Génération rapide d'accélerateurs matériels par synthèse d'architecture sous contraintes de ressources

Adrien Prost-Boucle

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Nano Électronique, Nano Technologies**

Arrêté ministériel : 7 août 2006

Présentée par

Adrien PROST-BOUCLE

Thèse dirigée par **Frédéric ROUSSEAU**
et co-encadrée par **Olivier MULLER**

Préparée au sein du **Laboratoire TIMA, CNRS / Grenoble INP / UJF**
dans **l'École Doctorale Électronique, Électrotechnique, Automatique et Traitement du Signal (EEATS)**

Génération rapide d'accélérateurs matériels par synthèse d'architecture sous contraintes de ressources

Thèse soutenue publiquement le **8 janvier 2014**,
devant le jury composé de :

M. Philippe COUSSY
MdC HDR, Université de Bretagne-Sud, Lorient, Rapporteur
M. Ivan AUGÉ
MdC HDR, Université Pierre et Marie Curie, Paris, Rapporteur
M. El-Bay BOURENNANE
Professeur, Université de Bourgogne, Dijon, Examinateur
M. Régis LEVEUGLE
Professeur, Université de Grenoble, Président
M. Pascal RÉMY
PhD, Directeur général de ADACSYS, Palaiseau, Invité
M. Frédéric ROUSSEAU
Professeur, Université de Grenoble, Directeur de thèse
M. Olivier MULLER
MdC, Université de Grenoble, Co-encadrant de thèse

Remerciements

Mes remerciements vont tout d'abord à Frédéric Rousseau et Olivier Muller, qui m'ont permis d'effectuer cette thèse au laboratoire TIMA. Sous leur encadrement, leurs conseils et leur patience ont grandement contribué à l'aboutissement de ces travaux.

Je remercie également Ivan Augé et Philippe Coussy, qui ont accepté de rapporter sur ma thèse, Régis Leveugle, qui a présidé le jury, El-Bay Bourennane, qui a examiné mes travaux, et Pascal Rémy, en tant qu'invité au jury, qui y a apporté son expertise industrielle.

Je remercie très chaleureusement mes collègues du laboratoire TIMA, qui ont apporté à ces quatre années de thèse beaucoup de bonne humeur et de challenge. Merci donc à Florentine, Yan, Maryam, Damien, Luc, Clément, Nicolas, Sahar, Frédéric Pétrot, Stéphane, merci à Fabien, Asma, Salma et Mohammed, merci à Laurence et Raoul, ainsi qu'à tous ceux du laboratoire ici oubliés.

Côté activité d'enseignement, je remercie Mounir Benabdenbi, Andrea Battistella, Katell Morin-Allory, Lorena Anghel et Laurent Fesquet, de l'école Phelma, ainsi qu'Alejandro Chagoya et Robin Rolland, du CIME.

Enfin, merci à toute la famille et aux amis proches pour leurs encouragements sans relâche, tout particulièrement à ceux qui ont pu faire le déplacement le jour de la soutenance. Et merci à Laure, ma chérie, pour sa patience et sa tolérance envers ma grande implication dans ces travaux !

Résumé

Dans le domaine du calcul générique, les circuits FPGA sont très attrayants pour leur performance et leur faible consommation. Cependant, leur présence reste marginale, notamment à cause des limitations des logiciels de développement actuels. En effet, ces limitations obligent les utilisateurs à bien maîtriser de nombreux concepts techniques. Ils obligent à diriger manuellement les processus de synthèse, de façon à obtenir une solution à la fois rapide et conforme aux contraintes des cibles matérielles visées.

Une nouvelle méthodologie de génération basée sur la synthèse d'architecture est proposée afin de repousser ces limites. L'exploration des solutions consiste en l'application de transformations itératives à un circuit initial, ce qui accroît progressivement sa rapidité et sa consommation en ressources. La rapidité de ce processus, ainsi que sa convergence sous contraintes de ressources, sont ainsi garanties. L'exploration est également guidée vers les solutions les plus pertinentes grâce à la détection, dans les applications à synthétiser, des sections les plus critiques pour le contexte d'utilisation réel. Cette information peut être affinée à travers un scénario d'exécution transmis par l'utilisateur.

Un logiciel démonstrateur pour cette méthodologie, AUGH, est construit. Des expérimentations sont menées sur plusieurs applications reconnues dans le domaine de la synthèse d'architecture. De tailles très différentes, ces applications confirment la pertinence de la méthodologie proposée pour la génération rapide et autonome d'accélérateurs matériels complexes, sous des contraintes de ressources strictes.

La méthodologie proposée est très proche du processus de compilation pour les microprocesseurs, ce qui permet son utilisation même par des utilisateurs non spécialistes de la conception de circuits numériques. Ces travaux constituent donc une avancée significative pour une plus large adoption des FPGA comme accélérateurs matériels génériques, afin de rendre les machines de calcul simultanément plus rapides et plus économes en énergie.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

Dans de nombreux domaines des sciences, d'importants moyens de calcul sont exploités. Parmi les principaux types de besoins, on cite notamment les simulations physiques (météorologie, modélisation des fluides, astrophysique, etc), la bioinformatique (analyses génétiques, repliement des protéines, etc), le prototypage de circuits, les traitements multimédia, le chiffrement et l'analyse de données (en plein essor grâce aux technologies liées à Internet).

Pour effectuer ces calculs de haute performance, des machines très puissantes sont employées : les supercalculateurs. Le plus souvent constituées d'un très grand nombre de microprocesseurs d'usage général (CPU), ces machines consomment une énorme quantité d'énergie électrique. Leurs coûts d'acquisition et d'exploitation sont critiques. Dans le choix et la conception de ces machines, le plus grand intérêt est porté au rapport entre la puissance de calcul et la consommation. Cette problématique existe également pour les plateformes embarquées, dans le cadre de l'optimisation de l'autonomie sur batterie.

Certes, l'augmentation de la finesse de gravure des circuits apporte de grands progrès au niveau de l'efficacité énergétique. Mais les besoins augmentent beaucoup plus vite, au point que des ruptures technologiques sont nécessaires. Une rupture technologique a déjà eu lieu grâce à l'adoption des processeurs graphiques d'usage général (GPGPU) comme accélérateurs matériels. Cette rupture a exigé de revoir la plupart des applications de calcul, afin de distribuer au mieux l'exécution des différents types de calcul sur les unités de calcul (processeur central ou bien accélérateur matériel). De nos jours, la plupart des supercalculateurs sont basés sur des noeuds de calcul composés de processeurs et d'accélérateurs matériels GPGPU. Ces accélérateurs sont également exploités sur les téléphones mobiles, notamment pour l'encodage et le décodage de la vidéo aussi bien que les jeux.

Cependant, pour de nombreuses catégories d'applications, les GPGPU sont encore trop énergivores. Ils sont constitués d'un grand nombre de processeurs simples, basés sur des opérateurs à virgule flottante, et ils fonctionnent sur le principe SIMD (application d'un même traitement sur plusieurs données). Ils sont donc relativement peu performants pour des applications avec beaucoup de contrôle, et leur consommation est encore inappropriée pour des opérations logiques ou lorsque la précision des opérateurs matériels est supérieure à la précision des calculs à effectuer.

Pour ces catégories de calcul, un autre type de circuit existe : le FPGA. Contrairement aux microprocesseurs et aux GPGPU, un FPGA n'a pas de jeu d'instructions propre et figé. Il est plutôt

constitué d'un très grand nombre d'opérateurs logiques élémentaires programmables, connectés entre eux par un tissu d'interconnexion, également programmable. Pour utiliser un FPGA afin de faire des calculs, ce circuit doit être programmé de façon à ce que ses éléments internes *implémentent* des opérateurs de calcul conçus et optimisés spécifiquement pour l'application à exécuter.

Le fait de pouvoir configurer la totalité de la surface du FPGA de façon optimisée pour les applications confère à ces circuits une capacité de calcul immense. Utilisés comme accélérateurs matériels, ils sont déjà considérés comme des solutions efficaces, qui peuvent typiquement apporter des gains en consommation et en accélération de un à trois ordres de grandeur, par rapport à des implémentations sur CPU ou GPGPU [PTD13]. Bien que ce soit encore de façon marginale, ils sont utilisés pour du calcul de hautes performances [TB08], et dans des supercalculateurs spécialisés [GLS11].

Développement pour FPGA

La conception d'un accélérateur matériel pour FPGA passe traditionnellement par des flots de conception très proches de la conception de circuits numériques dédiés (ASIC). Les langages de description de niveau transferts entre registres (RTL), tels VHDL et Verilog, en sont les points d'entrée. Ces langages permettent de décrire précisément n'importe quelle architecture de circuit numérique. L'emploi de logiciels spécialisés, couramment fournis par les fabricants des FPGA, permet de générer les données de programmation du FPGA à partir d'une telle description. Cependant, à cause du niveau de détail très élevé de ces langages, pour des applications complexes, la tâche de développement au niveau RTL est extrêmement fastidieuse.

Pour cette raison, le développement au niveau RTL est délaissé, au profit des techniques de synthèse de haut niveau (HLS). Les descriptions d'entrée des flots de HLS sont des algorithmes, d'un haut niveau d'abstraction. Autrement dit, les circuits sont construits non pas en travaillant leur structure, mais en indiquant la fonctionnalité attendue. Les langages d'entrée sont ainsi beaucoup plus proches des langages employés pour le développement logiciel. Ces descriptions algorithmiques sont converties, par des logiciels de HLS, en circuits de calcul décrits au niveau RTL. Les techniques de HLS permettent donc d'atteindre une productivité élevée, c'est-à-dire que le temps d'obtention d'une solution convenable est très court, comparé au développement directement au niveau RTL [CGMT09].

Cependant, pour une description d'entrée donnée, il existe une très large gamme de circuits remplissant la fonction décrite, et présentant des caractéristiques de surface, fréquence, latence et consommation différentes. Or, pour une application donnée à accélérer sur un FPGA, les utilisateurs souhaitent obtenir la version la plus performante qu'il soit possible d'implémenter dans le FPGA ciblé. L'obtention de cette solution, ou d'une solution proche, est un problème d'une très grande complexité algorithmique. De plus, l'espace des solutions possibles est en général beaucoup trop vaste pour considérer une exploration exhaustive des solutions en vue de trouver la meilleure.

Afin de pallier les limites des capacités d'exploration des logiciels de HLS, l'exploration des solutions reste pilotée de façon humaine. En agissant sur des commandes spécifiques au logiciel de HLS, l'utilisateur peut imposer certaines caractéristiques structurelles du circuit généré, ou commander la transformation de certaines parties critiques du circuit. Ce processus n'est donc

acceptable que pour les utilisateurs avec un bon niveau d'expertise, tant au niveau de la technologie ciblée, que du logiciel de HLS employé et de l'application à accélérer, et qui peuvent y consacrer le temps.

Ce flot de développement est toujours très loin de la compilation pour CPU ou GPGPU. En effet, cette compilation est effectuée de façon automatisée, sans pilotage manuel, sans aucune connaissance du compilateur, et très rapidement (voire au moment de l'utilisation, grâce aux techniques *just-in-time*). Pour une cible CPU, il n'est même pas besoin de connaître l'architecture du processeur ciblé ni de comprendre le code à compiler. Malgré leurs relativement pauvres performances intrinsèques, les architectures à base de CPU et GPGPU sont bien mieux acceptées par les utilisateurs, grâce à la souplesse des flots de développement existants.

Il faut donc construire un environnement de HLS pour FPGA, qui soit capable de produire rapidement une solution RTL performante, avec ou sans intervention de l'utilisateur. Ainsi, afin d'être autonome, un tel flot doit pouvoir s'adapter de façon transparente aux caractéristiques de la cible matérielle donnée. Notamment, la fréquence de fonctionnement ciblée et la quantité de ressources matérielles disponibles (un FPGA entier ou bien une partition) doivent être respectées strictement.

Ce mémoire présente une méthodologie d'exploration des solutions capable de générer rapidement et de façon automatique des circuits respectant des contraintes matérielles données. Cette méthodologie est applicable à toute technologie de FPGA, voire aux circuits dédiés (ASIC), et peut également être intégrée au sein des logiciels de HLS existants.

Plan du mémoire

Le chapitre 2 définit précisément le problème et les contributions de ces travaux. Le chapitre 3 présente l'état de l'art des travaux dans le contexte de la HLS rapide, autonome et sous contraintes de ressources. Le chapitre 4 décrit la méthodologie proposée. Le chapitre 5 présente la construction d'un logiciel de HLS démonstrateur. Le chapitre 6 détaille les résultats obtenus pour différentes applications. Le chapitre 7 conclut ce mémoire.

Chapitre 2

Problématique

Ce chapitre présente la problématique générale de cette thèse, et en détaille les enjeux fondamentaux. Il discute la difficulté de construire un flot de développement en synthèse de haut niveau, qui soit réellement capable de cibler une plateforme matérielle donnée, de manière automatisée.

En appuyant nos réflexions sur les techniques actuelles de conception en HLS, nous identifierons précisément les catégories d'utilisateurs, et les contextes d'utilisation, pour lesquels les solutions existantes ne sont pas adaptées. Nous décomposerons la problématique générale en enjeux élémentaires. Nous discuterons leur multiplicité et leur antagonisme, la complexité théorique du problème, et des compromis qu'il sera nécessaire de faire pour rendre le problème accessible.

Sommaire

2.1 La synthèse de haut niveau

2.1.1 Évolution depuis la conception RTL

Les langages de description de niveau transferts entre registres (RTL) sont les points d'entrée traditionnels pour la conception d'accélérateurs matériels sur FPGA. Pour référence, le flot de développement RTL courant est illustré en Figure 2.1. Le concepteur, expert en accélérateurs matériels et circuits numériques, génère manuellement une première version du circuit en utilisant un langage adapté, tels VHDL et Verilog. Une opération de synthèse logique, avec un logiciel approprié, décompose cette description RTL selon les primitives matérielles réelles du FPGA (LUT, bascules D, etc). Le niveau de description obtenu est appelé *netlist*. Cette opération fournit également les caractéristiques matérielles précises du circuit (classiquement, en surface et fréquence). Le concepteur évalue la conformité par rapport à ses contraintes et objectifs et, au besoin, modifie la description RTL afin de chercher une meilleure solution.

La dernière opération est le placement-routage, qui consiste à disposer les éléments de la *netlist* dans le FPGA ciblé, et à les connecter entre eux via le tissu d'interconnexion du FPGA. Le résultat final, appelé *bitstream*, est l'ensemble des données binaires permettant de programmer le FPGA.

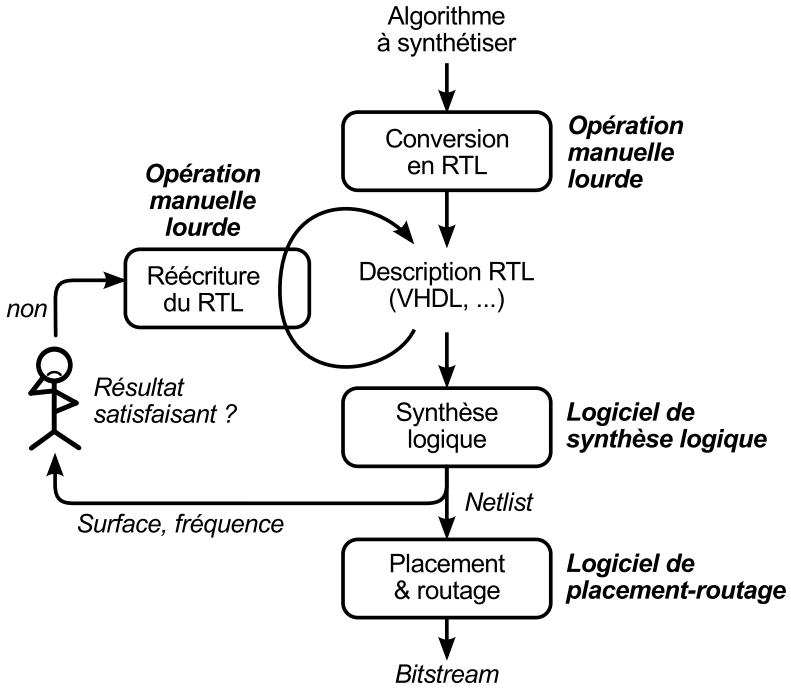


FIGURE 2.1 – Conception au niveau RTL

Durant les quelques dernières dizaines d'années, les avancées en synthèse de haut niveau (HLS) ont permis à ces techniques de remplacer la conception RTL dans la génération de certains accélérateurs matériels. Comme illustré en Figure 2.2, l'introduction d'un logiciel de HLS dans le flot de conception permet d'automatiser une partie de la génération de la description RTL. La description de la fonctionnalité du circuit à concevoir est donnée sous forme d'un algorithme.

C'est une description de haut niveau, purement fonctionnelle et sans référence au temps ou à une horloge explicite. Les tâches les plus fastidieuses et les plus répétitives de l'écriture de la description RTL sont alors confiées au logiciel de HLS.

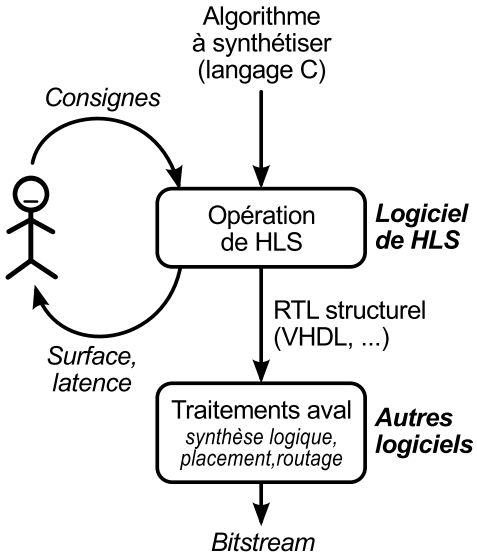


FIGURE 2.2 – Flot de conception actuel utilisant la HLS

Dans ce flot de HLS, l'utilisateur peut donner des consignes spécifiques sur la structure du circuit à construire (type des opérateurs et leur nombre, degré de partage des opérateurs, dimensionnement des mémoires, etc). Après génération, les caractéristiques matérielles du circuit final (principalement surface, fréquence et latence en cycles d'horloges) sont exposées à l'utilisateur pour analyse. Celui-ci peut alors choisir de conserver la solution obtenue, ou bien de tenter d'en trouver une meilleure en agissant sur des commandes spécifiques du logiciel de HLS.

2.1.2 Contexte d'utilisation et public visé

La technique de HLS permet à l'utilisateur d'explorer relativement rapidement l'espace des solutions [CGMT09], comparé à un développement directement au niveau RTL. Elle améliore la productivité des concepteurs d'accélérateurs matériels. En particulier, dans le domaine industriel, elle permet de réduire le temps de mise sur le marché des circuits.

Grâce au pilotage manuel, le concepteur conserve une forte maîtrise de la structure des circuits générés. Ainsi, l'expertise du concepteur est mise à contribution en vue de générer une solution optimisée. Les utilisateurs sont, presque autant que pour le développement RTL, des spécialistes du domaine de la conception de circuits et bons connaisseurs de la technologie matérielle ciblée. La HLS actuelle cible donc des domaines où le niveau d'optimisation des circuits est décisif. Sans toutefois parvenir à une génération immédiate du circuit, le logiciel de HLS assiste les concepteurs et les soulage des tâches les plus fastidieuses et les moins gratifiantes. Le niveau d'optimisation du circuit final mérite d'y consacrer des moyens humains et de prendre le temps d'explorer un certain nombre de solutions.

Mais dans de nombreux domaines d'applications où la puissance de calcul des FPGA serait bienvenue (calculs de haute performance, simulation, analyse audio et vidéo, etc), le domaine

d'expertise des utilisateurs directs n'est pas la conception de circuits numériques sur FPGA. Les accélérateurs matériels sont conçus pour des applications très spécifiques, et éventuellement sujets à de fréquentes évolutions. Le temps de génération des accélérateurs matériels et la complexité de ce processus revêtent donc une grande importance dans le choix des machines de calcul, et en conséquence pour l'acceptation des FPGA en général. Ces utilisateurs non experts ont donc besoin de flots de développement pour FPGA suffisamment performants pour leur permettre d'exploiter efficacement la puissance de leur machine.

Cette thèse se place dans le contexte de la synthèse de haut niveau de circuits complexes, pour des utilisateurs peu experts en conception de circuits sur FPGA.

Les machines de ces utilisateurs prennent généralement la forme illustrée en Figure 2.3. L'accélérateur matériel avec FPGA est contrôlé de façon logicielle, depuis un microprocesseur. La machine est exploitée selon les besoins des utilisateurs, ce qui peut correspondre à une ou plusieurs applications exigeant une accélération. Le FPGA est donc éventuellement partitionné, et chaque accélérateur matériel se voit attribuer une zone reconfigurable donnée (par exemple par un gestionnaire d'allocation des ressources dans le système d'exploitation). Ces zones sont alors programmées par la technique de reconfiguration dynamique partielle.

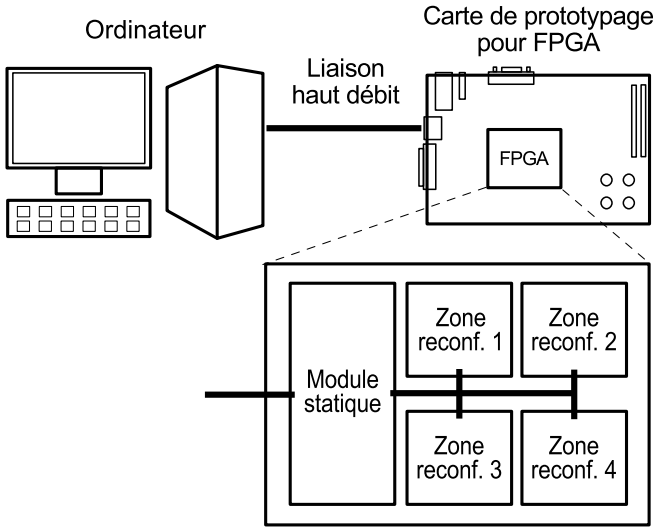


FIGURE 2.3 – Plateforme avec un FPGA partitionné

Certaines techniques liées à la reconfiguration dynamique partielle, à l'allocation des ressources matérielles et même à leur virtualisation (par multiplexage temporel sur les zones reconfigurables) sont encore à l'étude. Ces techniques, ainsi que les techniques liées à la conception des circuits accélérateurs destinés aux zones reconfigurables (et donc la HLS), sont les briques de base impliquées dans l'exploitation de la puissance des machines accélérées par FPGA. Cette thèse est focalisée sur la partie conception des coeurs accélérateurs, plus particulièrement via l'utilisation de la HLS.

2.1.3 Limites des flots de HLS actuels

Autant qu'avec les langages RTL, la HLS actuelle est destinée à des experts du domaine de la conception, et exige qu'une attention particulière soit accordée au pilotage du processus d'exploration des solutions. De plus, le processus de conception est toujours spécifique à une cible matérielle donnée, et les efforts, en particulier humains, doivent être répétés pour chaque cible. Cela aide donc peu les utilisateurs de domaines complètement différents, qui souhaiteraient bénéficier de la puissance des FPGA pour leurs applications. En ce sens, la HLS actuelle est insuffisante pour provoquer un élargissement des domaines d'utilisation des FPGA et du public utilisateur.

De plus, même pour les utilisateurs avertis, les flots de HLS actuels n'exposent que très peu d'indicateurs de qualité des solutions. Ceux-ci (surface, fréquence, latence en cycles d'horloge) sont des métriques agrégats représentant la qualité globale du circuit généré. Elles fournissent donc très peu d'indices pour guider l'utilisateur dans ses choix structurels. Les données exposées étant à la fois maigres et peu représentatives pour l'utilisateur, celui-ci contrôle alors la progression seulement par tâtonnement.

Ainsi, dans le contexte de la génération d'un accélérateur matériel pour une application logicielle existante, ce procédé reste très fastidieux. Afin d'étendre l'utilisation des FPGA comme accélérateurs matériels d'usage général, les logiciels de développement devraient être utilisables de façon plus souple, et accessibles aux utilisateurs non initiés. Les processus de HLS doivent évoluer de façon à ressembler aux processus de compilation.

Dans les flots de HLS actuels, l'utilisateur doit être expert à la fois de la technologie ciblée, du logiciel de HLS utilisé et de l'application à synthétiser. Globalement, on souhaite passer à des flots où il est seulement exigé de l'utilisateur qu'il connaisse bien l'application à synthétiser. L'environnement de HLS devant alors se charger de toutes les tracasseries liées aux spécificités de la cible matérielle. Dans ce contexte, où une grande autonomie du flot de génération est attendue, la problématique peut se décomposer selon plusieurs plans.

- Comment générer automatiquement des circuits rapides ?
- Comment respecter les caractéristiques technologiques de la cible matérielle ?
- Comment effectuer la génération de façon rapide ?
- Comment intégrer un tel flot de HLS dans une chaîne de génération impliquant couramment d'autres logiciels ?
- Un flot de HLS automatisé peut-il également permettre aux experts de contrôler l'exploration ?

Nous verrons que ces problématiques sont interdépendantes. Pour l'intégrité des travaux, nous aborderons chacune de ces problématiques, et chercherons à faire cohabiter des solutions respectives, de façon cohérente, au sein d'un unique flot de génération. Ces enjeux sont détaillés dans les sections suivantes.

2.2 Enjeux de la problématique

2.2.1 Génération de circuits rapides

La rapidité d'un circuit est une mesure de la vitesse à laquelle il effectue les calculs. À une fréquence d'exécution donnée, un circuit est rapide si le temps (en cycles d'horloge) entre l'acquisition des données d'entrée et la mise à disposition des résultats est faible.

Dans le contexte visé, celui de la synthèse de circuits relativement complexes, la description d'entrée contient très souvent des structures de contrôle dépendant des données. En conséquence, selon les données à traiter (les stimuli), le temps d'exécution du circuit final est variable. Dans le cas général, il n'existe donc pas de structure du circuit final qui soit optimale pour tous les stimuli possibles. Or, le logiciel de HLS doit produire un circuit unique. Il faut pouvoir résoudre ce dilemme.

On introduit alors la notion de *scénario d'exécution*. Un scénario d'exécution représente tout jeu de stimuli d'entrée possibles qui correspondent à des comportements similaires pour le circuit à générer. Un scénario d'exécution peut par exemple faire référence à un protocole de communication ou un certain codage de l'information. Cette notion de scénario d'exécution, conceptuellement, n'a de signification que pour l'utilisateur. Cependant, un scénario d'exécution donné reflète un certain fonctionnement du circuit à générer, concernant notamment des prises de branchements conditionnels plus fréquents que d'autres au sein de la description d'entrée.

Seul l'utilisateur connaît les scénarios d'exécutions auxquels le circuit final sera confronté. L'utilisateur peut également envisager des objectifs d'optimisation particuliers, impliquant éventuellement plusieurs scénarios d'exécution. Par exemple, il peut souhaiter des performances en moyenne, pour une certaine catégorie, ou bien dans le pire cas des scénarios d'exécution possibles. Pour permettre au logiciel de HLS d'être autonome, il semble donc que l'utilisateur doive transmettre au logiciel de nouvelles données, contenant les scénarios d'exécution utiles et des consignes d'optimisation (Figure 2.4). Cela demande toutefois un effort nouveau de la part de l'utilisateur, consistant en la génération, la structuration et la transmission de ces nouvelles informations au logiciel de HLS. Cette étape semble malheureusement inévitable.

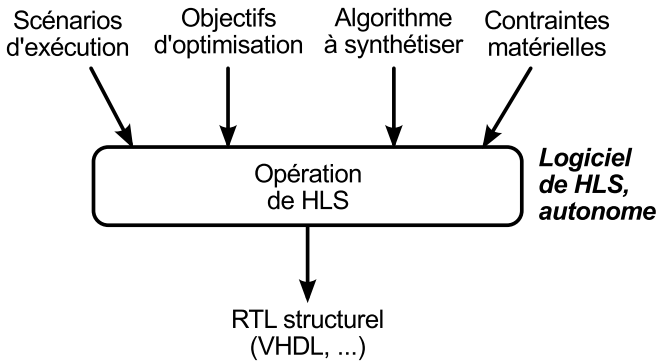


FIGURE 2.4 – Flot de HLS souhaité

Ces interrogations peuvent être exprimées par les problématiques suivantes :

- Quelles informations les scénarios d'exécution doivent-ils contenir ?
- Comment faciliter, pour l'utilisateur, les tâches de construction et de transmission des scénarios d'exécution au logiciel de HLS ?
- Comment gérer un objectif d'optimisation impliquant de multiples scénarios d'exécution ?

Réciproquement, pour une solution structurelle donnée, le logiciel de HLS doit être capable d'évaluer sa rapidité. En effet, afin de produire un circuit rapide, le logiciel de HLS est amené à effectuer, en interne, une certaine exploration de l'espace des solutions. Cela mène à de nouvelles problématiques.

- Comment évaluer et comparer la rapidité relative de plusieurs solutions générées ?
- Quelles étapes algorithmiques permettent d'atteindre l'objectif de rapidité du circuit final ?

2.2.2 Respect de contraintes matérielles

La génération d'un circuit rapide implique de faire bon usage des ressources matérielles disponibles. En effet, on peut généralement considérer que pour une architecture efficace, plus la quantité de ressources exploitées est grande, plus le circuit peut effectuer de calculs simultanément, et donc plus le circuit généré est rapide. Cependant, dans notre contexte, les ressources matérielles sont limitées, et la problématique associée n'est pas qu'une simple notion de quantité. La cible matérielle pour le flot de HLS étant un circuit FPGA donné, ou bien une zone particulière d'un FPGA donné, les conséquences sur le circuit à construire sont fortes :

— critère de ressources : il doit pouvoir être contenu dans le FPGA (ou la zone ciblée), au sens de la quantité de ressources matérielles pour le calcul (on parle aussi, abusivement, de surface du circuit),
— critère de fréquence : il doit pouvoir fonctionner correctement à une fréquence donnée (celle imposée par la carte de prototypage, ou bien par les interfaces entre les zones reconfigurables),
— critère de routabilité : il doit exister au moins une façon de placer les constituants du circuit dans le FPGA, et de les connecter en suivant la topologie et la capacité du tissu d'interconnexion interne au FPGA.

En plus de construire un circuit répondant à la fonctionnalité de calcul demandée, il est exigé du logiciel de HLS que le circuit en question respecte ces contraintes matérielles. Cela mène à de nouvelles problématiques :

- Comment évaluer la conformité des solutions considérées vis-à-vis des contraintes matérielles ?
- Comment évaluer l'optimalité des solutions considérées vis-à-vis des ressources disponibles ?

Cependant, en général, le logiciel de HLS n'est pas le seul logiciel du flot de génération. Comme illustré en Figure 2.4, l'étape de HLS est suivie des étapes de synthèse logique et de

placement-routage, effectuées par des logiciels spécialisés. Or ces logiciels, désignés par la suite comme *logiciels aval*, peuvent effectuer des transformations locales dans le circuit qui leur est transmis. On peut notamment citer la duplication de registres ou de LUT, qui permet de limiter la sortance *fanout* et d'optimiser la fréquence atteignable. Similairement, selon le niveau structurel du code RTL généré par HLS, la sélection de la structure de la FSM et de certains opérateurs peut être volontairement déportée au logiciel de synthèse logique. De plus, seul le logiciel de placement-routage (en général, seul le vendeur du FPGA fournit ce logiciel) peut conclure définitivement sur la latence exacte des interconnexions et la routabilité du circuit.

Ainsi, le logiciel de HLS peut être amené à prendre en compte l'existence de ces logiciels aval, voire à embarquer, pour certaines de ses fonctionnalités internes, une calibration spécifique aux logiciels utilisés en aval. D'une certaine façon, on peut considérer que ces logiciels aval *font partie* des contraintes matérielles. En particulier, afin de garantir le respect des contraintes imposées par l'utilisateur (ce qui est impératif), le logiciel de HLS peut être amené à estimer (voire à sur-estimer par sécurité) l'impact des éventuelles transformations effectuées par les logiciels aval. La Figure 2.5 illustre les compromis à effectuer entre la sécurité et la précision. La sécurité, pour garantir autant que possible, dès la sortie du logiciel de HLS, que le circuit après placement-routage respectera les contraintes imposées par l'utilisateur. Et la précision, afin de maximiser l'utilisation finale des ressources disponibles, en vue de générer des circuits rapides.

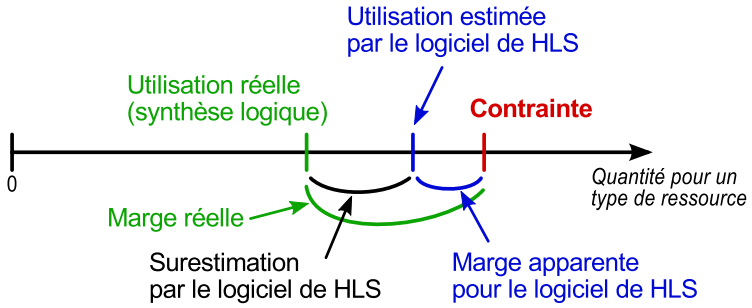


FIGURE 2.5 – Estimation des ressources utilisées et ressources réelles

- Comment prendre en compte l'impact des logiciels aval dans l'étape de HLS ?
- Comment répartir, entre les différents logiciels du flot de génération, la responsabilité du respect des contraintes matérielles ?
- Quel compromis effectuer entre la précision et la sécurité ?

2.2.3 Rapidité du flot de génération

La problématique autour du respect des contraintes matérielles a également un impact fort sur la rapidité globale du flot de génération. En effet, couramment, le flot de génération complet fait intervenir plusieurs logiciels spécialisés, notamment un logiciel de synthèse logique et un logiciel de placement-routage. Étant situés en aval de l'opération de HLS, ces logiciels peuvent produire des évaluations très précises sur les caractéristiques matérielles des circuits générés.

Couramment, les concepteurs estiment que les indicateurs de surface et de fréquence fournis par le logiciel de synthèse logique sont suffisamment fiables. Le circuit final reste en général valide pour le placement-routage. C'est ce flot qui est utilisé traditionnellement dans la conception

RTL, illustrée précédemment en Figure 2.1. Ainsi, pour les objectifs de précision et de sécurité, le logiciel de HLS pourrait être amené à exploiter les indicateurs fournis par ces logiciels aval. Ces logiciels aval seraient alors inclus dans la boucle d'exploration des solutions, pilotée par le logiciel de HLS (Figure 2.6).

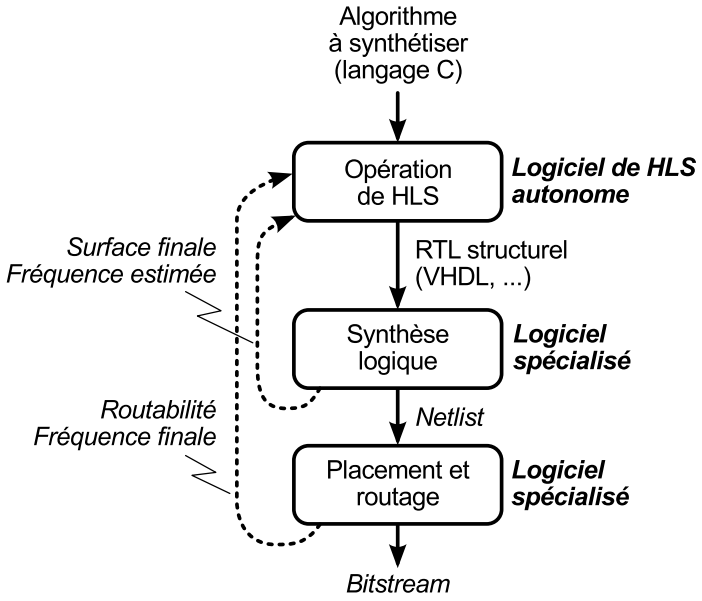


FIGURE 2.6 – Logiciel de HLS exploitant les indicateurs du flot aval

Cependant, les étapes de synthèse logique et surtout de placement-routage sont très longues. Pour des circuits complexes, ces opérations peuvent prendre plusieurs heures, voire dizaines d'heures. Ainsi, pour la rapidité globale du flot de génération, éviter autant que possible ces itérations impliquant les logiciels aval semble essentiel.

Par exemple, le logiciel de HLS pourrait être amené à surestimer (par sécurité) l'évaluation des caractéristiques des circuits. Cela correspond à réserver une certaine marge par rapport aux contraintes données, comme illustré précédemment en Figure 2.5 (page 24). Et dans ce cas, utiliser des méthodes d'évaluations peu précises, donc potentiellement rapides, est possible. Par exemple, envisager systématiquement le pire cas contribuerait à la rapidité du flot de génération général, mais nuirait généralement à la rapidité des circuits générés.

- Quel compromis effectuer entre la précision des évaluations internes et la rapidité du flot de génération ?
- Comment, dès la fin de l'opération de HLS, garantir la validité du circuit après placement-routage ?

Cependant, réduire le temps de calcul à l'intérieur de la boucle d'exploration des solutions n'est peut-être pas la seule mesure essentielle à prendre. Notamment, même en considérant le logiciel de HLS seul, la complexité algorithmique du processus d'exploration des solutions peut exploser à cause de la quantité de combinaisons des possibilités structurelles possibles. Parmi les transformations structurelles les plus courantes, on peut citer :

— transformation des boucles (déroulement séquentiel, exécution parallèle, pipeline, fusion, recouvrement, transformations polyédriques),
— transformation des conditions (câblage, incorporation du test dans les instructions),
— renommage de registres et insertion de registres temporaires,
— implantation des tableaux de données (en mémoire LUTRAM, dans les blocs BRAM du FPGA, ou dans les registres du FPGA),
— sélection de la structure des mémoires (nombre de ports, fusion, duplication, scission, etc),
— implémentation de la FSM (codage "1 parmi N", "2 parmi N", binaire, etc),
— sélection du nombre, de la taille et du degré de partage des opérateurs de chaque type,
— sélection de la structure des opérateurs de chaque type (polymorphisme selon implémentation en LUT ou avec les coeurs DSP, avec pipeline ou combinatoire, etc),
— création d'opérateurs spécifiques optimisés pour l'application.

En combinant ces paramètres, avec toutes leurs variations possibles, l'espace des solutions est gigantesque. Il est même connu que, pour un jeu d'opérateurs donné, la seule recherche d'un ordonnancement optimal est un problème NP-complet. Or, là où un flot de HLS ayant les qualités souhaitées est le plus utile, c'est justement pour la synthèse de circuits complexes. Pour ces problèmes, le nombre de possibilités structurelles est immense. Parcourir l'espace des solutions exhaustivement est alors un processus d'une complexité algorithmique extrême.

Dans notre contexte, les enjeux sont réduits au seul objectif de rapidité du circuit. Les contraintes matérielles réduisent la taille de l'espace des solutions à explorer. Cependant, il semble que la recherche des solutions optimales réelles reste tout de même une quête déraisonnablement longue.

