiplexeur M1 de façons qu'il chemine la variable désirée vers l'entrée droite de l'additionneur A1.

Dans notre étude nous allons nous focaliser sur la minimisation de la surface de l'unité de traitement.

3.2.1.1 Dénombrement des composants du chemin de données

Pour dénombrer les composants de l'architecture on procède de la manière suivante. Pour les opérateurs, le nombre est fixé de manière explicite à la fin des phases d'ordonnancement

et d'assignation des opérations. Quant au nombre de registres, il est également défini de façon explicite à la fin de la phase d'assignation des variables. Reste à trouver le type et le nombre de multiplexeurs. Celui-ci peut être déduit à partir de la grille de connexions que nous proposons.

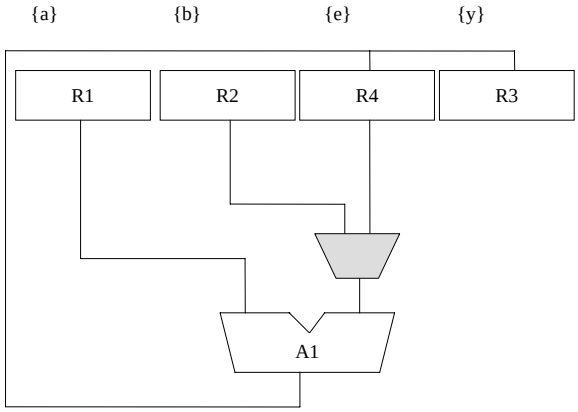


FIGURE 3.2 – Exemple d'un circuit

Dest./Sourc.		$A1$	$R1$	$R2$	$R3$	$R4$	Type du Mux
$A1$	$A11$	0	1	0	0	1	
	$A12$	0	0	1	1	0	(2 :1)
	$R1$	0	0	0	0	0	
	$R2$	0	0	0	0	0	
	$R3$	1	0	0	0	0	
	$R4$	1	0	0	0	0	
	Composants	1add	1reg	1reg	1reg	1reg	1 mux (2 :1)

TABLE 3.1 – Évaluation d'une solution

Le tableau 3.2.1.1 est une fidèle représentation de l'architecture de la figure 3.2. Les lignes du tableau correspondent aux entrées des opérateurs (A11 et A12 sont les entrées de l'additionneur A1) et des registres, alors que les colonnes correspondent à leurs sorties. Une case du tableau contient le nombre de variables qui transitent entre la sortie et l'entrée correspondante. Une connexion sera créée entre une source et une destination si la case correspondante contient un entier non nul. Si une destination est desservie par plus d'une seule source, un multiplexeur devra être introduit.

La deuxième ligne du tableau 3.2.1.1 nous indique l'entrée A12 de l'additionneur A1 nécessite un multiplexeur de deux entrées et une sortie noté (2 :1). Celui-ci se chargera d'aiguiller les

variables arrivant des sorties des registres R2 et R3. Ainsi, ce tableau nous permet de compter tous les composants de l'architecture : opérateurs, registres et multiplexeurs. Si l'on connaît la surface de chacun de ces composants, on peut estimer la surface totale occupée par le chemin de données, une fois l'application implémentée sur le FPGA cible.

3.2.1.2 Caractérisation des composants

La caractérisation est une étape indispensable pour la construction d'une bibliothèque qui nous indiquera les caractéristiques (surface occupée, latence...) de chaque composant (additionneur, multiplieur...) une fois implémenté sur le FPGA. La mesure de la surface occupée dans un FPGA de la famille Virtex-II Pro ou un Virtex-II Pro x de Xilinx peut se faire en utilisant plusieurs unités : CLB, *slices*, *LUT* et *gates* [Xilinx 2007].

En effet, la structure de base des FPGA des différentes familles Xilinx (voir figure 3.3) est

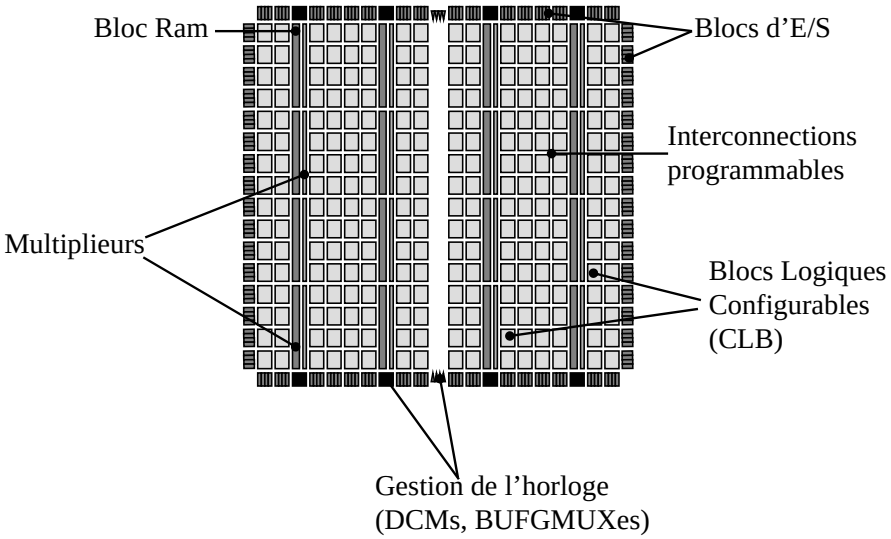


FIGURE 3.3 – FPGA

généralement la même, à savoir constituée de :

- CLB : blocs logiques programmables (configurables),
- IOB : blocs logiques d'entrées/sorties,
- Mémoires RAM,
- Horloges DLL pour le réseau de distribution,
- ressources de routages,
- opérateurs câblés,
- ...

Ainsi, la configuration des blocs logiques programmables consiste à mettre en oeuvre les opérateurs, les registres, les multiplexeurs et les connecter de façon à obtenir un circuit capable d'exécuter l'application désirée. Comme le montre la figure 3.4, les CLB disposent de quatre *slices*.

Dans les FPGA utilisant les LUT à quatre entrées, on trouve dans chaque *slice* :
(1) deux générateurs de fonction à 4 entrées (LUT : Look Up Table) permettant la réalisation de toute fonction booléenne des 4 entrées. Ces LUT peuvent être considérées comme des cellules de RAM de 16 fois 1 bit, réalisant la table de vérité de la fonction logique souhaitée ;
(2) deux registres de un bit.
(3) des multiplexeurs de largeur un bit appelés MUXF5 et MUXFi (avec i>5). Ces multiplexeurs connectent les LUTs entre eux pour créer les composants [Xilinx 2007]. Ces éléments peuvent être aussi quantifiés en nombre de portes logiques. Une porte logique désigne un circuit qui effectue une opération booléenne.

Le choix de l'unité que nous utiliserons pour évaluer le coût de la surface totale d'un circuit est celle que nous utiliserons pour la caractérisation des composants. N'ayant encore pas fait ce choix, à ce niveau, nous construirons trois bibliothèques. Dans la première le champ surface est caractérisé en *slice*, dans la deuxième en LUT et dans la troisième en portes logiques (*gates*). Concrètement, nous implémentons (jusqu'à la synthèse logique) chaque composant à part, pour pouvoir relever ses caractéristiques et remplir nos trois bibliothèques.

3.2.2 Validation de l'estimateur

Une fois construit, le tableau dénombrant les composants d'une architecture et la bibliothèque de composants complétée, il convient de vérifier la validité de notre estimateur. Pour ce faire, nous avons choisi une batterie de tests de trente instances. Une instance est définie par l'application qu'elle implémente, l'intervalle d'initiation (II) (i.e. la cadence) de l'application et la technique d'allocation utilisée dans le flot de synthèse (allocation globale/distribuée détaillées dans la section suivante). L'intervalle d'initiation est le temps qui sépare le traitement de deux jeux données successifs. En effet, en variant la cadence pertinemment nous obtenons un échantillon de cas de figures dissemblables d'utilisation des applications testées (voir 3.4.1.1).

Pour valider notre estimateur, il faut lancer la synthèse de haut niveau suivie de la synthèse logique pour ces trente instances. À la fin de la phase de la synthèse de haut niveau, nous obtenons une estimation de la surface, alors qu'à la fin de la synthèse logique nous obtenons la surface réelle occupée par l'implémentation de l'application sur le FPGA cible. L'étape suivante consiste à comparer les valeurs réelles avec celles estimées. Les tests de validation doivent être

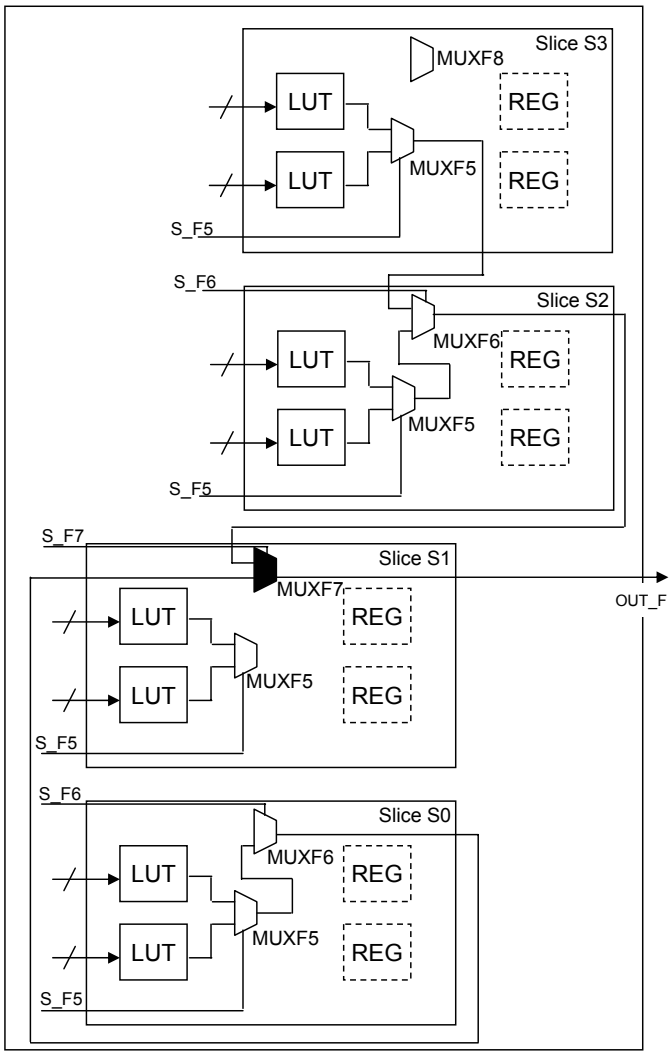


FIGURE 3.4 – Un CLB

réalisés pour les trois unités (*slice*, LUT et *gate*). Mais avant de lancer ces tests de validation de l'estimateur, nous avons fait une petite étude sur les correspondances entre options de synthèse logique et unité de mesure.

3.2.2.1 Quelles sont les options de synthèse logique ?

Pour faire la synthèse logique d'une architecture RTL, deux options sont possibles : l'option hiérarchique et l'option à plat. L'option hiérarchique n'exploite pas les parties non consommées des *slices* déjà utilisés pour réaliser un autre composant (opérateur ou multiplexeur). Ceci se traduit pratiquement par le fait que dans un *slice* sont soit les registres ou les LUTs qui sont utilisés et que deux composants différents ne peuvent pas partager les LUTs d'un même *slice*. A l'opposé de l'option hiérarchique, l'option à plat exploite les parties non consommées des

slices déjà utilisés pour faire un nouveau composant.
La figure 3.5 montre un additionneur de huit bits (A1) et un registre de huit bits (R1), implémentés avec l'option hiérarchique. La figure 3.5 montre les même composants implémentés cette fois avec l'option à plat. La première implémentation utilise huit *slices* pour réaliser les deux composants, quatre pour implémenter l'additionneur (A1) et quatre pour implémenter le registre (R1). La deuxième implémentation utilise quatre *slices* pour réaliser les deux composants.