Afin de favoriser la rapidité de génération, au détriment de la rapidité des circuits générés, on introduit la notion de rapidité *satisfaisante*. L'idée générale est que, pour des applications compatibles, l'utilisation d'un accélérateur matériel sur FPGA peut apporter des gains très importants par rapport à une solution basée sur un microprocesseur. Une implantation efficace peut exhiber des gains en rapidité et en consommation de un à trois ordres de grandeur [PTD13]. Dans ce contexte, sans contraintes du domaine du temps-réel, une sous-optimalité de quelques pourcents ou même de plusieurs dizaines de pourcents reste une solution intéressante pour le public visé.

Tout le problème est alors de concevoir le coeur d'exploration du logiciel de HLS de façon à assurer, autant que possible, que la rapidité de la solution finale est relativement proche d'une solution optimale. Il existe alors un espace des solutions satisfaisantes, illustré en Figure 2.7. On rappelle qu'une solution est dite optimale au sens de Pareto s'il n'existe aucune autre solution ayant de meilleures caractéristiques selon tous les paramètres simultanément (ici, rapidité et surface). L'espace des solutions satisfaisantes est donc la surface bornée par les solutions de Pareto, la contrainte en ressources, et une limite subjective représentant la sous-optimalité acceptable sur le critère de rapidité. Il semble alors beaucoup plus facile de trouver une solution appartenant à cet espace, qui est largement plus grand que l'ensemble des solutions optimales pour la contrainte en ressources donnée (qui peut être restreint à une unique solution).

- Est-il possible de garantir que le circuit final est proche d'une solution optimale ?
- Comment exploiter la notion de rapidité *satisfaisante* pour réduire la complexité algorithmique du processus d'exploration des solutions ?
- Quel compromis effectuer entre la rapidité du circuit final et la rapidité du flot de génération ?

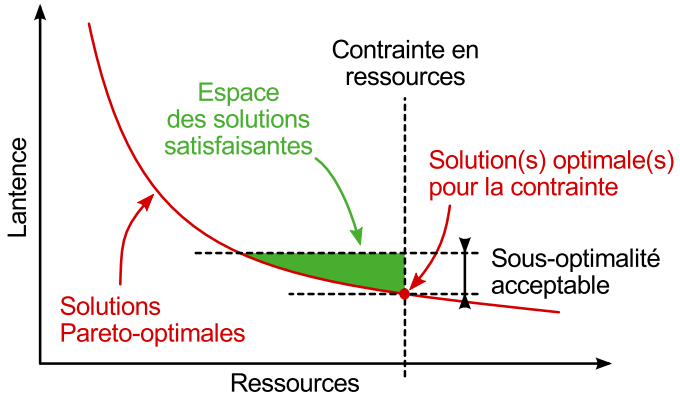


FIGURE 2.7 – Espace des solutions satisfaisantes

2.2.4 Autonomie et polyvalence

La notion de rapidité *satisfaisante* n'est que l'une des facettes représentant les différences entre un flot de HLS rapide et autonome par rapport aux flots de HLS usuels. Ces flots diffèrent sensiblement tant par les techniques exploitées en interne que par l'interface avec l'utilisateur.

L'interface utilisateur-logiciel revêt une grande importance, principalement pour la compatibilité entre les flots de HLS destinés aux experts et ceux destinés aux non-experts. Prévoir la compatibilité de ces flots, leur complémentarité et surtout leur éventuelle coexistence au sein d'un même logiciel de HLS permettrait de réconcilier les mondes des utilisateurs issus de la conception matérielle et ceux issus du développement logiciel. De la même façon, pour les développeurs de logiciels de HLS, il serait préférable de n'avoir qu'une seule déclinaison de logiciels à maintenir. Ces logiciels posséderaient alors une certaine polyvalence.

La différence du point de vue des techniques de génération internes se trouve au niveau de la capacité décisionnelle du processus contrôlant l'exploration des solutions. Sans interactivité avec l'utilisateur, un processus d'exploration autonome ne peut bénéficier ni de l'intelligence, ni de l'expertise, ni de l'intuition humaine pour guider la progression vers une implémentation respectant les contraintes et correspondant aux objectifs. Heureusement, dans notre contexte, le logiciel de HLS contrôle le processus l'exploration. Il a donc à sa disposition bien plus d'indicateurs et de données internes que ce qui est couramment exposé aux utilisateurs (par exemple les dépendances de données, les contentions d'accès aux mémoires et aux opérateurs partagés, la latence de l'ensemble des chemins de données, etc). La qualité de la capacité décisionnelle du processus d'exploration repose alors sa capacité à faire bon usage de toutes ces données internes.

Pour conclure, les aspects d'autonomie et de polyvalence sont reliés aux autres aspects de la problématique. En particulier, afin d'explorer des solutions variées et de trouver un circuit rapide, le logiciel de HLS devra sans doute être capable d'apprécier, d'une part, leurs qualités respectives (rapidité vis-à-vis des scénarios d'exécution envisagés), et leur conformité vis-à-vis des contraintes matérielles données. Réciproquement, un système capable d'apprécier la rapidité et la conformité des solutions sera à même de sélectionner les branches les plus pertinentes de l'espace des solutions, rendant ainsi le processus d'exploration plus efficace et plus rapide. Ces deux capacités, à apprécier la qualité des solutions et à sélectionner les branches de l'arbre des solutions supposées être intéressantes à explorer, sont usuellement déportés sur l'utilisateur,

comme illustré précédemment en Figure 2.2 (page 19). Un logiciel de HLS autonome doit donc compenser en reproduisant ces capacités au sein du processus d'exploration.

- Comment exploiter les données internes pour diriger intelligemment l'exploration des solutions ?
- Quelles sont les données et statistiques internes les plus utiles ?
- Dans l'objectif d'éviter une trop grande disparité des flots de HLS, peut-on laisser la possibilité aux utilisateurs experts d'imposer des choix structurels ?

2.3 Synthèse de la problématique

La thématique des présents travaux est la synthèse de haut niveau performante sans intervention de l'utilisateur. Plusieurs sous-problèmes correspondants ont été identifiés. À notre connaissance, à l'heure actuelle, il n'existe pas de flot de HLS répondant à tous ces critères. La problématique générale des présents travaux est donc de répondre à ces questions :

- Comment former un flot de génération autonome, s'intégrant de façon cohérente avec les autres logiciels du flot de génération ?
- Comment équilibrer les objectifs de rapidité du circuit final et de rapidité du flot de génération, sous une contrainte de ressources matérielles ?
- Quel niveau de polyvalence est-il possible d'atteindre avec un tel flot de conception ?

Chapitre 3

État de l'art

Ce chapitre présente les techniques existantes pour la conception de circuits pour FPGA, et les moyens d'exploration des solutions les plus adaptés à la problématique présente. Cet état de l'art aborde notamment les techniques de HLS orientées rapidité de génération, automatisation, gestion de contraintes matérielles, optimisation ou exploration des solutions.

Sommaire

3.1 Techniques de génération en HLS

Comme décrit dans [CM08], historiquement, le flot de génération HLS est communément composé de trois étapes : l'allocation, l'ordonnancement et l'assignation. L'allocation définit le type des unités matérielles dans le circuit final (i.e. calcul, mémorisation, routage), ainsi que leur nombre. L'ordonnancement décide de l'instant d'exécution de toutes les instructions, tout en garantissant le respect des dépendances de données. L'assignation associe chaque instruction à un jeu d'unités matérielles, pendant les cycles d'horloge définis par l'ordonnancement, et sur les unités matérielles retenues par l'allocation.

Ces trois étapes sont en réalité interdépendantes. Selon les objectifs d'optimisation et les contraintes du circuit, ces étapes sont couramment effectuées dans un ordre prédéfini. Effectuer l'ordonnancement en premier convient à un objectif de minimisation d'unités matérielles, sous une contrainte de temps d'exécution maximal. Effectuer l'allocation en premier convient pour l'optimisation du temps d'exécution du circuit, sous une contrainte de quantité d'unités matérielles. L'optimisation multi-critères implique généralement une exploration de l'espace de conception pouvant être très complexe et très longue.

Ces opérations sont effectuées sur une représentation interne du circuit, spécifique au logiciel de HLS. Aux débuts de la HLS, alors que la description d'entrée était composée seulement d'une succession d'instructions à exécuter, la représentation interne était un graphe de flot de données (DFG). Le flot de contrôle a été introduit par la suite pour gérer les sauts conditionnels, ce qui introduisit les représentations CDFG (DFG avec Contrôle). Les noeuds du graphe sont des DFG, également appelés *blocs de base* dans ce contexte. De nos jours, les logiciels de HLS utilisent une version étendue des CDFG, permettant de modéliser des structures hiérarchiques. Ces graphes sont appelés HCDFG (CDFG avec Hiérarchie [BGPB06, MDGP05]) ou HTG (Graphe de Tâches Hiérarchique [GDGN03]). Ils identifient la structure à haut niveau de la description d'entrée, notamment le corps des boucles, les branches des sauts conditionnels, le corps des fonctions, etc. Ces types de représentation révèlent des opportunités de parallélisation à plus haut niveau.

Basé sur cette représentation interne, le logiciel de HLS effectue les opérations traditionnelles d'allocation, d'ordonnancement et d'assignation. Il peut même transformer certaines parties de l'algorithme donné à synthétiser, de façon à les rendre plus appropriées à une exécution sur un FPGA. Les instructions à exécuter aussi bien que le flot de contrôle peuvent ainsi être remaniées par le logiciel. Les possibilités de transformation étant nombreuses, générer un circuit performant à partir d'une description de haut niveau et sous contraintes de ressources est une tâche complexe d'exploration de l'espace des solutions.

Comme présenté dans [CGMT09], les logiciels de HLS courants effectuent ces tâches internes suivant la méthodologie illustrée en Figure 3.1. Tout d'abord, la description d'entrée est compilée et convertie en une description interne, propre au logiciel. Ensuite, des transformations peuvent être appliquées selon des directives précises de l'utilisateur. Le logiciel aide l'utilisateur dans le processus d'exploration en effectuant les tâches de transformation spécifiées et en évaluant la qualité des circuits résultants (surface, fréquence, parfois latence). Une bibliothèque interne de composants, caractérisée pour la technologie ciblée, est exploitée pour produire ces évaluations. Le circuit est finalement généré dans un langage RTL standard (VHDL ou Verilog).

Il existe de nombreux logiciels capables d'effectuer ces transformations. Les logiciels Catapult C, PICO, Forte's Cynthesizer, AutoESL (maintenant Vivado HLS), CyberWorkBench,

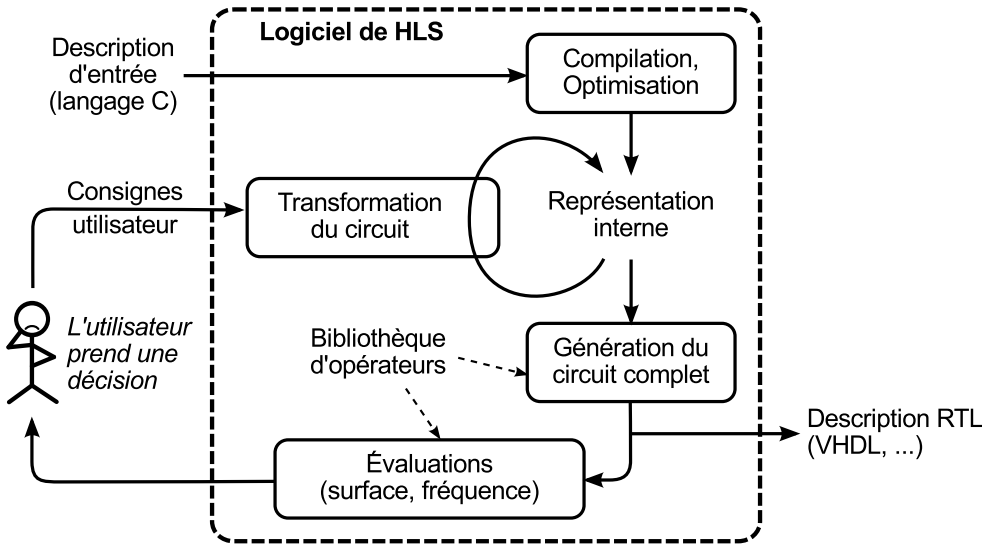


FIGURE 3.1 – Flot de génération des logiciels de HLS actuels

BlueSpec, GAUT et UGH sont présentés dans [CM08]. GAUT et UGH sont des projets académiques et *open source*. Parmi les autres projets académiques et/ou *open source*, on peut citer SPARK [GDGN03], ROCCC [VPNH10], Trident Compiler [TGP07] et LegUp [CCA+11]. Pour tous ces logiciels, le processus d'exploration des solutions suit la méthodologie dirigée par l'utilisateur décrite en Figure 3.1.

La suite de ce chapitre présente les diverses techniques employées en HLS pour la génération des circuits, dans le respect des contraintes matérielles, et avec les objectifs de rapidité des circuits et de rapidité de génération.

Les sections 3.1.1 et 3.1.2 rappellent les techniques courantes d'ordonnancement, d'allocation et d'assignation. Les principales transformations possibles sur les circuits sont décrites en section 3.1.3. La section 3.1.4 présente les moyens d'identification des sections critiques pour la latence des circuits. Les problématiques liées aux contraintes matérielles sont décrites en section 3.2. Les flots de génération présentant des capacités d'exploration autonome des solutions sont présentés en section 3.3.

3.1.1 Ordonnancement

L'ordonnancement est la technique principale révélant les différences fondamentales entre un flot de conception en HLS, et le flot de conception traditionnel (au niveau RTL), où le circuit est construit de façon structurelle. En HLS, la description d'entrée est algorithmique et, en particulier, ne contient aucune dépendance ni référence au temps d'exécution.

C'est l'opération d'ordonnancement qui introduit la notion de temps d'exécution. Elle se réfère implicitement à un circuit numérique cadencé par un signal d'horloge. À chaque instruction de la description d'entrée, elle associe un état particulier (au sens de l'automate d'états) du circuit pour son exécution. L'ordre des instructions est éventuellement modifié, afin de paralléliser les calculs. Cette opération est effectuée dans le respect des dépendances de données, qui imposent des relations de précédence entre certaines instructions.

L'opération d'ordonnancement est effectuée sur chaque bloc de base des représentations internes de type CDFG ou HDCFG. Les objectifs courants sont :

— l'optimisation de l'allocation des opérateurs, sous un nombre de cycles d'horloges maximal imposé pour l'exécution (on parle d'ordonnancement sous contrainte de temps),
— et l'optimisation du temps d'exécution du bloc de base, l'allocation des opérateurs étant imposée (on parle d'ordonnancement sous contrainte de ressources).

De nombreux algorithmes d'ordonnancement ont été étudiés. Historiquement, comme expliqué dans [HLH91], les premiers algorithmes d'ordonnancement utilisés furent ASAP (pour *As Soon As Possible*) et ALAP (pour *As Late As Possible*). Ils sont très simples et leur résultat est optimal en nombre de cycles d'horloge vis-à-vis des dépendances de données. Cependant, ils ne permettent pas de gérer des contraintes en ressources, et les circuits résultants ont souvent une taille excessive.

L'ordonnancement par liste [WGK08] permet de suivre une allocation déterminée, donnée sous la forme du nombre d'opérateurs de chaque type disponible (additionneur, multiplieur, etc). C'est une extension de ASAP, qui consiste à reporter l'exécution de certaines instructions lorsque le nombre d'opérateurs disponibles est insuffisant. La sélection des instructions à reporter peut être effectuée selon une priorité extraite des dépendances de données, noté PBLS (pour *priority-based list scheduling*), ou selon la criticité des opérateurs disponibles, noté FDLS (pour *force-directed list scheduling*) [PK89, TGP07]. Des variantes opérant sous contrainte de temps peuvent également être employées, par exemple TCLS (pour *time-constrained list scheduling*) [BGPB06]. D'autres techniques exploitant la priorité des opérations due à l'utilisation des registres ont également été proposées [BMZ11]. Ces techniques d'ordonnancement construisent le résultat final de façon gloutonne. Elles sont relativement rapides mais conduisent rarement à l'optimalité vis-à-vis de l'allocation donnée.

Les approches ILP (pour *Integer Linear Programming*) [Sil94], et CLP (pour *Constraint Logic Programming*) [Kuc98], voire leur combinaison [AAM06], sont mieux adaptées à la recherche de solutions optimales. La recherche exhaustive par l'algorithme B&B (pour *Branch And Bound*) [CDF06] a également été proposée. Ce sont des processus généralement complexes qui ne sont considérés que pour des DFG de taille réduite.

Des approches par raffinement itératif ont été proposées. Elles visent à obtenir des résultats meilleurs que les techniques gloutonnes, tout en présentant une complexité algorithmique moins élevée que la recherche exhaustive. On citera notamment le recuit simulé [Sil94] et les algorithmes génétiques [KK06]. Ces techniques font intervenir une certaine part d'aléa dans la construction des solutions. Celles-ci pouvant alors violer certaines dépendances de données ou contraintes d'unités matérielles, une rectification par ré-ordonnancement avec un algorithme fiable peut être effectuée (par exemple PBLS dans [KK06]).

3.1.2 Allocation et assignation

Dans le contexte de la synthèse sous contraintes de ressources, l'opération d'ordonnancement est le plus souvent effectuée pour une allocation des opérateurs donnée. L'allocation revêt donc une grande importance pour la génération de circuits rapides. La parallélisation des calculs implique notamment que les opérateurs sont disponibles en nombre suffisant pour exécuter plusieurs opérations simultanément.

La plupart des logiciels de HLS permettent à l'utilisateur de limiter manuellement le nombre maximal d'opérateurs de chaque type à utiliser. Cette contrainte est utilisée directement par l'algorithme d'ordonnancement. Cependant, certaines ressources matérielles restent souvent sans contrôle, en raison de leur caractère implicite dans la description d'entrée. C'est le cas pour les multiplexeurs, la machine à états (FSM - *Finite State Machine*) ainsi que pour la plupart des opérateurs logiques.

Le contrôle de l'allocation se fait également au niveau des mémoires [TGP07], car un banc de mémoire est une ressource partagée entre les instructions à exécuter. Le contrôle du nombre de ports d'accès à ces mémoires permet d'agir sur les possibilités de parallélisation, et donc sur la rapidité du circuit généré.

Les interfaces de communication jouent également un rôle important. L'impact de la profondeur des canaux FIFO a été étudié dans [LLZ+12]. L'insertion de mémoires tampon ping-pong permet également de paralléliser la communication et le calcul dans le circuit.

Les techniques permettant d'économiser des ressources matérielles contribuent également à l'accélération des circuits, en autorisant l'utilisation d'opérateurs supplémentaires. On citera en particulier l'élimination de sous-expressions communes [GRS+02], qui permet la réutilisation des chemins de données communs à plusieurs instructions. La taille de chaque opérateur peut également être optimisée pour la précision exacte des opérations à exécuter [LLCHM10]. Cela est effectué par exemple dans le logiciel UGH [AP08].

3.1.3 Transformation des circuits

En jouant sur les possibilités d'allocation, d'ordonnancement et d'assignation, il est possible d'optimiser le parallélisme dans les circuits générés, améliorant ainsi leur rapidité. Mais avec ces seules techniques, la forme et la rapidité des circuits générés sont fortement dépendantes de la forme de la description d'entrée.

Il est possible de pousser beaucoup plus loin la rapidité des circuits générés, en transformant cette description. Les transformations possibles peuvent cibler les instructions à exécuter et/ou le flot de contrôle de l'application. Dans certains cas, ces transformations peuvent être appliquées automatiquement si le gain est suffisamment sûr pour le logiciel de HLS.

Transformation des instructions

Les transformations des instructions des blocs de base permettent d'améliorer les possibilités d'ordonnancement.

En particulier, les techniques de propagation de constantes et d'assignations simples [NP91] permettent à la fois d'éliminer des dépendances de données et de réduire le nombre d'instructions à ordonnancer. Le remplacement scalaire [SHD02, DHP+03] élimine les accès redondants aux bancs de mémoire, en conservant une copie en registre des valeurs les plus fréquemment utilisées.

Le renommage de registres permet d'optimiser les dépendances de données sur l'utilisation des registres. Cette technique est appliquée dans LegUp [CCA+11], qui exploite les capacités du compilateur LLVM.

L'insertion de registres temporaires permet de mieux partager les opérateurs, et dans certains cas de pipeliner l'exécution des instructions. À l'inverse, le chaînage de plusieurs opérateurs dans un même cycle d'horloge permet de réduire le nombre d'instructions à exécuter, ce qui réduit potentiellement la latence. Cette opération n'a d'intérêt que si la latence combinatoire résultante est toujours compatible avec la période d'horloge ciblée.

Le déplacement d'instructions, notamment hors des corps des boucles et de certains branchements conditionnels, permet de diminuer le nombre d'instructions dans les sections les plus critiques pour le temps d'exécution [GDGN03].

Transformation du flot de contrôle

Le flot de contrôle des représentations CDFG et HCDFG est constitué des règles de branchement régissant l'enchaînement de l'exécution des blocs de base. Il ordonne et hiérarchise les blocs de base à exécuter. Il peut être transformé afin d'extraire toujours plus de parallélisme.

Beaucoup de techniques héritées de la compilation ont été proposées. Le déroulement de boucle [SHD02, CM08, SW12, Inc13] permet de réduire le temps d'exécution d'une boucle. Deux formes existent : la forme parallèle et la forme séquentielle. Le déroulement parallèle consiste à dupliquer les instructions du corps de boucle, ce qui permet d'exécuter simultanément plusieurs itérations de la boucle d'origine. Le temps de génération est faible, mais le coût en matériel est élevé, car le nombre d'opérateurs nécessaires est multiplié par le facteur de déroulement. Dans la version séquentielle, les duplicata du corps de la boucle sont ajoutés les uns à la suite des autres dans le graphe de la représentation interne. Cela expose des opportunités de parallélisation pour l'ordonnancement, et ne nécessite pas d'opérateurs supplémentaires.

Dans [CM08], la génération de pipeline est décrite pour les logiciels Catapult, PICO et Cynthesizer. Cette opération implique d'employer une technique d'ordonnancement spécialement conçue, par exemple *modulo scheduling* dans Trident Compiler [TGP07]. La fusion de boucles est appliquée par Catapult [Bol08] et ROCCC [Inc13]. Le pavage permet d'optimiser la localité des données et de réduire les accès à la mémoire externe. En contexte multi-boucles, le pavage peut être appliqué lors d'un déroulement ou d'une fusion de boucles [DHP+03]. Les transformations polyédriques [ZLL+13, Ple10, DRQR08] peuvent être employées pour les applications plus complexes présentant notamment des boucles imbriquées.

Un cas particulier du flot de contrôle est l'exécution conditionnée d'un bloc de base. En langage C, elle est créée notamment à partir des mots-clés *if* et *switch*. Cette structure crée une barrière à l'exécution entre les instructions avant, après et dans les blocs de base conditionnés. Certaines transformations permettent d'absorber ces branchements dans les instructions conditionnées. Les blocs de base résultants peuvent alors être mieux ordonnancés. Dans [GSK+01] et [Inc13], une technique basée sur des multiplexeurs est employée. Pour le logiciel UGH, François Donnet décrit dans sa thèse [Don04] une technique étendue pouvant également utiliser l'entrée d'activation des registres.

La génération de pipeline peut également être effectuée au niveau de l'application entière. L'application est alors scindée en plusieurs blocs, à travers lesquels les données transitent de façon sérielle. Les blocs peuvent alors fonctionner en parallèle. Cela est décrit pour le logiciel Catapult [Bol08].

Différentes implémentations des fonctions sont possibles. Dans [SW12], ces possibilités sont l'*inlining*, la génération de composant combinatoire ou séquentiel, et le saut conditionnel.

3.1.4 Détection des sections critiques pour la latence

Il est utile de pouvoir identifier les points chauds des applications à synthétiser. Cela permet de construire une allocation des opérateurs favorable aux blocs de base exécutés le plus souvent (pour une exécution typique de l'application).

Dans ce but, la plupart des logiciels de HLS actuels sont capables de reconnaître le nombre d'itérations des boucles *for* aux limites connues. Ils traitent alors préférentiellement ces boucles et leur corps, par exemple par déroulement dans [SHD02].

Des solutions ont également été proposées pour les cas non connus à la compilation. En particulier dans [BGPB06], le logiciel de HLS est capable de reconnaître des annotations de l'utilisateur, indiquant des probabilités de branchement. De nombreux logiciels de HLS gèrent également une indication du nombre d'itérations moyen des boucles.

3.2 Gestion des contraintes matérielles

3.2.1 Contraintes matérielles imposées par l'utilisateur

Pour les logiciels Catapult, Vivado HLS [Xil13a], UGH [AP08] et SPARK [GDGN03], l'utilisateur contrôle l'allocation des circuits générés. En particulier, le type d'implantation des mémoires peut être sélectionné et le nombre maximal d'instances de certains types d'opérateurs (tels que les additionneurs ou les multiplieurs) peut être imposé. Ces contraintes permettent d'exercer un certain contrôle sur la taille des circuits générés. Sans celles-ci, le logiciel de HLS considère qu'il n'a pas de contrainte, et génère le circuit le plus rapide possible (ordonnancement ASAP/ALAP et allocation des opérateurs en conséquence).

Pour Catapult, un objectif en ressources globales (surface) peut être donné. Cependant, cette consigne ne prend pas en compte le caractère multi-dimensionnel des ressources des FPGA (ils contiennent des LUT, des registres, des blocs DSP et des blocs de mémoire RAM), et est souvent excédée par les circuits générés.

Pour UGH, spécifier l'allocation est obligatoire. Cette étape, certes contraignante, permet toutefois de régler finement la forme des connexions internes du circuit. Pour cela, l'utilisateur peut spécifier les connexions entre opérateurs qui sont autorisées. Outre le contrôle de la surface, cela permet également d'optimiser les temps de propagation, notamment dans les multiplexeurs.

3.2.2 Estimation des caractéristiques globales des circuits

Afin de générer un circuit sous des contraintes de ressources, le logiciel de génération doit être capable d'évaluer le coût en ressources du circuit considéré.

Dans [SHD02, LPC12], la conformité du circuit vis-à-vis des contraintes matérielles est vérifiée grâce à l'intégration, dans la boucle d'exploration des solutions, d'une étape de synthèse

logique effectuée par un logiciel externe. Cette technique est très fiable et permet de simplifier l'outil de HLS, mais elle est très lente.

Des techniques plus rapides ont été proposées. Elles consistent généralement à intégrer, dans l'outil de HLS, une modélisation de chaque type de composant (en ressources matérielles et latence combinatoire). Cette modélisation permet de calculer la taille et la latence de tous les composants.

Dans GAUT [CCB+08], chaque type de composant est synthétisé préalablement à toute opération de HLS, par les logiciels de synthèse logique respectifs de Xilinx et d'Altera. Une base de données des caractéristiques des composants est ainsi construite, ce qui permet à GAUT de connaître les caractéristiques de tous les composants utilisés.

Dans [KNRK02], une analyse préalable de nombreux types de DFG est effectuée en faisant varier la largeur des données et le nombre d'entrées. Une régression linéaire, quadratique ou bilinéaire sur la consommation en LUT est alors extraite, et cette formule est utilisée comme estimateur durant les processus de HLS. L'impact réel de la non-exhaustivité de cette analyse préalable est cependant inconnu.

Dans [DVSVC01], une modélisation des temps de propagation des signaux dans les interconnexions est proposée. Elle consiste à estimer la distance moyenne à parcourir et le nombre de noeuds de routage moyens à traverser, selon la surface du circuit généré. Dans le cadre d'un FPGA, cette surface est proportionnelle au nombre de cellules logiques configurables élémentaires utilisées.

3.2.3 Corrections afin de cibler une fréquence donnée

Afin d'évaluer correctement la fréquence atteignable par un circuit donné, il faut prendre en compte la latence combinatoire des multiplexeurs et de la logique de calcul des sorties de la FSM. Ces valeurs ne peuvent cependant être obtenues qu'une fois les opérations de synthèse logique, de placement et de routage effectuées. Il est donc possible que les suppositions durant la HLS sur la latence de certains chemins de données se trouvent être fausses à la fin de la génération. Des corrections doivent alors être effectuées sur le circuit afin de générer une solution réellement fonctionnelle sous la contrainte en fréquence imposée par l'utilisateur.

Cette opération de correction, souvent appelée *retiming*, est insérée dans le flot de génération de la façon illustrée en Figure 3.2. Le circuit obtenu par un flot de HLS passe par les étapes aval de synthèse logique, placement et routage. Un contrôle des temps de propagation est alors effectué afin de vérifier si les estimations effectuées lors de la HLS sont toujours valides. Si des violations de la période d'horloge sont détectées, la FSM du circuit est re-générée (avec parfois une plus grande partie du circuit) de façon à compenser ces violations. Le flot de génération en aval de la HLS est alors relancé sur le circuit corrigé. Il existe plusieurs techniques de correction.

Dans [WMPW03], le circuit généré est instrumenté avec des éléments facilitant les corrections ultérieures. Il est proposé de dupliquer tous les registres un nombre donné de fois, noté k, préalablement à la première synthèse logique. Pour chaque registre, ces duplicata sont connectés de façon à former un registre à décalage, permettant de conserver la valeur du registre d'origine pendant k cycles d'horloge. Après placement et éventuellement routage, les temps de propagation sont analysés. La machine à états est alors modifiée de façon à ce que les registres adéquats

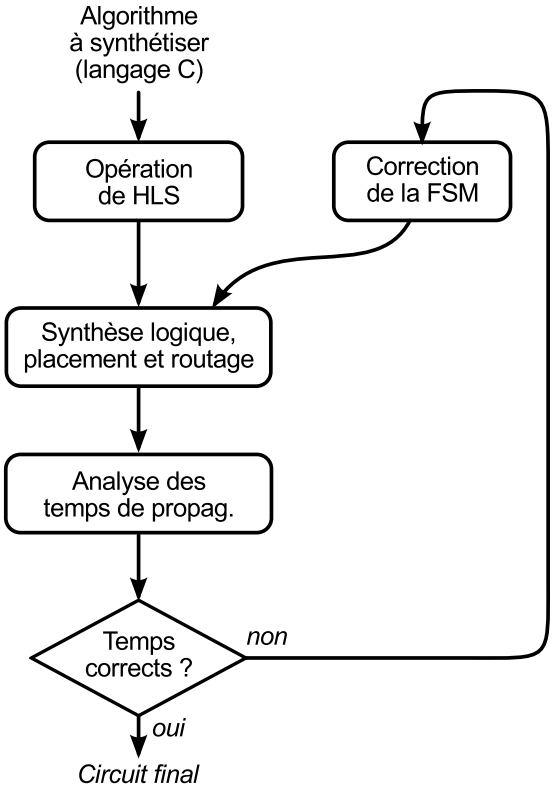


FIGURE 3.2 – Flot de génération actuel pour respecter une fréquence donnée

soient utilisés pour les instructions dont la latence combinatoire excède la période d'horloge. Cependant, dans le contexte visé, cette technique présente des défauts majeurs :

— Les éléments insérés dans le circuit générés afin de faciliter les corrections ultérieures consomment une grande quantité de ressources matérielles.
— Les corrections impactent les temps de propagation dans la machine à états. Afin de garantir que le circuit corrigé est toujours fonctionnel, de nouvelles vérifications et éventuellement corrections sont nécessaires. Ce processus itératif, en plus d'être long, pourrait ne pas jamais converger vers une solution satisfaisant toutes les contraintes.
— Le coût en ressources exact des corrections est difficilement prédictible au moment de la HLS. Afin d'effectuer rapidement une exploration des solutions au moment de la HLS (i.e. sans effectuer de synthèse logique ni placement et routage de chaque solution) seule une estimation de ce coût peut être prise en compte. Le respect de toutes les contraintes par le circuit final n'est donc pas garanti.

Le logiciel UGH [AP08] emploie une autre technique : le gel d'état. Cela consiste à générer une machine à états spécialement instrumentée de façon à pouvoir être "gelée" durant une certaine durée. De plus, chaque état de la FSM peut avoir une durée de gel spécifique. Cette durée est calculée grâce à une analyse des temps de propagation extraits après synthèse logique (voire après placement et routage, pour une meilleure précision). Les durées de gel d'état sont stockées dans une mémoire interne à la FSM. La correction est alors effectuée au moment de l'utilisation du circuit généré. Cette technique est relativement peu coûteuse en ressources, et son coût en ressources est prédictible au moment de la HLS. Mais le temps d'exécution réel du circuit n'est

pas connu au moment de la HLS.

En conclusion, il existe des techniques permettant de répondre à une contrainte en fréquence donnée. Dans le contexte présent, où le respect des contraintes matérielles prime sur le temps d'exécution du circuit obtenu depuis la HLS, la technique du gel d'état semble être une solution convenable.

3.2.4 Routabilité

Le tissu d'interconnexion interne du FPGA ciblé est une contrainte matérielle implicite. En effet, certains circuits peuvent nécessiter une quantité d'interconnexions supérieure à ce qu'il est possible d'implémenter dans le FPGA. La topologie des interconnexions est également à prendre en compte. Un inconvénient majeur dans la conception de circuits sur FPGA est que la conclusion finale sur la possibilité réelle de placer et connecter tous les opérateurs dans le FPGA ciblé (la routabilité) arrive très tard dans le flot de conception. Une conclusion négative obligerait le concepteur à tout recommencer.

Afin d'éviter cette situation, une pratique courante est de conserver une marge de sécurité sur les ressources matérielles et la fréquence d'horloge (par exemple, ne cibler que 80% des ressources réellement disponibles, et viser une fréquence majorée de 10%). Cela a pour effet de réduire la pression sur le logiciel de placement-routage, qui aura alors plus de liberté et de marge pour trouver une solution fonctionnelle.

Cependant, à notre connaissance, il n'existe actuellement pas de méthode fiable, autre que lancer le processus de placement-routage, qui permette de garantir la routabilité finale dans le cas général. Pour la majorité (si ce n'est la totalité) des flots de génération en HLS, cette limitation est avouée : si un circuit généré par HLS se trouve ne pas être routable pour le FPGA ciblé, l'utilisateur est censé imposer des contraintes plus strictes au logiciel de HLS, ou bien modifier la description d'entrée, puis relancer la génération. Non seulement cette méthode est très peu élégante, elle ne convient que très peu aux utilisateurs peu familiers avec les techniques de conception de circuits.

3.3 Exploration autonome des solutions

À ce jour, il n'existe pas de formalisme permettant de calculer le circuit optimal à un problème de HLS sous contraintes de ressources. La théorie faisant défaut, l'obtention du circuit final passe par la génération d'un certain nombre de solutions. Ces solutions sont construites et comparées par le logiciel de HLS, qui retient la meilleure.

Cependant, lorsque les possibilités structurelles sont nombreuses, l'espace des solutions peut être trop vaste pour une exploration exhaustive. Afin d'obtenir une solution finale en un temps raisonnable, des techniques permettant de limiter la taille de l'espace des solutions effectivement exploré sont mises en oeuvre.

Les environnements d'exploration impliquant de la synthèse de haut niveau ont été proposées à divers niveaux, allant jusqu'au niveau système. Afin de justifier les choix qui ont mené à proposer une nouvelle méthode, les techniques les plus appropriées au contexte visé sont présentées.

3.3.1 Techniques d'exploration au niveau système

Les techniques d'exploration au niveau système présentent des propriétés intéressantes. Elles visent généralement à décider quelles fonctionnalités du système considéré doivent être implémentées sous forme d'accélérateurs matériels. Les plateformes d'exécution ciblées sont donc hétérogènes (microprocesseurs et accélérateurs). Comme décrit dans [BPL01], cette hétérogénéité mène généralement à des environnements d'exploration de l'espace de conception très modulaires. Un logiciel spécialement optimisé pour l'exploration exploite des logiciels externes en tant qu'estimateurs spécialisés par type de cible matérielle. Les estimations peuvent être par exemple le temps d'exécution ou la consommation.

Un environnement d'exploration pour prototypage virtuel de systèmes sur puce (SoC) est proposé dans [GHH+12]. L'objectif des auteurs est de considérer le comportement fonctionnel, énergétique et temporel au niveau système, pour un partitionnement et une assignation particuliers sur une plateforme donnée. Ils proposent d'utiliser des modèles MARTE/UML sous Matlab/Symulink. Les estimations sont effectuées par des logiciels de HLS commerciaux. Cependant, les algorithmes d'exploration ne sont pas décrits et aucun résultat concret n'est fourni.

Des méthodologies très complexes peuvent être considérées avec des logiciels dédiés à l'exploration. Dans [JPT+10], un environnement très modulaire est proposé pour une exploration multi-dimensionnelle pour des SoC multi-processeurs (MP-SoC). Différents algorithmes peuvent être employés simultanément pour l'exploration des différentes dimensions de l'espace de conception, avec pour objectif de trouver des solutions meilleures qu'avec un unique algorithme d'exploration.

Les approches d'exploration au niveau système sont certainement intéressantes pour la modularité et la réutilisabilité des éléments logiciels mis en oeuvre. Leur spécialisation dans l'exploration peut en faire des approches très efficaces, au moins pour des plate-formes hétérogènes. Cependant, dans le contexte des présents travaux, ces propriétés ont un coût. L'usage de modèles de haut niveau génériques peut apporter des problèmes de précision et un surcoût en temps de communication inter-logiciels dû aux conversions entre modèles. Les résultats peuvent également présenter de sérieux problèmes de sous-optimalité à cause d'une faible exploitation des propriétés spécifiques aux cibles matérielles. Le niveau d'abstraction rend également difficile la génération des plateformes matérielles physiques.

Un niveau de généricité peu élevé peut être employé pour atteindre des objectifs de précision. Dans [CHD+12], un environnement d'exploration pour le partitionnement logiciel/matériel est proposé. Le logiciel d'exploration est étroitement couplé à un logiciel de HLS extérieur, GAUT [CCB+08], qui genère des accélérateurs matériels sous des contraintes en latence ou en débit de données. Pour un accélérateur matériel donné, la méthodologie d'exploration consiste en l'incrémentation itérative d'une contrainte en latence (en cycles d'horloge). Pour chaque valeur de cette contrainte, GAUT génère une solution. La consommation en ressources matérielles de cette solution est alors évaluée. Cependant, les contraintes en ressources ne sont gérées qu'indirectement, et seuls des circuits de type flot de données sont supportés.

Effectuer des simulations de performance pour des systèmes hétérogènes est également possible. La méthodologie proposée dans [dAMGBE11] facilite la simulation d'un système logiciel/matériel afin de décider si transformer un coeur de calcul donné en un accélérateur matériel

est nécessaire. Cependant, leur approche est focalisée sur la latence et ne garantit pas que l'application considérée est synthétisable dans un FPGA donné.

Les travaux présents sont focalisés sur l'exploration au niveau HLS (notée HLS-DSE), ce qui signifie qu'il est pris pour acquis que l'application considérée doit être implémentée en matériel. Dans le contexte présent, les inconvénients des approches d'exploration au niveau système (la précision est faible, la capacité à générer les plateformes finales est limitée et le surcoût en communication inter-logiciels peut être élevé) ne sont pas acceptables. C'est pourquoi dans la méthodologie de HLS proposée, l'exploration et l'estimation sont indissociables et embarqués dans le logiciel de HLS. Par ailleurs, un tel logiciel de HLS constituerait un bon estimateur pour FPGA, au sein d'un environnement d'exploration au niveau système.

3.3.2 Techniques d'exploration au niveau HLS

Dans [SHD02], le logiciel de HLS est capable de dérouler une boucle itérativement, jusqu'à atteindre une limite en surface. La description d'entrée doit être très structurée, et les bornes des boucles doivent être connues à la compilation. Le flot est limité à un unique type de transformation (déroulement de boucle). Chaque solution est également synthétisée par le logiciel de synthèse logique Monet, de Mentor Graphics. Ces opérations, fréquentes et très longues, font que cette approche est très lente.

Explorer l'espace des solutions grâce à des algorithmes génétiques (ou dérivés) a également été proposé. Dans [FLL+08, SS11, RBL11] les flots proposés permettent de synthétiser des blocs de base avec éventuellement du contrôle. Les transformations sur les structures de contrôle (branchements conditionnels et boucles) ne sont cependant pas gérées. Dans [STW09], le logiciel d'exploration est basé sur un environnement de HLS, CyberWorkBench, qui effectue la génération des solutions considérées. Le flot permet de gérer l'allocation et certaines transformations de haut niveau. Cependant, l'utilisateur est supposé fournir une configuration initiale *intuitive* des transformations possibles, ce qui exige une certaine expertise. De plus, en règle générale, ces techniques d'explorations génèrent en interne un grand nombre de solutions (la *population*), ceci à de nombreuse reprises (les *générations*). Ce processus peut donc être très long. Le temps d'exploration nécessaire à l'obtention d'une solution satisfaisante n'est également pas prévisible, en particulier pour des circuits complexes.

Dans [BGPB06], le logiciel de HLS peut évaluer de façon fiable le temps d'exécution de toutes les sections de l'algorithme donné à synthétiser. Des annotations permettent de transmettre au logiciel les probabilités de branchement et le nombre d'itérations moyen des boucles. Propagées récursivement dans la représentation interne hiérarchique, celles-ci révèlent les sections les plus critiques pour le temps d'exécution, ce qui oriente l'exploration vers les solutions les plus pertinentes. La technique employée consiste à générer toutes les implémentations possibles des blocs de base, puis à les combiner en remontant les niveaux hiérarchiques de la représentation interne. Cependant, cette approche peut présenter un passage à l'échelle limité, principalement à cause du grand nombre d'implémentations des blocs de base manipulés. Les auteurs soulignent également que la simplicité des heuristiques de combinaison peut impacter la qualité des solutions.

Dans [DHP+03], le système DEFACTO est employé pour l'exploration automatisée des solutions pour des applications de traitement du signal. Les circuits générés sont pipelinés, et le

contrôle dépendant des données n'est pas supporté. L'exploration des solutions est un processus itératif qui applique de façon successive des transformations à un circuit initial. Les transformations considérées ciblent les boucles (déroulement, pavage) et les accès à la mémoire (remplacement scalaire et optimisation du contrôleur). Le but est d'arriver à équilibrer le temps d'accès à la mémoire et le temps de calcul. Cependant, les techniques de choix des transformations employées ainsi que les temps de génération restent inconnus.

3.3.3 Techniques exploitant OpenCL

Les approches exploitant OpenCL comme description d'entrée, relativement récentes, sont à la frontière entre le niveau système et le niveau HLS. Altera propose un flot de génération en HLS acceptant en entrée une description au standard OpenCL [CAD+12]. Xilinx a également annoncé l'arrivée prochaine d'un logiciel de développement à partir d'OpenCL [Xil13b].

Originellement pensée pour cibler les processeurs graphiques, une description OpenCL possède une structure particulière. Le noyau de programme destiné à être accéléré est censé cibler un microprocesseur à jeu d'instructions SIMD, et très peu de contrôle dépendant des données est toléré (voire pas du tout). De façon générale, avec le standard OpenCL, les noyaux de programme à accélérer sont identifiés et extraits du reste de l'application par l'utilisateur. Le noyau OpenCL est supposé être exécuté sur un accélérateur dédié pourvu d'une mémoire locale relativement massive. La compilation, le déploiement et l'exécution de ces accélérateurs matériels, ainsi que les communications avec le reste de l'application, sont contrôlés par un pilote logiciel, potentiellement très complexe,

Dans le processus de construction d'un accélérateur matériel sur FPGA à partir d'une description OpenCL, la rapidité de la génération est favorisée par le fait que les possibilités structurelles sont intrinsèquement peu nombreuses. Tout d'abord, le noyau à accélérer est synthétisé. C'est une tâche d'allocation / ordonnancement / assignation courante, et ce problème est de nos jours relativement bien maîtrisé. Ce noyau synthétisé est ensuite instancié autant de fois que le permettent les ressources matérielles disponibles dans le FPGA. Conceptuellement, c'est donc une opération relativement simple.

Toutefois, le modèle d'utilisation de la mémoire impose d'intégrer également un contrôleur d'accès à la mémoire externe associée au FPGA. Ce contrôleur est en général un composant massif et complexe, mais dont la structure est prédéfinie. En particulier, sa surface peut être connue au moment de la HLS. Il a donc à priori très peu d'impact sur la complexité du processus d'exploration des solutions.

Cette solution est donc adaptée pour une exploration des solutions rapide, mais au prix de conditions fortes sur la structure des programmes à accélérer et sur les machines accélératrices utilisables pour l'exécution.

3.4 Synthèse sur les approches existantes

Les techniques de HLS existantes qui incluent une exploration des solutions présentent des propriétés extrêmement diverses. Dans le contexte visé, un certain nombre de propriétés sont

attendues pour le logiciel de HLS (conformité vis-à-vis des contrainte, rapidité, autonomie, optimisation, polyvalence). En général, des solutions appropriées existent à travers les flots de génération existants.

Pour respecter une contrainte en ressources, chaque solution explorée peut être évaluée par le logiciel de HLS grâce à une bibliothèque de composants caractérisée. Une correction des circuits générés peut être nécessaire afin de garantir la fréquence de fonctionnement. Les circuits générés peuvent être instrumentés spécialement dans ce but. Mais cela ne résout toutefois pas le problème de la routabilité, pour lequel aucune solution fiable n'existe.

Pour la rapidité de l'exploration des solutions, trois points sont primordiaux : autant que possible, le sous-ensemble de l'espace des solutions à explorer doit être réduit, les itérations avec les logiciels en aval du flot de conception doivent être évitées, et des algorithmes d'ordonnancement de faible complexité algorithmiques doivent être employés.

Pour l'autonomie, le logiciel de HLS doit effectuer seul la sélection des solutions à explorer, et être capable d'évaluer leurs caractéristiques globales (ressources, latence, fréquence).

Pour l'optimisation, les solutions explorées doivent être aussi pertinentes que possible. Pour les applications contenant du contrôle dépendant des données, il est préférable de transmettre au logiciel de HLS des indications sur l'emplacement des points chauds pour le temps d'exécution.

Pour la polyvalence, la gestion du contrôle dépendant des données est un point important. Les applications pour lesquelles un accélérateur matériel est souhaité sont de plus en plus complexes.

Aucune approche existante ne répond simultanément aux exigences du contexte visé. Pour cette raison, une nouvelle méthodologie d'exploration des solutions en HLS est nécessaire. Cette contribution est présentée dans le chapitre suivant.

Chapitre 4

Proposition d'une nouvelle méthode

Ce chapitre présente, de façon conceptuelle, la méthodologie de HLS proposée. Les principaux objectifs sont identifiés et un algorithme d'exploration des solutions est proposé. Les principales propriétés de cet algorithme sont discutées, notamment sa complexité algorithmique, sa capacité à générer des circuits optimisés vis-à-vis des objectifs et contraintes, ainsi que ses principaux points faibles potentiels. Finalement, des variantes permettant d'améliorer l'optimalité des circuits ou d'accélérer l'exploration sont proposées.

Sommaire

4.1 Le flot de synthèse proposé

Dans le contexte visé, les utilisateurs sont potentiellement peu experts en HLS. Leur but principal n'est pas de construire un circuit extrêmement optimisé (ce qui reste une tâche longue), mais plutôt d'obtenir une accélération significative par rapport à une solution purement logicielle. La recherche d'une solution encore plus rapide est alors une étape optionnelle, considérée uniquement si les contextes de la conception et de l'utilisation de l'application le permettent.

Cependant, avec les solutions de HLS courantes où le flot de génération est piloté par l'utilisateur, cet objectif est difficile à atteindre, pour deux raisons principales.

Premièrement, avec l'accroissement de la puissance des FPGA, les accélérateurs matériels considérés se complexifient. Les interactions entre les choix d'implémentation deviennent de plus en plus difficiles à appréhender pour les concepteurs. De plus, comme illustré précédemment en Figure 2.2 (page 19), les données couramment exposées à l'utilisateur (latence, fréquence et utilisation en ressources) sont peu nombreuses et sont des métriques agrégats. Cela ne guide que très peu les concepteurs dans le pilotage de l'exploration des solutions. Ainsi, la génération d'un accélérateur matériel complexe par HLS implique souvent une certaine part de tâtonnement.

Deuxièmement, les incertitudes dues au contrôle dépendant des données présent dans la description d'entrée empêchent le logiciel de HLS d'évaluer la latence du circuit. Ainsi, avec les flots de HLS courants, il est nécessaire d'effectuer une simulation de chaque solution explorée afin de connaître sa latence en situation réelle. Le temps total avant obtention de la solution finale peut donc être très long.

Or, en réalité, les logiciels de HLS manipulent en interne une grande quantité de données sur les caractéristiques du circuit, ses dépendances de données et les possibilités de transformation. Ces données sont notamment beaucoup plus nombreuses, plus précises et plus complètes que ce qui est couramment exposé à l'utilisateur.

La méthodologie de HLS présentée dans ce chapitre propose une réponse aux problèmes mentionnés ci-dessus, en cherchant à exploiter au maximum les données internes pour effectuer une exploration des solutions pertinentes. Les objectifs techniques sont les suivants :

— effectuer une exploration des solutions pertinente sans interaction avec l'utilisateur,
— ne laisser aucune place au tâtonnement dans l'exploration des solutions,
— permettre à l'utilisateur de spécifier le comportement de l'application au niveau du contrôle dépendant des données,
— éliminer le besoin de simulation de chaque solution explorée,
— intégrer le processus dirigeant l'exploration des solutions au sein du logiciel de HLS afin de permettre l'accès illimité aux données internes du logiciel,
— et présenter une faible complexité algorithmique afin de pouvoir traiter des circuits complexes.

4.1.1 Principe général

Le principe de la méthodologie de HLS proposée est le suivant : en partant d'une solution initiale de faible surface (avec peu de parallélisme, donc relativement lente), des transformations sont appliquées successivement de façon à accélérer le circuit. Comme illustré en Figure 4.1, en

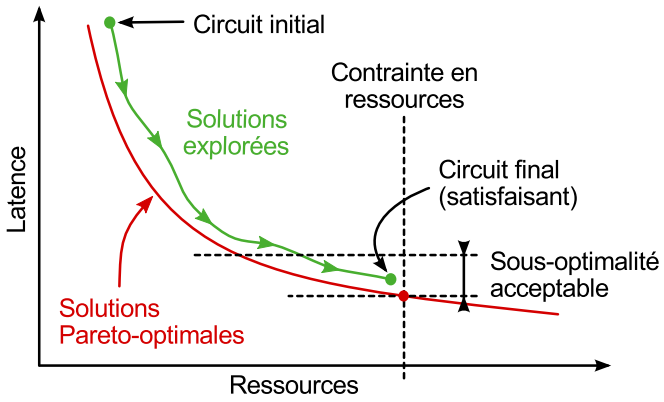


FIGURE 4.1 – Évolution du circuit durant l'exploration

augmentant progressivement le taux de parallélisme dans le circuit, ces transformations tendent également à augmenter la surface du circuit. Le respect de la contrainte de ressources est alors effectué en stoppant ce processus de transformation lorsque la contrainte est atteinte. Le circuit final est alors le dernier considéré dont l'utilisation en ressources respecte la contrainte. Le processus d'exploration des solutions proposé est donc basé sur un algorithme glouton.

La méthodologie proposée est détaillée plus formellement dans l'Algorithme 1. L'étape de synthèse initiale produit de manière rapide une première solution, de faible surface (détails en section 4.2.1, page 55). La conformité de cette solution avec les contraintes en ressources est aussitôt contrôlée. En cas de dépassement, le processus de HLS est avorté : l'opération de synthèse est considérée impossible, en tous cas pour le logiciel employé et sous les contraintes imposées. Ainsi, pour le reste du processus d'exploration des solutions, l'existence d'au moins une solution conforme est garantie.

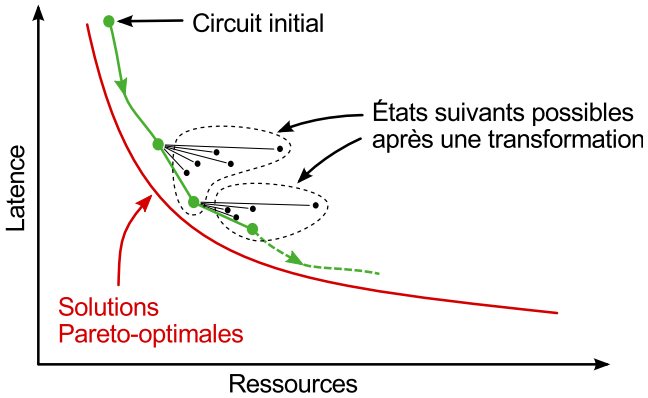


FIGURE 4.2 – Exploration de proche en proche

Le processus d'exploration des solutions est itératif. À chaque itération, une transformation est appliquée. La sélection de cette transformation, parmi celles possibles, est basée sur des pondérations produites par des estimateurs. Ces estimateurs sont des méthodes effectuant une prédiction de l'impact d'une transformation donnée sur le circuit, notamment au niveau du coût en ressources et du gain en latence (détails en section 4.2.2). Ce sont ces estimateurs qui exploitent les données internes mentionnées précédemment, afin de produire une pondération la plus précise

Algorithme 1 Algorithme d'exploration des solutions

Entrées : Description en langage C et contraintes en ressources
Sorties : Circuit en langage RTL ou netlist

Compiler le code C vers la représentation interne
Effectuer une synthèse initiale
si les contraintes en ressources sont dépassées **alors**
    Informer l'utilisateur : la synthèse est impossible pour les contraintes données
    Stopper le processus de HLS
fin si
répéter *// BOUCLE-EXPLORE*
    $\Omega_{tr} \leftarrow \emptyset$
    pour chaque transformation possible T du circuit **faire**
        Estimer le coût en surface et le gain en latence de T
        si les estimations respectent les contraintes **alors**
            ajouter T à Ω_{tr}
        fin si
    fin pour
    si $\Omega_{tr} = \emptyset$ **alors**
        SORTIR de la boucle *BOUCLE-EXPLORE // L'exploration est terminée*
    fin si
    $T \leftarrow$ la meilleure transformation de Ω_{tr}
    Appliquer T dans une copie de la représentation interne actuelle
    Effectuer l'ordonnancement et l'affectation de cette copie
    si le circuit copié respecte les contraintes **alors**
        Appliquer T à la représentation interne actuelle
    sinon
        Marquer T comme étant impossible
    fin si
fin répéter *// BOUCLE-EXPLORE*
Effectuer l'ordonnancement et l'affectation du circuit actuel
Générer le circuit actuel en tant que RTL ou netlist

possible. Comme illustré en Figure 4.2, la transformation sélectionnée est celle présentant le gain en latence le plus élevé, tout en ayant un coût en ressources limité (détails en section 4.2.3). Cela tend à diriger l'exploration aussi près que possible des solutions de Pareto, même si la position de celles-ci reste inconnue. Afin d'éviter les dérives potentielles dues à l'accumulation des biais des estimations, une évaluation précise du circuit est effectuée à la fin de chaque itération. Cette étape est détaillée en section 4.2.4. Finalement, le processus d'exploration est stoppé lorsque plus aucune transformation n'est possible, ou bien lorsque toutes les transformations possibles ont un coût en ressources estimé excessif.

La méthodologie proposée a des propriétés intéressantes pour le contexte visé (synthèse automatique et rapide sous contraintes de ressources). La complexité algorithmique de l'exploration des solutions est analysée en section 4.1.2. La prise en compte d'un scénario d'exécution spécifié par l'utilisateur est décrite en section 4.1.3. La pertinence de la méthodologie proposée pour des types de contraintes multiples est discutée en section 4.1.5. Une technique de correction rapide visant à garantir le fonctionnement à une fréquence donnée, même après placement et routage, est proposée en section 4.1.4. L'intégration de la méthodologie proposée au sein de logiciels de HLS existants est présentée en section 4.1.6. Les nouveaux éléments logiciels sont décrits en section 4.2. Le niveau d'optimalité des circuits générés est discuté en section 4.3. Des optimisations possibles de la méthodologie proposée sont proposées en section 4.4.

4.1.2 Complexité

Le temps d'exécution de la méthodologie proposée peut varier notablement selon le contexte d'utilisation. Les facteurs influents ont plusieurs origines :

— le circuit à synthétiser : le nombre et le type des transformations possibles à chaque itération,
— les contraintes de ressources (cela limite le nombre d'itérations),
— l'implémentation de la méthodologie dans le logiciel de HLS hôte (précision des estimateurs),
— le logiciel de HLS hôte lui-même : les algorithmes d'ordonnancement et d'affectation employés, les transformations qu'il est capable d'appliquer, le niveau de simplification et d'optimisation des circuits générés, etc.

L'analyse de la complexité algorithmique globale du flot de HLS proposé se prête donc malheureusement peu à des considérations formelles. Certaines analyses sur le comportement de la méthodologie proposée restent possibles. Certes assez qualitatives, elles sont néanmoins essentielles :

— L'algorithme d'exploration des solutions est glouton. Cela implique que le processus d'exploration ne revient jamais à une itération précédente, au cas où des choix différents auraient pu mener à de meilleures solutions. De plus, les transformations appliquées ne sont jamais *défaites*. En particulier, aucun type de transformation n'est censé pouvoir défaire ce qu'une transformation précédente a effectué. Chaque solution n'est donc envisagée qu'une seule fois.
— Le nombre de transformations qu'il est possible d'appliquer à un circuit donné est fini. En effet, le nombre d'opérateurs utilisables en parallèle est fini (lié à l'ordonnancement des instructions des blocs de base), le nombre de composants de stockage également (lié

au nombre de variables et de tableaux déclarés dans la description d'entrée), de même que le nombre de boucles (pour déroulement) et le nombre de branchements conditionnels (pour câblage). Ainsi, de façon très approximative, le nombre de transformations qu'il est possible d'appliquer à un circuit donné peut être considéré comme évoluant linéairement en fonction de la taille de la description d'entrée (en lignes de programme).

— L'algorithme d'exploration étant glouton, et le nombre de transformations qu'il est possible d'appliquer étant fini, le processus d'exploration des solutions est assuré d'avoir une fin.

— Les transformations tendent à accroître la taille du circuit. Les cas où une réduction de l'utilisation en ressources est obtenue sont normalement exceptionnels. Ainsi, sous une contrainte en ressources imposée par l'utilisateur, la solution finale est obtenue plus vite que sans contrainte.

Ces propriétés garantissent que, pour une description d'entrée donnée, le processus d'exploration aura une fin. Il a également une complexité algorithmique linéaire en fonction de la "taille" du programme à synthétiser. Cette propriété est cruciale, car elle permet le passage à l'échelle pour générer des circuits complexes relativement rapidement.

Un autre facteur essentiel concerne la puissance de la machine nécessaire à l'exécution de la méthodologie proposée, autant pour la puissance de calcul (processeur) que pour la quantité de mémoire nécessaire.

La complexité algorithmique est relativement faible et les étapes les plus lourdes en calcul sont en principe similaires à ce qui est effectué par les logiciels de HLS actuels (ordonnancement, affectation, simplifications diverses, etc). Certains calculs peuvent également être parallélisés, principalement au niveau de la pondération des transformations et de l'ordonnancement des blocs de base.

Les besoins en mémoire sont également très modérés. Durant l'exploration, le pic de consommation en mémoire correspond à l'évaluation précise des caractéristiques d'une solution (détails en section 4.2.4). La représentation interne est alors dupliquée, et une modélisation au niveau *netlist* est construite. Les besoins en mémoire correspondent donc à deux fois la taille de la représentation interne, plus la taille de la modélisation au niveau *netlist*. Par rapport à un logiciel de HLS actuel, la consommation en mémoire augmente de l'équivalent de la taille de la représentation interne. Cette augmentation est donc inférieure à 100 %.

Cette valeur peut être considérée comme modérée comparée à d'autres algorithmes employés dans la littérature, en particulier les algorithmes génétiques, qui impliquent de manipuler simultanément un grand nombre de solutions (la taille de la population).

4.1.3 Scénario d'exécution

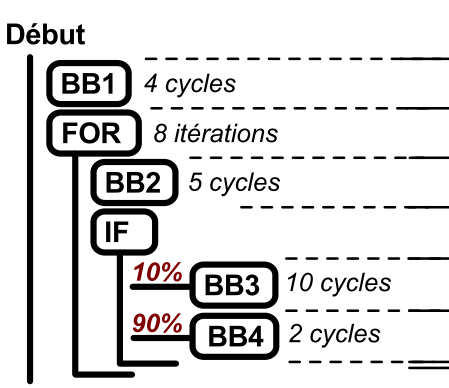
Comme mentionné dans la problématique en section 2.2.1 (page 22), indiquer un scénario d'exécution au logiciel de HLS peut être crucial pour générer des circuits rapides.

Les travaux présentés dans [BGPB06] montrent qu'une représentation interne hiérarchique permet d'estimer facilement le temps d'exécution d'un circuit. En connaissant la probabilité de branchement de chaque noeud conditionnel et le nombre moyen d'itérations de chaque boucle, il est possible de calculer le temps d'exécution moyen de toutes les sections de la représentation.

Cela permet de focaliser l'exploration des solutions sur les sections qui contribuent le plus au temps d'exécution global du circuit.

Cette méthode, basée sur des annotations de la description d'entrée, est celle retenue dans la méthodologie proposée pour transmettre le scénario d'exécution considéré au logiciel de HLS. Certaines de ces indications peuvent être inférées automatiquement depuis la représentation interne (en particulier le nombre d'itérations des boucles de type *for* aux limites statiques). Les indications relatives au contrôle dépendant des données peuvent être fournies par des annotations. Celles-ci restent facultatives : par défaut, les différentes branches des noeuds de contrôle sont considérées équiprobables, et les boucles aux limites dépendantes des données sont considérées comme itérant une seule fois.

Il est donc possible de générer un circuit sans ces annotations. Cependant, l'évaluation de la latence du circuit par le logiciel de HLS ne correspondra pas à la réalité. Cela n'est également pas favorable à la pertinence de l'exploration des solutions et donc à l'optimalité de la solution finale.



	Temps d'exécution (cycles d'horloge)			
	Pire cas	Meilleur cas	Par défaut	D'après les annotations
BB1	4	4	4	4
FOR	120	56	88	62.4
BB2	40	40	40	40
IF	80	16	48	22.4
BB3	80	0	40	8
BB4	0	16	8	14.4
Latence du circuit	124	60	92	66.4
Contribution relative des BB à la latence du circuit	BB3 > BB2 > BB1 > BB4	BB2 > BB4 > BB1 > BB3	BB2 = BB3 > BB2 > BB1	BB2 > BB4 > BB3 > BB1

FIGURE 4.3 – Importance des annotations

La Figure 4.3 illustre l'importance des annotations sur un petit exemple. Sur la gauche de la figure, le graphe HCDFG correspondant à une petite application est donné. Les feuilles de ce graphe sont des blocs de base (BB), et leur latence est obtenue par une opération d'ordonnancement. Le nombre d'itérations de la boucle est inféré à partir du graphe. Les annotations considérées sont les probabilités de branchement du noeud *if*. Le tableau à droite de la figure indique les temps d'exécution des différents noeuds du graphe, selon différents scénarios d'exécution.

Le point intéressant est le comportement du circuit présumé par le logiciel de HLS, dans ces différents scénarios. Dans le pire cas, la branche la plus longue (*BB3*) est toujours prise, alors que dans le meilleur cas, la branche la plus courte (*BB4*) est toujours prise. Par défaut, le logiciel de HLS considère que les branches sont prises de façon équiprobable. Enfin, les annotations représentent un scénario supplémentaire, le seul représentant la réalité.

Le tableau de la Figure 4.3 donne le temps d'exécution de chaque noeud du graphe, pour chaque scénario d'exécution. Par exemple, d'après les annotations, le nombre moyen de cycles d'horloge passés dans *BB3* vaut $10 \times 0.1 \times 8 = 8$ cycles d'horloge. Donc en moyenne, *BB3* contribue moins que *BB2* and *BB4* à la latence globale du circuit. De façon similaire, la latence du circuit entier peut être calculée : elle vaut $4 + 8 \times (5 + 0.1 \times 10 + 0.9 \times 2) = 66.4$ cycles d'horloge (en moyenne).

L'exemple de la Figure 4.3 révèle deux points importants. Premièrement, la contribution relative de chaque bloc de base à la latence du circuit dépend du comportement présumé pour le circuit en situation réelle. Ce point est crucial pour la pertinence de l'exploration des solutions, car il reflète directement où les ressources matérielles doivent être utilisées en priorité : il faut accélérer les blocs de base qui contribuent le plus à la latence du circuit entier. Deuxièmement, la latence du circuit entier (en moyenne) est connue au moment de la synthèse, ce qui élimine le besoin de simulations au niveau RTL. Cela permet à l'utilisateur de rechercher les solutions optimales de Pareto réelles avec uniquement le logiciel de HLS. En parallèle, le logiciel de HLS peut aussi calculer la latence du circuit dans les pire et meilleur cas. Ces informations sont particulièrement utiles pour des applications du domaine du temps-réel.

4.1.4 Correction pour le respect d'une fréquence donnée

Le flot de synthèse visé est voulu proche de la transparence d'un flot de compilation. Cela implique que le circuit généré doit respecter strictement toutes les contraintes matérielles imposées par l'utilisateur. En particulier, la fréquence de fonctionnement visée doit être garantie même après les étapes aval de placement et le routage. Cela est possible avec des techniques de correction des circuits générés, grâce à une analyse des temps de propagation des signaux après placement et routage.

Le flot de correction idéal est illustré en Figure 4.4 : il ne contient pas d'itérations avec regénération de certains composants du circuit, et le flot de génération reste entièrement glouton, ce qui garantit notamment sa convergence.

La technique existante du gel d'état est la plus appropriée au contexte visé. En effet, en principe, elle a un coût en ressources modéré, ce coût peut être pris en compte au moment de la HLS, et c'est un procédé très rapide.

La suite de cette section présente une implémentation possible de la technique du gel d'état, dont le coût en ressources est à la fois relativement faible et prévisible de façon exacte au moment de la HLS. Des optimisations seraient probablement possibles, mais cette problématique est hors du contexte des présents travaux. La solution proposée est prévue pour une FSM de type "un parmi N". Cette FSM est générée, depuis la HLS, avec une instrumentation spéciale de façon à permettre les corrections voulues, même après placement et routage.

Pour référence, la Figure 4.5 illustre une partie d'une FSM de type "un parmi N" normale. À chaque état du circuit est associé un registre de un bit. En fonctionnement, à tout instant, seul le registre associé à l'état courant est à 1. Chaque état dure un unique cycle d'horloge. Grâce à ces propriétés, la taille de la logique nécessaire à la transition d'un état à un autre est très faible. Les éléments spécifiques à chaque état de la FSM sont également clairement identifiés. En particulier, à un état donné, le signal permettant l'activation de l'état suivant (plus exactement du registre associé à l'état suivant) n'est autre que la sortie du registre de l'état courant.

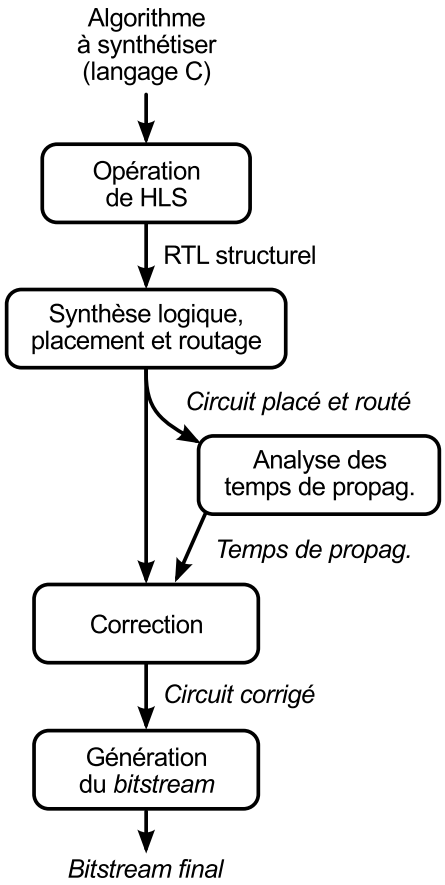


FIGURE 4.4 – Flot de génération idéal pour respecter une fréquence donnée

Le principe du gel d'état est le suivant : pour tout état dont la latence d'au moins une instruction à exécuter dépasse la période d'horloge, l'état de la FSM est "gelé" durant un certain nombre de cycles d'horloge supplémentaires. Ce laps de temps supplémentaire est mesuré grâce à un compteur spécial, placé dans la FSM. Ce compteur est initialisé à zéro à chaque changement d'état, et est incrémenté à chaque cycle d'horloge en situation de gel d'état.

La Figure 4.6 illustre comment une FSM de type "un parmi N" peut être instrumentée pour présenter les propriétés voulues. En plus du compteur dédié au gel d'état ajouté dans la FSM, un nouveau bloc logique est associé à chaque registre des états d'origine. Ce bloc contrôle la fin du gel d'état, et c'est maintenant lui qui génère le signal d'activation de l'état suivant. Ce signal intervient également dans le calcul des sorties de la FSM. En particulier, en situation de gel d'état, les opérateurs pipelinés du chemin de données (le cas échéant) doivent être figés, et la mémorisation des résultats des calculs et la synchronisation avec l'extérieur doivent être désactivées. Le circuit ne fonctionne normalement que lors du dernier cycle d'horloge de l'état.

Le fonctionnement est le suivant. Prenons l'exemple d'un état nécessitant d'ajouter une durée de gel de k cycles d'horloge. Dès l'arrivée dans cet état, le signal d'activation de l'état suivant reste à zéro durant k cycles, puis il passe à 1 durant un cycle, ce qui active la transition vers l'état suivant. Ce signal reste ensuite à zéro tant que le circuit ne revient pas à cet état.

Le nouveau bloc logique implémente un comparateur. Celui-ci compare la valeur du compteur du gel d'état avec une valeur particulière, qui est la durée du gel d'état pour le cycle d'horloge

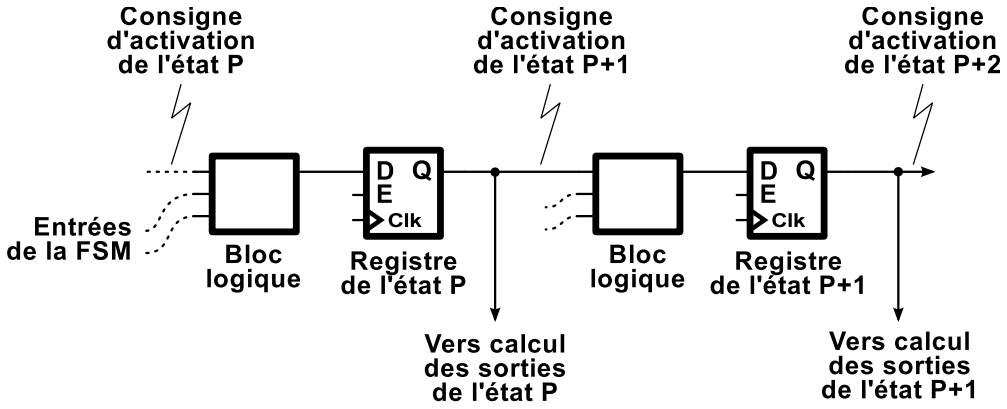


FIGURE 4.5 – Structure d'une FSM de type "un parmi N"

courant. Ainsi, à chaque état de la FSM, une valeur particulière de la durée du gel d'état peut être spécifiée.

Le circuit généré initialement par le logiciel de HLS est configuré de façon à ce que toutes les durées de gel d'état valent zéro (les états durent un unique cycle d'horloge). Afin que la durée du gel des états puisse être réglée même après placement et routage, et ce sans altération des propriétés du circuit (ressources et fréquence), une contrainte d'implémentation particulière est imposée à ce nouveau bloc logique : il doit être entièrement contenu dans une LUT. Le réglage d'une durée particulière de gel d'état consiste alors en un simple changement de la configuration par défaut des LUT ciblées. Il est donc possible d'effectuer cette opération après placement et routage, voire directement dans le *bitstream* généré.

Il est important de souligner que, si l'analyse après placement et routage conclut sur la nécessité d'activer le gel d'état, alors cette durée est censée être faible, par exemple un ou deux cycles d'horloge. En effet, cette durée représente uniquement la partie des temps de propagation des signaux qui n'a pas déjà été prévue et prise en compte par le logiciel de HLS. Une LUT à N entrées permettant d'activer jusqu'à $2^{N-1} - 1$ cycles d'horloge de temps de gel d'état, cette technique devrait convenir à toute technologie de FPGA à base de LUT.

Le flot de correction obtenu est alors bien conforme à la Figure 4.4. De plus, les opérations d'analyse et de correction sont en principe très courtes devant la durée de la synthèse logique et du placement-routage. Cette technique de correction de la fréquence de fonctionnement s'insère donc de façon indolore dans le flot de génération, qui reste entièrement glouton. Son coût en ressources peut être évalué facilement et rapidement pour toutes les solutions explorées lors de la HLS. Au niveau du chemin de données, elle n'impacte que les entrées de validation des pipelines et des unités de mémorisation (mémoires et registres) et les signaux de synchronisation avec l'extérieur. Ces éléments ne sont en général pas dans le chemin critique, ce qui rend cette technique également indolore pour la fréquence maximale de fonctionnement des circuits générés.

Enfin, la structure de FSM illustrée en Figure 4.6 peut probablement être optimisée, notamment au niveau du *fanout* de la sortie du compteur du gel d'état. Il existe des solutions simples qui ne remettent pas en cause les propriétés annoncées, par exemple employer plusieurs compteurs. Cette optimisation est cependant hors du contexte des présents travaux.

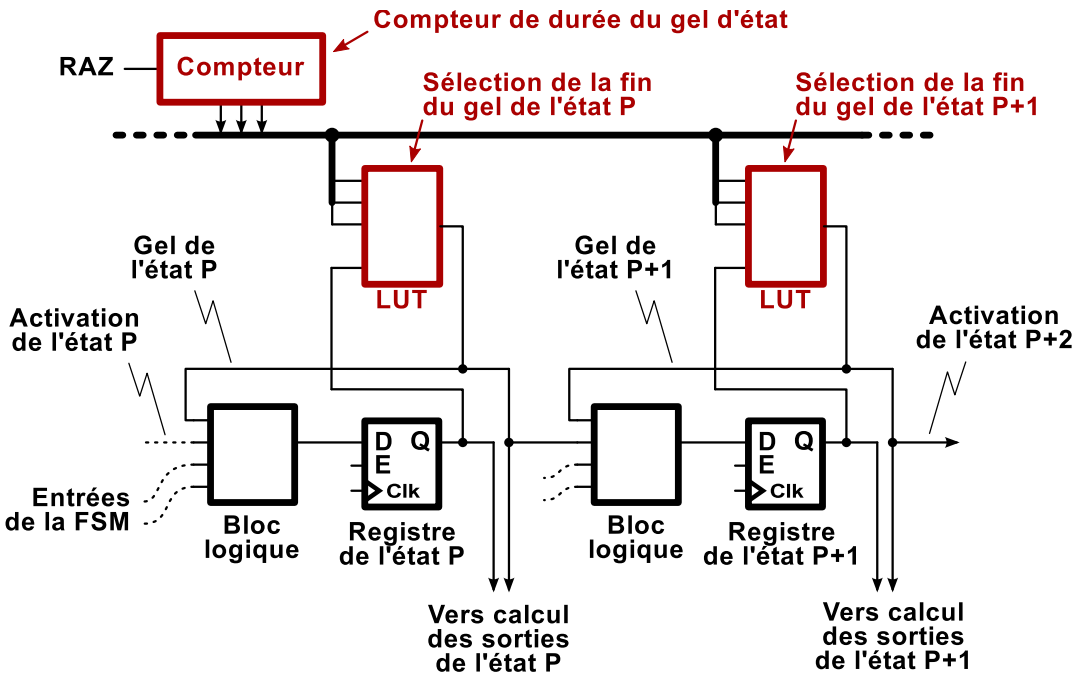


FIGURE 4.6 – Structure de la FSM pour le gel d'état

4.1.5 Pertinence pour d'autres types de contraintes

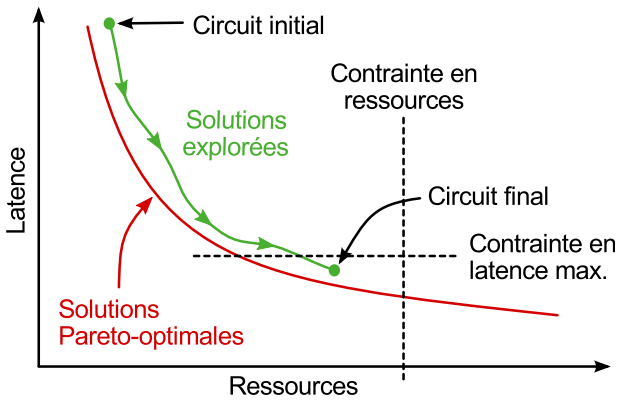


FIGURE 4.7 – Réponse à une contrainte en latence

La méthodologie de HLS proposée est nativement pensée pour répondre à des contraintes en ressources. Cependant, comme mentionné précédemment en section 4.1.3 (page 48), elle permet également de calculer le temps d'exécution des circuits générés, ainsi que pour toutes les solutions intermédiaires envisagées. Cette propriété la rend bien adaptée à des problèmes impliquant des contraintes de latence maximale. Dans ce cas, comme illustré en Figure 4.7, une telle contrainte peut être gérée en stoppant l'exploration dès que la latence limite est atteinte. Prendre une certaine marge de sécurité peut toutefois être nécessaire, afin de prendre en compte l'impact d'éventuelles corrections de la FSM (comme expliqué en section précédente).

Des objectifs en consommation sont souvent considérés dans le domaine de la conception

de circuits. Sous la forme présentée dans ce document, la méthodologie proposée ne permet pas de répondre à de tels objectifs. Éventuellement, des adaptations similaires au respect de la contrainte en ressources seraient possibles (caractérisation des composants, estimateurs, etc) mais ce problème est hors du contexte des travaux présentés ici.

La méthodologie présentée autorise les utilisateurs ayant l'expertise (et le temps) nécessaire, à imposer des conditions structurelles particulières. Le processus d'exploration peut être employé à partir d'une solution initiale donnée par l'utilisateur. Dans ce sens, le flot de HLS proposé reste pleinement compatible avec les flots de génération actuels.