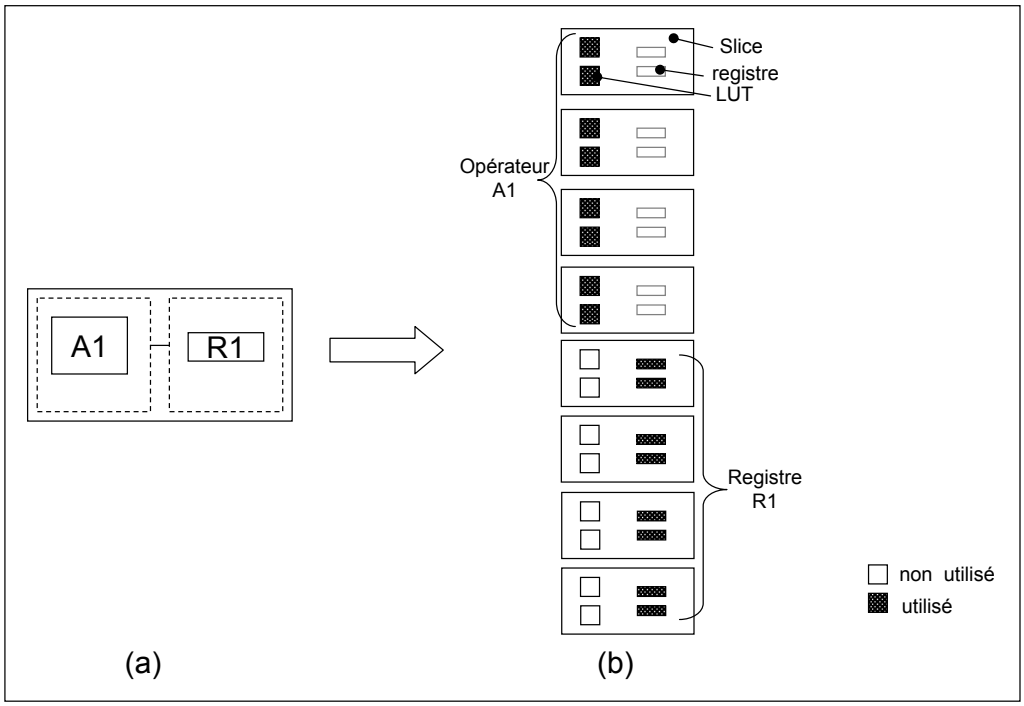


FIGURE 3.5 – Implémentation avec l'option hiérarchique

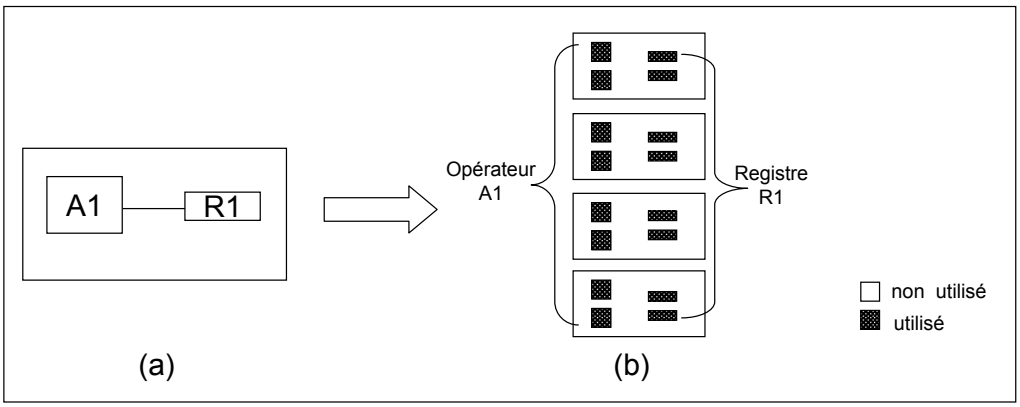


FIGURE 3.6 – Implémentation avec l'option à plat

3.2.2.2 Correspondances (options de synthèse logique/ unité de mesure)

Le choix de la caractérisation de la bibliothèque (*slice*, LUT or *gates*) sera influencé par les options de synthèse logique (à plat ou hiérarchique). En effet, certaines combinaisons (unité de caractérisation/option d'implémentation) ne sont pas valides comme le montre le tableau3.2. Les combinaisons (2) et (4) ne sont pas pertinentes. Comme le principe de la synthèse hiérarchique est d'implémenter le composant en condamnant la valeur entière supérieure du nombre de *slices* nécessaires à son implémentation, le nombre de *gates* (respectivement LUT) pouvant être condamnés peut être considérablement supérieur au nombre de *gates* (respectivement LUT) estimé. Ce dernier se rapproche plutôt du nombre de *gates* (respectivement LUT) réellement utilisées. La combinaison (5) n'est pas valide non plus. Dans ce cas, nous comparons la somme des parties entières correspondant à la surface totale estimée, à la partie entière de la somme correspondant à la surface totale réelle avec une implémentation à plat. Par exemple, prenons une architecture simple composée d'un registre de taille un bit et un opérateur qui utilise deux LUT. La bibliothèque nous indiquera que chacun de ces composants utilise un *slice*. La surface totale estimée sera ainsi égale à deux *slices*, alors que avec l'option à plat ces deux composants seront implémentés dans le même *slice*.

Carac./Option.	à plat	hiérarchique
gates	(1)OK	(2)NOK
LUTs	(3)OK	(4)NOK
slices	(5)NOK	(6)OK

TABLE 3.2 – Compatibilité (unité de caractérisation/option d'implémentation)

3.2.2.3 Courbes de validation de l'estimateur

Pour valider notre approche d'évaluation, nous avons comparé une estimation de la surface d'une solution donnée, avec la surface réelle retournée à la fin de la phase de la synthèse logique. Construire un estimateur précis est difficile, car nous ne maitrisons pas ce qui se passe durant la phase de la synthèse logique. Nous avons ainsi choisi d'utiliser l'unité de mesure qui nous rapproche le plus des valeurs réelles. Les figures 3.7, 3.8 et 3.9 montrent respectivement les courbes de validation des trois séries de tests (*gates*/à plat) (LUTs/à plat) (*slices*/hiérarchique). Les instances utilisées pour valider l'estimateur sont les instances définies dans la section .

La courbe de validation de l'estimateur en utilisant l'unité *slice* (voir figure 3.7) montre une erreur maximum de 7%, une erreur minimim de -2% et une erreur moyenne égale à 0,1%. La courbe de validation de l'estimateur en utilisant l'unité LUT (voir figure 3.8) montre une

erreur maximum de -83%, une erreur minimim de -10% et une erreur moyenne égale à -51%. La courbe de validation de l'estimateur en utilisant l'unité *gate* (voir figure 3.9) montre une erreur maximum de 2%, une erreur minimim de -3% et une erreur moyenne égale à 0,5%.

Les trois mesures sont pertinentes, cependant nous ne pouvons pas utiliser l'unité LUT car le modèle correspondant peut largement s'écarter des valeurs réelles. Par conséquent, notre choix doit se faire entre les unités *slice* et *gate*. Nous avons enfin choisi d'utiliser l'unité de mesure *gate* dont le modèle présente un minimum d'écart entre l'erreur maximum et l'erreur minimum.

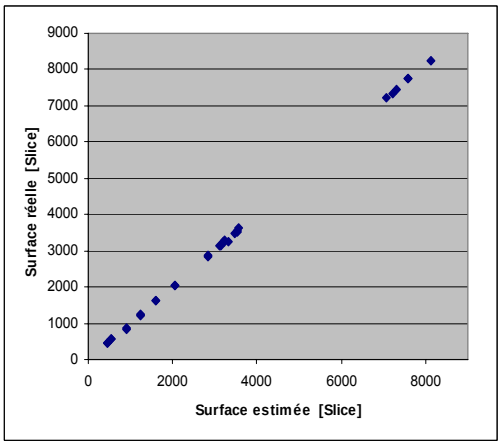


FIGURE 3.7 – Validation du modèle d'estimation en *slice*/ hiérarchique

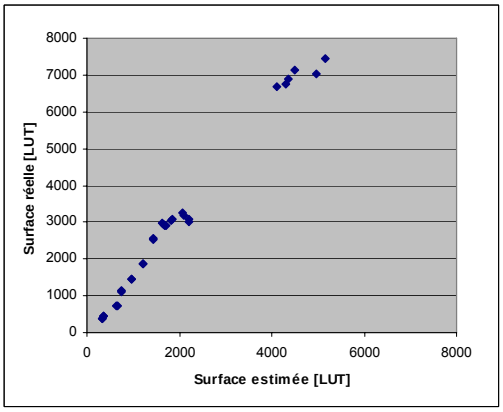


FIGURE 3.8 – Validation du modèle d'estimation en *LUT*/ à plat

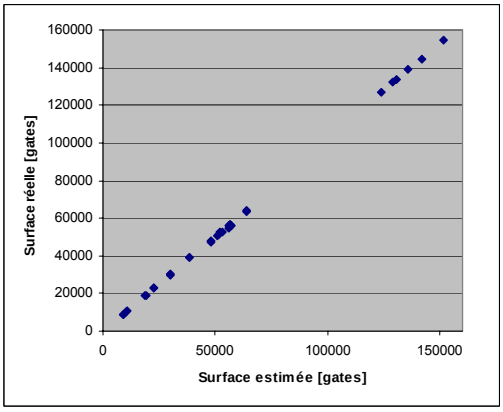


FIGURE 3.9 – Validation du modèle d'estimation en *gate*/ à plat

3.3 Méthode de descente

Le modèle d'estimation de la surface validé, l'évaluation de la qualité d'une solution se fait rapidement, sans passer par la longue phase de synthèse logique. L'application d'une méthode de descende devient donc possible. Dans cette section, nous allons décrire le principe des méthodes de descente et appliquer cette approche à la problématique de la minimisation de la surface des circuits.

3.3.1 Présentation de la méthode

À partir d'une solution initiale trouvée par une heuristique par exemple, on peut très facilement implémenter des méthodes de descente. Les méthodes de descente s'articulent toutes autour d'un principe simple. Partir d'une solution existante, chercher une solution dans le voisinage et accepter cette solution si elle améliore la solution courante (voir figure 3.10). L'algorithme 3 présente les étapes d'une méthode de descente simple. À partir d'une solution

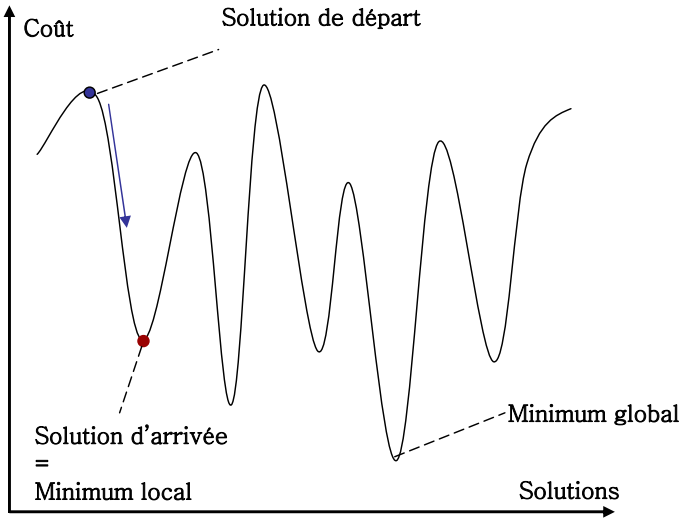


FIGURE 3.10 – Illustration de la méthode de descente

initiale x, on choisit une solution x' dans le voisinage $N(x)$ de x. Si cette solution est meilleure que x, $(f(x') < f(x))$ alors on accepte cette solution comme nouvelle solution x et on répète le processus jusqu'à ce qu'il n'y ait plus aucune solution améliorante dans le voisinage de x.

Une autre version de la méthode de descente est la méthode de plus grande descente. Au lieu de choisir une solution x' dans le voisinage de x, on choisit toujours la meilleure solution x' du voisinage de x. L'algorithme 4 donne une description de cette méthode.

Algorithme 3 : Algorithme de descente simple

Initialisation : Trouver une solution initiale x;

Répéter;

exploration du voisinage : trouver une solution $x' \in N(x)$;

Si $f(x') < f(x)$ **alors**;

$x \leftarrow x'$;

Fin si;

Jusqu'à $f(y) \geq f(x), \forall y \in N(x)$;

Algorithme 4 : Algorithme de la plus grande descente

Initialisation : Trouver une solution initiale x

Répéter

exploration du voisinage : trouver une solution $x' \in N(x)$,

$f(x') \leq f(x"), \forall x" \in N(x)$

Si $f(x') < f(x)$ **alors**

$x \leftarrow x'$

Fin si

Jusqu'à $f(x') \geq f(x), \forall x' \in N(x)$

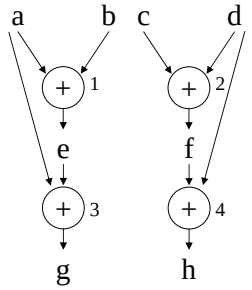
3.3.2 Application de la méthode de descente à la HLS

Dans cette section nous allons détailler le processus de construction de la solution initiale et les voisinages explorables. Nous illustrerons chacune des étapes de construction de la solution initiale ainsi que les différentes techniques de passage aux solutions voisines. Nous utiliserons l'exemple de la figure 3.11 avec une période d'horloge de 10 ns et un intervalle d'itération de 20 ns (soit deux cycles) .

```
Void  example (int a, int b, int g,
                    int c, int d, int h){

          int e,f;
          e = a+b;
          g = a+e;
          f = c+d;
          h = f+d;
}
```

(a) Description algorithmique



(b) DFG

FIGURE 3.11 – Exemple

3.3.2.1 La solution initiale

La solution initiale est générée par l'outil de synthèse de haut niveau GAUT [Coussy 2008]. Cette solution est obtenue suite au processus suivant :

La sélection : la procédure de sélection construit :

- une structure de données contenant les caractéristiques des différents opérateurs (surface, temps d'exécution et la liste des fonctions réalisables),
- une deuxième structure de données contenant les différentes alternatives de sélection. Une alternative valide doit contenir des opérateurs, choisis dans la première structure de données, de façon à exécuter toutes les fonctions du graphe.

Une fois toutes les alternatives définies, une évaluation du coût de chaque possibilité en terme de surface est faite. Celle-ci s'appuie sur un calcul du nombre d'opérateurs utilisé lors de l'ordonnancement ASAP. À la fin du processus d'évaluation des alternatives, le choix se fait sur celle qui a le moindre coût (surface opérateurs) tel que les latences des opérateurs respectent la contrainte de temps.