4.1.6 Intégration dans un logiciel de HLS

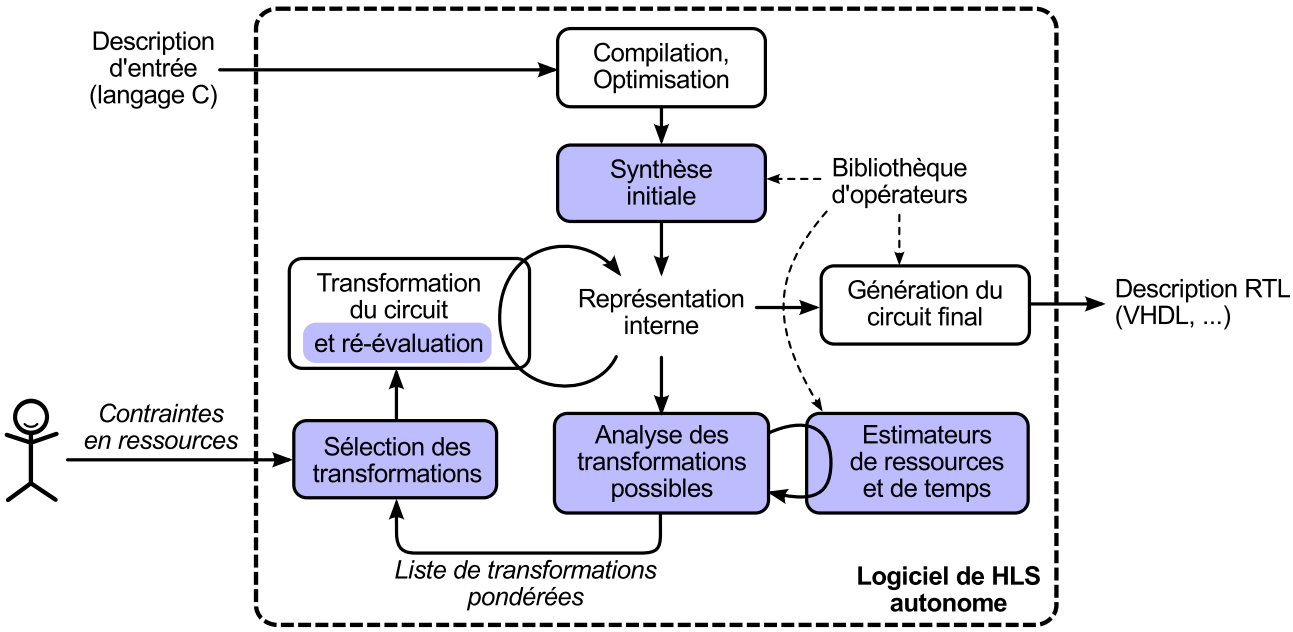


FIGURE 4.8 – Structure du logiciel de HLS proposé

La méthodologie proposée peut être intégrée dans un logiciel de HLS existant. Pour référence, la structure interne des logiciels de HLS les plus courants est illustrée en Figure 3.1 (page 31). La Figure 4.8 présente une façon d'intégrer, dans un tel logiciel, la méthodologie proposée.

Le point principal est que la boucle d'exploration est maintenant entièrement interne au logiciel. Sa seule interaction directe avec l'utilisateur est via un paramètre, spécifié uniquement au lancement du logiciel : les contraintes de ressources.

Les éléments nouveaux, ainsi que ceux hérités du logiciel de HLS hôte et employés d'une façon spécifique (synthèse initiale), sont surlignés. Ils sont présentés dans les sous-sections suivantes. La synthèse initiale est détaillée en section 4.2.1 (page 55). L'analyse des transformations possibles, et leur pondération via des estimateurs, sont décrits en section 4.2.2 (page 55). La sélection des transformations à appliquer est détaillée section 4.2.3 (page 56). La ré-évaluation des solutions considérées est présentée section 4.2.4 (page 57).

4.2 Détail des étapes principales

4.2.1 Synthèse initiale

La synthèse initiale a pour but de générer une solution initiale de surface réduite, de façon à respecter les contraintes en ressources données. Cette solution a donc également une latence élevée, mais cela est sans importance à ce stade. Cette étape de synthèse initiale exploite donc des capacités normalement héritées du logiciel de HLS hôte, mais employées sous un objectif spécifique de surface minimale.

L'obtention de cette solution doit surtout être rapide et, dans ce but, les choix d'allocation et d'affectation peuvent être largement sous-optimaux. Cela est également jugé non critique et est supposé être rectifié durant le processus d'exploration des solutions.

La solution initiale est obtenue en trois étapes :

1. Allocation des unités de mémorisation (registres et mémoires) : chaque variable déclarée est un registre et chaque tableau déclaré est un banc de mémoire.
2. Allocation des opérateurs : un seul opérateur de chaque type nécessaire (additionneur, multiplieur, etc) est instancié. Ces opérateurs sont partagés entre les différentes instructions des blocs de base.
3. Évaluation des caractéristiques du circuit correspondant, par la méthode décrite en section 4.2.4 (page 57).

Si le circuit obtenu dépasse les contraintes, alors la synthèse est considérée impossible. En effet, l'utilisateur souhaite obtenir une certaine accélération de son application, par rapport à une implémentation logicielle. Or, dans le contexte visé, la cible matérielle est supposée être un FPGA. La fréquence atteignable en pratique sur les FPGA pouvant être dix fois plus faible que sur un microprocesseur, un parallélisme très élevé est nécessaire pour justifier l'emploi d'un FPGA. Ce niveau de parallélisme implique qu'une relativement grande quantité de ressources est disponible. Dans ce cas, même avec une certaine sous-optimalité, si la solution initiale dépasse les contraintes, alors ces contraintes peuvent être considérées comme trop limitées pour pouvoir espérer obtenir l'accélération attendue.

4.2.2 Analyse des transformations possibles

Les transformations possibles sont celles que le logiciel de HLS hôte est nativement capable d'effectuer. Ces transformations peuvent concerner aussi bien l'allocation (insertion d'opérateurs de calcul supplémentaires, ajout de ports à des bancs de mémoire, etc) que la structure de la représentation interne (déroulement de boucles, câblage de conditions, etc).

Chaque type de transformation est associé à un jeu d'estimateurs. Ces estimateurs sont des méthodes qui effectuent une prévision de l'impact (en latence et en ressource matérielles) d'une transformation donnée en un endroit donné de la représentation interne. Leur implémentation est fortement liée aux capacités du logiciel de HLS hôte. Les détails des travaux effectués pour les expérimentations sont donnés en section 5.6 (page 71).

L'emploi d'estimateurs, plutôt que des solutions plus exactes (mentionné en section 4.2.4), est un compromis entre précision et rapidité du flot. Les biais d'estimation peuvent contribuer à

la sous-optimalité des circuits générés. Cependant, les estimateurs ont seulement besoin d'être *fidèles*, plutôt que *précis*. Cela signifie que les estimations doivent seulement conserver la relation d'ordre entre les transformations pondérées. En particulier, la transformation sélectionnée à une itération donnée est celle présentant les pondérations les plus avantageuses. L'important est uniquement de sélectionner la meilleure transformation. La valeur exacte des pondérations importe peu, tant que la meilleure transformation reste la mieux pondérée. Cela peut être exploité afin de simplifier et/ou accélérer les estimateurs.

L'utilisation d'estimateurs permet également une grande souplesse dans les considérations prises en compte. Il est envisageable de prendre en compte plusieurs transformations d'avance, et éventuellement d'avantager certaines transformations bien plus bénéfiques sur le long terme qu'elles ne le semblent en les considérant de façon isolée.

Exemple : couramment, le déroulement d'une boucle n'est pas considéré si son corps contient du contrôle dépendant des données. Une transformation de câblage de condition peut faire disparaître ce noeud de contrôle. Cependant, si cette opération de câblage a un coût en ressources estimé trop élevé, elle ne sera pas appliquée, le déroulement de la boucle parente non plus, et le circuit final pourrait être fortement sous-optimal. Cela illustre l'intérêt potentiel d'estimateurs adaptés à des prévisions complexes.

4.2.3 Sélection des transformations

Afin de sélectionner et appliquer la "meilleure" transformation disponible, une pondération supplémentaire est calculée pour chaque transformation. Cette pondération indique la "qualité" globale des transformations, et est calculée à partir des diverses pondérations données par les estimateurs.

L'objectif étant de produire un circuit aussi optimal que possible, les transformations les plus intéressantes sont celles qui *semblent* diriger l'exploration vers les meilleures solutions selon le critère de Pareto. La fonction de pondération globale est donc de la forme suivante :

$$\text{Poids global} = \frac{\text{Gain en latence estimé}}{\text{Coût en ressources estimé}} \tag{4.1}$$

La transformation sélectionnée à une itération donnée est celle présentant le poids global le plus élevé. La formule présentée reste très générale car sa forme est fortement dépendante du logiciel de HLS hôte, de ses capacités de transformation, de la précision des estimateurs employés, et de la technologie matérielle ciblée. Les principales raisons sont les suivantes :

— Le "coût en ressources" est en réalité un vecteur, donnant le coût pour chaque type de ressource matérielle considérée. En particulier, un FPGA contient couramment des LUT, des bascules (FF), des blocs DSP, des bancs de mémoire (BRAM), etc. La formule de pondération globale doit donc faire intervenir une forme d'agrégation de ces coûts élémentaires (par exemple en calculant la moyenne, ou le pire cas, etc).
— Les pondérations sont obtenues avec des estimateurs et, selon la façon dont ils sont implémentés, ils peuvent présenter des biais importants. Or, dans l'objectif de générer des circuits rapides, il faut éviter de consommer les ressources disponibles pour des transformations aux pondérations potentiellement biaisées. S'il est possible de détecter cette

situation, la formule de pondération globale peut inclure des facteurs supplémentaires afin de désavantager les transformations aux pondérations peu fiables.

Des facteurs supplémentaires pourraient être considérés sous des objectifs d'optimisation supplémentaires, par exemple la consommation. Cela est toutefois hors du contexte des présents travaux.

4.2.4 Évaluation précise des caractéristiques d'un circuit

Lors de l'exploration des solutions, des transformations sont appliquées en fonction d'estimations. Sous cette forme brute, après plusieurs transformations, les caractéristiques réelles des solutions considérées peuvent donc diverger notablement des caractéristiques estimées. De plus, les contraintes de ressources indiquées par l'utilisateur sont strictes. Pour le circuit final, tout dépassement, aussi minime soit-il, est interdit. Pour ces raisons, à chaque itération du processus d'exploration ainsi que pour la synthèse initiale, une évaluation précise des caractéristiques du circuit est effectuée.

Cette étape consiste en la génération, à partir de la représentation interne courante, du circuit qu'elle représente. Ce circuit est généré à un niveau très structurel, proche du niveau *netlist*. À ce niveau, le circuit est modélisé sous la forme de composants élémentaires connectés entre eux par des fils. Cela inclut les composants dont l'existence était seulement implicite jusque-là : les multiplexeurs, la FSM, les opérateurs logiques divers, etc. Grâce à la bibliothèque d'opérateurs interne au logiciel de HLS, et à sa calibration pour la technologie ciblée, les caractéristiques précises (en latence combinatoire et en ressources) de chaque composant peuvent être évaluées très précisément.

Ainsi, l'évaluation précise des caractéristiques d'une solution donnée est effectuée selon les étapes suivantes :

1. duplication de la représentation interne,
2. ordonnancement des blocs de base,
3. calcul de la latence du circuit considéré, grâce à la méthode présentée précédemment en section 4.1.3 (page 48),
4. affectation des opérations sur les opérateurs et création des composants implicites (multiplexeurs, FSM, etc),
5. évaluation de l'utilisation en ressource matérielles,
6. et enfin libération des données de la copie de la représentation interne ainsi que du circuit généré.

Les étapes 2, 3, 4 et 5 sont des capacités normalement héritées du logiciel de HLS hôte. En effectuant les opérations d'ordonnancement et d'affectation, une masse importante de données internes est générée, notamment les dépendances de données, la taille des multiplexeurs, la latence combinatoire de chaque instruction du graphe, etc. Ces données sont précieuses pour l'itération suivante du processus d'exploration des solutions. Elles sont exploitées pour la détection des transformations possibles, ainsi que par les estimateurs, pour lesquels elles forment le référentiel permettant la pondération des transformations possibles. C'est pourquoi cette opération d'évaluation précise est effectuée durant la synthèse initiale, ainsi qu'après les transformations de chaque itération.

4.3 Discussion sur l'optimalité des solutions

Dans le cas idéal, les solutions explorées sont des solutions Pareto-optimales et le circuit finalement généré est optimal. Malheureusement, en général, cela a peu de chances de se produire. Cette section décrit les principales situations menant à des solutions sous-optimales, et présente des méthodes permettant de limiter leur impact.

4.3.1 Distance aux solutions de Pareto

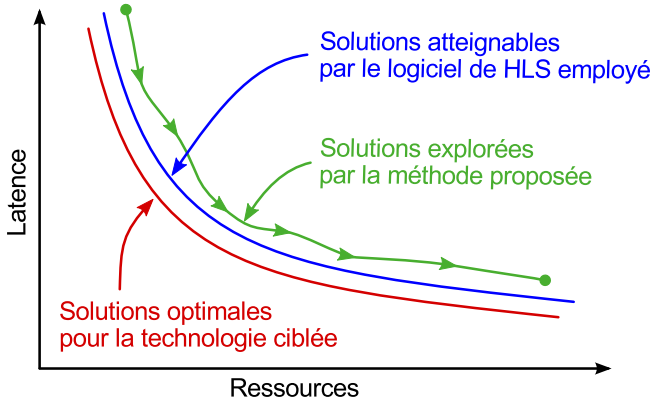


FIGURE 4.9 – Distance à la courbe de Pareto

Pour étudier l'optimalité de la méthodologie proposée pour l'exploration des solutions, on considère trois ensembles de solutions (illustration en Figure 4.9) :

— la courbe de Pareto des solutions pour la technologie ciblée,
— la courbe de Pareto des solutions qu'il est possible d'obtenir avec le logiciel de HLS employé,
— et la courbe des solutions explorées par le logiciel de HLS employé et avec l'algorithme d'exploration proposé.

Obtenir les solutions Pareto-optimales pour la technologie ciblée est théoriquement possible, par un raffinement manuel poussé, et en y consacrant le temps nécessaire. Ce niveau d'optimisation n'est en général pas atteignable avec les seuls paramètres et les possibilités de transformation permises par la HLS. Les solutions obtenues par les logiciels de HLS ont donc un certain degré de sous-optimalité.

Supposons maintenant que la méthodologie proposée est implantée dans un logiciel de HLS donné. Durant l'exploration des solutions, des transformations sont appliquées de façon itérative à la solution initiale. Deux solutions considérées successivement par la méthodologie proposée sont donc fortement similaires (elles diffèrent d'une seule transformation). Or en général, deux solutions très proches sur la courbe de Pareto pour le logiciel de HLS employé peuvent comporter beaucoup plus de différences. Cela signifie que les solutions considérées par la méthodologie proposée ne sont en général pas parmi ces solutions optimales pour le logiciel de HLS employé (elles-mêmes sous-optimales pour la technologie ciblée).

En conséquence, la méthodologie proposée n'est intrinsèquement pas prévue pour produire des solutions optimales. De plus, la position de la courbe de Pareto pour la technologie ciblée restant inconnue, le niveau d'optimalité des solutions produites reste inconnu et l'exploration des solutions progresse "en aveugle". Cette sous-optimalité est le prix de la rapidité de l'exploration, et c'est le compromis effectué par la méthodologie proposée.

4.3.2 Optima locaux

Le processus d'exploration proposé est basé sur un algorithme glouton. Une des principales propriétés de ce type d'algorithme est que le processus d'exploration ne revient jamais en arrière pour tester d'autres solutions afin de tenter de trouver une meilleure progression.

De plus, sous la forme décrite en section 4.1 (page 44), la progression ne prévoit les effets des transformations qu'avec une seule itération d'avance. La méthodologie proposée est donc malheureusement particulièrement prédisposée à tomber dans des optima locaux.

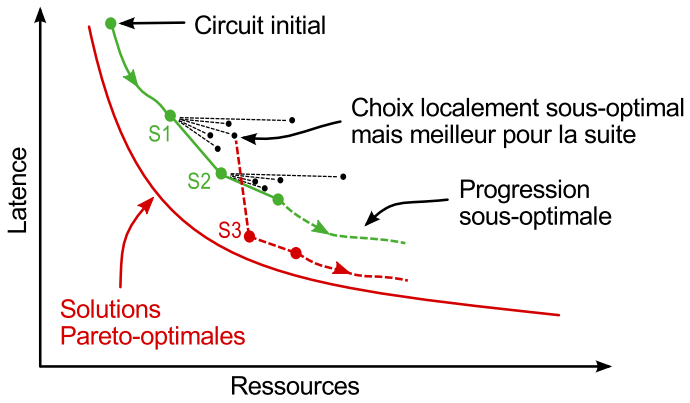


FIGURE 4.10 – Exemple d'optimum local

Un exemple d'optimum local est illustré en Figure 4.10. En *S1*, les transformations possibles sont pondérées. L'application de la transformation localement optimale mène en *S2* et l'exploration se poursuit alors selon la courbe verte (indiquée "Progression sous-optimale"). Cependant, effectuer un choix sous-optimal en *S1* aurait pu révéler de nouvelles transformations possibles, et mener en *S3*. Depuis *S3*, l'exploration des solutions se serait alors poursuivie selon la courbe rouge, qui est meilleure que celle issue de *S2* selon le critère de Pareto.

En poursuivant l'exploration, de nombreuses situations de ce type peuvent être rencontrées. L'incertitude sur le long terme due aux décisions locales est sérieuse, et l'optimalité de la solution finale ne peut pas être présumée tant qu'elle n'est pas atteinte. Or, les choix sont effectués d'après les pondérations associées aux transformations possibles. Pour cette raison, la précision des estimateurs, et même leur fidélité, pourrait n'avoir qu'une importance relative sur l'optimalité des circuits générés. Cela est traité par les expériences décrites en section 6.6 (page 93).

4.3.3 Progression non monotone du coût en ressources durant l'exploration

Le taux de ressources utilisées pour un type de ressources donné (par rapport à la quantité autorisée par les contraintes de l'utilisateur) peut progresser de manière non monotone au cours de l'exploration des solutions. Cela signifie qu'à une itération donnée du processus d'exploration, il peut exister des solutions à la fois plus rapides et plus économes en ressources. La Figure 4.11 illustre une telle situation, entre les points *S2* et *S4*.

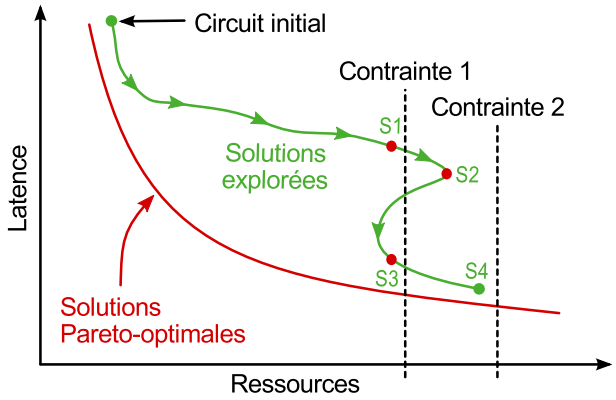


FIGURE 4.11 – Progression non monotone

Cette situation peut apparaître pour plusieurs raisons :

— à cause de la méthodologie proposée elle-même, qui peut parfois imposer un ordre d'application des transformations peu optimal,
— à cause des biais d'estimation,
— à cause de la formule de pondération globale, agrégeant les pondérations des ressources hétérogènes (mentionné en section 4.2.3).

Selon les contraintes en ressources données, ces situations peuvent être problématiques pour la méthodologie proposée. Notamment, sur la Figure 4.11 et pour la contrainte 1, l'exploration des solutions s'arrêterait à la solution *S1*, alors que la solution *S3*, bien meilleure car plus rapide, pourrait tout-à-fait convenir. Pour la contrainte 2, le problème est que certaines transformations peuvent avoir été marquées comme impossibles lors de l'exploration précédant *S2*. En effet, en se rapprochant de *S2*, la quantité de ressources disponibles s'amenuisant, le processus d'exploration marque comme impossibles les transformations dont le coût excède les ressources disponibles. Or, si la solution *S3* est tout de même atteinte (grâce à d'autres transformations), le processus d'exploration peut s'arrêter car il n'y a plus de transformation possible. Dans ce cas, la solution *S4* pourrait ne jamais être atteinte, alors qu'elle l'aurait été si certaines transformations n'avaient pas été marquées impossibles un peu trop promptement, avant d'atteindre *S2*.

Ce problème de progression non monotone peut être atténué de plusieurs manières. Par exemple, le logiciel de HLS peut être étendu avec des types de transformations supplémentaires, permettant d'atteindre *S3* selon une progression plus optimale selon le critère de Pareto. D'autres solutions liées à la méthodologie proposée sont décrites en section 4.4.

4.4 Variantes

La description de la méthodologie d'exploration décrite dans la section 4.1 (page 44) peut être considérée comme la principale ligne directrice. Certaines variantes méritent d'être considérées, en vue d'améliorer l'optimalité des circuits générés, ou la vitesse de génération des circuits (mais probablement pas les deux simultanément). Les sous-sections suivantes présentent les principales variantes possibles.

4.4.1 Pondération exacte des transformations

Le biais des estimateurs peut être une cause de sous-optimalité des circuits générés. En pondérant les transformations possibles non pas avec des estimations, mais avec les valeurs réelles, la transformation appliquée à chaque itération est assurée d'être la meilleure possible, au moins localement.

Il est possible d'effectuer une pondération exacte d'une transformation donnée avec le processus suivant :

1. duplication de la représentation interne,
2. application de la transformation considérée sur cette copie,
3. puis évaluation des caractéristiques du circuit correspondant avec la méthode précise décrite en section 4.2.4.

Cette méthode garantit que les choix localement optimaux sont effectués, ce qui, en général, devrait être favorable à l'optimalité des circuits générés. L'implémentation logicielle de cette technique de pondération est légère, car la technique d'évaluation précise est de toutes façons requise par la méthodologie proposée. De plus, une même implémentation logicielle permet de pondérer tout type de transformation, là où un jeu d'estimateurs, potentiellement complexes et spécialisés par type de transformation, est requis.

Cependant, cette méthode est sans doute extrêmement gourmande en temps de calcul, et est donc à employer avec prudence. De plus, elle ne résout pas tous les cas d'incertitude relatifs à la progression de l'exploration des solutions. En particulier, il est toujours possible de tomber dans des optima locaux (section 4.3.2). Dans des cas pathologiques, une pondération exacte peut même mener à un optimum local, qui aurait été évité "grâce" à une erreur d'estimation.

4.4.2 Application de plusieurs transformations par itération

Sous la forme brute proposés pour l'algorithme d'exploration des solutions, une seule transformation est appliquée à chaque itération. Or, un des objectifs affichés de la méthodologie proposée est de générer le circuit final très rapidement. De ce point de vue, l'algorithme proposé peut être considéré comme peu efficace, et plusieurs transformations devraient être appliquées à chaque itération. Cette discussion est en réalité un problème d'équilibre entre rapidité de la génération, et d'optimalité des circuits générés.

En appliquant une unique transformation à chaque itération, les pondérations des transformations sont exploitées au plus juste. En revanche, l'estimation de l'impact de multiples transformations sur le circuit, effectuée à partir des pondérations des transformations élémentaires

(elles-même des approximations), peut être hasardeuse. Une solution possible est de sélectionner des transformations relativement indépendantes les unes des autres. Dans ce cas, la pondération du groupe de transformations est la somme des pondérations des transformations élémentaires (pour le gain en latence et le coût en ressources). Il est alors envisageable de sélectionner plusieurs transformations, parmi les mieux pondérées, à chaque itération. Une autre solution est de construire des estimateurs spécialisés à des groupes de transformations présentant des dépendances particulières (par exemple, déroulement d'une boucle avec un noeud de contrôle imbriqué). Cependant, ce cas peut être complexe à mettre en oeuvre.

Une adaptation de l'algorithme d'exploration est alors nécessaire. En effet, les biais d'estimation peuvent mener à des cas où un groupe de transformations estimé compatible avec les contraintes se trouve être en réalité trop coûteux. Dans ce cas, il n'est pas possible de simplement marquer comme impossible chaque transformation considérée pour continuer l'exploration, comme proposé initialement. En effet, même si le groupe de transformations excède les contraintes, certaines de ces transformations peuvent tout de même être appliquées individuellement. Il faut donc revenir en arrière, et former une sélection de transformations différente. Ce contretemps potentiel peut limiter le gain de l'application de plusieurs transformations, voire faire perdre du temps par rapport à la sélection d'une seule transformation.

Afin d'éviter ces contretemps, des limites peuvent éventuellement être imposées sur la taille des groupes de transformations formés :

— Limite sur le coût en ressources : interdire la formation de groupes dont le coût en ressources estimé dépasse une certaine proportion des ressources restantes.
— Limite sur le nombre de transformations sélectionnées : interdire la formation de groupes dont la taille (en nombre de transformations) dépasse une certaine proportion des transformations possibles à l'itération courante.
— Limite sur la pondération globale des transformations sélectionnées : interdire la sélection de transformations ayant un poids global inférieur à un certain seuil, par rapport au poids de la meilleure transformation.

De telles limites peuvent réduire les risques de dépassement des contraintes et la consommation des ressources disponibles pour des transformations peu utiles.

Ainsi, ce procédé peut être très bon pour la rapidité de l'exploration. Mais il peut être relativement complexe à implémenter dans le logiciel hôte et, à cause de l'ajout d'un biais d'estimation potentiel supplémentaire, il peut être néfaste à l'optimalité des circuits générés.

4.4.3 Re-considérer les transformations marquées impossibles

Durant le processus d'exploration, certaines transformations peuvent être marquées impossibles pour cause de coût en ressources jugé excessif (comme expliqué en section 4.1, page 44). Cependant, cette pratique peut écarter, à tort, certaines transformations qui seraient en réalité très intéressantes. Les raisons sont les suivantes :

— Les estimateurs de coût en ressources peuvent être biaisés. Le coût pourrait en réalité être acceptable, dans le sens du respect des contraintes imposées.

— Le coût d'une transformation est évalué pour un état du circuit donné. Or, en appliquant d'autres transformations, la structure du circuit peut changer notablement. Une transformation présentant un coût excessif à une itération donnée peut avoir un coût convenable une ou plusieurs itérations plus tard, en particulier lorsque l'exploration progresse de façon non monotone selon les types de ressources concernés (comme mentionné en section 4.3.3).

Deux solutions sont proposées afin de compenser ces problèmes.

La première solution consiste à reconsidérer les transformations marquées impossibles, éventuellement en effectuant des pondérations précises (comme proposé en section 4.4.1). Ainsi, le processus d'exploration serait relancé plusieurs fois, selon l'Algorithme 2 : lorsque plus aucune transformation n'est disponible, le marquage des transformations comme étant impossibles peut être retiré, et l'exploration relancée.

Algorithme 2 Algorithme d'exploration relancé plusieurs fois

Entrées : Description en langage C + contraintes en ressources
Sorties : Circuit en langage RTL ou netlist

```
répéter // BOUCLE-RELANCE
    Lancer l'exploration des solutions
    si aucune transformation n'a été appliquée alors
        SORTIR de la boucle BOUCLE-RELANCE // L'exploration est terminée
    fin si
    Réactiver les transformations désactivées lors de l'exploration
fin répéter // BOUCLE-RELANCE
Générer le circuit final
```

La deuxième solution consiste à ne marquer aucune transformation comme étant impossible durant l'exploration. Dans ce cas, à chaque itération, toutes les transformations possibles seraient considérées et pondérées, ceci incluant celles qui ont été évaluées comme trop coûteuses aux itérations précédentes.

Ces deux solutions ont des impacts différents sur le processus d'exploration, en particulier sur la durée d'exécution. En proposant de recalculer aveuglément toutes les pondérations à chaque itération, la deuxième solution est potentiellement plus lourde que la première. Mais cela peut s'inverser si, pour la première solution, les transformations sont reconsidérées avec des pondérations précises.

4.4.4 Raffinement autour de la solution finale

Comme expliqué en section 4.3, le circuit final peut être sous-optimal, la position des solutions optimales restant inconnue. Dans ces conditions, une recherche par recuit simulé (ou techniques similaires) peut permettre de trouver une meilleure solution [STW09]. Éventuellement, l'exploration pourrait être poursuivie à partir d'une nouvelle solution, obtenue par ce type d'algorithme.

4.5 Synthèse sur le flot proposé

La méthodologie de HLS répond, au moins conceptuellement, aux objectifs identifiés au début du présent chapitre. Elle permet l'exploration des solutions de façon pertinente et sans interaction avec l'utilisateur. Sa faible complexité algorithmique autorise le passage à l'échelle pour générer des circuit complexes. Elle élimine le besoin de simulation de chaque solution explorée, et offre à l'utilisateur des possibilités étendues pour spécifier le comportement de l'application au niveau du contrôle dépendant des données. La fréquence de fonctionnement exigée par l'utilisateur peut également être garantie, même après placement et routage.

Les principaux points faibles potentiels sont identifiés, et des variantes de l'algorithme d'exploration sont proposées afin d'en amoindrir les conséquences.

Dans les prochains chapitres, la méthodologie proposée est implantée dans un logiciel de HLS démonstrateur en vue d'effectuer des expérimentations et d'analyser les propriétés de la méthodologie proposée en situation réelle.

Chapitre 5

Construction d'un démonstrateur

Ce chapitre décrit la construction d'un logiciel démonstrateur pour la méthodologie de HLS proposée. Un logiciel de HLS existant est choisi et des modifications y sont apportées. Des choix techniques sont effectués, notamment sur des considérations algorithmiques, avec l'ordonnancement, l'affectation, l'allocation initiale, la sélection des transformations et l'évaluation précise des solutions considérées. Les types de transformations disponibles et les techniques de pondération employées font également partie des choix d'implémentation spécifiques au logiciel, ainsi que les modélisations des types de composants et la calibration pour une technologie particulière.

Sommaire

5.1 Logiciel de HLS hôte

La méthodologie proposée pour l'exploration des solutions peut être intégrée au sein de la plupart des logiciels de HLS existants. Ainsi, afin de construire un logiciel démonstrateur pour la méthodologie proposée et de limiter l'ampleur de la tâche de développement, un logiciel de HLS existant a été utilisé comme point de départ.

Le logiciel UGH [AP08] a été choisi. C'est un logiciel académique *open source*, qui a été développé aux laboratoires LIP6 (principalement), à Paris, et TIMA, à Grenoble. Ce logiciel a été choisi pour les raisons suivantes.

— C'est un logiciel *open source* : le code source est disponible et peut être modifié librement. Le logiciel modifié pourra également être diffusé sans limitation.
— Il présente une stratégie d'allocation en premier (par rapport à l'ordonnancement et l'affectation), connue pour être bien adaptée au respect de contraintes de ressources.
— Le processus de génération est divisé en deux phases principales : l'ordonnancement à gros grain, puis à grain fin pour respecter une fréquence de fonctionnement donnée. Cette séparation peut faciliter l'intégration de nouveaux processus de génération.
— Les circuits générés ont une structure interne clairement scindée en composants connectés par des fils (détails en section 5.2, page 67). Les modèles de composants sont précisément paramétrés, ce qui est approprié pour le besoin en évaluation précise des caractéristiques des circuits générés.

De nombreuses modifications ont été effectuées afin d'intégrer la méthodologie proposée :

— extension de la représentation interne afin d'intégrer la notion de hiérarchie,
— calibration de la bibliothèque d'opérateurs pour les technologies Xilinx Virtex-5 et Virtex-7 (détails en section 5.3, page 68),
— création du module de synthèse initiale (détails en section 5.4, page 70),
— implantation de plusieurs types de transformations et des estimateurs associés (détails en section 5.6, page 71),
— implantation de la gestion d'annotations de l'utilisateur (probabilités de branchement et nombre moyen d'itérations des boucles),
— création du module de sélection des transformations à appliquer,
— création d'une fonction d'évaluation précise du circuit (détails en section 5.8, page 80).

Faute de temps, la technique de *retiming* proposée en section 4.1.4 (page 50) n'a pas été implémentée. Les travaux sont focalisés sur la méthodologie d'exploration des solutions. Pour chaque état du circuit pour lequel la latence combinatoire estimée d'une instructions à exécuter dépasse la période d'horloge, des états de FSM sont ajoutés de façon à couvrir la latence nécessaire.

D'autres modifications non spécifiques à la méthodologie d'exploration proposée ont été apportées (détails en section 5.9, page 81). L'ampleur des changements (environ 41k lignes de programme en langage C) est telle qu'un nouveau nom a été donné au logiciel : AUGH, pour *Autonomous and User-Guided High-level synthesis*.

5.2 Structure du circuit généré

Le circuit généré est composé de plusieurs composants matériels connectés entre eux au sein d'une entité parente. Ces composants peuvent être classés en cinq catégories.

— Contrôle : il y a un unique composant de contrôle, qui est une machine à états (FSM) de style Moore. Cette FSM utilise le codage "un parmi N", pour sa simplicité de modélisation et les possibilités de correction après placement et routage décrites précédemment en section 4.1.4 (page 50).

— Stockage : les registres et les bancs de mémoire. Dans le cadre de ce démonstrateur, les bancs de mémoire sont toujours implantés en LUTRAM, et les bancs de mémoire BRAM du FPGA ne sont pas utilisés. Les opérations de lecture et d'écriture sont possibles simultanément.

— Routage et sélection des chemins de données : les multiplexeurs.

— Calcul, composants *partagés* : additionneurs, soustracteurs, multiplieurs, opérateurs de décalage et de rotation.

— Calcul, composants *non partagés* : unités logiques et comparateurs.

On retrouve la séparation traditionnelle en une partie commande (la FSM) et une partie opérative (tous les autres composants). Les composants *partagés* et *non partagés* sont gérés différemment. Les composants *partagés* sont créés lors de l'allocation initiale (détails en section section 5.4, page 70), ou bien lors de l'application de certaines transformations, durant l'exploration des solutions. Ces composants sont partagés entre les différentes instructions de l'algorithme à synthétiser. Les composants *non partagés* sont gérés de façon similaire aux multiplexeurs : ils forment la "colle logique" qui connecte la FSM, les composants de stockage et les composants *partagés* en un tout cohérent qui répond à la fonctionnalité demandée. Ils sont créés et dimensionnés lors du processus d'affectation.

Les types d'opérateurs partagés sont ceux utilisant traditionnellement le plus de ressources (multiplieurs, opérateurs de décalage et de rotation), ainsi que ceux qui contiennent une chaîne de propagation de retenue rapide (additionneurs, soustracteurs). Pour ces derniers, la raison est que les chaînes de propagation de retenue ne peuvent être placées qu'à des emplacements spécifiques du FPGA, ce qui représente une contrainte forte pour les étapes aval du flot de conception (placement et routage). Le fait de les considérer comme partagés permet de contrôler leur nombre dans le circuit final.

Le logiciel ne supporte pas (encore) les architectures pipelinées. Pour cette raison, tous les chemins de données considérés sont combinatoires, y compris les opérations de lecture dans les bancs de mémoire. Ces limitations sont uniquement dues aux capacités du logiciel démonstrateur, et ne sont pas des caractéristiques de la méthodologie proposée. En effet, la grande majorité des opérateurs pipelinés sont pourvus d'une entrée de validation (tels les blocs matériels DSP dont les FPGA Xilinx et Altera sont pourvus), ce qui reste compatible avec la technique du gel d'état envisagée (section 4.1.4, page 50).

La structure générale du circuit généré est illustrée en Figure 5.1. Les entrées de la FSM (les compte-rendus de la partie opérative) peuvent être n'importe quel signal de un bit disponible dans le circuit. Les sorties de la FSM (les commandes pour la partie opérative) contrôlent tous les multiplexeurs du circuit, ainsi que les entrées de validation des registres et bancs de mémoire

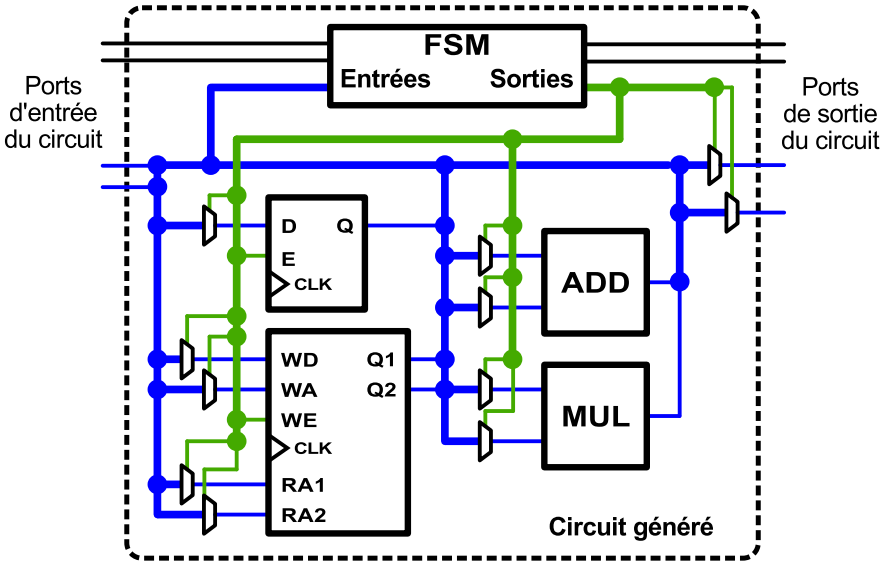


FIGURE 5.1 – Structure interne du circuit généré

(*write enable*) et la synchronisation des communications avec l'extérieur. Les données d'entrée des composants de calcul sont sélectionnées à travers des multiplexeurs. Le chaînage d'opérateurs dans un même cycle d'horloge est permis.

5.3 Bibliothèque d'opérateurs et calibration pour une technologie donnée

Afin de pouvoir évaluer précisément les caractéristiques des circuits, les composants de la bibliothèque d'opérateurs sont modélisés au niveau *netlist*. À ce niveau, chaque composant est constitué des éléments matériels de base du FPGA (LUT, FF, etc), connectés entre eux par des fils.

La calibration de la bibliothèque d'opérateurs pour la technologie de FPGA ciblée consiste en la caractérisation, en latence combinatoire, de chaque élément matériel de base. Ces informations sont extraites de la documentation technique officielle du fabricant du FPGA [Xil13c]. On peut citer en particulier la latence d'une LUT, le temps de pré-sélection d'un registre, etc.

Chaque opérateur est modélisé selon la structure interne recommandée pour la fonctionnalité en question et pour la technologie ciblée. Par exemple, la Figure 5.2 représente la structure d'un additionneur 4 bits pour une technologie Xilinx, où chaque LUT possède deux bits de sortie. Cette structure est décrite dans la documentation du fabricant [Xil12].

Couramment, les éléments spécifiques à la chaîne de propagation de retenue (les portes logiques XOR et les multiplexeurs) ne sont utilisables que via les LUT associées. Pour cette raison, le coût en ressources matérielles d'un additionneur de largeur N vaut N LUT.

La latence combinatoire du composant dépend des calculs effectués. Par exemple, le chemin critique sur la Figure 5.2 correspond au cas où la retenue d'entrée *Cin* reste à zéro et lorsque la valeur de la retenue de sortie *Cout* n'est pas lue (utilisation la plus courante).

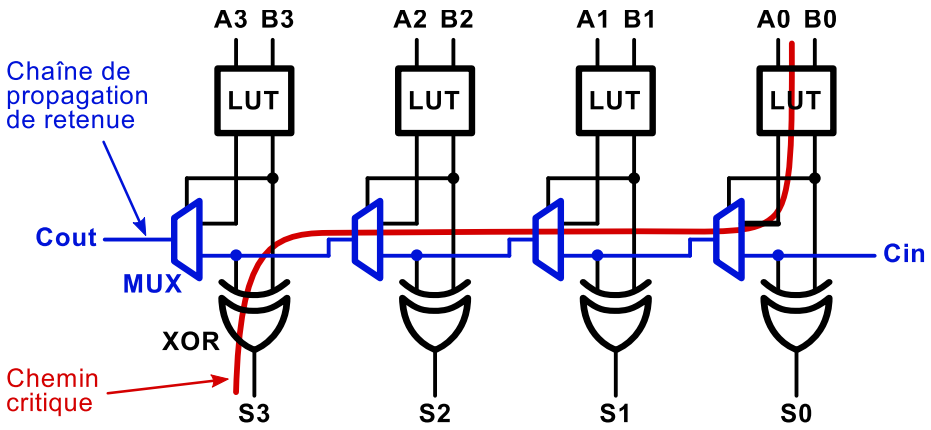


FIGURE 5.2 – Modélisation niveau *netlist* du composant Additionneur

De plus, la documentation du fabricant du FPGA [Xil13c] indique que les temps de propagation des signaux entre les composants spécifiques à la chaîne de propagation sont négligeables. Le calcul de la latence combinatoire du composant est donc :

$$L_{comp} = L_{lut} + (N-1) \times L_{mux} + L_{xor} \tag{5.1}$$

où

— L_{comp} est la latence du composant pour le calcul considéré,
— N est la largeur des opérandes pour le calcul considéré,
— L_{lut}, L_{xor} et L_{mux} sont les latences combinatoires respectives d'une LUT, d'une porte XOR et d'un multiplexeur (données par la documentation du fabricant [Xil13c]).

Pour d'autres calculs à effectuer, la valeur de la retenue de sortie peut être exploitée. Dans ce cas, la latence combinatoire du composant s'exprime ainsi :

$$L_{comp} = L_{lut} + (N-1) \times L_{mux} + max(L_{xor}, L_{mux}) \tag{5.2}$$

Pour d'autres types d'opérateurs, les temps de propagation des signaux internes peuvent ne pas être négligeables. Cela se produit notamment lorsque les signaux internes au composant sont routés via le tissu d'interconnexion configurable du FPGA. Dans ce cas, une valeur de temps de propagation arbitraire est employée. Celle-ci est obtenue lors d'expériences de synthèse menées avec le logiciel de synthèse logique ISE de Xilinx. La valeur correspondant au pire cas observé a été retenue, afin d'assurer, autant que possible, que les temps de propagation seront toujours valides après placement et routage. Cette valeur est également employée pour modéliser les temps de propagation des signaux entre les composants, dans le circuit entier.

Dans la technologie de FPGA ciblée, il existe également des éléments matériels optimisés pour effectuer des opérations arithmétiques, appelés blocs DSP (issu de l'anglais *Digital Signal Processing*). Ces blocs contiennent notamment un multiplieur de 18×25 bits et un additionneur de 48 bits. Dans AUGH, ils sont utilisés comme élément de base (au même titre qu'une LUT) pour les opérateurs de multiplication. L'usage des DSP peut aussi être désactivé manuellement, et dans ce cas une structure basée sur les LUT est utilisés.

Cette technique de modélisation des types de composant au niveau *netlist* permet d'évaluer rapidement, et de façon relativement fiable, la taille du circuit final ainsi que la latence combinatoire de toutes les opérations. Cette propriété permet au logiciel de HLS de respecter des contraintes matérielles diverses, notamment la surface du circuit ainsi qu'une fréquence de fonctionnement donnée.

5.4 Allocation initiale

Juste après le chargement et la compilation de l'algorithme d'entrée, des composants initiaux sont créés. Ceci concerne les composants de stockage et les opérateurs partagés.

Pour les composants de stockage, l'allocation suit une règle simple : les variables déclarées sont stockées dans des registres, et les tableaux dans des bancs de mémoire (incluant les ROM). Ces composants de stockage ont chacun leurs propres paramètres (largeur des données et profondeur d'adresse). Cette allocation initiale fige donc l'assignation des variables et tableaux associés.

AUGH n'effectue en effet pas de partage des registres. Certes, une telle fonctionnalité permettrait de réduire l'impact de la forme de la description d'entrée sur la rapidité des circuits générés, notamment au niveau de l'emploi de variables de confort pour faciliter la lisibilité des programmes. Mais ce comportement, hérité du logiciel UGH, a été conservé pour plusieurs raisons.

— Partager les registres réduirait le nombre de registres dans le circuit final, mais au prix d'une augmentation de la taille de leurs multiplexeurs d'entrée, et par conséquent également d'une augmentation de la latence des instructions concernées. En effet, les registres seraient alors utilisés pour des calculs potentiellement très différents, issus de nombreuses instructions.
— Les FPGA sont très riches en registres. Cette ressource est souvent sous-utilisée comparé aux LUT notamment.
— Le processus de partage des registres est potentiellement coûteux en temps de calcul, ce qui est contraire à l'objectif de génération rapide.
— Mettre en place le partage des registres est une lourde tâche de développement logiciel, jugée non critique pour le logiciel démonstrateur proposé.

Les opérateurs de calcul sont partagés entre les instructions des blocs de base. Afin de factoriser les expressions des calculs à effectuer, AUGH effectue une détection des sous-expressions communes entre les instructions. Cette opération est effectuée en insérant toutes les expressions des instructions dans un dictionnaire de sous-expressions communes. Cela révèle le nombre minimal nécessaire d'opérateurs de chaque type, ainsi que le nombre de ports d'accès à chaque banc de mémoire. Tous les opérateurs d'un même type sont créés avec les mêmes paramètres, correspondant au pire cas, c'est-à-dire à l'opération effectuée sur les opérandes les plus larges en nombre de bits. Cette phase d'initialisation est gardée relativement peu élaborée car, après l'étape d'assignation, il est possible de réduire chaque opérateur selon son utilisation réelle.

Ensuite, durant le processus d'exploration des solutions, cette allocation initiale est étendue ou simplifiée lorsque des transformations sont appliquées (ajout d'opérateurs ou de ports aux bancs de mémoire, simplification des registres, remplacement de mémoires, etc).

5.5 Ordonnancement et assignation

L'ordonnancement décide quelles instructions sont exécutées à quels états du circuit, tout en garantissant le respect des dépendances de données. Cette opération est effectuée sur chaque bloc de base de la représentation interne. L'algorithme d'ordonnancement employé est un ordonnancement par liste basé sur la priorité des instructions (PBLS), pour sa rapidité et la performance de ses résultats. L'implémentation est similaire à celle du logiciel GAUT [CCB+08].

L'assignation est effectuée juste après l'ordonnancement de tous les blocs de base. Cette opération affecte un opérateur à chaque opération à exécuter. Pour les opérations correspondant à des composants non partagés, (i.e. opérations logiques), de nouveaux composants sont créés au besoin. Le partage des chemins de données est effectué grâce à deux dictionnaires : l'un contient tous les chemins de données existants et l'autre contient toutes les expressions des instructions à exécuter. Chaque élément de chaque dictionnaire est lié aux éléments compatibles de l'autre dictionnaire, ce qui permet une recherche très rapide. Le nombre d'entrées des multiplexeurs est étendu au besoin. La structure de la FSM est construite à la volée, notamment en ce qui concerne le nombre d'états, les ports d'entrée (pour les branchement conditionnels) et de sortie (les commandes des multiplexeurs), ainsi que le calcul des sorties.

À la suite de l'assignation, une optimisation de la *netlist* est effectuée. Les multiplexeurs n'ayant qu'une entrée sont éliminés. Ceux de largeur 1 bit et qui n'ont en entrée que des valeurs binaires connues statiquement voient également leur fonctionnalité transférée à la FSM. Les fonctionnalités optionnelles de certains composants sont éliminées (par exemple, une indication de débordement de capacité (*overflow*) pour une opération arithmétique). Le circuit est alors prêt à être généré dans un langage RTL, en l'occurrence le VHDL.

5.6 Transformations disponibles et pondérations

Cinq types de transformations sont implantées dans AUGH : l'extension de l'allocation (ajout d'opérateurs et de ports de lecture aux bancs de mémoire), le câblage de condition, déroulement de boucle, le remplacement de bancs de mémoire par des registres indépendants, et le remplacement des mémoires ROM par les valeurs littérales.

Les sections suivantes décrivent chaque type de transformation, ainsi que les techniques d'estimation employées pour les pondérations.

5.6.1 Extension de l'allocation : ajout d'opérateurs et de ports aux mémoires

L'extension de l'allocation consiste à ajouter au circuit des composants de calcul supplémentaires, ou bien à ajouter des ports de lecture aux bancs de mémoire. Elle cible donc les composants dont l'accès est partagé entre les instructions des blocs de base. L'extension de l'allocation permet d'augmenter le parallélisme des calculs dans le circuit. L'objectif est de réduire le nombre de cycles d'horloge nécessaires à l'exécution des blocs de base.

Pour cela, une analyse des dépendances de données entre les instructions est effectuée, dans le but de détecter les goulots d'étranglement dus aux ressources partagées. Dans AUGH, les ressources partagées sont de deux types : les opérateurs de calcul partagés, et les ports d'accès en lecture aux bancs de mémoire. L'ajout de ports d'écriture aux bancs de mémoire n'est pas géré car (à l'heure actuelle), les bancs de mémoire ne possèdent qu'un seul port d'écriture.

Le processus d'analyse recherche les instructions dont l'exécution pourrait être avancée d'un cycle d'horloge, à condition que l'accès à certaines ressources partagées soit possible. Les dépendances de données avec les autres instructions sont donc vérifiées. Ces informations (dépendances de données et accès à des ressources partagées) sont présentes dans les données internes du logiciel de HLS car elles sont produites par l'ordonnanceur, à l'itération précédente du processus d'exploration des solutions (ou bien lors de la synthèse initiale).

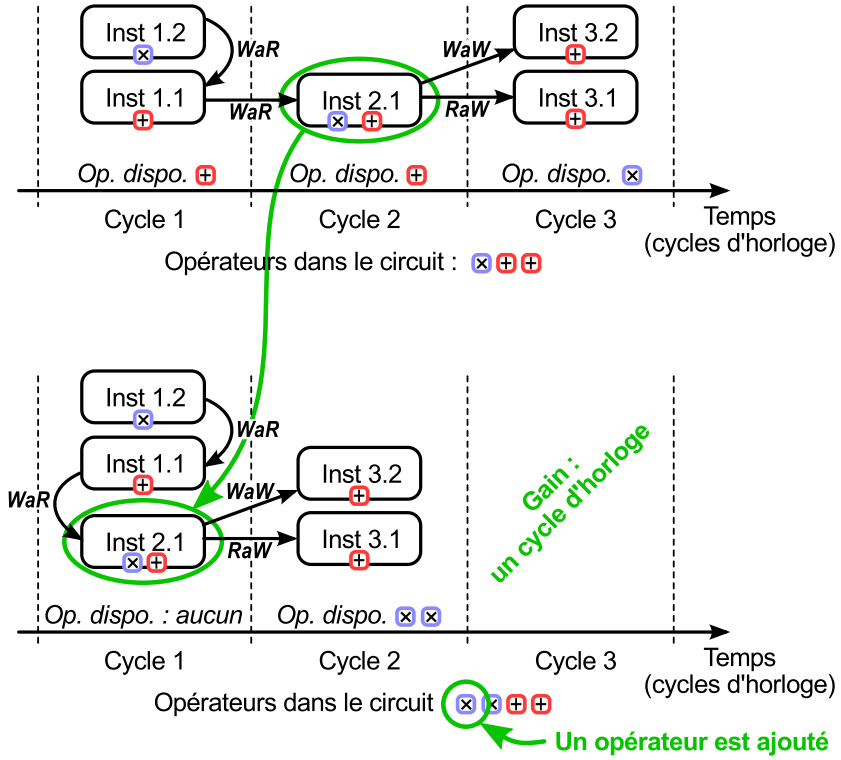


FIGURE 5.3 – Ré-ordonnancement suite à l'ajout d'un opérateur

La Figure 5.3 illustre le cas d'une instruction dont l'exécution pourrait être avancée d'un cycle. Il s'agit de l'instruction 2.1. Elle a été ordonnancée au cycle 2 au lieu du cycle 1, parce qu'il manquait un opérateur, en l'occurrence un multiplieur. Cette instruction étant la seule dans le cycle d'horloge 2, avancer son exécution d'un cycle signifie que le temps d'exécution du bloc de base est également réduit d'un cycle.

Cependant, cette opération doit être permise par les dépendances de données entre les instructions (symbolisées par les flèches sur la Figure 5.3). Une dépendance est une relation d'ordre entre une instruction source et une instruction cible. Il en existe trois types (également décrits dans [Don04]) :

— *RaW* (de l'anglais *Read after Write*) indique qu'une variable est écrite dans l'instruction source de la dépendance et qu'elle est lue dans l'instruction cible. Les registres enregistrent

les valeurs sur front montant de l'horloge. Afin que la valeur soit effectivement mémorisée dans l'instruction source, au moins un front montant de l'horloge doit se produire entre les deux instructions.

— *WaW* (de l'anglais *Write after Write*) indique qu'une variable est écrite à la fois dans les instructions source et cible de la dépendance. Cette dépendance peut être cumulée avec *RaW* dans des instructions du genre "$i = i + 1$". Elle est également utilisée pour des accès à des tableaux, dont la valeur des adresses peut être inconnue au moment de la compilation. Cette dépendance est alors ajoutée par sécurité au cas où les adresses accédées dans les deux instructions soient identiques. Cette dépendance exige également au moins un front d'horloge entre les deux instructions.

— *WaR* (de l'anglais *Write after Read*) indique qu'une variable est lue dans l'instruction source et écrite dans l'instruction cible. Ces deux instructions peuvent être séparées par un ou plusieurs fronts d'horloge, mais elles peuvent également être exécutées au même cycle d'horloge. En effet, l'écriture n'est effective qu'au moment du front d'horloge à la fin du cycle, ce qui ne perturbe pas la lecture, qui est effectuée durant le cycle. Pour cette raison, dans l'exemple de la Figure 5.3, l'instruction 2.1 peut être avancée et être exécutée au même cycle que l'instruction 1.1.

Ainsi, pour chaque instruction du circuit dont les dépendances permettent d'avancer l'exécution d'un cycle (ou plus), un jeu de ressources partagées "manquantes" est formé. Le processus d'analyse des transformations possibles forme ainsi un certain nombre de jeux de ressources partagées. Ceux-ci sont ensuite pondérés, en coût en ressources matérielles et en gain en latence sur le circuit entier.

Coût en ressources

On distingue deux cas : les opérateurs à ajouter au circuit, et les bancs de mémoire auxquels il faut ajouter des ports d'accès.

Pour l'ajout d'un opérateur, le coût en ressources a deux sources : le nouvel opérateur lui-même, et les multiplexeurs pour ses entrées. La taille des opérateurs, ainsi que la taille des multiplexeurs, sont estimées au pire cas parmi les opérateurs déjà présents dans le circuit. Ces informations sont disponibles grâce à l'évaluation précise des caractéristiques du circuit (décrite précédemment en section 4.2.4, page 57) effectuée à l'itération précédente (ou bien lors de la synthèse initiale).

Pour l'ajout de ports de lecture à un banc de mémoire, le coût en ressources est également réparti entre le composant principal (le banc de mémoire) et les multiplexeurs (en l'occurrence, un seul multiplexeur, pour le port d'adresse). Le multiplexeur est estimé au pire cas parmi les ports de lecture déjà présents dans le banc de mémoire. L'augmentation de la taille du composant est estimée grâce à la bibliothèque d'opérateurs caractérisés : cette bibliothèque fournit la taille du composant courant, et la taille du composant avec un port de lecture supplémentaire. Le coût en ressources, pour le composant, est la différence entre ces deux tailles.

Gain en latence

Pour chaque jeu de ressources à ajouter, le gain en latence potentiel sur le circuit entier est estimé. Cela est effectué en sommant les contributions de toutes les instructions que l'ajout du jeu de ressources considéré permettrait d'avancer d'un cycle d'horloge.

Pour chacune de ces instructions, deux cas sont considérés : l'instruction est la seule dans son cycle d'horloge, ou bien il y a d'autres instructions.

Le cas où l'instruction est seule dans son cycle d'horloge est le plus simple : ajouter le jeu de ressources considéré permet, de façon assurée, de diminuer la latence du bloc de base dont elle fait partie d'au moins un cycle d'horloge.

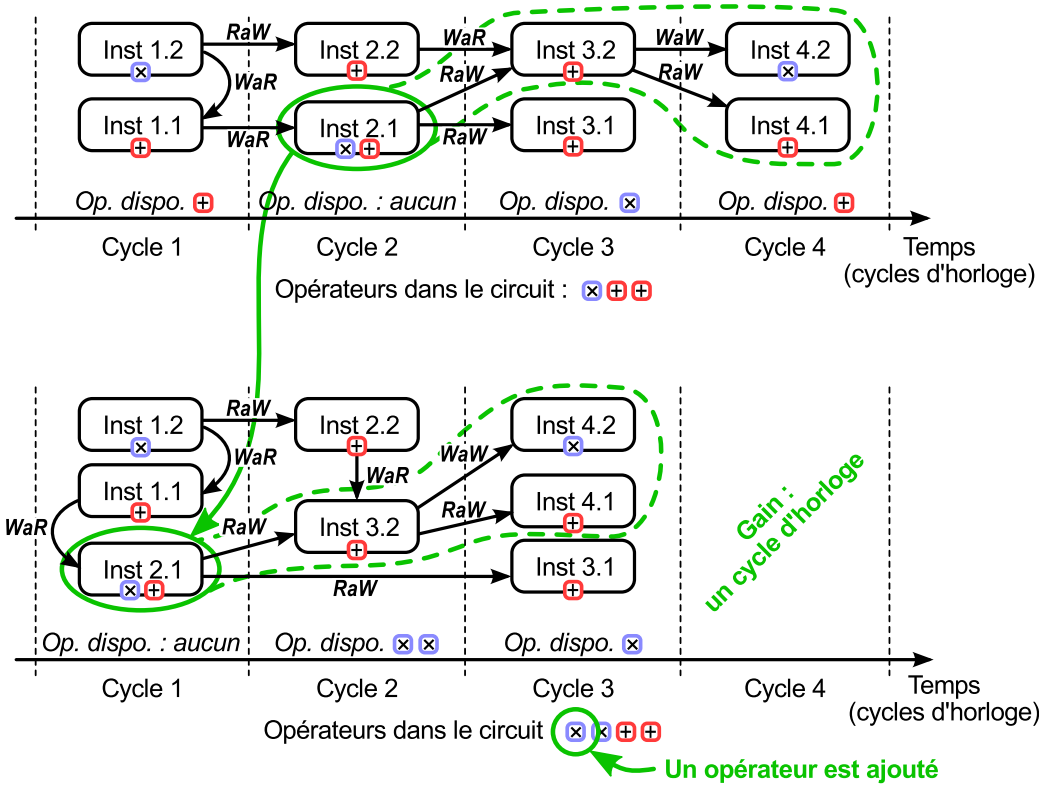


FIGURE 5.4 – Ré-ordonnancement en cascade suite à l'ajout d'un opérateur

Le cas où l'instruction n'est pas seule dans son cycle d'horloge est illustré en Figure 5.4. L'instruction qu'il serait possible d'avancer d'un cycle est la 2.1. Elle est actuellement ordonnancée en même temps que l'instruction 2.2. Dans ce cas, des cycles d'horloge peuvent tout de même être économisés grâce à un effet de ré-ordonnancement en cascade : avancer l'instruction 2.1 permet d'avancer également l'instruction 3.2, ce qui permet à son tour d'avancer les instructions 4.1 et 4.2. Ainsi, le cycle économisé n'est pas le cycle 2, mais le cycle 4.

Détecter cette possibilité de façon certaine nécessiterait cependant une analyse poussée des dépendances de données, dont la complexité algorithmique pourrait être comparable à un ré-ordonnancement du bloc de base entier. Cela est jugé trop lourd pour la méthodologie de HLS rapide souhaitée, pour les raisons suivantes :

— À chaque itération du processus d'exploration, un très grand nombre de ces transformations possibles peut être détecté. Celles-ci diffèrent par le type, le nombre et la taille des opérateurs à ajouter, ainsi que par les bancs de mémoire ciblés et le nombre de ports à y ajouter. Potentiellement, chaque instruction qu'il est possible d'avancer peut avoir une transformation associée.
— Les opérateurs et les ports des mémoires sont partagés entre les instructions et entre les blocs de base. Pour chaque transformation possible, tous les blocs de base du circuit utilisant les opérateurs et les bancs de mémoire ciblés devraient être ré-ordonnancés. Cela pourrait potentiellement représenter le circuit entier.

Pour ces raisons, une solution plus légère, mais approximative, est employée : avec N le nombre d'instructions présentes au cycle d'horloge considéré (sur l'exemple de la Figure 5.4, pour le cycle 2, $N = 2$ instructions), le nombre de cycles d'horloge potentiellement économisés dans le bloc de base est estimé à $1/N$.

Cette valeur ne représente pas seulement la possibilité d'apparition de l'effet de ré-ordonnancement en cascade. Même si aucun cycle d'horloge n'est immédiatement gagné, cela pourrait se produire en effectuant de nouvelles extensions de l'allocation, à des itérations ultérieures du processus d'exploration des solutions. La pondération $1/N$ a donc une signification probabiliste. Elle est également valable dans le cas où l'instruction est seule dans son cycle d'horloge (cas particulier).

Cette pondération estimée (qui est pour l'instant locale au bloc de base) est ensuite convertie en un gain en latence potentiel au niveau du circuit entier. Pour cela, la pondération est multipliée par la contribution du bloc de base à la latence du circuit, qui est connue (détails en section 4.1.3, page 48).

Finalement, le gain en latence estimé du jeu de ressources considéré est la somme des gains apportés par chaque instruction dont il permettrait d'avancer l'exécution. La formule complète est :

$$P_{lat} = \sum_{i \in \mathfrak{I}} \frac{factorBB(i)}{nbInstCycle(i)} \tag{5.3}$$

où :

— P_{lat} est la pondération en gain en latence du jeu de ressources considéré,
— $\mathfrak{I}$ est l'ensemble des instructions que ce jeu de ressources permettrait d'avancer d'un cycle,
— $nbInstCycle(i)$ est le nombre d'instructions actuellement ordonnancées en même temps que l'instruction i,
— $factorBB(i)$ est le nombre de fois que le bloc de base contenant l'instruction i est exécuté, dans le scénario d'exécution considéré.

5.6.2 Câblage de condition

Le câblage de condition consiste à éliminer un branchement conditionnel, en incorporant le test dans les expressions à calculer dans chaque branche. Elle apporte des opportunités de parallélisation des instructions et potentiellement des possibilités d'extension de l'allocation (c.f. section 5.6.1).

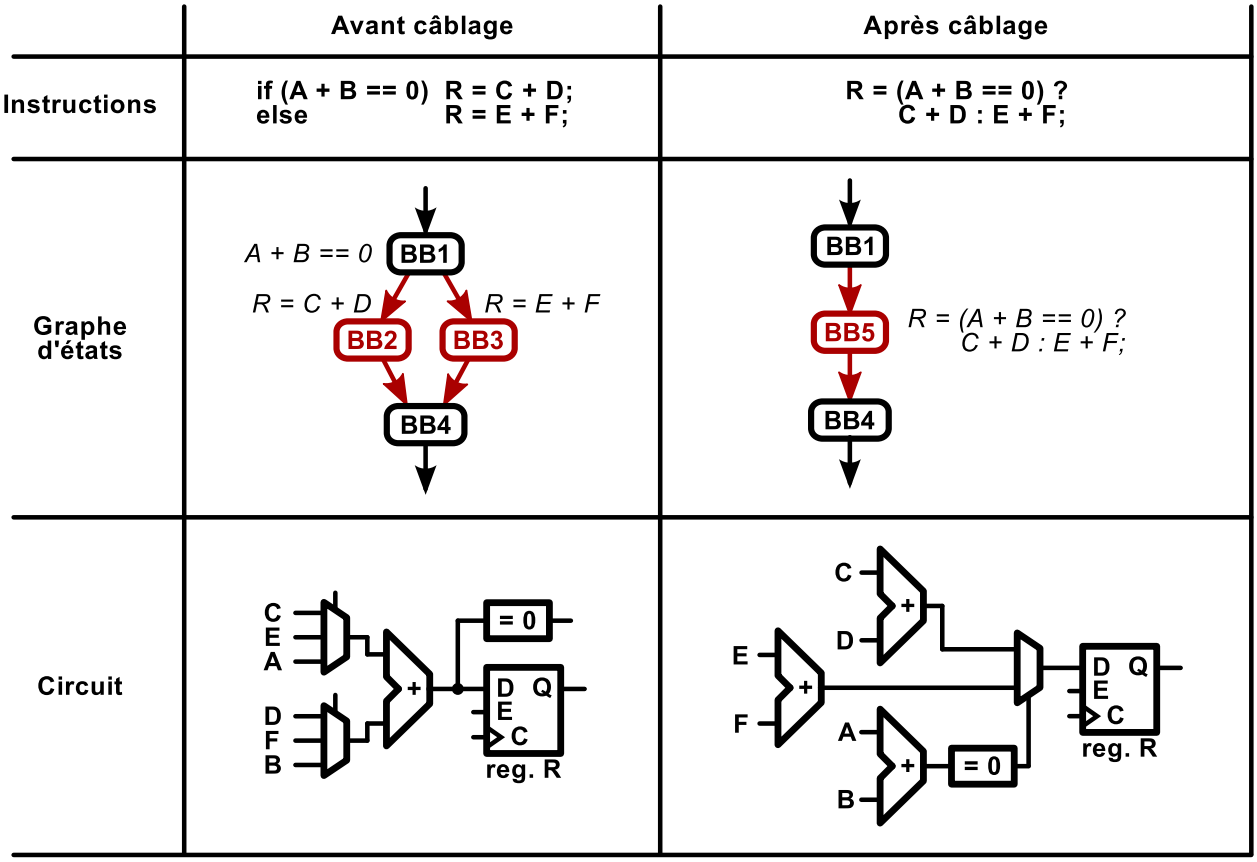


FIGURE 5.5 – Câblage de condition

Comme illustré en Figure 5.5, le branchement conditionnel est éliminé de la machine à états, et l'instruction effectuant le test est éliminée du bloc de base précédant le branchement (*BB1*). Les blocs de base conditionnés sont remplacés par un nouveau bloc de base (*BB5*), composé des instructions modifiées de chaque branche (initialement *BB2* et *BB3*). Par la suite, ce bloc de base pourra être fusionné avec les blocs de base précédent (*BB1*) et suivant (*BB4*), apportant ainsi des nouvelles opportunités de parallélisation pour l'ordonnanceur.

Le coût en ressources est évalué selon la quantité d'opérateurs nécessaires à l'évaluation simultanée du test et des expressions de chaque branche. Par exemple, sur l'exemple de la Figure 5.5, deux additionneurs supplémentaires sont nécessaires.

L'estimation du nombre de cycles d'horloge que cette transformation peut faire économiser est effectuée en appliquant localement la transformation, sur une copie temporaire des blocs de base impliqués. Une étape d'ordonnancement sur ces données dupliquées permet d'obtenir le gain en cycles d'horloges. Cette opération étant effectuée uniquement sur les instructions conditionnées, et ne nécessitant pas de refaire l'affectation du circuit, elle est jugée acceptable en temps de calcul.

5.6.3 Déroulement de boucle

Cette transformation bien connue permet de réduire le nombre d'itérations d'une boucle, en dupliquant le corps de la boucle. Elle apporte des opportunités de parallélisation des instructions et potentiellement des possibilités d'extension de l'allocation (c.f. section 5.6.1).

Le type de déroulement effectué est un déroulement séquentiel. Cela consiste à dupliquer le corps de la boucle, les duplicata créés étant concaténés les uns aux autres. Ainsi, l'ordonnancement et l'affectation du circuit peuvent toujours être effectués en utilisant les opérateurs déjà alloués. L'extension de l'allocation n'est considérée qu'à partir de l'itération suivante du processus d'exploration des solutions. Le déroulement de boucle n'est considéré que lorsque le corps de la boucle ne contient pas de flot de contrôle, car dans ce cas le gain potentiel est très faible.

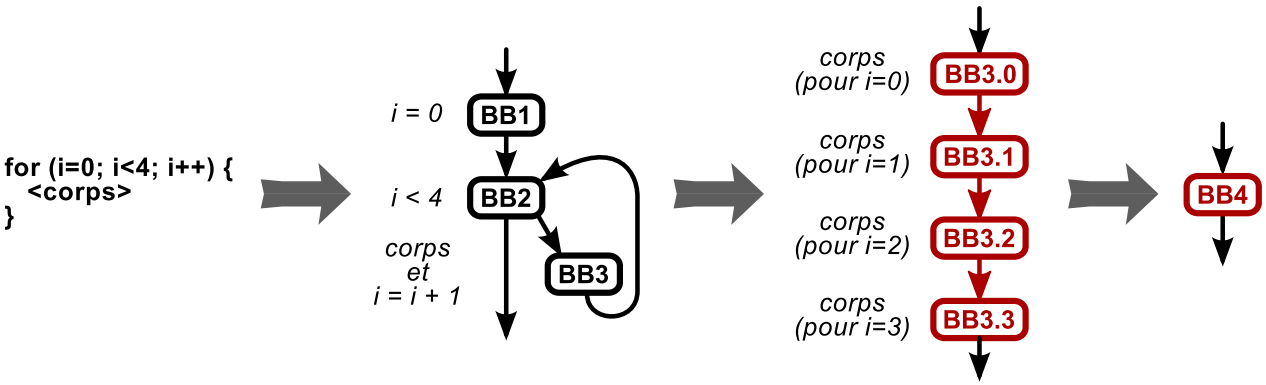


FIGURE 5.6 – Déroulement total d'une boucle

La Figure 5.6 illustre le cas d'un déroulement total. Avant déroulement, la boucle est constituée de plusieurs blocs de base, le corps principal étant *BB3*. Le déroulement crée autant de duplicata de *BB3* que la boucle possède d'itérations (4 dans l'exemple). Dans chaque duplicata de *BB3*, les instructions d'incrémentation de l'itérateur ($i = i + 1$) sont éliminées, et toutes les références à l'itérateur *i* restantes sont remplacées par la valeur de l'itérateur, soit 0 dans *BB3.0*, 1 dans *BB3.1*, 2 dans *BB3.2* et 3 dans *BB3.3*. L'initialisation de l'itérateur ($i = 0$) disparaît également. Les blocs de base résultants sont fusionnés en un nouveau bloc de base, *BB4*.

Puisque l'allocation n'est pas modifiée, l'impact en ressources est supposé être principalement localisé dans la FSM. La structure de cette FSM étant de type "un parmi N", le coût est estimé proportionnel au nombre d'états ajoutés par la duplication du corps de la boucle. L'estimation du coût est effectuée de la façon suivante : chaque nouvel état coûte un registre et une LUT, et cause également une augmentation de la taille de la logique de calcul des sorties utilisées dans le corps de la boucle. Cette augmentation de la taille des sorties est estimée arbitrairement à une LUT par sortie.

L'évaluation du nombre de cycles d'horloge que cette transformation peut apporter est évaluée en appliquant localement la transformation, sur une copie temporaire des blocs de base impliqués. Une étape d'ordonnancement sur ces données dupliquées permet d'obtenir le gain en cycles d'horloges. Cette opération étant effectuée uniquement sur le corps de boucle, et ne nécessitant pas de refaire l'affectation du circuit, elle est jugée acceptable en temps de calcul.

5.6.4 Remplacement d'un banc de mémoire par des registres indépendants

Cette transformation consiste à remplacer une mémoire donnée par autant de registres que la mémoire contient de cellules. Dans toutes les instructions des blocs de base, toutes les opérations de lecture ou d'écriture effectuées sur cette mémoire sont remplacées par des accès sur le registre correspondante. Cette transformation n'est donc possible que lorsque, pour tous les accès à la mémoire considérée, l'adresse de la cellule ciblée est connue à la compilation.

Cette transformation permet d'éliminer les contentions d'accès à la mémoire dues à un nombre de ports trop limité. Elle autorise toutes les opérations de lecture et d'écriture simultanément sur toutes les cellules de la mémoire. Elle permet donc d'augmenter le nombre d'opérations possibles à chaque cycle d'horloge, réduisant ainsi la latence du circuit.

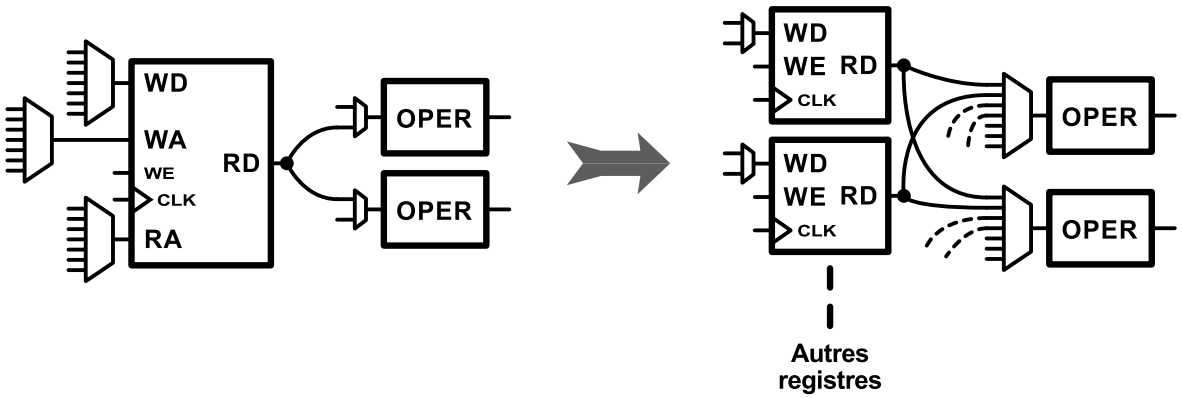


FIGURE 5.7 – Remplacement d'un banc de mémoire

L'impact sur le circuit est illustré en Figure 5.7, pour une mémoire présentant à l'origine un port d'écriture et un port de lecture. Outre le composant de mémoire ciblé, la transformation impacte également les multiplexeurs.

Avant transformation, les multiplexeurs d'entrée de la mémoire sont potentiellement gros, car ils sélectionnent tous les chemins de données de tous les résultats à mémoriser dans toutes les cellules. Les multiplexeurs en entrée des opérateurs sont potentiellement petits, car ils ne sélectionnent que les données issues de l'unique port de lecture de la mémoire.

Après transformation, la situation sur la taille des multiplexeurs est inversée. Ceux en entrée des registres sont généralement plus petits, car ils sélectionnent uniquement les chemins de données des calculs à enregistrer dans le registre auquel ils sont associés. Les multiplexeurs en entrée des opérateurs sont généralement plus gros, car ils peuvent sélectionner les données issues de chacun des registres créés.

Afin de prévoir précisément l'ampleur de ces changements, il faudrait considérer les résultats des différents calculs à mémoriser dans chaque cellule, et prendre en compte l'affectation des opérations sur les opérateurs. Cette analyse est potentiellement très complexe, et le seul moyen d'obtenir des valeurs fiables est probablement d'appliquer la transformation, et de refaire l'ordonnancement et l'affectation du circuit complet. C'est une opération trop coûteuse.

L'approximation effectuée est que, globalement, le fait que certains multiplexeurs grossissent est compensé par le fait que d'autres rapetissent. Sous cette hypothèse, le coût en ressources de la

transformation est celui lié à la mémoire (qui disparaît) et aux registres (qui sont créés). Ce coût est obtenu par deux appels directs à la bibliothèque de composants embarquée dans le logiciel : un pour évaluer la taille de la mémoire d'origine, et un pour évaluer la taille des registres. Le coût en ressources est la différence entre ces deux évaluations.

L'estimation de l'accélération du circuit est similaire à l'ajout de ports de lecture, à plus large échelle : toute contention d'accès à la mémoire est éliminée.

5.6.5 Remplacement de mémoire ROM

Cette transformation consiste à remplacer, dans toutes les instructions des blocs de base, toutes les opérations de lecture depuis le banc de mémoire par la valeur contenue dans la cellule correspondante de la mémoire ROM. Elle n'est donc possible que lorsque, pour tous les accès à la mémoire ROM considérée, l'adresse de la cellule lue est connue à la compilation.

Cette transformation permet d'éliminer toute contention d'accès à la mémoire ROM qui serait due à un nombre de ports de lecture trop limité. Cette transformation est donc très similaire au remplacement d'un banc de mémoire par des registres indépendants. Pour cette raison, la technique d'estimation du gain en latence est identique.

L'estimation du coût en ressources est cependant différent, puisqu'en remplaçant une mémoire ROM, aucun registre n'est créé. L'impact en ressources est donc situé au niveau du composant de mémoire (qui disparaît), des multiplexeurs en entrée des ports d'adresse (qui disparaissent), et des multiplexeurs en entrée des opérateurs (qui grossissent).

Similairement au remplacement d'une mémoire par des registres indépendants, une approximation est effectuée afin d'éviter une analyse potentiellement complexe du circuit. Dans AUGH, les mémoires ROM et les multiplexeurs utilisent le même type de ressources matérielles : les LUT. L'approximation consiste à estimer que la disparition de certains composants compense le fait que d'autres grossissent. Le coût en ressources matérielles de cette transformation est donc estimé nul, en moyenne.

5.7 Sélection des transformations à appliquer

La sélection de la (ou des) transformation(s) à appliquer à chaque itération est effectuée à partir des pondérations en coût en ressources et en gain en temps d'exécution du circuit. Afin de pouvoir comparer la "qualité" relative de toutes ces transformations possibles, une pondération globale est calculée pour chacune. Cette évaluation est une métrique sans unité représentant le rapport du gain divisé par le coût. La formule exacte utilisée est la suivante :

$$PG_{tr} = \underset{R \in \mathfrak{R}}{max} \left(\frac{C_R - R_{act,R}}{PR_{tr,R}} \right) \times \frac{PT_{tr}}{T_{act}} \tag{5.4}$$

où :

— PG_{tr} est le poids global pour la transformation considérée,
— $\mathfrak{R}$ est l'ensemble des types de ressources considérés (LUT, FF, etc),

— $PR_{tr,R}$ est le coût de la transformation tr pour le type de ressource R,
— $R_{act,R}$ est la quantité de ressources de type R utilisée par le circuit actuel,
— C_R est la contrainte utilisateur sur les ressources de type R,
— PT_{tr} est le gain de la transformation tr pour le temps d'exécution du circuit, en cycles d'horloge,
— T_{act} est le temps d'exécution du circuit actuel, en cycles d'horloge.