Application à l'exemple de la figure 3.11

Dans l'exemple de la figure 3.11, nous avons un seul type d'opération (addition). Prenons

```
...
operator(add_op) {
                    function            add;
                    component           add_op;
                    type                combinatorial;
                    propagation_time    5;
                    combinational_delay 7;
                    area                186;
                    power               0;
}

operator(addsub_op) {
                    function            add ,sub;
                    component           addsub_op;
                    type                combinatorial;
                    propagation_time    5 ,5;
                    combinational_delay 7 ,7;
                    area                471;
                    power               0 ,0;
}
...
```

FIGURE 3.12 – Bibliothèque

l'exemple d'une bibliothèque (figure 3.12) qui ne contient que deux opérateurs (add_op et addsub_op) capables d'exécuter ce type d'opération. Pour cet exemple le processus de sélection est très simple. En effet, les différents opérateurs de la bibliothèque de la figure 3.12 ainsi que leurs caractéristiques seront recopiés fidèlement dans une première structure de donnée. Ensuite, une

deuxième structure de donnée regroupera les différentes alternatives. Chaque alternative est caractérisée par sa latence et sa surface. Dans cet exemple deux alternatives sont possibles, soit utiliser l'opérateur add_op ou bien l'opérateur addsub_op. Les latences des ces deux alternatives, suite à un ordonnancement ASAP, sont égales à 20ns. Les surfaces des deux alternatives sont respectivement 189 *gates* et 471 *gates*. Les deux alternatives ont la même latence lors de l'ordonnancement ASAP. Cependant, elles ont des coûts (surfaces) différents. Le choix du processus de sélection se porte donc sur la première alternative qui consiste à utiliser l'opérateur add_op.

L'allocation : Deux techniques d'allocation sont mises en oeuvre dans GAUT :

- L'allocation distribuée : consiste à allouer un nombre initial minimum (une borne inférieure) d'opérateurs sur chaque tranche de pipeline. Deux étapes pour trouver le nombre d'opérateurs à allouer par tranche :
  (1) Calculer le nombre de tranches *a priori*. Celui-ci est obtenu par le ratio entre la latence minimum L (définie par le chemin critique du graphe) et la cadence C (donnée par les contraintes du problème, elle correspond à l'intervalle d'itération de l'application) : $\lceil L/C \rceil$.
  (2) Calculer le parallélisme moyen de l'application. Celui-ci est déduit suite à l'ordonnancement ASAP. Le parallélisme moyen est calculé pour chaque tranche de pipeline et pour chaque type d'opération appartenant à la tranche de pipeline considérée séparément. Pour chaque type d'opération et chaque tranche de pipeline, le parallélisme moyen calculé définit le nombre d'opérateur minimum à allouer pour la tranche considérée ;
- l'allocation globale : la même technique d'allocation est utilisé pour l'allocation globale, cependant celle-ci considère l'ensemble des opérations du graphe indépendamment des tranches auxquelles elles appartiennent.

Application à l'exemple de la figure 3.11

- L'allocation distribuée : Le nombre de tranches dans l'exemple de la figure 3.11 est égal à un. Ce nombre est le résultat du ratio $\lceil L/C \rceil$, avec $L = 20$ns et $C = 20$ns. Suite à un ordonnancement ASAP, une répartition des opérations sur les tranches de pipeline est faite (dans notre exemple une tranche). On observe que, au plus, deux additions peuvent s'exécuter en même temps. Le résultat de l'allocation distribuée est ainsi égal à deux.
  Si nous varions la cadence à $C = 10$ ns, nous obtiendrons un nombre de tranches égal à deux. Après la répartition des opérations sur les tranches, nous observons que **pour chaque tranche** deux additionneurs sont nécessaires. Le résultat de l'allocation distribuée pour les opérations d'addition, dans ce cas, sera égal à quatre (deux pour chaque tranche de pipeline).
- L'allocation globale : divise le nombre total d'opérations de tout le graphe par le nombre

de cycle de la cadence. Pour une cadence $C = 10$ ns, le nombre de cycles dans une tranche de pipeline sera donc égal à un. Le nombre d'additionneurs nécessaires est ainsi égal à quatre.

L'ordonnancement : Cette étape s'appuie sur un algorithme de liste. Pour un cycle donné, les opérations dites prêtes (opérations dont l'exécution de tous les prédécesseurs a été achevée) sont ordonnées par priorité. La priorité est définie par ce que l'on appelle la mobilité. Celle-ci est l'intervalle de temps [cycle courant, date de début d'ordonnancement ALAP]. Ainsi, les opérations les plus prioritaires sont ordonnancées dans le cycle courant.

Application à l'exemple de la figure 3.11

Le nombre d'opérateurs (additionneurs) étant fixé à deux, la cadence $C = 20$ns, cet algorithme de liste donne le résultat illustré dans la figure 3.13.

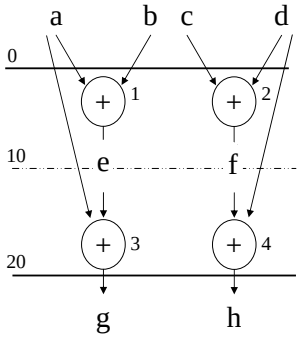
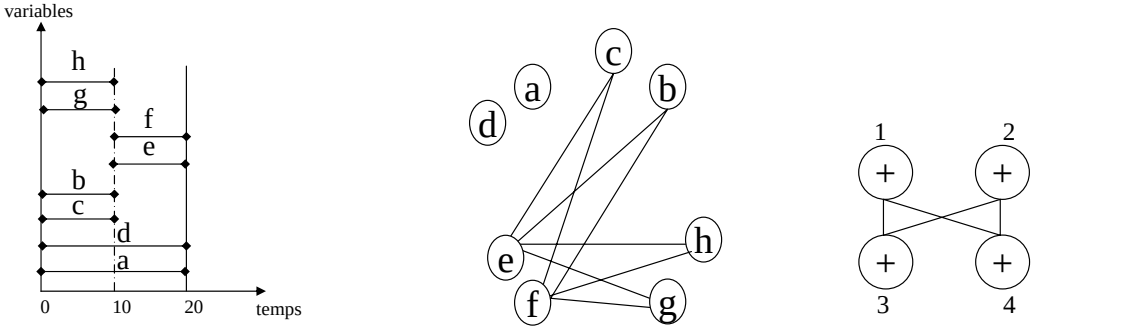


FIGURE 3.13 – Le DFG de la figure 3.11 ordonnancé



(a) Durées de vie des variables

(b) Graphe de compatibilité des variables

(c) Graphe de compatibilité des opérations

FIGURE 3.14 – Graphes de compatibilité déduis de l'ordonnancement de la figure 3.13

L'option retenue pour l'assignation des opérations : Au moment de l'ordonnancement d'une opération on choisit le premier opérateur rencontré capable d'exécuter l'opération. Celui-ci doit être libre pendant le cycle considéré et capable d'exécuter la fonction voulue. Ainsi l'assignation des opérations se fait aléatoirement, sans se soucier de la création de multiplexeurs.

Application à l'exemple de la figure 3.11

En s'appuyant sur les informations résultant de la phase d'ordonnancement (voir figure 3.14-c) une alternative d'assignation des opérations est illustrée dans la figure 3.15. Les opérations (1) et (4) sont assignées à l'opérateur A_1, tandis que les opérations (2) et (3) sont assignées à l'opérateur A_2. Cette assignation ne change pas le nombre d'opérateurs défini lors de la phase d'allocation.

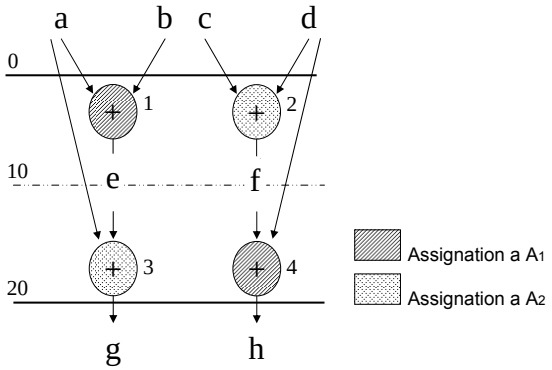


FIGURE 3.15 – Assignation des opérations de l'exemple de la figure 3.11

L'option retenue pour l'assignation des variables aux registres : chaque variable est assignée à un nouveau registre. On aura, par conséquent, autant de registres que de variables.

Application à l'exemple de la figure 3.11

Ainsi la solution de départ (voir figure 3.16) est obtenue par un ordonnancement de liste et des assignations des opérations et des variables expliquées. Dans cette solution, deux opérateurs, huit registres et quatre multiplexeurs (2 :1) sont utilisés. En supposant que les coûts en *gates* d'un registre, d'un additionneur et d'un multiplexeur (2 :1) sont respectivement 131, 186 et 96 le coût total éstimé de la solution initiale est égal à 1804 *gates*.

3.3.2.2 Les voisinages

Définir un voisinage revient à définir une technique simple et rapide qui transforme une solution admissible x en une solution différente x' toujours admissible.

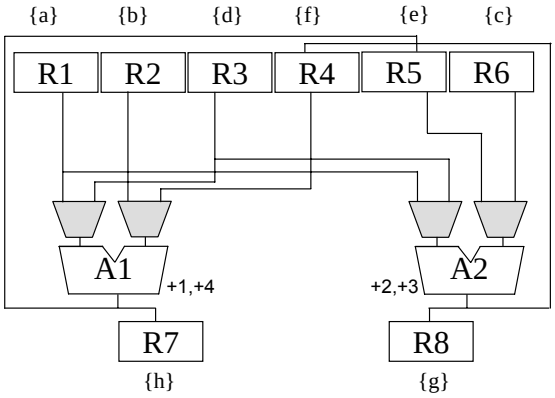


FIGURE 3.16 – Solution de départ

Réassignation des registres : comme nous partons d'une solution qui utilise autant de registres que de variables, ce voisinage est le plus large. Pour obtenir la nouvelle solution x', nous choisissons une variable d_i assignée à un registre R_i et nous l'assignons à un registre existant R_j si toute les variables déjà assignées à R_j sont compatibles avec la variable d_i. Comme expliqué dans le paragraphe 2.1.3 du chapitre II, deux variables sont compatibles si leurs durées de vie ne se chevauchent pas. Une fois l'assignation de la variable changée, trois scénarios sont possibles :
(1) une connexion est supprimée. La surface totale diminue, la nouvelle solution est acceptée,
(2) le nombre d'interconnexions ne diminue pas. La solution obtenue n'est pas acceptée,
(3) un registre est vide. Si le coût des interconnexions n'augmente pas alors la nouvelle solution est acceptée, sinon elle est rejetée.

Application à l'exemple de la figure 3.11

Prenons la variable e assignée au registre R_5, et essayons de la déplacer vers un autre registre. En se basant sur le graphe de compatibilité des variables de la figure 3.14, la variable e peut être déplacée vers un des registres suivants : R_2, R_6, R_7 ou R_8. La figure 3.17 montre la nouvelle architecture obtenue après la déplacement de la variable e vers R_2.

La nouvelle architecture utilise un registre de moins que la solution initiale. Ainsi le coût de la nouvelle architecture est égal à 1673 *gates*. Cette suppression du registre a réduit la surface de l'unité de traitement, mais elle doit aussi réduire la surface de l'unité de contrôle comme il y aura un registre de moins à piloter.

Permutation de l'assignation de deux opérations : pour obtenir une nouvelle solution, il suffit de choisir deux opérations, vérifier si elles sont compatibles et intervertir l'assignation des deux opérations. Dans ce voisinage, deux opérations sont compatibles si, d'une part, elles peuvent être exécutées par le même opérateur et si elles ont la même date de début d'exécution d'autre part. La nouvelle solution n'est acceptée que si le coût des intreconnecteurs décroit

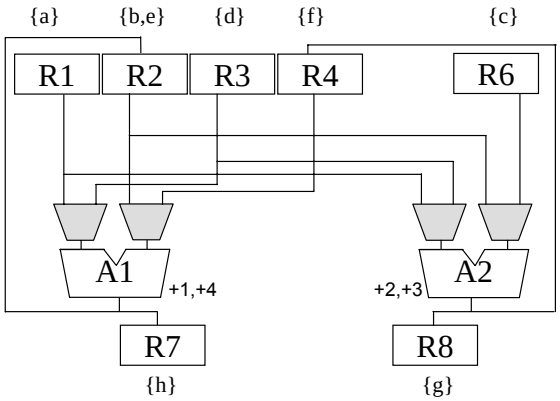


FIGURE 3.17 – Architecture obtenue après la déplacement de la variable e vers R_2

strictement. Le nombre d'opérateurs à la fin de cette étape restera inchangé.

Application à l'exemple de la figure 3.11

Prenons l'opération +4 assignée à l'additionneur A_1, et l'opération +3 assignée à l'additionneur A_2 et permutons les. La figure 3.18 montre la nouvelle architecture obtenue après la permutation des opérations +4 et +3.

La nouvelle architecture utilise deux multiplexeurs de moins que la solution initiale. Ainsi le

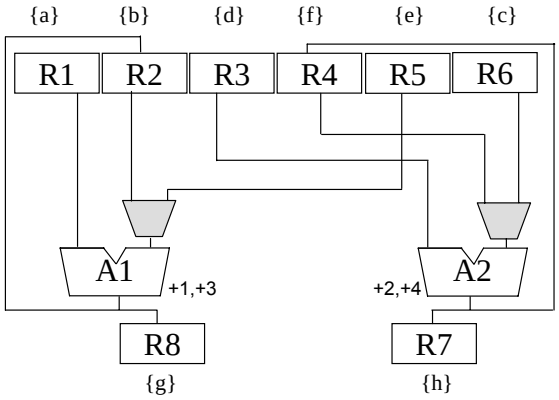


FIGURE 3.18 – Architecture obtenue après la permutation des assignations des opérations A_1 et A_2

coût de la nouvelle architecture est égal à 1673 *gates*. La suppression des deux multiplexeurs a réduit la surface de l'unité de traitement, mais elle doit aussi réduire la surface de l'unité de contrôle comme il y aura deux multiplexeurs de moins à piloter.

Réassignation des opérations : ce voisinage est exploré de la manière suivante : dans un cycle donné, une opération est choisie ainsi qu'un opérateur inactif capable d'exécuter l'opéra-

tion sélectionnée sans décaler sa date de début. Si un tel opérateur existe, l'opération lui est assignée. On obtient, par conséquent, une nouvelle solution. Celle-ci n'est acceptée que si le coût des interconnecteurs décroit strictement.

Application à l'exemple de la figure 3.11

Dans notre cas, aucun mouvement de ce type n'est possible. En effet, les deux additionneurs sont constamment utilisés.

Réallocation des opérateurs : cette technique permet l'ajout d'opérateurs puis la réassignation des opérations comme expliqué dans la technique précédente.

Application à l'exemple de la figure 3.11

La figure 3.18 montre la nouvelle architecture obtenue après l'ajout d'un opérateur A_3 et l'assignation de l'opération +4 à celui-ci.

La nouvelle architecture utilise deux multiplexeurs de moins et un additionneur de plus que la

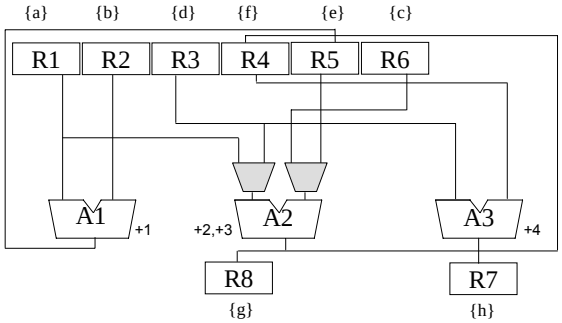


FIGURE 3.19 – Architecture obtenue après l'ajout d'un additionneur et le déplacement de l'opération +4 vers cet opérateur

solution initiale. Ainsi le coût de la nouvelle architecture est égal à 1798 *gates*.

Réordonnancement cette technique d'exploration du voisinage consiste à choisir une opération, lui définir un intervalle de mobilité en tenant compte des dates de fin de tous ses prédécesseurs et des dates de début de tous ses successeurs. Une fois l'opération réordonnancée, une nouvelle solution est obtenue. Celle-ci est acceptée si la somme du coût des opérateurs et de celui des interconnexions décroit strictement.

3.4 Résultats

Dans un premier temps, nous avons décidé d'implémenter l'algorithme de descente simple (N1 simple) et celui de la plus grande descente en explorant le voisinage des registres (N1 deepest). Ces deux algorithmes de recherche locale ont été implémentés en C++, dans un module d'optimisation, intégré à l'outil GAUT. Le tableau 3.3 illustre les coûts des solutions trouvées avec ces deux algorithmes et les compare aux coûts des solutions initiales S_0 et des coûts trouvés avec les algorithmes left-edge (LEA) et le left-edge modifié (MLEA), décrits dans le chapitre précédent.
Rappelons que les objectifs des algorithmes de ces algorithmes sont :

- la solution initiale utilise le nombre maximum de registres. Ce nombre est égal au nombre de variables utilisées dans l'application implémentée,
- le Left Edge utilise le nombre minimum de registres sans se soucier du coût en multiplexeur que celui-ci peut engendrer,
- le Left Edge Modifié utilise le nombre minimum de registres en tentant de minimiser le coût en multiplexeurs engendré par la réutilisation des registres,
- la descente simple et la plus grande descente tentent de trouver le nombre de registres qui minimise la surface totale de l'architecture.

3.4.1 Applications et technologie cible

3.4.1.1 Applications

Les applications choisies font partie des algorithmes typiques du domaine du traitement du signal et de l'image.
Le CORDIC sigle de COordinate Rotation DIgital Computer, pour calcul numérique par rotation de coordonnées, est un algorithme de calcul des fonctions trigonométriques et hyperboliques.
Une transformée de Fourier rapide(ou Fast Fourier Transform : FFT) est une réorganisation du calcul d'une transformée de Fourrier discrète (Direct Fourier Transform :DFT) dans le but de réduire la complexité calculatoire de celle-ci. Une DFT est utilisée pour convertir une séquence de données du domaine temporel vers sont équivalent en domaine fréquentiel.
Le produit de matrices : les instances que nous avons utilisés sont des produits de deux matrices carrées de tailles 2X2 pour le prodmat2x2, et 4x4 pour le prodmat4x4.
Les équations de Lotka-Volterra En mathématiques, les équations de Lotka-Volterra, que l'on désigne aussi sous le terme de "modèle proie-prédateur", sont un couple d'équations différentielles non-linéaires du premier ordre.

3.4.1.2 Choix de la technologie cible

Le choix de la technologie FPGA a été contraint par les outils de synthèse logique et les cibles matérielles disponibles au laboratoire. Nous avons utilisé un virtex II pro de chez Xilinx pour nos expériences.

3.4.2 Résultats et interprétations

Le tableau 3.3 montre les surfaces estimées en *gates* ainsi que les temps CPU des solutions obtenues après l'application des algorithmes Left Edge (LEA), en assignant une variable par registre (S_0) , une simple descente à partir de la solution S_0 (N1 simple) et une plus grande descente à partir de la solution S_0 (N1 deepest).

Le tableau 3.4 montre les pourcentages de gains en surface par rapport à la solution initiale, après l'application des algorithmes Left Edge (LEA), descente simple (N1 simple) et plus grande descente (N1 deepest).

En observant les gains obtenus par chaque algorithme, nous remarquons que dans 86% des cas, l'algorithme de plus grande descente avec le voisinage de la réassignation des registres permet d'atteindre les gains les plus importants. L'algorithme Left Edge réalise les meilleurs gains en surface pour seulement trois instances (3, 5 et 6). Dans ces cas, les algorithmes de descente simple et de plus grande descente ont été bloqués dans des minima locaux. Pour les instances 3, 6 et 29, les gains réalisés par l'algorithme de descente simple sont meilleurs que ceux trouvés par la plus grande descente. On observe ainsi que choisir toujours la meilleure solution voisine d'une solution donnée ne conduit pas forcément à la solution optimale.

L'amélioration des résultats passe par l'implémentation de méthodes plus avancées capables de combiner plusieurs voisinages. Nous proposons de comparer plusieurs méthaheuristiques à cet effet, dans le chapitre IV.

TABLE 3.3 – Résultats de l'algorithme de descente simple et de l'algorithme de plus grande descente pour le voisinage de réassignation des registres

	Instances	S_0	LEA	CPU (s)	N1 simple	CPU (s)	N1 deepest	CPU (s)
1	cordic_200_distributed	55944	36795	0,75	36252	10,64	25788	371,802
2	cordic_300_distributed	52274	26526	0,812	29613	8,234	18967	218,447
3	cordic_400_distributed	50975	13873	0,953	21562	19,717	31551	315,476
4	fft16_100_distributed	151566	130484	1,515	126800	69,949	117361	8970,113
5	fft16_150_distributed	135560	114176	1,64	110777	48,2	98614	7710,271
6	fft16_200_distributed	128901	107005	1,671	100266	113,073	90696	8031,919
7	prodmat2x2_30_distributed	19124	18041	0,062	17945	0,093	17945	0,109
8	prodmat2x2_60_distributed	10361	8597	0,062	8667	0,109	8379	0,109
9	prodmat2x2_100_distributed	8812	6219	0,078	6411	0,093	6110	0,156
10	prodmat4x4_280_distributed	56489	42351	0,609	41559	9,483	38251	290,992
11	prodmat4x4_360_distributed	48150	33114	0,64	32161	13,311	30966	281,445
12	prodmat4x4_100_distributed	64051	55466	0,39	52758	13,233	49563	241,728
13	volterra_80_distributed	38596	33259	0,218	32190	1,437	30628	8,077
14	volterra_120_distributed	29801	23355	0,171	23875	0,859	22426	6,093
15	volterra_160_distributed	22560	15741	0,156	16335	0,749	15528	4,124
16	cordic_200_global	56894	36992	0,656	38239	7,109	31836	427,879
17	cordic_300_global	53384	24727	0,734	28666	6,89	27799	294,461
18	cordic_400_global	51449	13537	0,859	17271	7,999	34350	361,834
19	fft16_100_global	141696	122255	1,296	121295	19,483	106352	7714,13
20	fft16_150_global	123603	99718	1,515	101020	25,201	85768	8038,187
21	fft16_200_global	130389	106919	1,656	104727	31,029	88762	11103,918
22	prodmat2x2_30_global	18708	17590	0,062	17014	0,093	17014	0,093
23	prodmat2x2_60_global	10463	8555	0,062	8590	0,093	8289	0,14
24	prodmat2x2_100_global	8812	6219	0,062	6411	0,093	6110	0,203
25	prodmat4x4_280_global	55805	41673	0,609	40890	9,608	36977	294,867
26	prodmat4x4_360_global	47970	33354	0,64	31934	10,343	30608	268,165
27	prodmat4x4_100_global	63877	54812	0,406	53951	9,671	48778	247,493
28	volterra_80_global	38692	33067	0,218	32116	1,421	30916	7,843
29	volterra_120_global	29903	23217	0,171	23117	0,874	23554	7,031
30	volterra_160_global	22380	15411	0,156	15468	0,703	15211	3,296

TABLE 3.4 – Gains en surface obtenues avec l'algorithme de descente simple et l'algorithme de plus grande descente pour le voisinage de réassignation des registres

	Instances	S_0	LEA	N1 simple	N1 deepest
1	cordic_200_distributed	55944	34%	35%	54%
2	cordic_300_distributed	52274	49%	43%	64%
3	cordic_400_distributed	50975	73%	58%	38%
4	cordic_200_global	56894	35%	33%	44%
5	cordic_300_global	53384	54%	46%	48%
6	cordic_400_global	51449	74%	66%	33%
7	fft16_100_distributed	151566	14%	16%	23%
8	fft16_150_distributed	135560	16%	18%	27%
9	fft16_200_distributed	128901	17%	22%	30%
10	fft16_100_global	141696	14%	14%	25%
11	fft16_150_global	123603	19%	18%	31%
12	fft16_200_global	130389	18%	20%	32%
13	prodmat2x2_30_distributed	19124	6%	6%	6%
14	prodmat2x2_60_distributed	10361	17%	16%	19%
15	prodmat2x2_100_distributed	8812	29%	27%	31%
16	prodmat2x2_30_global	18708	6%	9%	9%
17	prodmat2x2_60_global	10463	18%	18%	21%
18	prodmat2x2_100_global	8812	29%	27%	31%
19	prodmat4x4_280_distributed	56489	25%	26%	32%
20	prodmat4x4_360_distributed	48150	31%	33%	36%
21	prodmat4x4_100_distributed	64051	13%	18%	23%
22	prodmat4x4_280_global	55805	25%	27%	34%
23	prodmat4x4_360_global	47970	30%	33%	36%
24	prodmat4x4_100_global	63877	14%	16%	24%
25	volterra_80_distributed	38596	14%	17%	21%
26	volterra_120_distributed	29801	22%	20%	25%
27	volterra_160_distributed	22560	30%	28%	31%
28	volterra_80_global	38692	15%	17%	20%
29	volterra_120_global	29903	22%	23%	21%
30	volterra_160_global	22380	31%	31%	32%

3.4.3 Conclusion

Dans ce chapitre, nous avons défini un modèle d'estimation de la surface d'une solution donnée. Des tests numériques ont validé ce modèle. Dans une deuxième partie nous avons introduit deux algorithmes de recherche locale (algorithme de simple descente et algorithme de plus grande descente). Ces deux algorithmes ont été implémenté dans l'environnement de synthèse de haut niveau GAUT. Le voisinage exploré dans cette implémentation est celui de la réassignation des variables. Enfin une comparaison des surfaces en *gates* et des temps CPU résultants de l'application des approches proposées avec ceux des solutions initiales et des solutions qui s'appuient sur le LEA est faite. Ces comparaisons soulignent les avantages et les inconvénients des approches proposées.

Chapitre 4

Algorithmes avancés pour la synthèse de haut niveau

Sommaire

Dans ce chapitre, nous présentons les approches élaborées que nous proposons pour minimiser la surface des circuits intégrés. Ces approches appartiennent aussi à la famille des méthodes de recherche locale, sauf que celles-ci explorent un espace de solutions plus large que celui exploré par l'approche présentée dans le chapitre III .
Ces algorithmes utilisent aussi l'estimateur de surface présenté au chapitre III.

4.1 Introduction

Dans ce chapitre nous exposons trois métaheuristiques à bases de voisinages [Sorensen 2003], la descente *Multi-start*, la recherche à voisinage variable VNS et la recherche adaptative à base d'heuristique constructive randomisée GRASP. Pour chacune de ces trois méthodes :

- nous définissons d'abord le principe général de la méthode,
- nous expliquons la version adaptée et le paramétrage pour la problématique de la synthèse de haut niveau,
- et nous présentons les résultats (surfaces en *gates* et temps CPU) obtenus par l'approche.

Le jeu d'instances que nous avons utilisé a été introduit dans la section 3.4.2.

Enfin nous concluons le chapitre par une analyse de toutes les approches. Cette analyse met en avant les avantages et les limites de toutes les approches de résolution proposées dans ce manuscrit.

4.2 La descente *Multi-start*

Les méthodes de descente simple et celle de plus grande descente décrites dans le chapitre III avaient fait l'objet de plusieurs critiques. Toutes deux se basaient sur une amélioration progressive de la solution et donc resteraient bloquées dans un minimum local dès qu'elles en rencontreraient un. L'absence de technique de diversification est l'origine de ce comportement. L'équilibre nécessaire entre intensification et diversification n'existait pas. Un moyen simple de diversifier la recherche peut consister à re-exécuter un des algorithmes de descente en prenant un autre point de départ. Comme l'exécution de ces méthodes est souvent très rapide ; on peut inclure cette répétition au sein d'une boucle générale. On obtient alors un algorithme de type descente *Multi-start*, décrit par l'algorithme 5 [Martí 2003].