Cette formule donne une note élevée aux transformations qui apportent un gain en temps d'exécution élevé, et qui présentent un coût en ressources faible. Les types de ressources sont gérés séparément pour éviter que le processus d'élaboration se retrouve bloqué par une utilisation excessive d'un type de ressource particulier.

Cette formule n'est pas présentée comme une heuristique de décision optimale. Déterminer si elle est optimale, et même simplement s'il existe une formule optimale dans le cas général (ce qui est peu probable), est au-delà de la problématique des travaux présentés.

L'heuristique ci-dessus permet de trier rapidement toutes les transformations possibles selon leur poids global. Le processus d'exploration sélectionne alors la transformation de poids global le plus élevé.

Comme mentionné précédemment en section 4.4.2 (page 61), il est possible de sélectionner, et d'appliquer, plusieurs transformations par itération du processus d'exploration des solutions. Cela permet de réduire le nombre d'itérations total, et donc de réduire le temps de génération du circuit. Cela est effectué en sélectionnant d'autres transformations possibles parmi celles de plus haute pondération globale. Un groupe de transformations possibles est ainsi formé. Une limite est toutefois imposée : le coût en ressources cumulé de toutes les transformations sélectionnées doit rester inférieur à 10 % des ressources disponibles. Ce seuil doit être respecté pour chacun des types de ressources disponibles (LUT, FF, etc). La valeur de ce seuil, 10 %, est choisie arbitrairement pour les expérimentations.

Le module de sélection des transformations possibles est également capable d'effectuer une pondération précise des transformations possibles, selon la technique présentée en section 4.4.1 (page 61). Ce type de pondération peut permettre d'améliorer la pertinence des solutions considérées.

5.8 Évaluation précise des solutions durant l'exploration

De nombreuses solutions sont construites par le processus d'exploration. Il est impératif que chacune respecte les contraintes matérielles données par l'utilisateur. La progression étant basée sur des estimations, une vérification doit être effectuée à chaque itération, aussitôt après application des transformations sélectionnées.

L'objectif principal de cette opération est de fournir une évaluation précise et fiable. Dans ce but, cette vérification est implémentée en la génération d'une modélisation au niveau *netlist* de la solution courante. Peu de compromis sur la rapidité du processus sont possibles : le processus effectue l'ordonnancement et l'assignation du circuit complet. La modélisation au niveau *netlist* est alors évaluée de façon précise, tant en ressources qu'en latence combinatoire, par la bibliothèque d'opérateurs du logiciel.

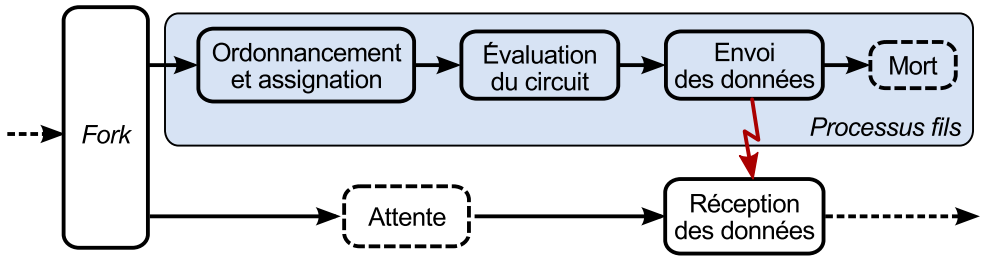


FIGURE 5.8 – Évaluation d'une solution via un *fork*

Le schéma de la Figure 5.8 illustre l'implémentation de cette fonctionnalité. Les structures de données décrivant le circuit courant sont dupliquées en utilisant la fonctionnalité *fork* du système d'exploitation (en l'occurrence GNU/Linux). Un appel à cette fonction provoque la duplication du processus système contenant à la fois le logiciel de HLS et ses structures de données courantes. Le processus fils est alors libre d'effectuer n'importe quelle opération sur ses structures de données. En l'occurrence, il effectue l'ordonnancement et l'assignation, génère la modélisation au niveau *netlist*, évalue précisément ses caractéristiques, transfère ces données au processus père, et meurt.

5.9 Synthèse sur le développement effectué à partir de UGH

Les modifications effectuées dans UGH sont nombreuses et ont touché la plupart des sections du logiciel. Dans un premier temps, seuls les travaux directement liés à la méthodologie d'exploration proposée ont été effectués. La preuve de concept pour la capacité à explorer différentes solutions de façon autonome a ainsi été établie et publiée dans [PBMR12]. Ces résultats préliminaires ont également révélé que, pour pouvoir appliquer cette technique sur des circuits complexes, les processus de génération propres au logiciel de HLS hôte devaient présenter un niveau de complexité algorithmique particulièrement bas. Or, ce point est crucial pour démontrer que la méthodologie proposée est techniquement réalisable. Des adaptations devaient donc être faites.

La principale cause de la faible vitesse de génération du logiciel modifié était héritée d'une fonctionnalité spécifique à UGH : la capacité à générer des circuits selon un schéma des chemins de données spécifié par l'utilisateur. Cette fonctionnalité imposait en effet des contraintes très fortes sur les processus d'ordonnancement et d'assignation. Plus précisément, la vérification de la conformité de tous les chemins de données considérés vis-a-vis des chemins autorisés ou interdits par l'utilisateur apportait un degré de complexité algorithmique inapproprié pour les besoins de la méthodologie proposée. Certaines simplifications des instructions des blocs de base ne pouvaient pas non plus être effectuées.

Pour ces raisons, cette capacité à suivre un schéma de chemin de données spécifié par l'utilisateur a été retirée du logiciel. Les processus d'ordonnancement, d'assignation, d'ordonnancement à grain fin et de génération des circuits ont été remplacés par des versions plus rapides. De même, pour les besoins en détection des transformations possibles, un processus de simplification minimaliste des instructions des blocs de base a été implémenté. Celui-ci détecte et élimine

le code mort, propage les constantes dans les instructions, et effectue également une propagation et élimination de certaines instructions simples.

Ces travaux ont apporté un gain en vitesse de génération de deux à trois ordres de grandeur (respectivement pour les applications *idct* et *mjpeg* utilisées dans les expériences au chapitre suivant). Ils ont certes nécessité d'importants efforts de développement, mais ils étaient indispensables afin d'arriver au niveau de complexité algorithmique annoncé.

Un interpréteur de commandes a également été intégré au logiciel, permettant notamment à l'utilisateur d'appliquer certaines transformations manuellement. Le logiciel final intègre ainsi la méthodologie d'exploration proposée, tout en conservant le traditionnel mode manuel. Le nouveau nom du logiciel, AUGH (pour *Autonomous and User-Guided High-level synthesis*), reflète cette versatilité.

De UGH, seuls subsistent deux librairies : le *parser* pour charger le code des applications en langage C, et la librairie de représentation et de manipulation d'expressions vectorielles *VEX*. Dans les travaux à venir, le *parser* sera remplacé (pour cause d'obsolescence) et des optimisations seront apportées à la librairie *VEX* pour les besoins techniques de la méthodologie proposée.

5.10 Synthèse sur la construction du logiciel démonstrateur

Un logiciel démonstrateur pour la méthodologie de HLS proposée a été construit, il est nommé AUGH. Il présente cinq types de transformations différentes, ce qui permet de traiter des circuits de types et de structures très variés. Le logiciel est conçu pour présenter un temps d'exécution relativement faible. Certaines opérations d'optimisation sont implantées afin d'adresser l'objectif de rapidité des circuits générés.

Dans le chapitre suivant, des expériences de synthèse avec ce logiciel démonstrateur sont menées sur plusieurs applications.

Chapitre 6

Expériences et résultats

La méthodologie proposée a été implantée dans un logiciel de HLS démonstrateur. Dans ce chapitre, de nombreuses expériences sont menées avec plusieurs applications de test, afin de tester le comportement de cette méthodologie le plus exhaustivement possible.

Sommaire

6.1 Applications de test et points d'intérêt

La méthodologie proposée a été appliquée sur plusieurs applications, choisies parmi des applications de test représentatives utilisées en HLS. Les applications *adpcm*, *aes*, *blowfish*, *gsm*, *motion*, *sha* sont issues de la suite CHStone [HTHT09]. Les vecteurs de test internes ont été retirés afin de ne considérer que les circuits réels, mais aussi afin d'éviter que des optimisations basées sur les données à traiter ne soient effectuées.

Deux applications supplémentaires ont été utilisées : un décodeur de vidéo au format MJpeg, et la transformée en cosinus inverse discrète 2D IDCT 8×8 (algorithme original de Loeffler), qui est extraite du décodeur MJpeg. Ils illustrent une situation réelle où un accélérateur matériel est généré à partir d'une application logicielle existante. L'ensemble des applications de test comprend des applications purement flot de données ainsi que d'autres plus orientées flot de contrôle.

Le code d'origine des applications a été modifié pour correspondre aux limitations du *parser* de AUGH, qui est hérité de UGH. En particulier, l'arithmétique des pointeurs est interdite, toutes les fonctions doivent être *inline*, et les interfaces externes sont des canaux de type FIFO. L'implémentation actuelle de AUGH exige également que les programmes soient fortement structurés (pas de *goto*, pas de *break* ni *continue* dans les boucles, le *return* n'apparaît qu'à la fin des fonctions, et toutes les branches des *switch* se terminent par un *break*). Cette tâche de réécriture a nécessité environ deux jours de travail, pour l'ensemble des applications. On rappelle que cela est uniquement dû à l'implémentation effectuée dans AUGH, et que cela ne constitue pas une restriction de la méthodologie de HLS proposée. La taille finale du code en langage C varie de 110 lignes (*idct*) à 1200 lines (*mjpeg*). Ces différences de taille illustrent la scalabilité de la méthodologie proposée.

La technologie ciblée est la plus performante de Xilinx à l'heure actuelle : Virtex-7, indice de vitesse 3. Cela permet d'effectuer des comparaisons avec un logiciel de HLS de référence pour cette technologie : la suite Vivado HLS de Xilinx [Xil13a]. Les expériences ont été effectuées sur un ordinateur équipé d'un processeur Intel Core2 Duo E5300 à 2,4GHz. AUGH est actuellement un programme entièrement *monothread*. Les calculs effectués via un *fork* sont également effectués séquentiellement.

Chaque application comprenant du contrôle non connu à la compilation a été annotée à partir des vecteurs de test de la suite CHStone, et à partir d'images de dimension 256×144 pixels pour *mjpeg*. Les applications ne nécessitant pas d'annotations sont *idct*, *aes* et *blowfish*. Les annotations ont été obtenues de la façon suivante, pour chaque application :

1. remplacement des canaux FIFO par les entrée/sortie standard *stdin* et *stdout*,
2. compilation de l'application avec le compilateur *gcc*,
3. exécution du programme obtenu avec les vecteurs de test considérés en entrée,
4. analyse des statistiques d'exécution avec le logiciel *gcov*,
5. insertion des annotations dans le code des applications.

Les logiciels *gcc* et *gcov* sont issus de la *GNU Compiler Collection* [Fre13]. Pour chaque application, l'adaptation du code a demandé au plus 5 minutes, la compilation et l'analyse ont été effectuées en moins d'une seconde, et l'insertion des annotations dans le code de l'application a demandé au plus 20 minutes (pour *mjpeg*). Ce processus pourrait être automatisé, mais ce n'était pas nécessaire dans le cadre de ces expériences.

Les expériences menées ont été focalisées sur plusieurs points d'intérêt :

— l'évolution de la latence et de la surface des solutions considérées durant le processus d'exploration,

— l'impact des annotations sur la latence des circuits finaux,

— la capacité à respecter les contraintes de ressources et à faire bon usage des ressources disponibles,

— la précision des estimateurs employés pour la pondération des transformations possibles,

— l'importance de la précision des pondérations dans le flot d'exploration,

— la précision de l'évaluation des caractéristiques des solutions,

— le temps d'exploration comparé aux logiciels en aval du flot de conception,

— et la comparaison avec d'autres approches en HLS.

Les sections suivantes correspondent à ces différents points d'intérêts.

6.2 Évolution de la latence et de la surface des solutions explorées

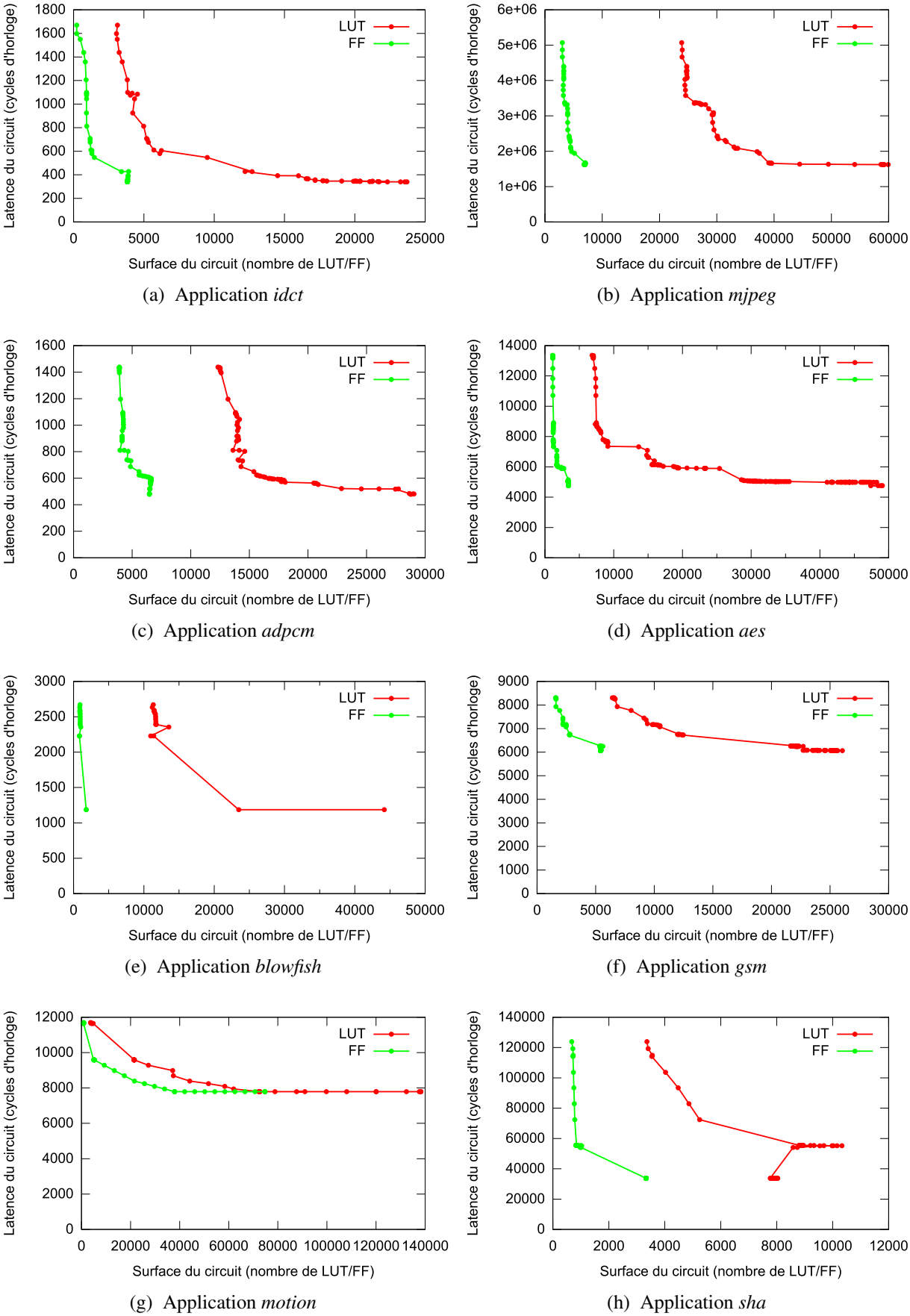
La Figure 6.1 illustre, pour chaque application, l'évolution de la latence et de la surface des solutions explorées par AUGH durant l'exploration. Pour la clarté des résultats, l'utilisation de coeurs DSP et des bancs de mémoire BRAM du FPGA a été désactivée. Les circuits générés étaient donc évalués en surface avec uniquement des LUT et des registres (FF).

La fréquence ciblée a été choisie à 100 MHz, une contrainte en ressources délibérément très haute a été utilisée (le FPGA Xilinx Virtex-7 xc7v585t-3 entier, qui contient 364 200 LUT à 6 entrées et le double en FF), une seule transformation était appliquée par itération, et les pondérations des transformations possibles étaient des estimations (comme décrit précédemment en section 5.6, page 71).

Les figures révèlent que AUGH explore des solutions variées, et que l'exploration s'arrête lorsque toutes les opportunités de parallélisme exploitables ont été utilisés. Pour l'application *mjpeg*, la latence du circuit ne s'améliore qu'imperceptiblement au-delà de 50k LUT (la fin réelle est à ≈300k LUT), la courbe a donc été générée sous une contrainte en ressources à 60k LUT pour révéler la section utile.

AUGH commence par générer un circuit de taille réduite et de faible rapidité, et progresse globalement vers des solutions plus rapides et de plus grande surface. Cependant, cette progression n'est pas toujours strictement monotone, en particulier pour les applications *adpcm* (Figure 6.1c), *blowfish* (Figure 6.1e), et *sha* (Figure 6.1h). Cela illustre que certaines transformations sur la structure du circuit peuvent parfois simultanément l'accélérer et réduire sa taille. Ces résultats illustrent ce qui a été décrit précédemment en section 4.3.3 (page 60).

Les résultats révèlent également que les circuits générés utilisent beaucoup moins de FF que de LUT. Cela est dû au fait que AUGH ne génère actuellement que des circuits non pipelinés.



(a) Application *idct*

(b) Application *mjpeg*

(c) Application *adpcm*

(d) Application *aes*

(e) Application *blowfish*

(f) Application *gsm*

(g) Application *motion*

(h) Application *sha*

FIGURE 6.1 – Évolution de la latence et de la surface des solutions explorées

6.3 Impact des annotations

Les annotations données par l'utilisateur révèlent la contribution relative de chaque bloc de base à la latence moyenne du circuit final. Elles devraient donc guider le logiciel de HLS vers les solutions les plus appropriées aux scénarios d'exécution réels dans lesquels le circuit final est censé être utilisé.

Afin d'observer l'impact des annotations de l'utilisateur sur la latence des circuits générés, AUGH a été spécialement instrumenté pour pouvoir effectuer la pondération des transformations possibles sans tenir compte des annotations. Cela est effectué de la façon suivante :

1. sauvegarde des indications de temps d'exécution de chaque noeud de la représentation interne,
2. re-calcul des temps d'exécution de chaque noeud en appliquant la règle par défaut décrite précédemment en section 4.1.3 (page 48),
3. détection et pondération des transformations possibles,
4. rappel des indications de temps sauvegardées.

Grâce à ce procédé, AUGH peut appliquer les transformations comme si aucune annotation n'était présente, tout en évaluant la latence des solutions en tenant compte des annotations.

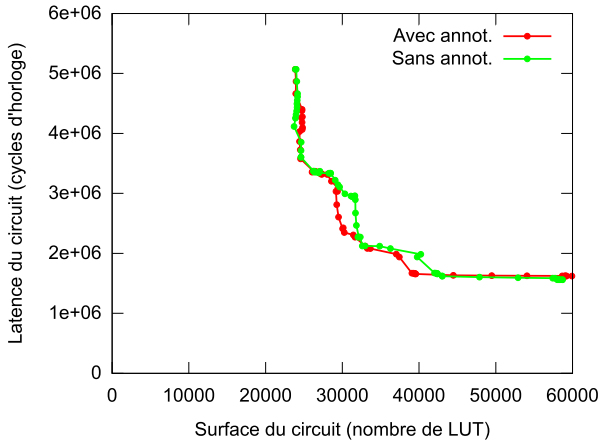
Des expériences dédiées ont été lancées sur les applications pourvues d'annotations (*mjpeg*, *adpcm*, *gsm*, *motion* et *sha*). Ce sont les applications qui possèdent du contrôle dépendant des données. Pour chacune de ces applications, AUGH a été lancé deux fois : une fois normalement, selon le procédé décrit en section 6.2 (page 85), et une fois sans tenir compte des annotations dans la pondération des transformations possibles. Aucune contrainte de ressources n'a été imposée pour observer les résultats sur la plus grande plage possible.

Les résultats de ces expériences sont donnés en Figure 6.2. Comme attendu, en général, les solutions explorées avec annotations sont meilleures, car plus Pareto-optimales, que celles explorées sans annotations. Mais ce n'est pas toujours strictement le cas : les solutions explorées sans annotations sont légèrement meilleures en de rares occasions, par exemple lors des premières itérations pour l'application *mjpeg*, et autour de 12 000 LUT pour l'application *gsm*. Dans ces cas, la différence est minime, et s'explique par le phénomène des minima locaux (expliqué précédemment en section 4.3.2, page 59) : à partir d'un minima local, une décision localement sous-optimale peut conduire à une meilleure progression qu'une décision localement optimale.

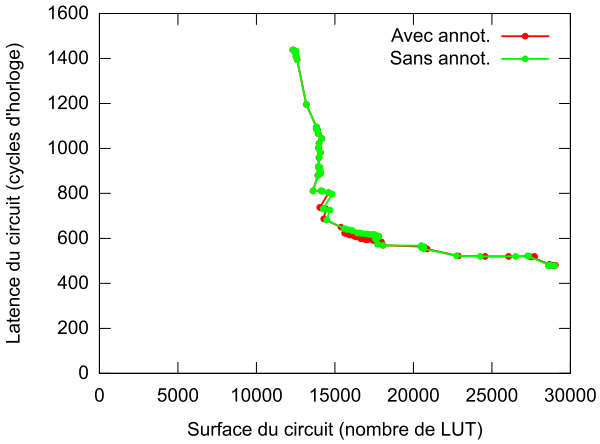
Les courbes de la Figure 6.2 suggèrent que, sans annotations, les circuits générés peuvent être moins bons que ceux obtenus avec annotations. Cependant, ces courbes seules sont insuffisantes pour conclure, car il est possible que les solutions explorées avec des contraintes en ressources soient différentes de celles explorées sans contrainte, notamment au voisinage desdites contraintes. Les deux courbes pourraient se rejoindre et les solutions finales pourraient donc être semblables.

De nouvelles expériences ont été lancées sur les applications présentant l'écart le plus visible entre les deux courbes : *mjpeg* (contrainte à 32k LUT), *motion* (contrainte à 80k LUT) et *sha* (contrainte à 9 850 LUT). Les résultats sont donnés en Figure 6.3.

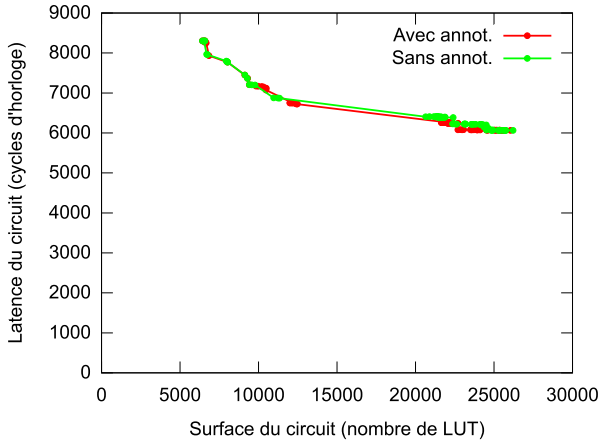
Pour les applications *motion* et *sha*, la courbe des solutions explorées est similaire à celle obtenue précédemment sans contrainte, et semble n'être qu'une simple troncature au niveau de la



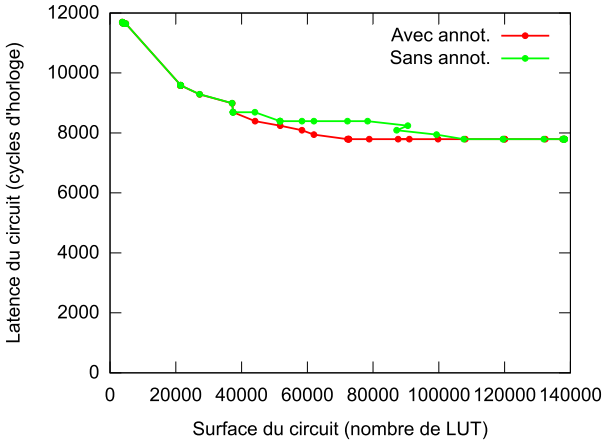
(a) Application *mjpeg*



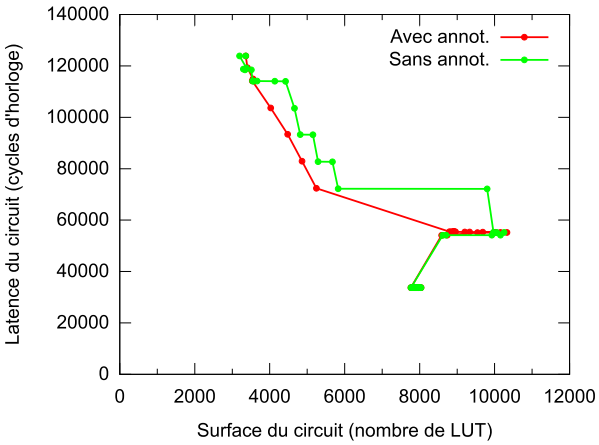
(b) Application *adpcm*



(c) Application *gsm*

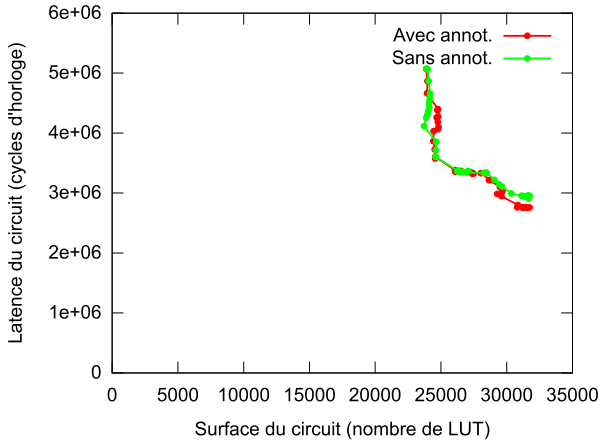


(d) Application *motion*

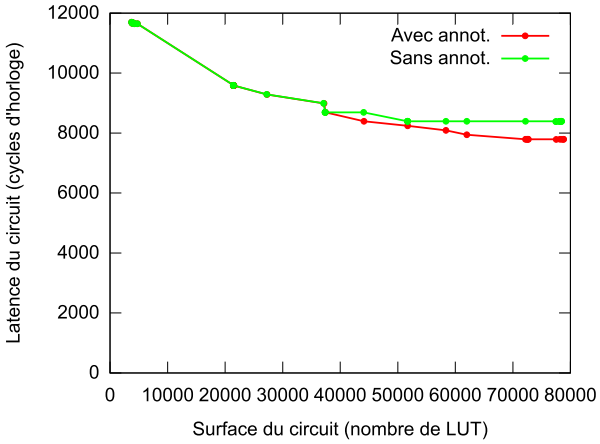


(e) Application *sha*

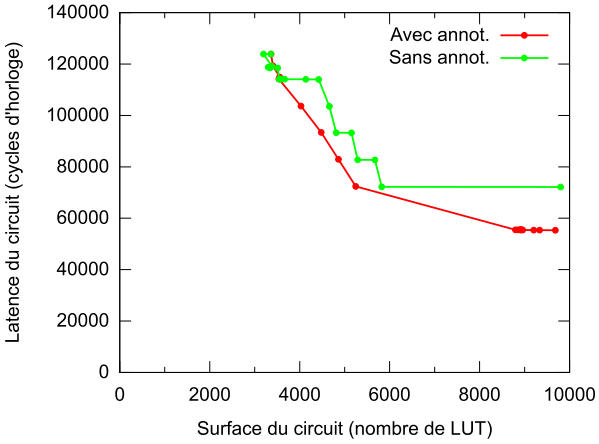
FIGURE 6.2 – Solutions explorées avec et sans annotations, sans contrainte en ressources



(a) Application *mjpeg*, contrainte à 32k LUT



(b) Application *motion*, contrainte à 80k LUT



(c) Application *sha*, contrainte à 9850 LUT

FIGURE 6.3 – Solutions explorées avec et sans annotations, avec contrainte en ressources

contrainte. Les solutions finales obtenues sans annotations présentent ainsi une latence notablement supérieure aux solutions finales obtenues avec annotations (+7,7 % pour *motion* et +30 % pour *sha*). Ces deux expériences confirment que des annotations réalistes sont parfois nécessaires afin d'obtenir les circuits les plus performants. Le cas de l'application *sha*, pour laquelle la progression de l'utilisation en ressources n'est pas monotone, illustre également un cas où une contrainte en ressources empêche AUGH de trouver des solutions simultanément plus rapides et de plus faible surface (comme décrit précédemment en section 4.3.3, page 60).

Le cas de l'application *mjpeg* est différent puisque, au voisinage de la contrainte, les solutions explorées ne sont pas les mêmes que celles explorées sans contrainte. L'écart entre les solutions finales est moindre (de 24 %) que l'écart visible sur les courbes obtenues sans contrainte. De plus, la solution obtenue avec annotations est plus lente (de 22 %) que celle de surface équivalente explorée sans contrainte. Cela illustre que la fonction de sélection des transformations, plus particulièrement la formule de pondération globale employée, (décrite précédemment en section 5.7, page 79), impacte sur la performance des circuits finaux. Cela pourrait être compensé

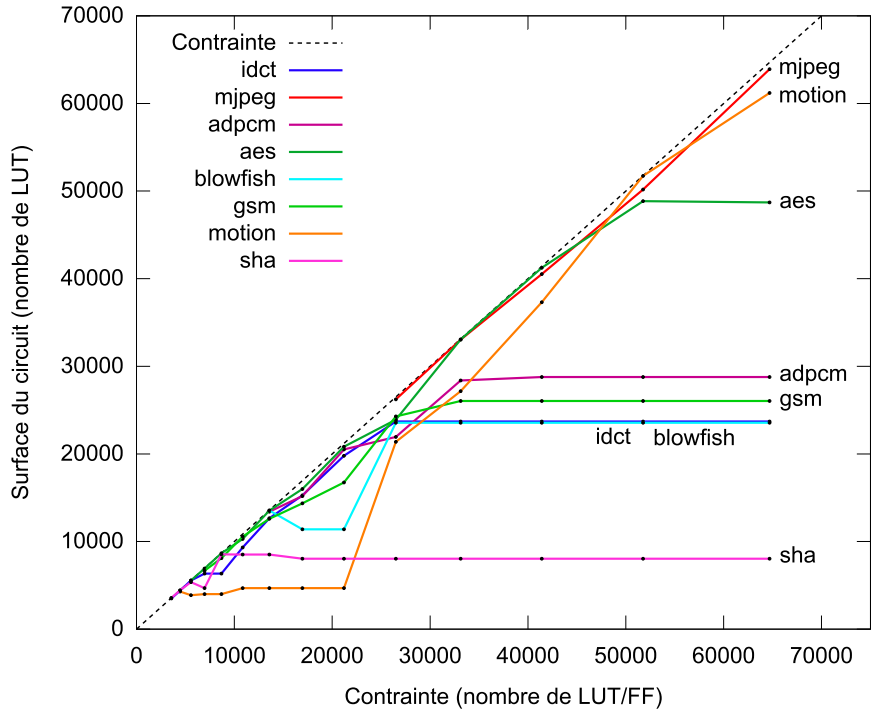


FIGURE 6.4 – Capacité de AUGH à suivre des contraintes de ressources

en effectuant un raffinement supplémentaire au voisinage de la solution finale, comme mentionné précédemment en section 4.4.4 (page 63).

6.4 Respect des contraintes de ressources

Chaque application a été synthétisée avec différentes contraintes de ressources, afin d'observer comment AUGH les respecte et comment ces ressources sont utilisées. Les résultats sont présentés en Figure 6.4. Plusieurs observations peuvent être faites.

Premièrement, comme attendu, AUGH n'a jamais dépassé la contrainte imposée.

Deuxièmement, pour chaque application, il existe une zone de *saturation*. Cette zone correspond aux cas où AUGH a appliqué toutes les transformations possibles au circuit. AUGH est donc incapable de produire un circuit plus gros (donc en général plus rapide) et, pour toute valeur supérieure de la contrainte en ressources, AUGH produit toujours le même circuit final. Pour chaque application, le début de la zone de saturation correspond au circuit final illustré en Figure 6.1. Par souci de clarté, la Figure 6.4 ne s'étend pas jusqu'aux zones de saturation des applications *mjpeg* et *motion*, qui commencent respectivement à environ 300k LUT et 140k LUT.

Troisièmement, dans la zone non saturée, les solutions générées par AUGH suivent grossièrement la contrainte. En effet, en général, la réduction de la latence du circuit est obtenue en augmentant sa taille. AUGH applique donc le plus de transformations possibles en-dessous de

la contrainte. De ce point de vue, la contrainte est un objectif à atteindre. Mais parfois, cet objectif ne peut pas être atteint, en particulier pour les applications *blowfish* et *motion*, autour de 20k LUT. Dans ces deux situations, les seules transformations disponibles ont un coût jugé trop élevé par AUGH. Elles ne sont appliquées que pour des contraintes en ressources supérieures.

Enfin, des minima locaux sont parfois observés, notamment pour les application *blowfish* et *sha*. Dans ces minima locaux, la progression de l'utilisation en ressources n'est pas monotone. Il y a donc une corrélation avec les courbes de la Figure 6.1.

6.5 Précision des estimateurs

Les biais des estimateurs sont mesurés comme l'écart entre les valeurs estimées et les valeurs obtenues avec la méthode précise utilisant *fork* (décrite précédemment en section 5.8, page 80). Afin de mesurer ces biais, des expériences dédiées ont été effectuées : AUGH a été lancé sur chaque application, en sélectionnant une seule transformation par itération. Le choix est basé sur les pondérations estimées. Chaque transformation sélectionnée est également pondérée de façon précise, avec la méthode utilisant *fork*.

La Figure 6.5 illustre, pour les deux types de transformation les plus fréquents (extension de l'allocation et déroulement de boucle) et pour chaque dimension (coût en LUT et en FF, gain en cycles d'horloge), les écarts mesurés sur l'ensemble des applications. La droite indiquant la valeur idéale (si les estimations étaient exactes) est également représentée. Les résultats complets, pour tous les types de transformation, sont donnés en Annexe A.

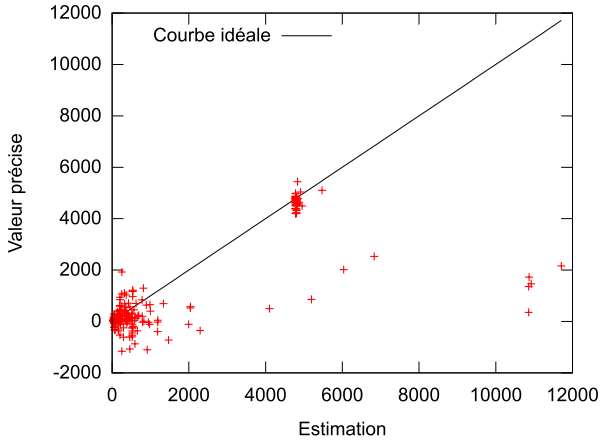
Les résultats varient notablement entre les types de transformation. Comme observations générales, on remarque que les estimations sont rarement exactes. Pour les LUT, des écarts à la courbe idéale sont souvent causés par des changements non prévus sur la taille des multiplexeurs dans le circuit. Cela entraîne une variation non prévue de la latence combinatoire de certaines instructions. Lorsque la latence augmente et nécessite qu'un cycle d'horloge supplémentaire soit utilisé, un état de FSM est ajouté, ce qui coûte une FF. De même, des états de FSM peuvent être économisés.

Une autre cause est le fait qu'après application d'une transformation, quelle qu'elle soit, une passe de simplification des instructions des blocs de base est effectuée (comme mentionné précédemment en section 5.9, page 81). Cela peut aboutir à la suppression de registres temporaires devenus inutiles, d'où une économie en FF et une simplification des multiplexeurs associés.

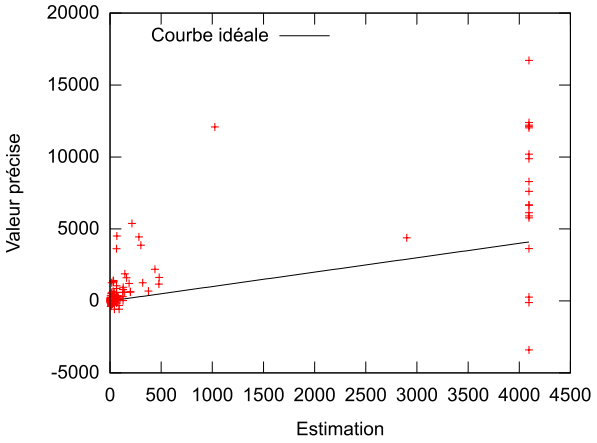
Extension de l'allocation

Les estimations en LUT (Figure 6.5a) sont globalement regroupées en deux masses de points, chacune est entourée de points localement diffus, dus aux effets non prévus sur les multiplexeurs. Le groupe de points autour de 4 500 LUT correspond aux ajouts de multiplieurs, qui consomment une grande quantité de LUT (puisque l'usage des DSP a été interdit pour ces expériences).