4.2.1 Description de la méthode

Dans l'algorithme 5, B représente la meilleure solution, en terme de coût, rencontrée jusqu'à la date considérée et $f(B)$ est son coût. Il s'agit donc de générer différents points de départ, comme le montre la figure 4.1 et de lancer une méthode de descente à partir de ces solutions initiales différentes pour obtenir différentes solutions d'arrivée. La dernière étape consiste à retenir la meilleure solution en terme de coût.

4.2.2 Implémentation

Pour trouver plusieurs solutions de départ, il suffit de jouer sur le résultat d'une des sous étapes du flot de synthèse qui génère la solution initiale, détaillé dans le chapitre III. Une des

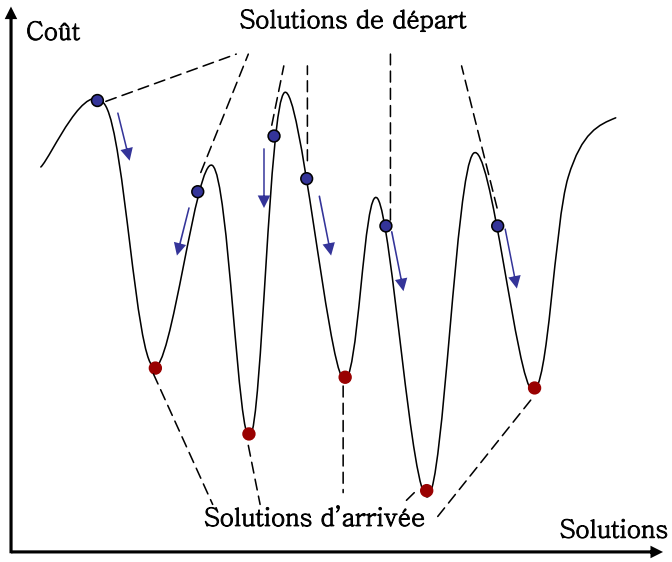


FIGURE 4.1 – Exploration de l'espace des solutions en utilisant la descente *Multi-start*

Algorithme 5 : Algorithme de descente *Multi-start*

Initialisation : Trouver une solution initiale x, $k \leftarrow 1$, $f(B) \leftarrow +\infty$;
Répéter;
    solution de départ : choisir aléatoirement une solution de départ x_0 ;
    $x \leftarrow$ résultat d'une simple descente ou d'une plus grande descente ;
    Si $f(x) < f(B)$ **alors**;
        $B \leftarrow x$;
    Fin si;
    $k \leftarrow k + 1$;
Jusqu'à Critère d'arrêt satisfait;

possibilités consiste à partir de différents résultats de l'étape de sélection. Pour cela, on peut par exemple changer le résultat de la sélection. Une autre alternative consiste à faire varier les méthodes d'allocation des opérateurs (allocation globale/ allocation distribuée), ou encore changer l'algorithme d'ordonnancement ou bien les techniques d'assignation des opérations ou des variables.

4.2.3 Résultats

Les quinze premières lignes du tableau 4.1 donnent la surface en *gates* des quatre solutions de départs obtenus en jouant sur les algorithmes d'allocation des opérateurs et d'assignation des variables. Nous avons utilisé deux techniques d'allocations des opérateurs (allocation globale/

allocation distribuée) et deux techniques d'assignation des variables (un registre pour chaque variable/ LEA).
Ces quatre combinaisons produisent quatre solutions initiales comme suit :

- l'utilisation de l'allocation distribuée pour les opérateurs suivie de l'assignation d'une variable par registre conduit à la solution initiale $S0_1$,
- l'utilisation de l'allocation globale pour les opérateurs suivie de l'assignation d'une variable par registre conduit à la solution initiale $S0_2$,
- l'utilisation de l'allocation distribuée pour les opérateurs suivie de l'assignation selon l'algorithme LEA des variables conduit à la solution initiale $S0_3$,
- l'utilisation de l'allocation globale pour les opérateurs suivie de l'assignation selon l'algorithme LEA des variables conduit à la solution initiale $S0_4$.

Les quinze dernières lignes du tableau 4.1 donnent la surface des quatre solutions ($S0_1+N1$, $S0_2+N1$, $S0_3+N1$ et $S0_4+N1$) obtenues suite à l'application d'une simple descente à partir des quatre solutions initiales définies précédemment.

Analyse des résultats La première partie du tableau 4.1 montre qu'en variant les techniques d'allocation des opérateurs et celles de l'assignation des variables nous trouvons bel et bien quatre solutions de départ. Ces solutions de départ ont des surfaces différentes.

En observant la deuxième partie du même tableau, nous notons qu'après l'application d'une descente simple à partir des différentes solutions de départ, nous obtenons quatre solutions différentes. Celles-ci ont des surfaces différentes. La dernière colonne représente pour chaque instance, la meilleure surface obtenue suite à l'application de la descente *Multi-start*. Dans l'échantillon que nous avons considéré, $S0_4+N1$ aboutit dans 80% des cas à la solution qui a la plus petite surface.

Le tableau 4.2 illustre les gains réalisés par la descente *Multi-start* ($BestSol_1$) par rapport à la descente simple (N1 simple) expliquée dans le chapitre III. Dans ce tableau la descente simple part d'une solution initiale obtenue en utilisant une allocation distribuée des opérateurs et une assignation d'une variable par registre ($S0_1$).

L'approche de la descente *Multi-start* explore un espace de solution plus large que celui exploré par la simple descente et trouve ainsi des solutions de meilleur qualités que cette dernière. Cependant, l'espace à explorer n'a pas atteint ces limites. Les approches qui serons présentées dans les sections suivantes explorent des espaces de plus en plus large.

TABLE 4.1 – Application de la descente *Multi-start* à la synthèse de haut niveau

	instances	$S0_1$	$S0_2$	$S0_3$	$S0_4$
1	cordic+200	55944	56894	36795	36992
2	cordic+300	52274	53384	26526	24727
3	cordic+400	50975	51449	13873	13537
4	fft16+100	151566	141696	130484	122255
5	fft16+150	135560	123603	114176	99718
6	fft16+200	128901	130389	107005	106919
7	prodmat2x2+30	19124	18708	18041	17590
8	prodmat2x2+60	10361	10463	8597	8555
9	prodmat2x2+100	8812	8812	6219	6219
10	prodmat4x4+280	56489	55805	42351	41673
11	prodmat4x4+360	48150	47970	33114	33354
12	prodmat4x4+100	64051	63877	55466	54812
13	volterra+80	38596	38692	33259	33067
14	volterra+120	29801	29903	23355	23217
15	volterra+160	22560	22380	15741	15411

	instances	$S0_1 + N1$	$S0_2 + N1$	$S0_3 + N1$	$S0_4 + N1$	$BestSol_1$
1	cordic+200	36252	38239	**29505**	29972	29505
2	cordic+300	29613	28666	20436	**18013**	18013
3	cordic+400	21562	17271	9937	**9265**	9265
4	fft16+100	126800	121295	123284	**115343**	115343
5	fft16+150	110777	101020	106844	**93256**	93256
6	fft16+200	100266	104727	99337	**98957**	98957
7	prodmat2x2+30	17945	**17014**	18041	17494	17014
8	prodmat2x2+60	8667	8590	8549	**8507**	8507
9	prodmat2x2+100	6411	6411	**6123**	**6123**	6123
10	prodmat4x4+280	41559	40890	40191	**38991**	38991
11	prodmat4x4+360	32161	31934	31488	**31194**	31194
12	prodmat4x4+100	52758	53951	53258	**52316**	52316
13	volterra+80	32190	32116	**31627**	**31627**	31627
14	volterra+120	23875	23117	22491	**22113**	22113
15	volterra+160	16335	15468	14445	**14019**	14019

TABLE 4.2 – Gains réalisés par la descente *Multi-start* par rapport à la simple descente

	instances	N1 simple	$BestSol_1$	gains
1	cordic+200	36252	29505	19%
2	cordic+300	29613	18013	39%
3	cordic+400	21562	9265	57%
4	fft16+100	126800	115343	9%
5	fft16+150	110777	93256	16%
6	fft16+200	100266	98957	1%
7	prodmat2x2+30	17945	17014	5%
8	prodmat2x2+60	8667	8507	2%
9	prodmat2x2+100	6411	6123	4%
10	prodmat4x4+280	41559	38991	6%
11	prodmat4x4+360	32161	31194	3%
12	prodmat4x4+100	52758	52316	1%
13	volterra+80	32190	31627	2%
14	volterra+120	23875	22113	7%
15	volterra+160	16335	14019	14%

4.3 La recherche à voisinages variables

La recherche à voisinages variables est une méthode relativement récente, basée sur la performance des méthodes de descente [Mladenović 1997]. Cette méthode suggère simplement d'utiliser plusieurs voisinages successifs dès qu'on se retrouve bloqué dans un minimum local.

4.3.1 Description de la méthode

Le processus de la recherche à voisinages variables est décrit par l'algorithme 6. La première étape consiste à définir un ensemble de n voisinages $N_{i=1...n}$ (de préférence tels que $N_i \subset N_{i+1}$). Dans l'étape qui suit, on choisit une solution de départ x à l'aide d'une heuristique. Ensuite, en partant d'une solution initiale x' choisie dans le premier voisinage de $N(x)$ de x, on applique une méthode de recherche locale (une méthode de descente ou une recherche tabou [Glover 1986, Sevaux 2007] par exemple) jusqu'à arriver dans un minimum local (ou l'arrêt de la recherche locale). Si la solution trouvée x'' est meilleure que x, on recentre la recherche en repartant du premier voisinage, sinon on passe au voisinage suivant (voir figure 4.2). La recherche s'arrête quand aucun des voisinages n'est capable d'améliorer la solution.

La figure 4.2 montre un espace de solution à deux dimensions ($N1$ et $N2$) qui représentent les deux voisinages. L'ensemble des points des courbes tracées dans cet espace constitue les solutions réalisables. L'application d'une méthode de descente qui explore le voisinage $N1$ à partir de la solution de départ $S0$ aboutit à une solution $S1$. Ensuite, l'application d'une méthode de descente qui explore le voisinage $N2$ à partir de la solution $S1$ aboutit à une solution $S2$. Ainsi, la solution d'arrivée de la première descente est le point de départ de la deuxième descente.

Un des points cruciaux dans un VNS, est le choix de l'ordre d'exploration des voisinages. Une bonne structure de voisinage conduit généralement à de bons résultats ou au moins à une recherche efficace [Sevaux 2004]. Dans cette méthode, la diversification est gérée par deux mécanismes. Le premier consiste à choisir dans le voisinage courant une solution aléatoirement (*Randomiser*) et le deuxième est le changement de voisinage lui même qui élargit l'espace exploré. L'intensification de la recherche est assurée par l'appel à une procédure de recherche locale.

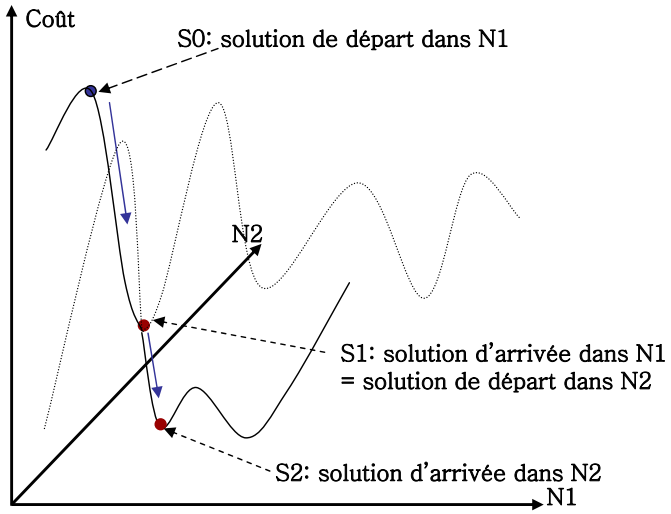


FIGURE 4.2 – Exploration de l'espace de solution en utilisant le VNS

4.3.2 Implémentation

La première étape consiste à choisir plusieurs voisinages. Nous avons choisi trois voisinages parmi ceux définis dans le chapitre III :

- (1) la réassignation des registres,
- (2) la permutation de l'assignation de deux opérations,
- (3) la réassignation des opérations.

Algorithme 6 : Algorithme de recherche à voisinages variables

Initialisation : Trouver une solution initiale x, $k \leftarrow 1$;

Répéter;

Randomiser : générer une solution x' aléatoirement dans le voisinage $N_k(x)$;

Recherche locale : appliquer la procédure de recherche locale en partant de la solution x' pour trouver une solution x'' ;

Si $f(x'') < f(x)$ **alors**;

$x \leftarrow x''$ et $k \leftarrow 1$ (centrer la recherche autour x'' et explorer le premier voisinage);

Sinon;

$k \leftarrow k + 1$ (élargir le voisinage);

Fin si;

Jusqu'à $i = n$;

L'omission des voisinages de la réallocation des opérateurs et de celui du réordonnancement est dû respectivement, à la stérilité du premier s'il n'est pas couplé avec le réordonnancement et la complexité de la mise en oeuvre du deuxième.

4.3.2.1 La solution initiale

Nous avons appliqué le principe de la descente *Multi-start*. Cet algorithme part des quatre solutions initiales ($S0_1$, $S0_2$, $S0_3$ et $S0_4$) définies dans le section précédente. Il utilise la recherche à voisinages variables comme méthode de recherche locale à la place de la descente simple appliquée dans la section 4.2.

4.3.2.2 La Randomisation

L'étape *Randomiser* que nous avons choisi d'implémenter consiste à prendre un voisin de la solution initiale sans se soucier si celui-ci a un coût inférieur à la solution de départ, et de répéter ce processus trois fois.
Ainsi pour le voisinage de la ré-assignation des registres, il s'agit de déplacer trois variables et d'accepter la solution obtenue quel que soit l'effet de ces déplacements sur la surface totale. De même pour le voisinage de la permutation de l'assignation des opérations, la randomisation consiste à choisir trois couples d'opérations compatibles deux à deux et à permuter leurs assignations. Enfin, pour le voisinage de la ré-assignation des opérations, la randomisation consiste à choisir trois opérations quelconques et à changer leurs assignations, tout en respectant les règles de compatibilités entre opérations.

4.3.2.3 L'ordre de l'exploration des voisinages

Dans notre application du VNS, les voisinages ne sont pas inclus les uns dans les autres, en revanche le choix de l'ordre de l'exploration des voisinages peut influer sur les résultats (voir tableau 4.4). Ceci peut s'expliquer par le fait que le voisinage de la réassignation des registres est le plus large car dans la solution de départ aucune optimisation n'est faite au niveau du nombre de registres.
Le choix de l'ordre de l'exploration des voisinages a était validé par des tests numériques. En effet, nous avons testé les différentes combinaisons d'ordres d'exploration des voisinages en passant ou pas par l'étape *Randomiser*.

Dans le tableau 4.4 les numéros de la première colonne correspondent aux numéros des instances comme définies dans le tableau 3.3 du chapitre III. La première ligne correspond aux différentes possibilités d'ordres d'exploration des trois voisinages (voir tableau 4.3).

TABLE 4.3 – Différentes combinaisons d'ordre d'exploration des trois voisinages

Appellation	Ordre d'exploration des voisinages
M1 :	l'ordre (1-2-3) sans la phase de randomisation
M1' :	l'ordre (1-2-3) avec une phase de randomisation qui précède chacun des voisinages
M2 :	l'ordre (2-3-1) sans la phase de randomisation
M2' :	l'ordre (2-3-1) avec une phase de randomisation qui précède chacun des voisinages
M3 :	l'ordre (3-1-2) sans la phase de randomisation
M3' :	l'ordre (3-1-2) avec une phase de randomisation qui précède chacun des voisinages
M4 :	l'ordre (1-3-2) sans la phase de randomisation
M4' :	l'ordre (1-3-2) avec une phase de randomisation qui précède chacun des voisinages
M5 :	l'ordre (2-1-3) sans la phase de randomisation
M5' :	l'ordre (2-1-3) avec une phase de randomisation qui précède chacun des voisinages
M6 :	l'ordre (3-2-1) sans la phase de randomisation
M6' :	l'ordre (3-2-1) avec une phase de randomisation qui précède chacun des voisinages

Ainsi, les valeurs du tableau 4.4 correspondent aux surfaces estimées en *gates* des architectures obtenues après l'application du VNS en explorant les trois voisinages selon les ordres prédéfinis dans le tableau 4.3. Enfin le tableau **??** présente les surfaces maximum et minimum trouvées avec les différents ordres d'exploration des trois voisinages, ainsi que la combinaison permettant d'aboutir à ces résultats.

En observant ce tableau, nous constatons que la combinaison M1 trouve la surface minimum dans 50% des cas, M4 dans 20% des cas et M3 dans 10% des cas. Par conséquent, notre choix se porte sur l'ordre d'exploration des voisinages (1-2-3) en ignorant la phase de Randomisation, comme celle-ci ne mène que dans 7% des cas aux plus petites surafces.

4.3.2.4 La recherche locale

En ce qui concerne la procédure de descente appliquée, nous avons opté pour la simple descente présentée dans le chapitre III. Nous avons choisi la descente simple plutôt que la plus grande descente, car le temps d'exécution de la deuxième peut être très long (un peu plus de 3 heures pour l'instance fft16_200_global par exemple). Ce temps important est dû principalement à la taille du voisinage qui est importante.

TABLE 4.4 – Influence de l'ordre de l'exploration des voisinages sur la surface

	M1	M1'	M2	M2'	M3	M3'	M4	M4'	M5	M5'	M6	M6'	Max	Min
1	34470	34614	38363	38036	37931	38267	34296	34392	34434	39906	38363	38036	M5'	M4
2	28707	28803	29397	29794	28785	29073	28515	28653	28557	29265	29397	29794	M2'	M4
3	21562	21562	24856	25258	24856	25426	21562	22138	21562	21857	24856	25258	M3'	M1
4	121280	121319	126951	127575	121568	120787	120272	120647	125578	126584	126868	127531	M2'	M4
5	106691	102746	114129	116355	110931	110508	106889	102794	110662	114309	114418	116091	M2'	M1'
6	96906	97002	105526	105574	95788	95980	97386	97434	105334	105395	102398	104746	M2'	M3
7	17945	18041	17945	18521	17945	18041	17945	18041	17945	18521	17945	18521	M2'	M1
8	8667	8872	8536	8405	8453	8885	8667	9016	8213	8213	8453	8693	M4'	M5
9	6075	6219	6363	6363	6267	6459	6123	6219	6315	6315	6363	6363	M3'	M1
10	38631	38758	42126	42326	41098	40268	38823	38950	41480	40293	40650	42032	M2'	M1
11	29707	30843	32731	32673	30786	30520	30139	31521	29983	31060	31524	33623	M6'	M1
12	50694	50506	51619	51619	51414	50746	50118	50554	50659	51619	51619	51619	M2	M4
13	30126	30414	30964	31514	30846	30942	30606	30702	30955	30964	31230	31422	M2'	M1
14	22291	22291	23648	24159	22771	22771	22387	22579	22880	23151	23404	23823	M2'	M1
15	15279	15183	16030	16497	15692	15740	14895	14991	15567	16462	15336	15401	M2'	M4
16	36001	36301	40453	37959	35592	35688	36013	36109	37835	38144	37462	38743	M2	M3
17	26320	26320	26853	26507	27574	27574	26608	26608	29143	28977	26853	26507	M5	M1
18	17271	17373	29058	18209	21684	22206	17271	17373	19382	21736	29058	18209	M2	M1
19	116015	118238	125039	125437	116565	119041	116159	117806	124489	124887	124694	125092	M2'	M1
20	97996	96854	103069	103271	98162	96566	97996	96560	102311	101644	103165	103367	M6'	M4'
21	100461	100773	110596	110027	102917	101885	100701	101205	108581	108851	110418	109456	M2	M1
22	17014	17014	17014	17014	17014	17014	17014	17014	17014	17014	17014	17014	M1	M1
23	8446	8782	8219	8219	8315	8686	8446	8926	8267	8267	8219	8219	M4'	M2
24	6075	6315	6315	6315	6075	6315	6075	6315	6315	6315	6315	6315	M2'	M1
25	39360	38395	41702	41218	39033	40319	39456	38251	41667	40711	40873	41100	M2	M4'
26	30968	32499	33710	31694	30933	32473	30968	32355	33029	31856	33710	31694	M2	M3
27	52031	50542	53270	53532	51904	52179	52175	50734	53222	53532	52790	53619	M6'	M1'
28	31204	31300	31335	31440	31230	31374	31060	31300	31523	31593	31335	31440	M5'	M4
29	22301	22493	22829	23331	22541	22637	22397	22493	22829	23187	22829	23331	M2'	M1
30	14748	14940	16215	16180	14748	14940	14748	14940	16215	16180	16215	16180	M2	M1

4.3.3 Résultats

Le tableau 4.5 montre pour chacune des instances les surfaces en *gates* des quatre solutions ($S0_1 + VNS$, $S0_2 + VNS$, $S0_3 + VNS$ et $S0_4 + VNS$). Nous avons obtenu ces solutions suite à l'application de la recherche à voisinage variable à partir des solutions initiales ($S0_1$, $S0_2$, $S0_3$ et $S0_4$) définies dans la section précédente.

Dans la colonne $BestSol_2$, nous trouvons les plus petites surfaces obtenues, trouvées dans 66% des cas par l'application du VNS à la solution $S0_4$, contre 33% par l'application du VNS à la solution $S0_3$.

Le tableau 4.6 compare les résultats de la simple descente et de la recherche à voisinage variable. Il affiche les meilleures solutions $BestSol_1$ trouvées en appliquant une simple descente partant des solutions initiales ($S0_1$, $S0_2$, $S0_3$ et $S0_4$), et les meilleures solutions trouvées en appliquant une recherche à voisinage variable VNS, en partant des mêmes solutions initiales.

La combinaison d'une descente *multi-start* et d'une recherche à voisinages variables semble rentable. Dans la section suivante nous allons implémenter une approche qui génère automatiquement plusieurs solutions de départ et applique à chacune de ces solutions une recherche à voisinages variables.

TABLE 4.5 – Application du VNS à la synthèse de haut niveau

	instances	$S0_1 + VNS$	$S0_2 + VNS$	$S0_3 + VNS$	$S0_4 + VNS$	$BestSol_2$
1	cordic+200	34470	36001	28491	**27812**	27812
2	cordic+300	28707	26320	19332	**17533**	17533
3	cordic+400	21562	17271	9361	**9265**	9265
4	fft16+100	121280	116015	117092	**111167**	111167
5	fft16+150	106691	97996	102428	**89050**	89050
6	fft16+200	96906	100461	94537	**93203**	93203
7	prodmat2x2+30	17945	**17014**	18041	17398	17014
8	prodmat2x2+60	8667	8446	8021	**7931**	7931
9	prodmat2x2+100	6075	6075	**5883**	**5883**	5883
10	prodmat4x4+280	38631	39360	**35055**	35967	35055
11	prodmat4x4+360	29707	30968	**28272**	29040	28272
12	prodmat4x4+100	50694	52031	**49610**	48668	49610
13	volterra+80	30126	31204	**30523**	30763	30523
14	volterra+120	22291	22301	21531	**21297**	21297
15	volterra+160	15279	14748	13341	**13155**	13155

TABLE 4.6 – Comparaison entre les résultats de la simple descente et de la recherche à voisinage variable

		$BestSol_1$	$BestSol_2$
1	cordic+200	29505	27812
2	cordic+300	18013	17533
3	cordic+400	9265	9265
4	fft16+100	115343	111167
5	fft16+150	93256	89050
6	fft16+200	98957	93203
7	prodmat2x2+30	17014	17014
8	prodmat2x2+60	8507	7931
9	prodmat2x2+100	6123	5883
10	prodmat4x4+280	38991	35055
11	prodmat4x4+360	31194	28272
12	prodmat4x4+100	53258	49610
13	volterra+80	31627	30523
14	volterra+120	22113	21297
15	volterra+160	14019	13155

4.4 Le GRASP

La méthode du Greedy Randomized Adaptative Search Procedure GRASP a été introduite en 1989 par Feo et Resende [Feo 1989] puis présentée dans sa forme définitive en 1995 [**?**]. Un des avantages de cette méthode est la simplicité avec laquelle le processus d'optimisation peut être compris.
Le document de Festa [Festa 2003] présente des implémentations réussies et un tutoriel de la méthode.

4.4.1 Description de la méthode

Cette méthode d'exploration combine une heuristique gloutonne et une recherche locale.
A chaque itération, on construit une solution en utilisant une heuristique gloutonne dont une partie est modifié aléatoirement. Ensuite, cette solution est raffinée par l'intermédiaire d'une méthode de descente.

Un algorithme glouton est un algorithme qui suit le principe de faire, étape par étape, un choix optimum local, dans l'espoir d'obtenir un résultat optimum global. Par exemple, dans le problème de l'assignation des variables, l'algorithme consistant à répéter le choix de l'assignation grande valeur qui ne dépasse pas la somme restante est un algorithme glouton. Dans les cas où l'algorithme ne fournit pas systématiquement la solution optimale, il est appelé une heuristique gloutonne.

4.4.2 Implémentation

Pour l'implémentation du GRASP nous avons choisi l'algorithme glouton d'assignation des variables : le LEA. Ainsi, la solution complète est obtenue en randomisant les étapes de l'algorithme LEA. Une étape du LEA consiste à choisir une variable non assignée dans une liste ordonnée et l'assigner dans le registre le plus à gauche. Cette assignation n'est valide que si les variables déjà assignées à ce registre sont toutes compatibles avec la variable à assigner.

La randomisation de ce processus consiste à tirer au sort un entier dans [0..100], si l'entier est inférieur à un certain *alpha*, l'assignation se fait comme le veut le LEA. Sinon, un nouveau registre doit être créé pour sauvegarder la variable.

Nous avons lancé deux GRASP (GRASP100 et GRASP10).
- Le GRASP100 est un GRASP à cent itérations. Ainsi, le critère d'arrêt de la boucle du GRASP est le nombre d'itérations. Chaque itération part d'une assignation différente des variables. Ces différentes assignations sont obtenues comme expliquée précédemment par

Algorithme 7 : GRASP

Choisir un algorithme gloutonne et la découper en étapes;
(nb_étapes est le nombre d'étapes de l'algorithme);
Fixer une valeur *alpha* dans [0..100];
étape = 0;
Répéter;
    Répéter;
        tirer au sort un entier *rand* dans [0..100];
        Si (*rand* < *alpha*);
            appliquer la logique de l'algorithme glouton pour étape;
        Sinon;
            appliquer une logique différente de celle de l'algorithme glouton pour étape;
        Fin si;
    Jusqu'à obtenir une solution x complète;
    Recherche locale ; trouver un optimum local x' *en partant de* x **Jusqu'à** critère d'arrêt satisfait;

la randomisation de l'algorithme LEA. Dans cette version du GRASP nous avons fait varier la valeur de alpha de zéro à cent. Le cas ou alpha est égal à zéro correspond à l'assignation normale trouvée par le LEA. Le cas ou alpha est égal à cent correspond à l'assignation d'une seule variable par registre.