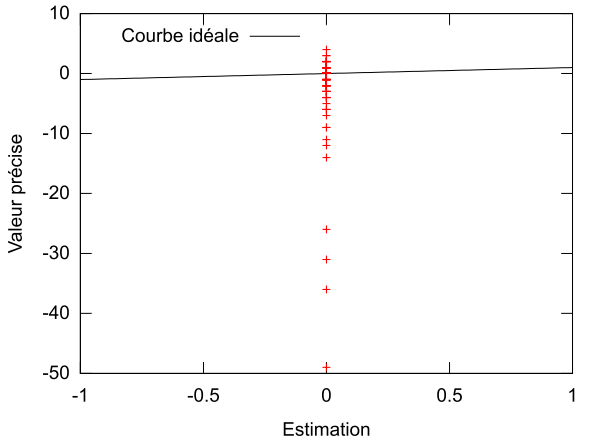
Les estimations en FF (Figure 6.5c) sont toujours nulles car les opérateurs ciblés n'utilisent que des LUT. Mais un coût (ou un gain) très modéré en FF peut tout de même être observé, à



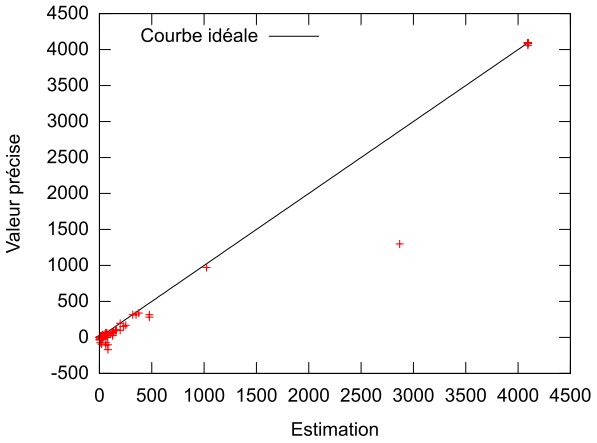
(a) Extension alloc. (LUT)



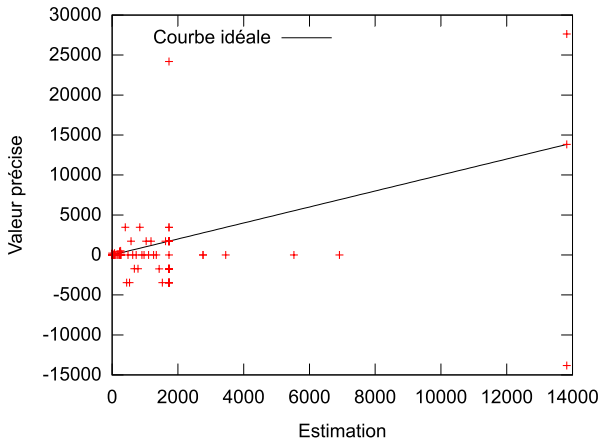
(b) Déroulement de boucle (LUT)



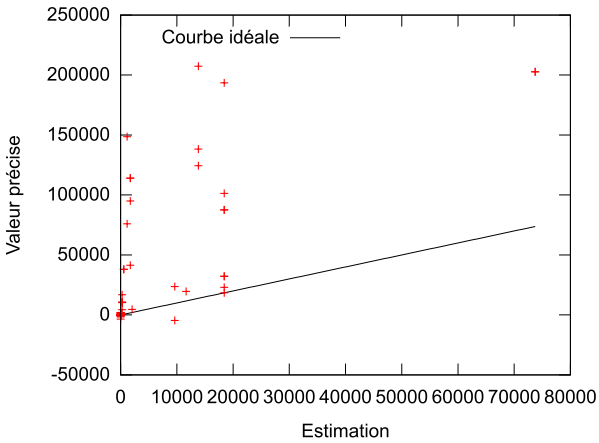
(c) Extension alloc. (FF)



(d) Déroulement de boucle (FF)



(e) Extension alloc. (cycles)



(f) Déroulement de boucle (cycles)

FIGURE 6.5 – Biais des estimateurs

cause des changements non prévus sur la latence de certaines instructions et la simplification de certains registres.

Pour l'estimateur en latence (Figure 6.5e), on remarque un certain nombre de points sur la valeur zéro en ordonnée. Dans ces cas, des opérateurs ou ports sont ajoutés, l'exécution de certaines instructions est effectivement avancée, mais cela reste sans effet sur la latence globale du circuit. Cela arrive lorsque les instructions avancées ne sont pas seules dans leur cycle d'horloge. La valeur fournie par l'estimateur a une signification probabiliste, comme décrit précédemment en section 5.6.1 (page 71).

Déroulement de boucle

L'estimateur en FF (Figure 6.5d) est relativement précis. En effet, il mesure l'impact de la transformation en termes d'états économisés dans la FSM. L'estimateur applique un ré-ordonnancement de la boucle déroulée, le résultat est donc relativement précis. Les écarts correspondent aux simplifications non prévues sur les registres.

En revanche, les estimations en LUT (Figure 6.5b) et en latence (Figure 6.5f) semblent très imprécises. L'implémentation des estimateurs devra sans doute être revue.

6.6 Impact de la précision des pondérations

Afin d'observer l'impact de la précision des pondérations sur la latence des circuits générés, de nouvelles expériences ont été effectuées avec différentes stratégies d'exploration. Il y a deux principales sources d'approximation : le fait d'appliquer plusieurs transformations par itération, et le fait de générer les pondérations des transformations avec des estimateurs. Afin d'isoler ces deux contributions, les quatre stratégies suivantes ont été employées :

— S.P. : Une unique transformation est appliquée par itération, et les pondérations sont obtenues avec la méthode précise utilisant *fork* (comme décrit en section 5.8, page 80).

— M.P. : Plusieurs transformations peuvent être appliquées à chaque itération (comme décrit précédemment en section 5.7, page 79), et les pondérations sont obtenues via un *fork*.

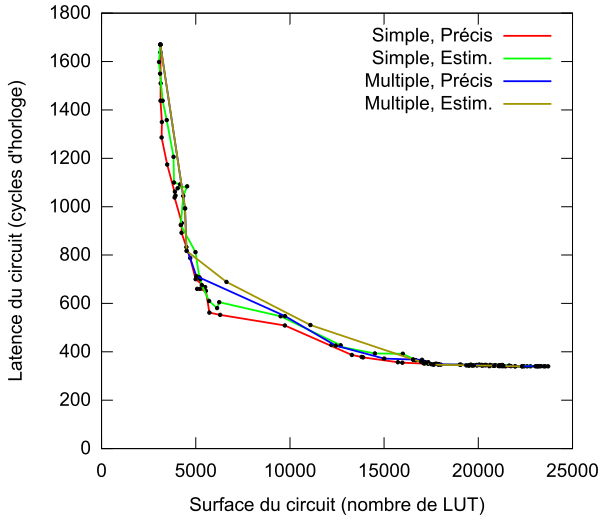
— S.E. : Une unique transformation est appliquée par itération, et les pondérations sont obtenues par estimation (comme décrit précédemment en section 5.6, page 71).

— M.E. : Plusieurs transformations peuvent être appliquées à chaque itération, et les pondérations sont des estimations.

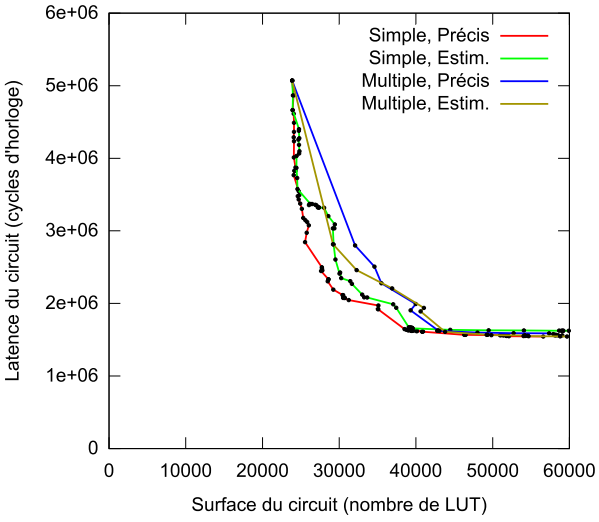
Toutes les applications ont été traitées par AUGH avec ces différentes stratégies, et sans contrainte en ressources. Les résultats sont illustrés en Figures 6.6 et 6.7, et les détails sont donnés en Tableau 6.1 et Tableau 6.2.

Pour chaque application, on observe (Figures 6.6 et 6.7) que les solutions explorées par les différentes stratégies d'exploration sont ne sont pas les mêmes.

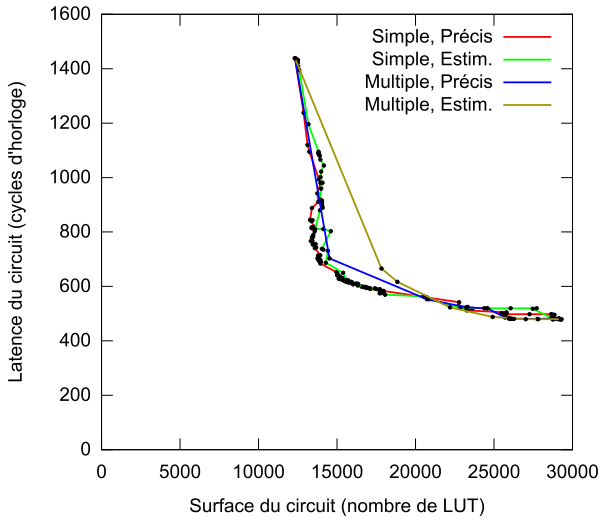
Cependant, dans la majorité des cas, la latence et la surface des circuits finaux obtenus avec les différentes stratégies sont très similaires. Ce résultat était attendu. En effet, sans contrainte



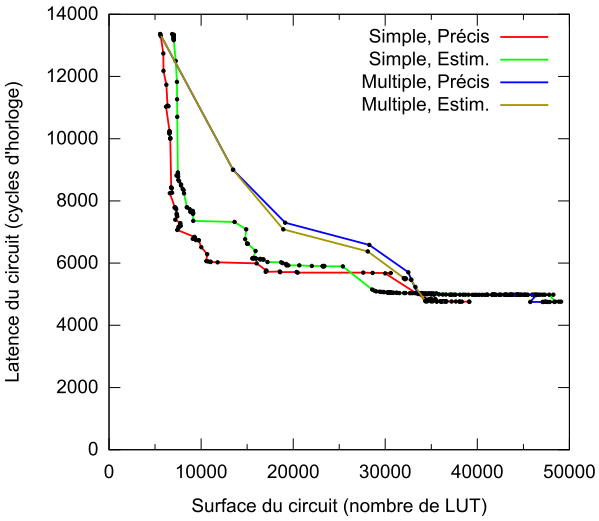
(a) Application *idct*



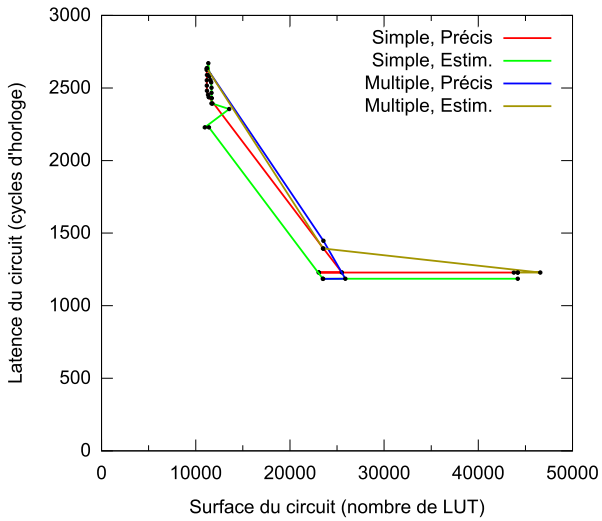
(b) Application *mjpeg*



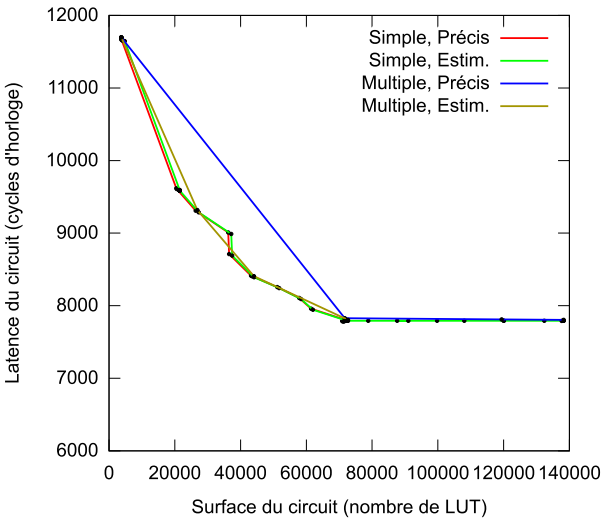
(c) Application *adpcm*



(d) Application *aes*

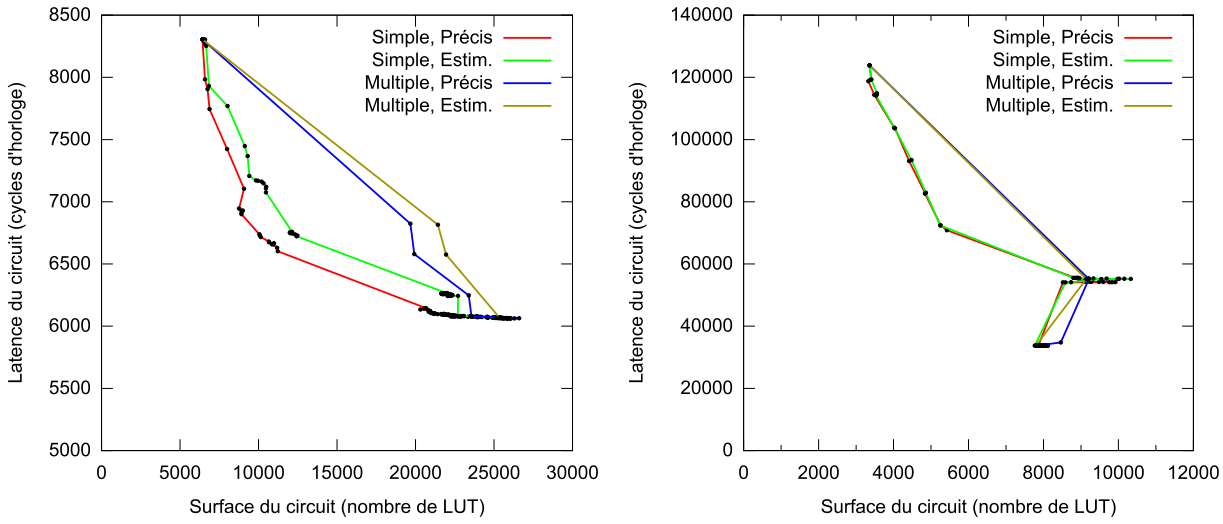


(e) Application *blowfish*



(f) Application *motion*

FIGURE 6.6 – Différentes stratégies d'exploration, sans contrainte en ressources



(a) Application *gsm* (b) Application *sha*

FIGURE 6.7 – Différentes stratégies d'exploration, sans contrainte en ressources (suite)

TABLE 6.1 – Différentes stratégies, pas de contrainte : surface (en LUT)

Application	idct	mjpeg	adpcm	aes	blowfish	gsm	motion	sha
Stratégie S.P.	22 429	56 585	28 857	39 132	44 180	26 009	71 390	8 120
Stratégie M.P.	22 326	59 687	29 270	36 600	44 180	25 891	71 390	8 052
Stratégie S.E.	23 707	59 928	28 772	48 855	44 181	26 048	137 655	8 039
Stratégie M.E.	23 120	59 066	29 145	47 869	23 517	25 842	138 255	8 054

en ressources, toutes les transformations possibles sont appliquées. L'ordre dans lequel elles sont appliquées semble donc, en général, n'avoir que peu d'importance. Mais les circuits ne sont toutefois pas strictement identiques. Les légères variations observées sur la taille et la latence des circuits finaux sont dues aux simplifications itératives des instructions des blocs de base. En effectuant les transformations dans des ordres différents, les simplifications effectuées varient, ainsi que la forme des expressions à calculer. L'opération d'assignation (qui crée également les multiplexeurs et les opérateurs logiques) produit donc des chemins de données légèrement différents, même si la fonctionnalité est identique.

Les applications *blowfish* et *motion* sont toutefois de notables exceptions. Pour *blowfish*, l'utilisation finale en LUT pour la stratégie M.E. est presque deux fois plus faible qu'avec toutes les autres stratégies. La Figure 6.6e montre qu'en réalité, pour les autres stratégies, AUGH a appliqué des transformations supplémentaires, qui visiblement n'apportent aucun gain en latence. Pour *motion*, l'utilisation finale en LUT pour les stratégies S.P. et M.P est presque deux fois plus faible qu'avec les autres stratégies. La Figure 6.6f montre une situation similaire au cas de *blowfish* : l'utilisation de ressources supplémentaires par les stratégies S.E. et M.E. est inutile. Ces situations sont dues au fait que AUGH considère même les transformations pour lesquelles les pondérations indiquent qu'elles n'apporteront aucun gain en latence, car d'autres transformations plus intéressantes pourraient être détectées par la suite. Dans ces situations, les légères variations

TABLE 6.2 – Différentes stratégies, pas de contrainte : latence (en cycles d'horloge)

Application	idct	mjpeg	adpcm	aes	blowfish	gsm	motion	sha
Stratégie S.P.	340	1 544 860	495,8	4 758	1 229	6 063,1	7 786	33 740
Stratégie M.P.	340	1 544 230	477,8	4 760	1 229	6 063,1	7 786	33 740
Stratégie S.E.	340	1 622 820	478,8	4 759	1 186	6 063,1	7 791	33 741
Stratégie M.E.	340	1 560 020	478,8	4 752	1 185	6 063,1	7 804	33 740

de la forme de la représentation interne selon les stratégies peuvent mener à des pondérations différentes, d'où une différence au niveau des transformations appliquées. Ces cas sont toutefois rares.

TABLE 6.3 – Différentes stratégies, sans contrainte : temps d'exploration (en secondes)

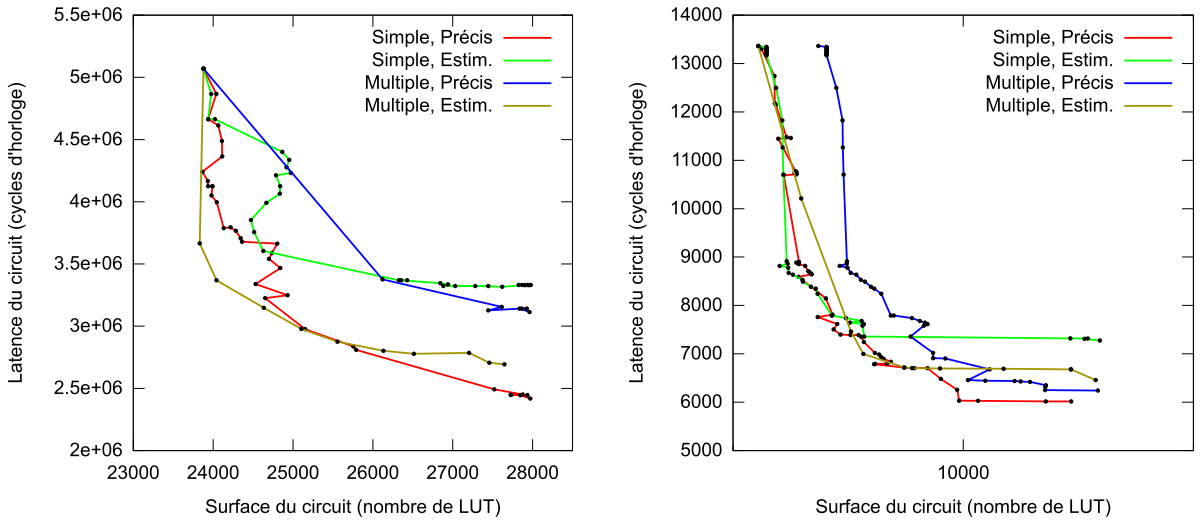
	idct	mjpeg	adpcm	aes	blowfish	gsm	motion	sha	Moy. géom.
S.P.	60,8	815	794	374	13,1	1 870	1 530	30,9	254
M.P.	50,3	105	50,3	208	3,93	72,8	402	5,55	49,4
S.E.	17	66	103	354	2,47	181	286	6,82	49,4
M.E.	14,4	40,5	13,5	266	1,98	23	31,1	2,78	17,4
S.P. / M.E.	4,24	20,1	58,7	1,41	6,63	81,3	49,1	11,1	14,6
M.P. / M.E.	3,5	2,59	3,72	0,781	1,99	3,16	12,9	2	2,84
S.E. / M.E.	1,19	1,63	7,6	1,33	1,25	7,83	9,2	2,46	2,85

Les temps d'exploration par AUGH sont donnés en Tableau 6.3. Comme attendu, dans la majorité des cas, la stratégie S.P. est la plus lente et M.E. est la plus rapide. Avec des rapports entre les temps d'exploration atteignant 14x, la stratégie M.E. semble extrêmement intéressante pour le critère du temps de génération.

Les Figures 6.6 et 6.7 révèlent cependant une autre différence entre les différentes stratégies : l'optimalité (selon le critère de Pareto) des solutions explorées. En général, les solutions explorées par la stratégie S.P. sont meilleures que celles explorées avec les autres stratégies. Cela est particulièrement visibles pour les applications *mjpeg* (Figure 6.6b) et *aes* (Figure 6.6d). Sous certaines contraintes en ressources, notamment là où l'écart entre les courbes des solutions explorées par les différentes stratégies est le plus grand, cela pourrait mener à des solutions finales présentant des différences en latence notables.

Afin d'analyser cette situation, de nouvelles expériences ont été effectuées. L'application *mjpeg* a été synthétisée sous une contrainte de 28k LUT, et l'application *aes* a été synthétisée sous une contrainte de 13k LUT. Les résultats sont illustrés en Figure 6.8 et les détails sont donnés en Tableau 6.4. Les temps d'exploration par AUGH sont donnés en Tableau 6.5.

Les circuits générés avec les différentes stratégies sont maintenant de latences très différentes. Cela illustre qu'avec la méthodologie proposée, la stratégie d'exploration importe sur la rapidité des circuits. Les courbes confirment que la stratégie S.P. donne en général les meilleurs résultats. Pour l'utilisateur, il y a donc un compromis à faire entre la rapidité du flot et l'assurance d'obtenir des circuits optimisés.



(a) Application *mjpeg* (b) Application *aes*

FIGURE 6.8 – Différentes stratégies d'exploration, avec contraintes en ressources

TABLE 6.4 – Différentes stratégies, avec contrainte

	Latence (cycles)		Surface (LUT)	
Application	mjpeg	aes	mjpeg	aes
Stratégie S.P.	2 417 890	6 017	27 972	12 344
Stratégie M.P.	2 693 090	6 456	27 647	12 877
Stratégie S.E.	3 330 490	7 276	27 952	12 968
Stratégie M.E.	3 113 420	6 242	27 962	12 923

TABLE 6.5 – Différentes stratégies, avec contrainte : temps d'exploration (en secondes)

	mjpeg	aes	Moy. géom.
S.P.	346	145	224
M.P.	78	24,9	44,1
S.E.	20,5	9,53	14
M.E.	7,4	11,4	9,17
S.P. / M.E.	46,8	12,7	24,4
M.P. / M.E.	10,5	2,19	4,81
S.E. / M.E.	2,78	0,838	1,53

Cependant, au voisinage de la contrainte, la forme des courbes suggère que, dans le cas général et avec l'implémentation proposée, on ne peut pas présumer des latences relatives obtenues avec les autres stratégies.

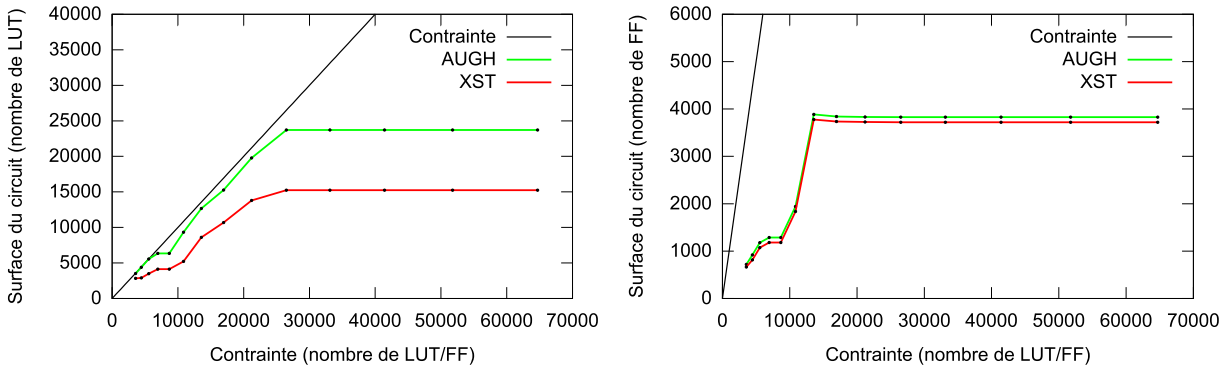
Comparé à un processus de compilation, la stratégie M.E. correspond au mode par défaut, où les efforts d'optimisation sont faibles (par exemple le mode *O0* pour *gcc*). La stratégie S.P. est un mode optimisé (par exemple le mode *O2* ou *O3* pour *gcc*), plutôt destinée aux utilisateurs prêts

à en payer le coût en temps d'exploration.

6.7 Précision de l'évaluation des circuits

La méthodologie de HLS proposée a été conçue avec notamment comme exigence que les circuits générés respectent strictement les contraintes matérielles spécifiées par l'utilisateur. En section 6.4 (page 90), il a été montré que les circuits obtenus après l'exploration des solutions sont conformes aux contraintes de ressources, mais seulement selon le processus d'évaluation interne à AUGH.

Or, afin de générer les données de programmation du FPGA ciblé, la description RTL générée par AUGH doit passer par des étapes de traitement aval supplémentaires : la synthèse logique, le placement et le routage. Il est donc très important de vérifier également si les évaluations de AUGH sont toujours valides après ces étapes de traitement aval. Seul le résultat après synthèse logique est abordé, faute de temps. Pour cela, le logiciel de synthèse logique de référence pour la technologie ciblée (Xilinx Virtex-7) a été utilisé : Xilinx XST.



(a) Surface après synthèse logique (LUT)

(b) Surface après synthèse logique (FF)

FIGURE 6.9 – Surface après synthèse logique

Tout d'abord, une expérience a été menée sur l'application *idct*, pour plusieurs contraintes en ressources (stratégie S.E.). Les solutions générées par AUGH pour chaque contrainte sont synthétisées par XST. Les résultats sont illustrés en Figure 6.9. XST a été configuré de façon à "mettre à plat" la hiérarchie de la description RTL. Cela consiste à fusionner la fonctionnalité de tous les composants, ce qui permet d'optimiser globalement la surface et la latence du circuit. Plusieurs observations peuvent être faites. Premièrement, la surface du circuit obtenu en sortie de XST est toujours inférieure à la surface prévue par AUGH. Les contraintes de ressources sont donc toujours respectées, même après synthèse logique. Même imprécise, l'évaluation de AUGH est fidèle. Deuxièmement, la surestimation de AUGH est beaucoup plus importante pour les LUT (+55,5 %) que pour les FF (+2,9 %). En effet, les opportunités de fusion (dans les LUT) des fonctionnalités des composants logiques et des multiplexeurs sont en général beaucoup plus nombreuses que les opportunités de simplification des registres ou du registre d'état de la FSM (seuls composants utilisant des FF). Enfin, la surestimation de l'utilisation en LUT par AUGH est importante. Les modèles de composants de AUGH, relativement simples, mènent ici à une

situation où une quantité non négligeable de ressources matérielles reste inutilisée alors qu'elle pourrait servir à accélérer davantage le circuit.

TABLE 6.6 – Rapport entre AUGH et XST

		idct	mjpeg	adpcm	aes	blowfish	gsm	motion	sha	Moy. géom.
LUT	Plat	1,55	1,08	1,61	2,42	3,96	2,57	3,2	1,67	2,08
	Hier.	1,22	0,941	1,37	1,67	1,72	1,81	1,28	1,5	1,41
FF	Plat	1,03	1,07	1,21	1,4	1,36	1,05	1,24	1	1,16
	Hier.	1	1,06	1,13	1,21	1,3	1,01	1,14	1	1,1
Latence combi.	Plat	0,953	0,035 3	0,642	1,04	1,36	0,988	1,07	1,15	0,663
	Hier.	0,771	0,033 1	0,656	0,904	0,935	0,898	0,888	1,12	0,579

Le Tableau 6.6 donne, pour chacune des applications considérées, le rapport entre les valeurs prévues par AUGH et celles obtenues après synthèse par XST. Les métriques considérées sont la surface, en LUT et FF, et la latence combinatoire maximale. Deux configurations différentes de XST ont été utilisées : dans l'une, XST conserve la hiérarchie des composants, et dans l'autre le circuit complet est "mis à plat". Les données sont celles des circuits générés par AUGH à la fin du processus d'exploration, car c'est en général le pire cas.

Afin de garantir le respect des contraintes matérielles, AUGH devrait légèrement surévaluer l'utilisation en ressources et les latences combinatoires. Les valeurs du Tableau 6.6 devraient donc notamment être toutes supérieures à 1. C'est le cas pour l'utilisation en ressources, sauf pour un unique cas : l'utilisation en LUT pour l'application *mjpeg* lorsque la hiérarchie est conservée. Toutefois, avec un facteur de 2,08x sur les LUT et 1,16x sur les FF, la précision des évaluations faites par AUGH pourrait être grandement améliorée.

L'évaluation de la latence combinatoire par XST doit être analysée différemment. Pour presque toutes les applications (*sha* fait exception), il semble que AUGH sous-estime les valeurs. Cependant, dans le cas de *mjpeg*, cette sous-estimation serait d'un facteur 30, ce qui est démesuré. Mais la cause principale ne vient pas d'une sous-estimation de la part de AUGH. En réalité, les valeurs de latence combinatoire fournies par XST ne peuvent pas être comparés à ceux de AUGH, car ces deux logiciels ne mesurent pas la même chose. En effet, les opérateurs manipulés par AUGH sont partagés entre les instructions exécutées par le circuit, et ces opérateurs peuvent être chaînés dans un même cycle. Cela signifie, même si seuls quelques opérateurs sont effectivement utilisés dans chaque instruction, il peut exister des chemins combinatoires traversant un très grand nombre d'opérateurs, en passant par les multiplexeurs associés. Or, XST mesure la latence du pire cas, c'est-à-dire celle du chemin combinatoire traversant le plus d'opérateurs (jusqu'à 1 500 niveaux de logique sont détectés pour le *mjpeg*).

En conséquence, les valeurs de latence indiquées par XST ne représentent pas la latence combinatoire des instructions manipulées par AUGH. Afin d'obtenir des valeurs fiables, un logiciel d'analyse différent devrait être utilisé, en spécifiant précisément les chemins de données à prendre en compte (le logiciel *trace* pour les technologies Xilinx). De telles valeurs pourraient alors être utilisées directement dans une étape de *retiming* après la génération du circuit, comme décrit précédemment en section 4.1.4 (page 50). Cela n'a pas pu être effectué dans le cadre des présents travaux, faute de temps.

Le Tableau 6.6 montre également que les valeurs fournies par XST sont fortement dépendantes de la configuration employée (ici, avec ou sans conservation de la hiérarchie des circuits). Cela confirme que les logiciels utilisés en aval de la HLS, et leur configuration, font partie des contraintes matérielles à gérer au niveau de la HLS.

6.8 Temps d'exploration comparé aux logiciels aval

AUGH n'est pas le seul logiciel présent dans le flot de génération. AUGH génère un circuit au niveau RTL, en langage VHDL. Ce code VHDL doit passer par un logiciel de synthèse logique, puis un logiciel de placement et de routage. Dans les mêmes conditions expérimentales que pour la section précédente (page 98), le temps d'exploration par AUGH est comparé au temps de synthèse logique par XST. Le placement et le routage des circuits n'ont pas (encore) été effectués, faute de temps.

TABLE 6.7 – Rapport de temps d'exécution AUGH / XST

	idct	mjpeg	adpcm	aes	blowfish	gsm	motion	sha	Moy. géom.
Plat	0,098 8	0,433	0,545	1,23	0,253	1,06	0,173	0,098 4	0,327
Hier.	0,221	0,681	1,09	1,85	0,256	1,41	0,192	0,119	0,473

Les résultats sont donnés en Tableau 6.7. Le traitement de AUGH est, en moyenne, 2 à 3 fois plus rapide que le traitement de XST, selon la configuration. Même sans avoir le temps de placement et routage (qui est généralement beaucoup plus élevé que la synthèse logique), on peut conclure que le temps d'exploration de AUGH est faible devant celui du traitement des logiciels aval.

La méthodologie proposée est donc très rapide. Cela peut être interprété de deux façons différentes. La première interprétation est que, avec un logiciel de HLS tel que AUGH, si les utilisateurs d'accélérateurs matériels sur FPGA souhaitent accélérer encore plus le flot de conception global, alors il faut employer des logiciels aval plus rapides, même si cela implique de faire de nouveaux compromis sur l'optimalité des résultats. Mais les seuls logiciels existants sont propriétaires, donc le fait que cela soit possible ou non reste inconnu. Dans la seconde interprétation possible, il y a une marge confortable pour améliorer l'implémentation proposée dans AUGH (l'emploi d'algorithmes de complexité raisonnablement plus élevée peut être considéré). Les principales pistes d'amélioration sont les suivantes :

— ajouter de nouveaux types de transformation (par exemple l'insertion de registres temporaires),
— utiliser des composants pipelinés,
— améliorer la précision des estimateurs,
— améliorer la précision de l'évaluation des caractéristiques des circuits,
— employer une technique de *retiming* plus élaborée, en particulier garantir la fréquence de fonctionnement même après placement et routage (la méthode employée dans UGH est décrite dans [AP08]),
— ajouter une étape de raffinement autour de la solution finale (mentionné précédemment en section 4.4.4, page 63).

6.9 Comparaison avec d'autres approches

TABLE 6.8 – Temps d'exploration par AUGH et nombre d'itérations

Application	idct	mjpeg	adpcm	aes	blowfish	gsm	motion	sha
Lignes de code C	113	1193	710	701	275	433	225	197
Nombre d'itérations	47	54	74	145	16	88	41	42
Temps d'exploration (s)	17	66	103	354	2,47	181	286	6,82
Temps moy. par itér. (s)	0,36	1,22	1,39	2,44	0,15	2,06	6,98	0,16

Le Tableau 6.8 détaille le temps d'exploration par AUGH pour chaque application (stratégie S.E.). AUGH effectue l'exploration des solutions en moyenne en 49,4 secondes (voir Tableau 6.3). L'application *mjpeg* est traitée beaucoup plus rapidement que dans [CHD+12] (66 secondes vs. 45 minutes). De plus, dans le cas présent, c'est l'application entière qui est traitée et non une paire d'accélérateurs matériels extraits manuellement.

Vivado HLS

D'autres comparaisons ont été effectuées avec le logiciel de synthèse de haut niveau Vivado HLS de Xilinx, pour les applications *idct* et *mjpeg*.

Pour *idct*, Vivado HLS détecte correctement le nombre d'itérations de toutes les boucles. Une opération de synthèse dure environ 59 secondes, ce qui est notablement plus lent qu'une exploration complète par AUGH (moins de 17 secondes), mais plus rapide qu'une synthèse logique par XST (quelques minutes). Vivado HLS produit une unique solution par lancement (manuel) et indique sa latence (en cycles d'horloge) et la consommation en ressources matérielles.

TABLE 6.9 – Circuits obtenus par AUGH et Vivado HLS pour l'application *idct*

	LUT	FF	DSP	BRAM	Latence (cycles)
AUGH	20 104	3 927	20	0	340
Vivado HLS	21 393	14 656	448	2	261

En effectuant différents essais en variant les consignes, l'utilisateur peut explorer différentes solutions. Vivado HLS a été lancé avec pour consigne de dérouler toutes les boucles, et sans contrainte sur le nombre d'opérateurs. AUGH a été lancé sans contraintes en ressources et en activant l'utilisation des blocs DSP. Les résultats sont donnés en Tableau 6.9. La solution produite par AUGH est moins rapide (+30,3 % par rapport à Vivado HLS), mais sensiblement plus réduite en surface. En réalité, AUGH est ici limité par le nombre de transformations implémentées et la qualité des simplifications des instructions.

Pour *mjpeg*, qui comprend de très nombreux branchements conditionnels, Vivado HLS ne fournit pas d'estimation du temps d'exécution moyen du circuit généré. Donc pour cette application, et pour toutes celles décrites avec un flot de contrôle non connu à la compilation, l'utilisateur

ne peut pas rechercher les solutions Pareto-optimales sans effectuer une simulation de la version RTL de chaque solution considérée. Chaque solution est générée en environ 13 secondes, ce qui est plus rapide qu'une exploration complète AUGH (66 secondes avec la stratégie S.E.).

Cependant, effectuer plusieurs itérations avec Vivado HLS peut prendre beaucoup plus de temps qu'une exploration entière par AUGH. Par exemple, quatre solutions ont pu être générées avec Vivado HLS en environ 20 minutes (sans effectuer de simulations RTL). Pour cela, les transformations qui semblaient les plus appropriées ont été appliquées, ce qui a nécessité de bien comprendre les points chauds de l'application.

D'après la Figure 6.1b (page 86), chaque solution devrait également être simulée sur environ 1,5 million à 5 millions de cycles d'horloge. En supposant une vitesse de simulation RTL de 500 cycles/s (vitesse atteinte par le simulateur GHDL [Gin13]), cela représenterait 3 000 à 10 000 secondes (environ une à trois heures) de temps supplémentaire pour *chaque* solution. Ces simulations n'ont toutefois pas été effectuées. Cela révèle à nouveau la grande importance des annotations de l'utilisateur pour la méthodologie de HLS proposée.

LegUp

Les résultats obtenus avec AUGH peuvent être comparés avec ceux publiés pour LegUp dans [CCA+11] et pour le flot *pure hardware*. Les valeurs de latence des circuits générés par AUGH sont prises sans contraintes en ressources, comme obtenu dans les expériences précédentes (Tableau 6.2 (page 96), stratégie S.E.).

La comparaison porte sur le temps d'exécution des circuits et non sur le nombre de cycles d'horloge, car les circuits générés par LegUp ne fonctionnent pas tous à la même fréquence. Les résultats sont donnés en Tableau 6.10.

TABLE 6.10 – Rapport de temps d'exécution des circuits entre AUGH et LegUp

	adpcm	aes	blowfish	gsm	motion	sha	Moy. géom.
LegUp / AUGH	168	4,85	270	1,86	1.2	8,45	12,7

Les circuits produits par AUGH sont en moyenne 12,7x plus rapides que ceux annoncés pour LegUp. Cependant, des facteurs autres que les capacités de AUGH doivent être pris en compte pour l'interprétation de ce résultat.

— Les expériences menées avec AUGH ciblaient une technologie Xilinx Virtex-7 au plus haut niveau de performance, qui est sensiblement plus rapide que la technologie ciblée dans [CCA+11], également plus ancienne (Altera Cyclone II).
— Le code des applications a été manuellement réécrit pour correspondre aux exigences du *parser* de AUGH (décrites précédemment en section 6.1, page 84). Cette réécriture pourrait parfois exhiber du parallélisme dans les blocs de base, ce qui favoriserait AUGH. Cette réécriture pourrait toutefois être automatisée, aussi bien dans AUGH que dans LegUp.
— Pour AUGH, les vecteurs de test ont été retirés du code des applications, et remplacés par une transmission par canal FIFO (une valeur est transmise par cycle d'horloge). Selon le type de mémoire employé par LegUp pour stocker ces vecteurs de test cela peut agir de

façon à le favoriser, ou bien à le défavoriser, dans l'interprétation de la performance apparente du coeur de calcul. Si les vecteurs de test sont stockés en mémoire externe au FPGA, la latence de communications est élevée, ce qui défavorise LegUp. S'ils sont stockés en ROM dans le FPGA, cela pourrait favoriser LegUp, qui pourrait par exemple lire plusieurs valeurs par cycle d'horloge depuis cette ROM. La technique employée n'est toutefois pas précisée dans [CCA+11].

Toutefois, ces éléments de contexte ne suffisent probablement pas à justifier les facteurs 168 et 270 obtenus pour les application *adpcm* et *blowfish*. Ces valeurs, très élevées, confirment l'intérêt, et même la nécessité, de l'exploration des solutions dans la génération d'accélérateurs matériels.