- Le GRASP10 est un GRASP à dix itérations. Chaque itération part d'une assignation différente des variables. Ces différentes assignations sont obtenues par la randomisation de l'algorithme LEA. Dans cette version du GRASP, alpha a été fixé à quatre-vingt. Ce nombre est la moyenne des alphas qui conduisent aux meilleures solutions en terme de surface dans la version GRASP100.

4.4.3 Résultats

En effet, le GRASP100 nous a aidé a déterminer les valeurs des paramètres génériques tel que alpha et le nombre d'itérations de notre méthode finale : le GRASP10. Nous avons défini le alpha qui a aboutit au meilleur résultat dans le GRASP100 pour chacune des trente instances du tableau 4.7. Ensuite, nous avons fait la moyenne des alphas obtenues pour trouver quatre-vingt. Ainsi, pour GRASP10 le alpha a été fixé à quatre-vingt et le nombre d'itérations à dix pour avoir un temps de calcul raisonnable. Le tableau 4.7 compare les temps CPU et les surfaces en *gates* obtenues avec les algorithmes suivants :

- le Left-Edge (LEA),
- le GRASP100.

TABLE 4.7 – Résultats du GRASP100

Instances	LEA	CPU(s)	GRASP100	CPU (s)	alpha
1	36795	0.75	24318	205.46	81
2	26526	0.81	17652	174.13	93
3	13873	0.95	9361	187.28	100
4	130484	1.51	115854	2569.61	75
5	114176	1.64	100695	2077.92	21
6	107005	1.67	92945	2196.22	49
7	18041	0.06	17945	1.28	0
8	8597	0.06	8021	0.73	70
9	6219	0.07	5643	0.57	89
10	42351	0.60	35055	165.47	100
11	33114	0.64	27587	149.00	93
12	55466	0.39	48554	196.58	96
13	33259	0.21	29515	22.10	94
14	23355	0.17	21038	13.49	98
15	15741	0.15	12896	8.20	84
16	36992	0.65	26818	186.24	78
17	24727	0.73	17293	148.56	98
18	13537	0.85	9265	186.77	100
19	122255	1.29	109151	1787.25	89
20	99718	1.51	88407	1581.37	93
21	106919	1.65	92153	1878.83	94
22	17590	0.06	17014	0.98	0
23	8555	0.06	7835	0.70	63
24	6219	0.06	5643	0.54	89
25	41673	0.60	35001	163.88	89
26	33354	0.64	28032	143.51	88
27	54812	0.40	48668	169.61	99
28	33067	0.21	29947	21.21	94
29	23217	0.17	21105	12.49	89
30	15411	0.15	13107	7.32	87
				moyenne	79,76666667

TABLE 4.8 – Résultats du GRASP10

Instances	LEA	CPU(s)	GRASP10	CPU (s)
1	36795	0.75	25223	8.04
2	26526	0.81	18737	8.40
3	13873	0.95	12063	9.21
4	130484	1.51	116695	80.15
5	114176	1.64	101165	68.98
6	107005	1.67	94254	54.18
7	18041	0.06	18041	0.15
8	8597	0.06	8021	0.10
9	6219	0.07	5787	0.12
10	42351	0.60	35570	5.32
11	33114	0.64	28170	6.39
12	55466	0.39	48026	6.60
13	33259	0.21	30235	1.37
14	23355	0.17	21457	0.78
15	15741	0.15	13219	0.49
16	36992	0.65	27475	7.35
17	24727	0.73	17745	6.67
18	13537	0.85	12043	9.27
19	122255	1.29	108955	66.18
20	99718	1.51	87650	51.71
21	106919	1.65	92942	52.48
22	17590	0.06	17302	0.12
23	8555	0.06	7835	0.09
24	6219	0.06	5787	0.09
25	41673	0.60	35330	5.35
26	33354	0.64	28307	6.20
27	54812	0.40	49100	6.21
28	33067	0.21	30187	1.34
29	23217	0.17	21485	0.74
30	15411	0.15	13155	0.46

Analyse des résultats Les résultats du tableau 4.8 montrent que plus on élargit l'espace des solutions explorées, meilleures sont les solutions trouvées en terme de surface. La version

GRASP10, présente un compromis entre la qualité de la solution et le temps d'exécution de l'algorithme qui reste raisonnable même pour les grandes instances.

4.5 Bilan

Le tableau 4.9 compare les temps CPU et les surfaces en *gates* obtenues avec les algorithmes suivants :

- la descente simple (N1 simple),
- la plus grande descente (N1 deepest),
- le Left-Edge (LEA),
- la recherche à voisinage variable qui part de la solution initiale dans laquelle on affecte une variable par registre (SI+VNS),
- la recherche à voisinage variable qui part de la solution initiale retourné par le LEA (LEA+VNS),
- le GRASP10.

Les approches proposées dans ce manuscrit sont toutes des métaheuristiques adaptées à la problématique de la minimisation de la surface d'un circuit intégré. Ces métaheuristiques ont été présentées dans un ordre de façon à explorer un espace de solution de plus en plus large. La descente simple explore ainsi le plus petit espace de solution alors que le GRASP10 explore un espace beaucoup plus grand.

TABLE 4.9 – Comparaison des différentes approches développées pour la HLS

	SI	N1 simple	CPU (s)	N1 deepest	CPU (s)	LEA	CPU (s)	SI+VNS	LEA+VNS	CPU (s)	GRASP10	CPU (s)	gain
1	55944	36252	10,64	25788	371,8	36795	0,75	34470	28491	0,83	25223	8,05	31%
2	52274	29613	8,23	18967	218,45	26526	0,81	28707	19332	0,84	18737	8,41	29%
3	50975	21562	19,72	31551	315,48	13873	0,95	21562	9361	0,95	12063	9,22	13%
4	151566	126800	69,95	117361	8970,11	130484	1,52	121280	117092	7,5	116695	80,15	11%
5	135560	110777	48,2	98614	7710,27	114176	1,64	106691	102428	6,58	101165	68,98	11%
6	128901	100266	113,07	90696	8031,92	107005	1,67	96906	94537	5,3	94254	54,19	12%
7	19124	17945	0,09	17945	0,11	18041	0,06	17945	18041	0,08	18041	0,16	0%
8	10361	8667	0,11	8379	0,11	8597	0,06	8667	8021	0,06	8021	0,11	7%
9	8812	6411	0,09	6110	0,16	6219	0,08	6075	5883	0,09	5787	0,12	7%
10	56489	41559	9,48	38251	290,99	42351	0,61	38631	35055	0,89	35570	5,33	16%
11	48150	32161	13,31	30966	281,45	33114	0,64	29707	28272	1,03	28170	6,39	15%
12	64051	52758	13,23	49563	241,73	55466	0,39	50694	49610	0,89	48026	6,61	13%
13	38596	32190	1,44	30628	8,08	33259	0,22	30126	30523	0,3	30235	1,37	9%
14	29801	23875	0,86	22426	6,09	23355	0,17	22291	21531	0,2	21457	0,78	8%
15	22560	16335	0,75	15528	4,12	15741	0,16	15279	13341	0,16	13219	0,5	16%
16	56894	38239	7,11	31836	427,88	36992	0,66	36001	27812	0,81	27475	7,36	26%
17	53384	28666	6,89	27799	294,46	24727	0,73	26320	17533	0,8	17745	6,67	28%
18	51449	17271	7,999	34350	361,83	13537	0,86	17271	9265	0,94	12043	9,28	11%
19	141696	121295	19,48	106352	7714,13	122255	1,3	116015	111167	6,17	108955	66,19	11%
20	123603	101020	25,2	85768	8038,19	99718	1,52	97996	89050	3,47	87650	51,72	12%
21	130389	104727	31,03	88762	11103,92	106919	1,66	100461	93203	4,62	92942	52,48	13%
22	18708	17014	0,09	17014	0,09	17590	0,06	17014	17398	0,06	17302	0,12	2%
23	10463	8590	0,09	8289	0,14	8555	0,06	8446	7931	0,08	7835	0,09	8%
24	8812	6411	0,09	6110	0,2	6219	0,06	6075	5883	0,06	5787	0,09	7%
25	55805	40890	9,61	36977	294,87	41673	0,61	39360	35967	0,91	35330	5,36	15%
26	47970	31934	10,34	30608	268,17	33354	0,64	30968	29040	1,03	28307	6,2	15%
27	63877	53951	9,67	48778	247,49	54812	0,41	52031	48668	0,87	49100	6,22	10%
28	38692	32116	1,42	30916	7,84	33067	0,22	31204	30763	0,3	30187	1,34	9%
29	29903	23117	0,87	23554	7,03	23217	0,17	22301	21297	0,2	21485	0,75	7%
30	22380	15468	0,7	15211	3,3	15411	0,16	14748	13155	0,17	13155	0,47	15%

Chapitre 5

Conclusions et Perspectives

Nous avons proposé dans ce mémoire des métaheuristiques pour la minimisation de la surface des circuits électroniques qui implémentent des applications du traitement du signal et de l'image. Cette optimisation s'inscrit dans le cadre de la miniaturisation des composants intégrés aux systèmes nomades qui envahissent notre vie quotidienne et l'automatisation du flot de conception de ces composants.

Les approches proposées sont appliquées à très haut niveau, lors du passage de la représentation algorithmique à la représentation RTL, appelé synthèse de haut niveau. Les entrées de ce processus consistent en une description algorithmique et les contraintes de cadence. Il passe par les étapes de sélection, allocation, ordonnancement et l'assignation.

Un outil de synthèse de haut niveau, GAUT, à été développée au LESTER puis au Lab-STICC. Cet outil est déjà capable de trouver plus d'une solution (= architecture) pour une application donnée. Ces différentes solutions sont obtenues en variant les techniques d'allocation et d'assignation. Le module d'optimisation, que nous avons intégré à GAUT, part des solutions proposées par ce dernier et cherche à explorer leurs voisinages en suivant différentes stratégies.

La première stratégie consiste à appliquer des méthodes de descente (descente simple ou plus grande descente) pour minimiser la surface d'une architecture de départ. La principale limite de cette approche est son blocage dans le premier minimum local rencontré. Par ailleurs, la méthode de plus grande descente est coûteuse en temps de calcul, puisqu'elle teste toutes les possibilités avant de faire une modification de l'architecture.

La deuxième stratégie à été proposée pour pallier aux limites de la première. Elle consiste à partir de plusieurs solutions initiales différentes, et à leur appliquer une simple descente pour choisir enfin le meilleur résultat. Le temps de calcul de cette approche dépend du nombre de

solutions initiales mais reste généralement raisonnable. Cette méthode est appelée la descente à départs multiples ou *Multi-start descent*.

La troisième approche, appelée recherche à voisinages variables, est une approche plus élaborée. Elle suggère de définir plusieurs voisinages, partir d'une solution de départ, de lancer une simple descente qui explore le premier voisinage et de changer de voisinage dès qu'on se trouve bloqué dans un minimum local. Cette approche a été combinée avec celle de la descente à départs multiples. Pour cela, dans l'algorithme de la descente à départs multiples décrit précédemment et la simple descente à été remplacé par la recherche à voisinages variables.

Enfin, la dernière approche proposée dans ce manuscrit, le GRASP pour *Greedy Randomized Adaptative Search Procedure*, consiste à randomiser une heuristique gloutonne de construction de la solution initiale. Cette randomisation nous permet d'obtenir plusieurs solutions de départ. Les solutions obtenues sont celles auxquelles sont appliquées des recherches à voisinages variables. À la fin du processus la meilleure solution en terme de surface est retenue.

Toutes ces approches s'appuient sur un modèle d'estimation de la surface de l'architecture implémentée sur le FPGA cible. Cet estimateur nous épargne le passage par la phase de synthèse logique , coûteuse en temps de calcul. Notre modèle d'estimation se fonde sur le dénombrement des opérateurs, des registres et des multiplexeurs utilisés dans la solution. Une fois ces informations collectées, l'estimateur est capable d'approximer la surface en nombre de portes logiques à l'aide d'une bibliothèque qui contient une caractérisations de tous les composants susceptibles d'être utilisés.

Bien que les approches proposées conduisent à des améliorations considérables au niveau surface sur les exemples testés, plusieurs améliorations et extensions peuvent être envisagées.

Ainsi un ensemble de tests intégrants une estimation de la taille du contrôleur doivent être menés. En effet, théoriquement la minimisation du nombre de registres et des multiplexeurs doit comprimer le contrôleur.

L'implémentation d'autres algorithmes de sélection et d'ordonnancement dans le processus de base de GAUT peut permettre de générer plus de solutions de départ pour l'approche de descente à départs multiples.

L'implémentation du voisinage du ré-ordonnancement apportera une autre dimension à l'espace des solutions explorées. L'exploration de ce voisinage nous permettra sans doute d'échapper

au dernier optimum local rencontré dans le VNS actuel. En plus, une fois ce voisinage implémenté nous pourrons retenter d'explorer le voisinage de la réallocation des opérateurs qui s'est montré stérile jusqu'à maintenant.

Enfin, des travaux qui s'appuient sur des expériences sur la consommation des différentes solutions proposées, pourraient mener à la définition de relations entre minimisation de la surface et variation de la consommation.

Bibliographie

[Andriamisaina 2008] C Andriamisaina. *Synthèse d'architecture multi-modes pour les applications du traitement du signal et de l'image.* PhD thesis, Université de Bretagne Sud, Novembre 2008.

[Bacon 1994] David F. Bacon, Susan L. Graham and Oliver J. Sharp. *Compiler transformations for high-performance computing.* ACM Comput. Surv., vol. 26, no. 4, pages 345–420, 1994.

[Bakshi 1996] S. Bakshi, D. Gajski and H. Juan. *Component selection in resource shared and pipelined DSP applications.* In EURO-DAC '96/EURO-VHDL '96 : Proceedings of the conference on European design automation, pages 370–375, Los Alamitos, CA, USA, 1996. IEEE Computer Society Press.

[Brown 1996] S. Brown and J. Rose. *FPGA and CPLD architectures : A tutorial.* IEEE Design & Test of Computers, vol. 13, no. 2, pages 42–57, 1996.

[Chen 2004] Deming Chen and Jason Cong. *Register binding and port assignment for multiplexer optimization.* In ASP-DAC '04 : Proceedings of the 2004 Asia and South Pacific Design Automation Conference, pages 68–73, Piscataway, NJ, USA, 2004. IEEE Press.

[Cong 2006] Jason Cong and Zhiru Zhang. *An efficient and versatile scheduling algorithm based on SDC formulation.* In DAC '06 : Proceedings of the 43rd annual Design Automation Conference, pages 433–438, New York, NY, USA, 2006. ACM.

[Cong 2008] Jason Cong and Junjuan Xu. *Simultaneous FU and register binding based on network flow method.* In DATE '08 : Proceedings of the conference on Design, automation and test in Europe, pages 1057–1062, New York, NY, USA, 2008. ACM.

[Coussy 2003] P Coussy. *Synthèse d'Interface de Communication pour les Composants Virtuels.* PhD thesis, Université de Bretagne Sud, Juin 2003.

[Coussy 2008] P. Coussy. High-Level Synthesis : From Algorithm to Digital Circuit. Springer Verlag, 2008.

[Deb 2001] K. Deb. Multi-objective optimization using evolutionary algorithms. Wiley, 2001.

[D'silva 2005] V. D'silva, S. Ramesh and A. Sowmya. *Synchronous protocol automata : a*

framework for modelling and verification of SoC communication architectures. IEE Proceedings-Computers and Digital Techniques, vol. 152, no. 1, pages 20–27, 2005.

[Ernst 1998] Rolf Ernst. *Codesign of Embedded Systems : Status and Trends.* IEEE Des. Test, vol. 15, no. 2, pages 45–54, 1998.

[Feo 1989] TA Feo and MGC Resende. *A probabilistic heuristic for a computationally difficult set covering problem.* Operations Research Letters, vol. 8, no. 2, pages 67–71, 1989.

[Ferrandi 2007] F. Ferrandi, PL Lanzi, G. Palermo, C. Pilato, D. Sciuto and A. Tumeo. *An evolutionary approach to area-time optimization of FPGA designs.* In Embedded Computer Systems : Architectures, Modeling and Simulation, 2007. IC-SAMOS 2007. International Conference on, pages 145–152, 2007.

[Festa 2003] P. Festa. *Greedy randomized adaptive search procedures.* AIROnews, vol. 7, no. 4, pages 7–11, 2003.

[Gajski 1992] D.D. Gajski, N.D. Dutt, C.H.W. Allen and Y.L.L. Steve. High-level synthesis : introduction to chip and system design. Kluwer Academic Publishers Norwell, MA, USA, 1992.

[Galil 1986] Z. Galil. *Efficient algorithms for finding maximum matching in graphs.* ACM Computing Surveys (CSUR), vol. 18, no. 1, pages 23–38, 1986.

[Garey 1979] M.R. Garey and D.S. Johnson. Computers and intractability : A guide to the theory of NP–completeness. Freeman, New York, NY, 1979.

[Glover 1986] F. Glover. *Future paths for integer programming and links to artificial intelligence.* Computers and Operations research, vol. 13, no. 5, pages 533–549, 1986.

[Guéret 2000] C. Guéret, C. Prins and M. Sevaux. *Programmation linéaire.* 2000.

[Gutberlet 1992] P. Gutberlet, J. Müller, H. Krämer and W. Rosenstiel. *Automatic module allocation in high level synthesis.* In EURO-DAC '92 : Proceedings of the conference on European design automation, pages 328–333, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[Hao 1999] J.K. Hao, P. Galinier and M. Habib. *Metaheuristiques pour l'optimisation combinatoire et l'affectation sous contraintes.* Revue d'Intelligence Artificielle, vol. 13, no. 2, pages 283–324, 1999.

[Hashimoto 1971] Akihiro Hashimoto and James Stevens. *Wire routing by optimizing channel assignment within large apertures.* In DAC '71 : Proceedings of the 8th Design Automation Workshop, pages 155–169, New York, NY, USA, 1971. ACM.

[Henkel 2007] J. Henkel, S. Parameswaran and SpringerLink (Online service. Designing Embedded Processors : A Low Power Perspective. Springer, 2007.

[itr 2007] *International Technology Roadmap for Semiconductors.* 2007.

[Jain 2001] MK Jain, M. Balakrishnan and A. Kumar. *ASIP design methodologies : survey and issues.* In VLSI Design, 2001. Fourteenth International Conference on, pages 76–81, 2001.

[Kurdahi 1987] FJ Kurdahi and AC Parker. *REAL : a program for REgister ALlocation.* In Proceedings of the 24th ACM/IEEE conference on Design automation, pages 210–215. ACM New York, NY, USA, 1987.

[Martí 2003] R. Martí. *Multi-start methods.* INTERNATIONAL SERIES IN OPERATIONS RESEARCH AND MANAGEMENT SCIENCE, pages 355–368, 2003.

[Martin 2009] G. Martin, G. Smith and G.S. EDA. *High-Level Synthesis : Past, Present, and Future.* IEEE Design & Test, vol. 26, no. 4, pages 18–25, 2009.

[Micheli 1994] G.D. Micheli. *Synthesis and optimization of digital circuits*, 1994.

[Micheli 2001] Giovanni De Micheli, Wayne Wolf and Rolf Ernst. Readings in hardware/software co-design. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[Mladenović 1997] N. Mladenović and P. Hansen. *Variable neighborhood search.* Computers and Operations Research, vol. 24, no. 11, pages 1097–1100, 1997.

[Moore 1998] GE Moore. *Cramming more components onto integrated circuits.* Proceedings of the IEEE, vol. 86, no. 1, pages 82–85, 1998.

[Nekoogar 2003] F. Nekoogar and F. Nekoogar. From ASICs to SOCs : a practical approach. Prentice Hall PTR, 2003.

[Orailoglu 1986] Alex Orailoglu and Daniel D. Gajski. *Flow graph representation.* In DAC '86 : Proceedings of the 23rd ACM/IEEE conference on Design automation, pages 503–509, Piscataway, NJ, USA, 1986. IEEE Press.

[Pangrle 1991] Pangrle. *On the Complexity of Connectivity Binding.* IEEETCAD : IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 10, 1991.

[Papadimitriou 1998] C.H. Papadimitriou and K. Steiglitz. Combinatorial optimization : algorithms and complexity. Dover Pubns, 1998.

[Park 1991] I.C. Park and C.M. Kyung. *Fast and near optimal scheduling in automatic data path synthesis.* In Proceedings of the 28th conference on ACM/IEEE design automation, pages 680–685. ACM New York, NY, USA, 1991.

[Paulin 1989a] PG Paulin, JP Knight, B.N. Res and O. Ottawa. *Force-directed scheduling for the behavioral synthesis of ASICs.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 8, no. 6, pages 661–679, 1989.

[Paulin 1989b] Pierre G. Paulin and John P. Knight. *Algorithms for High-Level Synthesis.* IEEE Des. Test, vol. 6, no. 6, pages 18–31, 1989.

[Rabaey 2000] J.M. Rabaey. *Low-power silicon architectures for wireless communications.* In Asia and South Pacific Design Automation Conference, pages 377–380, 2000.

[Rau 1994] B. Ramakrishna Rau. *Iterative modulo scheduling : an algorithm for software pipelining loops.* In MICRO 27 : Proceedings of the 27th annual international symposium on Microarchitecture, pages 63–74, New York, NY, USA, 1994. ACM.

[Sevaux 2004] M Sevaux. *Métaheuristiques : Stratégies pour l'optimisation de la production de biens et de services.* PhD thesis, Juillet 2004.

[Sevaux 2007] M. Sevaux, K. Sorensen and A. Singh. *Min-Cost electronic design using tabu search.* In Proceedings of 7th Metaheuristics International Conference, MIC, pages 951–952, 2007.

[Sorensen 2003] K Sorensen. *A framework for robust and flexible optimisation using metaheuristics-with applications in supply chain design.* PhD thesis, Avril 2003.

[Stojcev 2005] Mile K. Stojcev. *Design of Energy-Efficient Application-Specific Instruction Set Processors (ASIPs), Tilman Glokler, Heinrich Meyr, Kluwer Academic Publishers, Boston, 2004, ISBN 1-4020-7730-0, Hardcover, pp 234, plus XX.* Microelectronics Reliability, vol. 45, no. 7-8, pages 1270–1271, 2005.

[Urard 2007] Pascal Urard. *Discussion : Les Systèmes hétérogènes discret/continu.* In École d'hiver Francophone sur les technologies de Conception des systèmes embarqués Hétérogènes (FETCH), France, 2007.

[Walker 1995] Robert A. Walker and Samit Chaudhuri. *Introduction to the Scheduling Problem.* IEEE Des. Test, vol. 12, no. 2, pages 60–69, 1995.

[Wilson 1994] Thomas Charles Wilson, Gary William Grewal and Dilip K. Banerji. *An ILP Solution for Simultaneous Scheduling, Allocation, and Binding in Multiple Block Synthesis.* In ICCS '94 : Proceedings of the1994 IEEE International Conference on Computer Design : VLSI in Computer & Processors, pages 581–586, Washington, DC, USA, 1994. IEEE Computer Society.

[Wolf 2003] Wayne Wolf. *A Decade of Hardware/Software Codesign.* Computer, vol. 36, no. 4, pages 38–43, 2003.

[Xilinx 2007] I. Xilinx. *Virtex-II Pro and Virtex-II Pro X FPGA User Guide.* UG012, Version v4, vol. 1, page 28, 2007.

Glossaire

ALAP As Late As Possible, 26
ASAP As Soon As Possible, 26
ASIC Application-Specific Integrated Circuit, 5
ASIP Application Specific Instruction Processor, 5
ASSP Application Specific Standard Product, 6

CDFG Control Data Flow Graph, 17

DFG Data Flow Graph, 17

FDS Force Directed Scheduling, 34
FPGA Field Programmable Gate Arrays, 6

GPP General Purpose Preprocessor, 4

HDL Hardware Description Language, 8
HLS High level Synthesis, 16

IP Intellectual Property, 10

RTL Register Transfert Level, 8

SDC System of Difference Constraints, 31
SFR Simultaneous Functional unit and Register, 41
SoC System on Chip, 6

TDSI Traitement Digital du Signal et l'Image, 6

VLSI Very Large Scale Integration, 7, 16

Métaheuristiques pour la synthèse de haut niveau

Ce manuscrit présente des travaux à la croisée des domaines de la recherche opérationnelle et de la synthèse de haut niveau.
Le marché de la technologie numérique évoluant très rapidement, des nouvelles solutions pour l'automatisation du flot de conception des circuits intégrés doivent être trouvées.
Dans cette optique les outils de synthèse de haut niveau apparaissent pour combler le fossé entre la modélisation d'une architecture et la réalisation proprement dite du circuit VLSI.
Ces outils prennent en entrée une description algorithmique, les contraintes prédéterminées et les objectifs attendus et génèrent automatiquement la description RTL du circuit.
Dans ces travaux, nous proposons différentes approches pour l'exploration de l'espace des solutions pour une application du traitement du signale et de l'image donnée, afin de trouver l'architecture ayant une surface minimum.
Cependant, pour trouver la surface réelle d'une architecture, nous passons habituellement par la synthèse logique. Cette étape est très coûteuse en temps de calcul. Nous proposons, ainsi, une technique d'estimation de cette surface basée sur un tableau permettant le démembrement des composants de l'architecture considérée et une bibliothèque caractérisant notamment, en surface ces composants.
Les approches que nous proposons appartiennent à la famille des méthodes à base de voisinages. Celles-ci s'appuient sur des techniques permettant de passer d'une solution à une solution voisine par déplacements successifs, pour en choisir la solution de meilleure qualité.
Mots clés: Synthèse de haut niveau, métaheuristique, voisinage, FPGA, estimation, VNS, GRASP, recherche locale

Metaheuristics for high level synthesis

This dissertation presents a study at the cross-road of the operational research's area and the high level synthesis's one.
If considering current state-of-the-art methods for automated integrated circuits design, it is clear that new solutions for design flow must be found, to reply to the very fast growth of the digital technology market.
High level synthesis tools appear to bridge the gap between modelling architecture and the actual achievement of the integrated circuit.
Thus, given a C/C++ specification of an application, its associated throughput constraint, and the target goal, high-level synthesis tools allows to generate automatically an RTL architecture. However, the solutions provided by the high-level synthesis are not always area efficient, due to current selection of search algorithms.
In this dissertation we propose several new approaches to explore the solutions space. Yet, to find the real architecture area, we have to pass by the logical synthesis. But this way is very expensive in computing time. So we propose also an architecture area estimation technique, based on an array giving the different components of the architecture and a library characterising notably the different components on area.
The search approaches we propose, includ a simple descent, deepest descent, a variable neighbourhood search, a multi starts descent and finally a dedicated Greedy Randomized Adaptive Search Procedure which all aiming at minimizing the global area. We show through a set of test cases that our approach offers significant gain relative to the state-of-the-art
Key Words: High Level Synthesis, metaheuristic, neighborhood, FPGA, estimation, VNS, GRASP, local search