6.10 Synthèse sur les expérimentations

Des campagnes expérimentales ont été lancées avec le logiciel démonstrateur implémentant la méthodologie de HLS proposée. Les résultats obtenus confirment que la méthodologie proposée permet d'explorer de nombreuses solution de façon transparente, et de produire des circuits ajustés aux contraintes matérielles données. La capacité à traiter un décodeur MJpeg complet illustre la capacité à passer à l'échelle et à traiter des circuits complexes. Le processus d'exploration converge et s'exécute en un temps relativement faible, comparé aux techniques de HLS-DSE existantes et aux logiciels en aval du flot de génération.

L'implémentation proposée illustre également qu'il est possible d'intégrer un flot de HLS-DSE rapide et autonome dans les logiciels de HLS existants, tout en conservant la traditionnelle interface manuelle. La gestion des annotations telles que le nombre d'itération des boucles et les probabilités de branchement laisse également la possibilité de rechercher manuellement les solutions optimales de Pareto, pour les utilisateurs prêts à y consacrer le temps.

Cependant, pour des raisons de temps d'implémentation, certains résultats n'ont pas pu être obtenus. En particulier, la position des solutions générées par AUGH par rapport aux solutions optimales de Pareto pour la technologie ciblée reste inconnue. Les simulations RTL des circuits générés par AUGH n'ont pas toutes pu être effectuées. Néanmoins, des tests ont été menés pour l'application *idct* : toutes les solutions considérées durant l'exploration par AUGH ont été simulées au niveau RTL avec succès.

Chapitre 7

Conclusions et perspectives

Dans ces travaux de thèse, une nouvelle méthodologie de génération automatique d'accélérateurs matériels, via la synthèse d'architecture et sous contraintes de ressources, est proposée. Le flot de conception est conçu de façon à présenter simultanément, et de façon cohérente, plusieurs propriétés. Ces propriétés répondent aux exigences principales du domaine de l'accélération matérielle sur FPGA :

— la génération des circuits de façon autonome,
— l'exploration des solutions avec une faible complexité algorithmique,
— la prise en compte d'un scénario d'exécution particulier,
— la garantie du respect des contraintes matérielles en ressources et en fréquence,
— et la cohérence avec les logiciels en aval du flot de génération.

La génération autonome est accomplie par l'intégration de la boucle d'exploration entière à l'intérieur du logiciel de HLS. Pour la génération rapide, des algorithmes de faible complexité sont employés, notamment pour l'ordonnancement, l'assignation et l'exploration des solutions. Un scénario d'exécution peut être transmis au logiciel de HLS, sous la forme d'annotations indiquant le comportement au niveau du contrôle dépendant des données. La conformité vis-à-vis des contraintes en ressources est assurée par la vérification de toutes les solutions explorées. Une technique de correction légère et rapide a été proposée pour assurer le fonctionnement à une fréquence donnée, même après le placement-routage. Le flot de génération global ainsi formé, depuis la HLS jusqu'à la génération du *bitstream* de programmation du FPGA, est exécuté de façon linéaire, ce qui assure sa convergence et sa rapidité.

Pour démontrer la pertinence du flot proposé, le logiciel de HLS démonstrateur AUGH a été construit. Il intègre cinq types de transformations différentes, ce qui lui permet d'explorer des solutions de structures variées et avec différents niveaux de parallélisme. Huit applications de test ont été utilisées, dont six issues du *benchmark* CHStone, connu et déjà employé dans l'évaluation des performances des logiciels de HLS. Pour chacune de ces applications, AUGH a exploré des solutions de façon autonome et rapide, tout en respectant strictement les contraintes de ressources imposées.

Même si AUGH, en tant que preuve de concept, n'atteint pas le niveau d'optimisation des solutions commerciales sous pilotage manuel, il démontre la faisabilité du flot de conception envisagé. La méthodologie de HLS proposée est très proche de la compilation, du point de vue

de la génération automatique pour une cible matérielle donnée. Elle est accessible même aux utilisateurs ayant peu de connaissances dans le domaine de la conception de circuits.

Plus largement déployée au sein des logiciels de HLS existants, la méthodologie de HLS proposée peut grandement favoriser l'adoption des circuits FPGA comme accélérateurs matériels d'usage général. En apportant un gain en puissance de calcul et en consommation, les FPGA pourraient alors détrôner les GPGPU pour les classes d'applications appropriées.

Cependant, avant d'en arriver là, on souligne qu'il reste des challenges à relever.

En particulier, la position des circuits générés par AUGH par rapport aux solutions optimales de Pareto est encore inconnue. Il est possible que des techniques d'optimisation plus poussées puissent être employées, mais c'est un sujet qui reste à explorer.

Le problème de la routabilité des circuits générés sur le FPGA ciblé n'est également pas encore garanti. Ce point est le dernier encore non résolu afin que le processus de synthèse entier (depuis la HLS jusqu'à l'obtention du *bitstream* de programmation du FPGA), soit transparent et entièrement compréhensible même pour les utilisateurs non initiés. Cependant à notre connaissance, aucune approche efficace à ce problème n'est encore connue. C'est donc une problématique à très long terme.

Enfin, une technique de *retiming* permettant de garantir le fonctionnement du circuit généré même après placement et routage a été proposée en section 4.1.4, mais n'a pas été implémentée. Afin de conclure sur la validité des propriétés annoncées pour le flot de génération, notamment l'absence d'itérations avec les logiciels avals de synthèse logique, placement et routage, une implémentation devra être effectuée. Cette tâche est une des perspectives à court terme pour le projet AUGH.

En tant que l'un des rares logiciels de HLS libres et *open source*, le développement de AUGH continue. De nombreuses opportunités d'amélioration existent, notamment au niveau du module *parser* des descriptions d'entrée en C, de l'ordonnanceur et du processus d'assignation.

Les fonctionnalités de AUGH seront également étendues afin que ce projet reste vivant et que ses principales propriétés soient acceptées. On citera notamment :

— l'ajout de nouveaux types de transformation, notamment au niveau des ports d'écriture dans les bancs de mémoire (travaux initiés par Damien Hackett lors d'un stage sur les mémoires multi-ports génériques),
— la gestion de types d'interfaces avec le monde extérieure plus performantes (travaux initiés par Sébastien Martins lors d'un stage sur les mémoires *ping-pong*),
— l'application de techniques d'extraction et d'importation de contexte aux circuits générés, afin de cibler les plateformes dynamiquement reconfigurables (travaux initiés par Alexandre Ghiti durant un stage),
— la génération de modèles de simulation en SystemC afin de simuler des systèmes entiers (travaux en cours sous la forme de projets d'étudiants à l'école Phelma),
— l'amélioration de la précision des estimateurs,
— l'ajout de calibrations de référence pour d'autres technologies de FPGA,
— la gestion de composants pipelinés,
— et la gestion d'opérations en virgule flottante.

À plus large échelle, des discussions ont eu lieu avec les auteurs du projet FloPoCo [dDP11], laissant entrevoir une possible collaboration au niveau des trois derniers points. En effet, en

exploitant la bibliothèque technologique de FloPoCo au sein de AUGH, les deux projets deviendraient cohérents et interopérables.

AUGH est libre et *open source*, disponible sous la license GPLv3 sur le site internet du projet [Pro13].

Publications et présentations des travaux

Publications internationales

DSD 2013 Santander, Espagne (papier 8 pages + présentation orale)

"A Fast and Autonomous HLS Methodology for Hardware Accelerator Generation Under Resource Constraints"

Journal of Systems Architecture Édition spéciale *Design Space Exploration* (article accepté le 6 Octobre 2013 pour publication)

"Fast and Standalone Design Space Exploration for High-Level Synthesis under Resource Constraints"

Publications nationales

SympA 2013 Grenoble (papier 9 pages + présentation orale)

"Méthodologie de génération rapide et automatique d'accélérateurs matériels sous contraintes de ressources : progression itérative et gloutonne"

Autres colloques et groupements de recherche

HLS4HPC 2013 *Workshop* durant la conférence HiPEAC, Berlin, Allemagne (résumé 2 pages + présentation orale)

"A Fast and Stand-alone HLS Methodology for Hardware Accelerator Generation Under Resource Constraints"

JNRDM 2012 Marseille (résumé 4 pages + présentation de poster)

"Génération automatique d'accélérateurs matériels sur cible reconfigurable via la synthèse d'architecture"

GDR SoC-SIP 2010 Cergy-Pontoise (résumé 2 pages + présentation de poster)

"Méthodologie de Conception de Systèmes Matériels Reconfigurables à partir de la Synthèse de Haut Niveau"

Glossaire

ASIC Circuit intégré dédié, entièrement conçu pour une application spécifique. Acronyme issu de l'anglais *Application-Specific Integrated Circuit*.

BRAM Bloc de mémoire permettant de lire et d'écrire des valeurs par adresse. Acronyme issu de l'anglais *Block of Random Access Memory*.

CDFG Graphe modélisant des suites d'instructions à exécuter, avec des sauts conditionnels. Les noeuds du graphe sont des DFG, aussi appelés blocs de base dans ce contexte. Acronyme de *Graphe de Flot de Données et de Contrôle*.

CPU Représente le microprocesseur principal d'un ordinateur. Issu de l'anglais *Central Processing Unit*.

DFG Graphe modélisant une unique suite d'instructions à exécuter. Acronyme de *Graphe de Flot de Données*.

DSP Dans un FPGA, c'est un élément capable d'effectuer efficacement des opérations arithmétiques (couramment addition, soustraction, multiplication). Acronyme issu de l'anglais *Digital Signal Processor*.

FF Registre élémentaire permettant de mémoriser un bit, couramment implémenté sous la forme d'une bascule D. Acronyme issu de l'anglais *Flip-Flop*.

FPGA Circuit numérique reconfigurable capable d'adopter le comportement de circuits combinatoires ou séquentiels variés. Acronyme issu de l'anglais *Field Programmable Gate Array*.

FSM Machine à états. Ce composant contrôle le flot d'exécution du circuit auquel il appartient. Acronyme issu de l'anglais *Finite State Machine*.

GPGPU Processeur graphique dont la structure permet également d'effectuer des calcul plus génériques. Issu de l'anglais *Graphical Processing Unit*.

HCDFG Graphe de type CDFG étendu de façon à modéliser la structure hiérarchique des instructions à exécuter. Chaque noeud peut être un sous-HCDFG, les feuilles de l'arbre étant des DFG. Aussi appelé HTG.

HLS Domaine de la conception de circuits numériques utilisant des langages de haut niveau (tel le C) comme description d'entrée. Acronyme issu de l'anglais *High-Level Synthesis*.

HTG Graphe hiérarchique de tâches, aussi appelé HCDFG.

LUT Table de correspondance. Ce petit composant de mémoire peut modéliser n'importe quelle fonction logique d'un nombre donné d'entrées logiques (couramment 4 ou 6). Ces entrées forment l'adresse à lire dans la mémoire interne. Acronyme issu de l'anglais *Look-Up Table*.

LUTRAM Composant de mémoire construit en utilisant les LUT d'un FPGA. Acronyme issu de l'anglais *Look-Up Table based Random Access Memory*.

RTL Niveau d'abstraction d'un langage de description de circuits numériques, correspondant aux descriptions des transfert entre registres. Acronyme issu de l'anglais *Register Transfer Level*.

UGH Logiciel de synthèse de haut niveau développé aux laboratoires LIP6 (Paris) et TIMA (Grenoble). Libre et open-source, il permet à l'utilisateur de contrôler finement les chemins de données dans les circuits générés. Acronyme issu de l'anglais *User-Guided High-level synthesis*.

VHDL Langage de description de circuits numérique. Acronyme issu de l'anglais *Very-high-speed integrated circuits Hardware Description Language*.

Bibliographie

[AAM06] Ahmed, M.M. et H.L. Abdel-Malek: *Scheduling In High-Level Synthesis Using A Hybrid Constraint Logic Programming / Integer Programming Approach*. Dans *Computer Engineering and Systems, The 2006 International Conference on*, november 2006.

[AP08] Augé, Ivan et Frédéric Pétrot: *User guided high level synthesis*. Dans *High-Level Synthesis : from Algorithm to Digital Circuit*, pages 171–196. Springer, 2008.

[BGPB06] Bilavarn, S., G. Gogniat, J. L. Philippe et L. Bossuet: *Design Space Pruning Through Early Estimations of Area/Delay Tradeoffs for FPGA Implementations*. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 25(10), october 2006.

[BMZ11] Beidas, R., Wai Sum Mong et Jianwen Zhu: *Register pressure aware scheduling for high level synthesis*. Dans *Design Automation Conference (ASP-DAC), 2011 16th Asia and South Pacific*, pages 461–466, Jan. 2011.

[Bol08] Bollaert, Thomas: *Catapult synthesis : a practical introduction to interactive C synthesis*. Dans *High-Level Synthesis : from Algorithm to Digital Circuit*, pages 29–52. Springer, 2008.

[BPL01] Bakshi, A., V. K. Prasanna et A. Ledeczi: *MILAN : A Model Based Integrated Simulation Framework for Design of Embedded Systems*. Dans *Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems*, LCTES '01, pages 82–93. ACM, 2001.

[CAD+12] Czajkowski, T.S., U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras et D.P. Singh: *From OpenCL to high-performance hardware on FPGAs*. Dans *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, aug. 2012.

[CCA+11] Canis, Andrew, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown et Tomasz Czajkowski: *LegUp : high-level synthesis for FPGA-based processor/accelerator systems*. Dans *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '11, 2011.

[CCB+08] Coussy, Philippe, Cyrille Chavet, Pierre Bomel, Dominique Heller, Eric Senn et Eric Martin: *GAUT : A high-level synthesis tool for DSP applications*. Dans *High-Level Synthesis : from Algorithm to Digital Circuit*, pages 147–169. Springer, 2008.

[CDF06] Cherroun, H., A. Darte et P. Feautrier: *Scheduling under Resource Constraints using Dis-Equations*. Dans *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, tome 1, march 2006.

[CGMT09] Coussy, P., D.D. Gajski, M. Meredith et A. Takach: *An Introduction to High-Level Synthesis*. Design Test of Computers, IEEE, 26, 2009.

[CHD+12] Corre, Youenn, Van Trinh Hoang, Jean Philippe Diguet, Dominique Heller et Loic Lagadec: *HLS-based Fast Design Space Exploration of ad hoc hardware accelerators : a key tool for MPSoC Synthesis on FPGA*. Dans *International Conference on Design and Architectures for Signal and Image Processing (DASIP 2012)*, Oct 2012.

[CM08] Coussy, Philippe et Adam Morawiec: *High-Level Synthesis : from Algorithm to Digital Circuit*. Springer Publishing Company, Incorporated, 2008.

[dAMGBE11] Aledo Marugán, Pablo González de, Javier González-Bayón et Pablo Sánchez Espeso: *Hardware performance estimation by dynamic scheduling*. Dans *Specification and Design Languages (FDL), 2011 Forum on*, 2011.

[dDP11] Dinechin, F. de et B. Pasca: *Designing Custom Arithmetic Data Paths with FloPoCo*. Design Test of Computers, IEEE, 28(4) :18–27, 2011.

[DHP+03] Diniz, Pedro, Mary Hall, Joonseok Park, Byoungro So et Heidi Ziegler: *Bridging the Gap between Compilation and Synthesis in the DEFACTO System*. Dans Dietz, HenryG. (rédacteur) : *Languages and Compilers for Parallel Computing*, tome 2624 de *Lecture Notes in Computer Science*, pages 52–70. Springer Berlin Heidelberg, 2003.

[Don04] Donnet, François.: *Synthèse de Haut Niveau Contrôlée par l'Utilisateur*. Thèse de doctorat, Université Paris VI, 2004.

[DRQR08] Derrien, Steven, Sanjay Rajopadhye, Patrice Quinton et Tanguy Risset: *High-level synthesis of loops using the polyhedral model*. Dans *High-Level Synthesis : from Algorithm to Digital Circuit*, pages 215–230. Springer, 2008.

[DVSVC01] Dambre, J., P. Verplaetse, D. Stroobandt et J. Van Campenhout: *On rent's rule for rectangular regions*. Dans *Proceedings of the 2001 international workshop on System-level interconnect prediction*, SLIP '01, 2001.

[FLL+08] Ferrandi, F., P.L. Lanzi, D. Loiacono, C. Pilato et D. Sciuto: *A Multi-objective Genetic Algorithm for Design Space Exploration in High-Level Synthesis*. Dans *Symposium on VLSI, 2008. ISVLSI '08. IEEE Computer Society Annual*, april 2008.

[Fre13] Free Software Foundation, Inc.: *GCC, the GNU Compiler Collection*, 2013. `http://gcc.gnu.org/`, Consulté le 10 mai 2013.

[GDGN03] Gupta, S., N. Dutt, R. Gupta et A. Nicolau: *SPARK : a high-level synthesis framework for applying parallelizing compiler transformations*. Dans *VLSI Design, 2003. Proceedings. 16th International Conference on*, jan. 2003.

[GHH+12] Gruttner, K., P.A. Hartmann, K. Hylla, S. Rosinger, W. Nebel, F. Herrera, E. Villar, C. Brandolese, W. Fornaciari, G. Palermo, C. Ykman-Couvreur, D. Quaglia, F. Ferrero et R. Valencia: *COMPLEX : COdesign and Power Management in*

PLatform-Based Design Space EXploration. Dans *Digital System Design (DSD), 2012 15th Euromicro Conference on*, pages 349–358, 2012.

[Gin13] Gingold, Tristan: *GHDL*, 2013. `http://ghdl.free.fr/`, Consulté le 20 mai 2013.

[GLC94] Gray, C.T., Wentai Liu et R.K. Cavin: *Timing constraints for wave-pipelined systems*. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 13(8) :987–1004, 1994.

[GLS11] George, A., H. Lam et G. Stitt: *Novo-G : At the Forefront of Scalable Reconfigurable Supercomputing*. Computing in Science Engineering, 13(1), jan.-feb. 2011.

[GRS+02] Gupta, S., M. Reshadi, N. Savoiu, N. Duff, R. Gupta et A. Nicolau: *Dynamic common sub-expression elimination during scheduling in high-level synthesis*. Dans *System Synthesis, 2002. 15th International Symposium on*, pages 261–266, 2002.

[GSK+01] Gupta, Sumit, Nick Savoiu, Sunwoo Kim, Nikil Dutt, Rajesh Gupta et Alex Nicolau: *Speculation techniques for high level synthesis of control intensive designs*. Dans *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 269–272. ACM, 2001.

[HLH91] Hwang, C. T., J. H. Lee et Y. C. Hsu: *A formal approach to the scheduling problem in high level synthesis*. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 10(4), apr 1991.

[HTHT09] Hara, Yuko, Hiroyuki Tomiyama, Shinya Honda et Hiroaki Takada: *Proposal and Quantitative Analysis of the CHStone Benchmark Program Suite for Practical C-based High-level Synthesis*. Journal of Information Processing, 17, 2009.

[Inc13] Inc., Jacquard Computing: *ROCCC 2.0*, 2013. `http://www.jacquardcomputing.com/roccc/`, Consulté le 20 mai 2013.

[JPT+10] Jia, Zai Jian, A.D. Pimentel, M. Thompson, T. Bautista et A. Nunez: *NASA : A generic infrastructure for system-level MP-SoC design space exploration*. Dans *Embedded Systems for Real-Time Multimedia (ESTIMedia), 2010 8th IEEE Workshop on*, pages 41–50, 2010.

[KK06] Krishnan, V. et S. Katkoori: *A genetic algorithm for the design space exploration of datapaths during high-level synthesis*. Evolutionary Computation, IEEE Transactions on, 10(3), june 2006.

[KNRK02] Kulkarni, D., W.A. Najjar, R. Rinker et F.J. Kurdahi: *Fast area estimation to support compiler optimizations in FPGA-based reconfigurable systems*. Dans *Field-Programmable Custom Computing Machines, 2002. Proceedings. 10th Annual IEEE Symposium on*, 2002.

[Kuc98] Kuchcinski, K.: *An approach to high-level synthesis using constraint logic programming*. Dans *Euromicro Conference, 1998. Proceedings. 24th*, tome 1, aug 1998.

[LLCHM10] Lhairech-Lebreton, G., P. Coussy, D. Heller et E. Martin: *Bitwidth-aware high-level synthesis for designing low-power DSP applications*. Dans *Electronics,*

Circuits, and Systems (ICECS), 2010 17th IEEE International Conference on, pages 531–534, dec 2010.

[LLZ+12] Li, Shuangchen, Yongpan Liu, Daming Zhang, Xinyu He, Pei Zhang et Huazhong Yang: *A hierarchical C2RTL framework for FIFO-connected stream applications*. Dans *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, pages 133 –138, jan. 30 2012-feb. 2 2012 2012.

[LPC12] Liu, Hung Yi, M. Petracca et L.P. Carloni: *Compositional system-level design exploration with planning of high-level synthesis*. Dans *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pages 641–646, March 2012.

[MDGP05] Moullec, Yannick Le, Jean Philippe Diguet, Thierry Gourdeaux et Jean Luc Philippe: *Design-Trotter : System-level dynamic estimation task a first step towards platform architecture selection*. J. Embedded Comput., 1(4) :565–586, décembre 2005, ISSN 1740-4460.

[NP91] Nicolau, A. et R. Potasman: *Incremental tree height reduction for high level synthesis*. Dans *Design Automation Conference, 1991. 28th ACM/IEEE*, pages 770–774, june 1991.

[PBMR12] Prost-Boucle, Adrien, Olivier Muller et Frédéric Rousseau: *Méthodologie de génération rapide et automatique d'accélérateurs matériels sous contraintes de ressources : progression itérative et gloutonne*. Dans *Conférence d'informatique en Parallélisme, Architecture et Système, 2013 (ComPAS'2013)*, oct 2012.

[PK89] Paulin, P.G. et J.P. Knight: *Force-directed scheduling for the behavioral synthesis of ASICs*. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 8(6) :661 –679, juin 1989.

[Ple10] Plesco, Alexandru: *Program transformations and memory architecture optimizations for High-Level Synthesis of hardware accelerators*. Thèse de doctorat, Ph.D. thesis, École Normale Supérieure de Lyon, 2010.

[Pro13] Prost-Boucle, Adrien: *Site internet du projet AUGH*, 2013. `http://tima.imag.fr/sls/research-projects/augh/`, Consulté le 18 octobre 2013.

[PTD13] Pocek, Kenneth, Russell Tessier et André DeHon: *Birth and Adolescence of Reconfigurable Computing : A Survey of the First 20 Years of Field-Programmable Custom Computing Machines*, 2013. `http://tcfpga.org/fccm20/fccm20survey.pdf`.

[RBL11] Ram, D.S.H., M.C. Bhuvaneswari et S.M. Logesh: *A Novel Evolutionary Technique for Multi-objective Power, Area and Delay Optimization in High Level Synthesis of Datapaths*. Dans *VLSI (ISVLSI), 2011 IEEE Computer Society Annual Symposium on*, pages 290 –295, july 2011.

[SHD02] So, Byoungro, Mary W. Hall et Pedro C. Diniz: *A compiler approach to fast hardware design space exploration in FPGA-based systems*. SIGPLAN Not., 37, May 2002.

[Sil94] Silc, Jurij: *Scheduling strategies in high-level synthesis*. rapport technique, Laboratory for Computer Architecture, 1994.

[SS11] Sengupta, A. et R. Sedaghat: *Integrated scheduling, allocation and binding in High Level Synthesis using multi structure genetic algorithm based design space exploration*. Dans *Quality Electronic Design (ISQED), 2011 12th International Symposium on*, pages 1 –9, march 2011.

[STW09] Schafer, B.C., T. Takenaka et K. Wakabayashi: *Adaptive Simulated Annealer for high level synthesis design space exploration*. Dans *VLSI Design, Automation and Test, 2009. VLSI-DAT '09. International Symposium on*, april 2009.

[SW12] Schafer, B.C. et K. Wakabayashi: *Machine learning predictive modelling high-level synthesis design space exploration*. Computers Digital Techniques, IET, 6(3) :153 –159, may 2012.

[TB08] Tian, Xiang et K. Benkrid: *Design and implementation of a high performance financial Monte-Carlo simulation engine on an FPGA supercomputer*. Dans *ICECE Technology, 2008. FPT 2008. International Conference on*, dec. 2008.

[TGP07] Tripp, J.L., M.B. Gokhale et K.D. Peterson: *Trident : From High-Level Language to Hardware Circuitry*. Computer, 40(3) :28–37, march 2007.

[VPNH10] Villarreal, J., A. Park, W. Najjar et R. Halstead: *Designing Modular Hardware Accelerators in C with ROCCC 2.0*. Dans *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, may 2010.

[WGK08] Wang, Gang, Wenrui Gong et Ryan Kastner: *Operation Scheduling : Algorithms and Applications*. Dans Coussy, Philippe et Adam Morawiec (rédacteurs) : *High-Level Synthesis : from Algorithm to Digital Circuit*, pages 231–255. Springer Netherlands, 2008.

[WMPW03] Weaver, Nicholas, Yury Markovskiy, Yatish Patel et John Wawrzynek: *Post-placement C-slow retiming for the xilinx virtex FPGA*. Dans *Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, FPGAÓ3, pages 185–194. ACM, 2003.

[Xil12] Xilinx: *7 Series FPGAs Configurable Logic Block User Guide*, 2012. `http://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf`.

[Xil13a] Xilinx: *Introduction to FPGA Design with Vivado High-Level Synthesis*, jul 2013. `http://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpga-design-hls.pdf`.

[Xil13b] Xilinx: *Introduction to FPGA Design with Vivado High-Level Synthesis*, jul 2013. `http://www.xilinx.com/products/design-tools/all-programmable-abstractions.html#software-based`, Consulté le 25 septembre 2013.

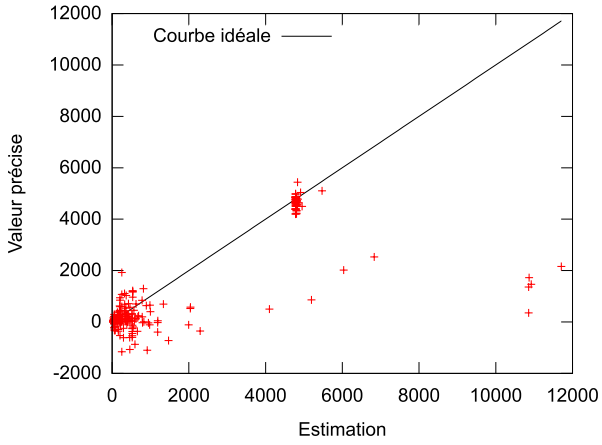
[Xil13c] Xilinx: *Virtex-7 T and XT FPGAs Data Sheet : DC and AC Switching Characteristics*, 2013. `http://www.xilinx.com/support/documentation/data_sheets/ds183_Virtex_7_Data_Sheet.pdf`.

[ZLL+13] Zuo, Wei, Yun Liang, Peng Li, Kyle Rupnow, Deming Chen et Jason Cong: *Improving high level synthesis optimization opportunity through polyhedral trans-*

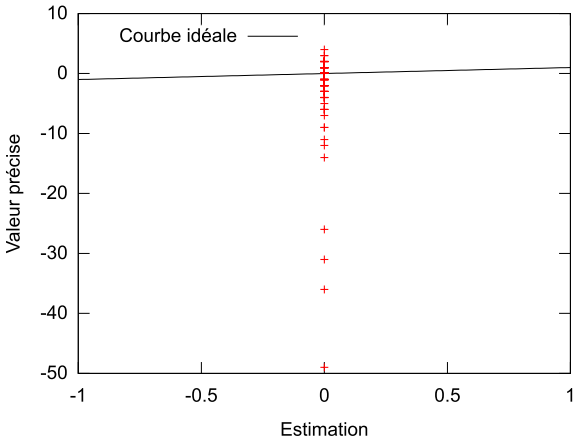
formations. Dans *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '13, pages 9–18. ACM, 2013.

Annexe A

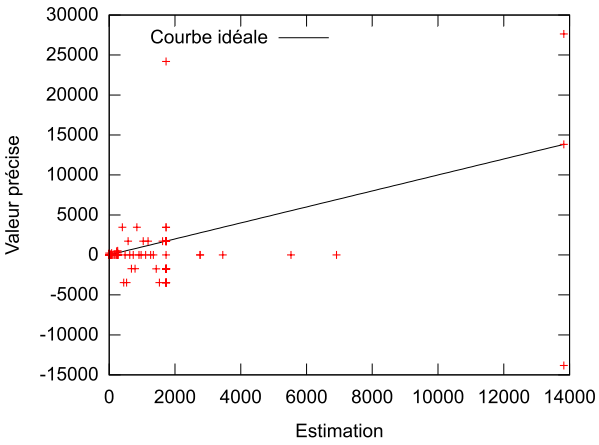
Précision des estimateurs



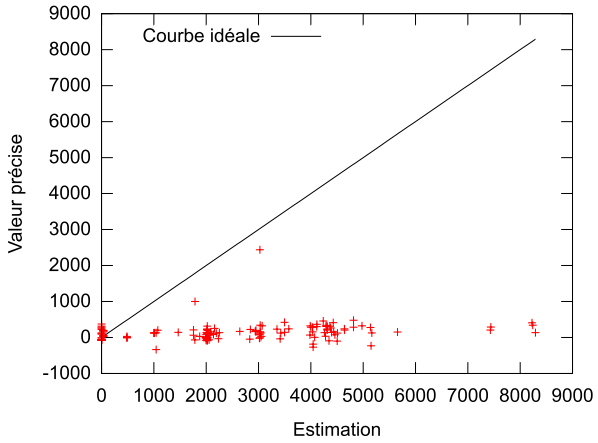
(a) Extension alloc. (LUT)



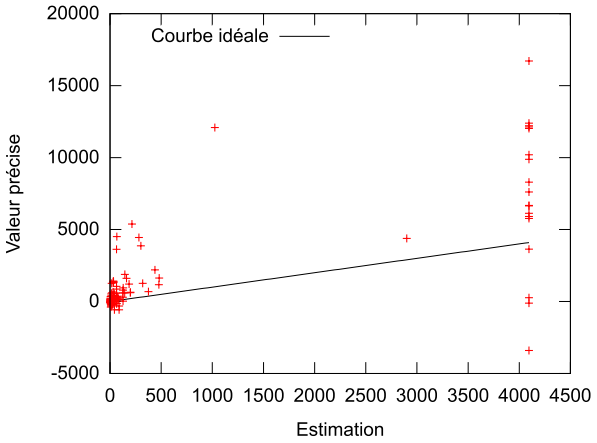
(b) Extension alloc. (FF)



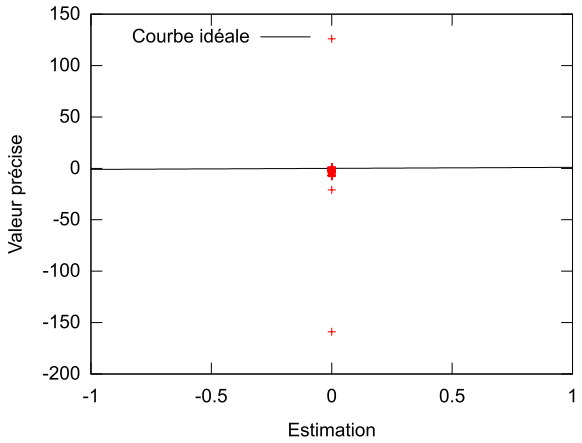
(c) Extension alloc. (cycles)



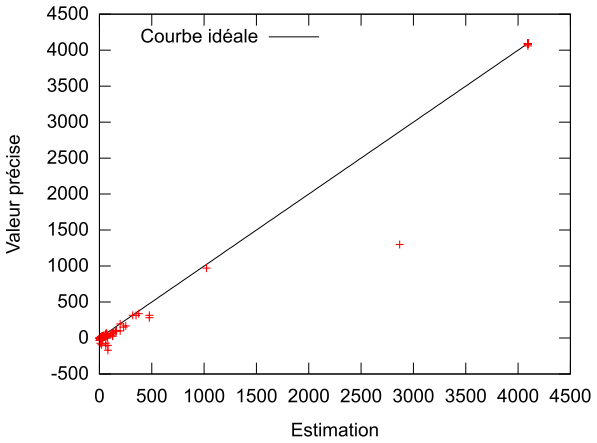
(a) Câblage de condition (LUT)



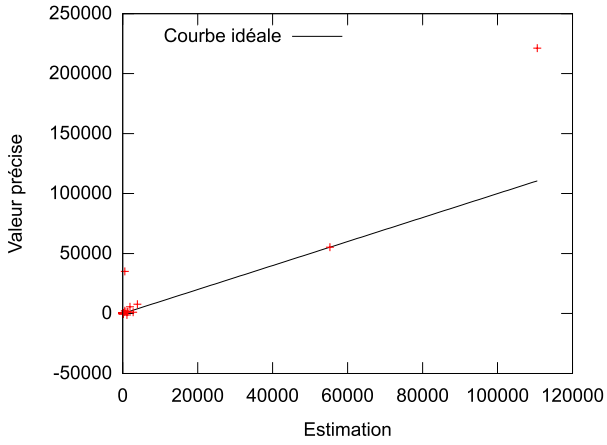
(b) Déroulement de boucle (LUT)



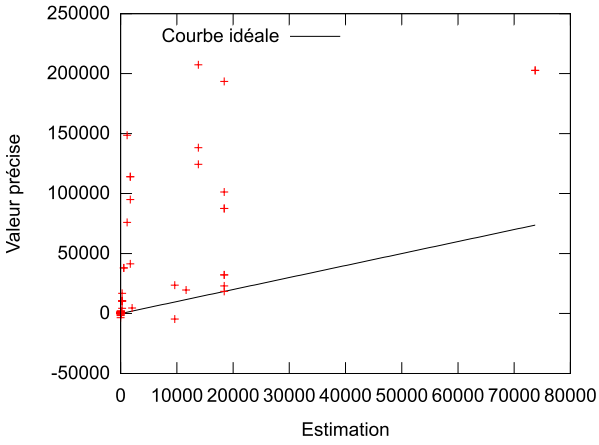
(c) Câblage de condition (FF)



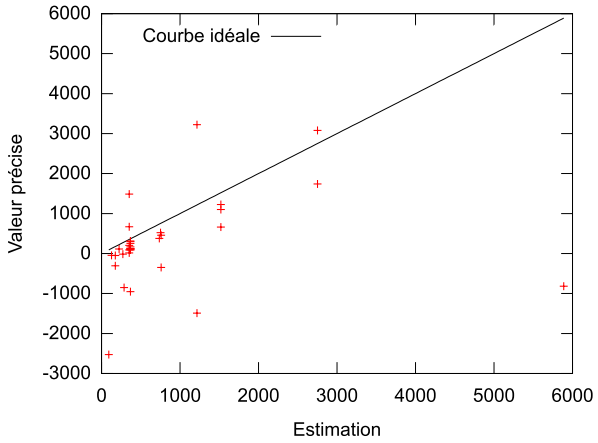
(d) Déroulement de boucle (FF)



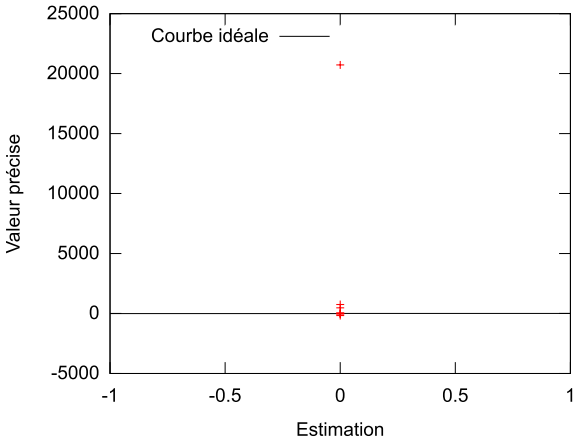
(e) Câblage de condition (cycles)



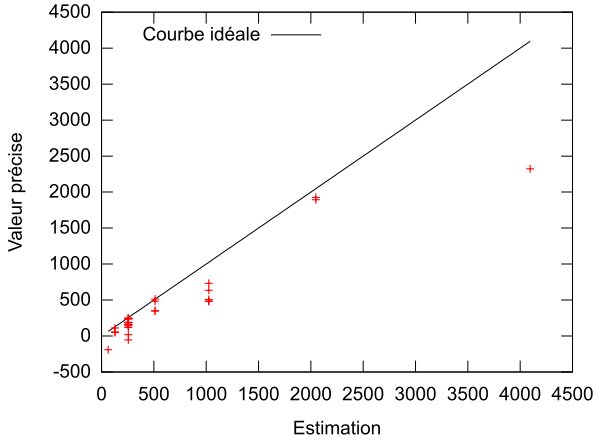
(f) Déroulement de boucle (cycles)



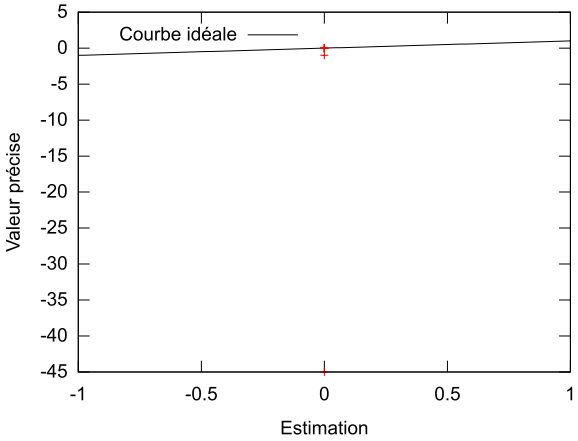
(a) Remplacement de RAM (LUT)



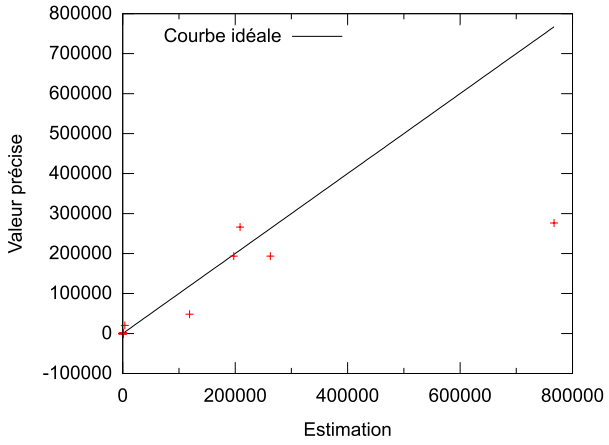
(b) Remplacement de ROM (LUT)



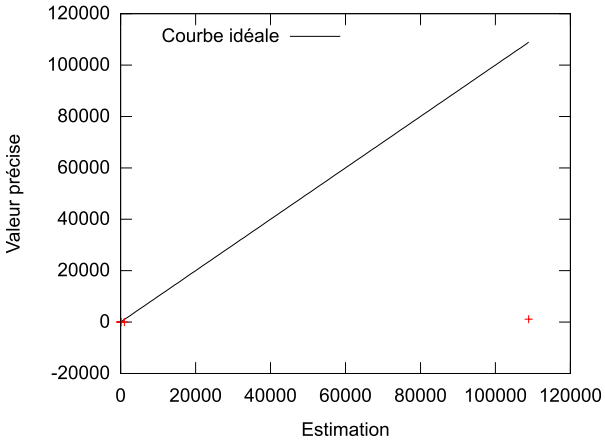
(c) Remplacement de RAM (FF)



(d) Remplacement de ROM (FF)



(e) Remplacement de RAM (cycles)



(f) Remplacement de ROM (cycles)