THESE DE DOCTORAT DE L'UNIVERSITE PARIS VI

Spécialité

INFORMATIQUE

Présentée par

Joachim PISTORIUS

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITE PARIS VI

Sujet

PARTITIONNEMENT DE TRES GRANDES NETLISTS SUR ARCHITECTURES HIERARCHIQUES MULTI-NIVEAUX

Soutenue le

16 Novembre 1999

Devant le jury composé de

M. Michel	**LANDAU**	**Président du Jury**
M. Kamel	**BARKAOUI**	**Rapporteur**
M. Yves	**MATHIEU**	**Rapporteur**
M. Luc	**BURGUN**	**Examinateur**
M. Habib	**MEHREZ**	**Examinateur**
M. Michel	**MINOUX**	**Directeur de Thèse**

A Catherine et à nos enfants
A mes parents
A mon frère et sa famille

Remerciements

Je remercie monsieur Michel Minoux, responsable de l'équipe Optimisation et Modélisation du département Algorithmique Numérique et Parallélisme du LIP6, qui m'a encadré durant ma thèse et dont les conseils et encouragements m'ont été très utiles dans la préparation de ce travail.

Ensuite, j'adresse mes remerciements à monsieur Michel Landau, directeur de l'IST, pour avoir accepté de présider le jury de cette thèse.

Je remercie aussi messieurs Yves Mathieu, directeur d'études du département électronique de l'ENST, et Kamel Barkaoui, maître de conférence au CNAM, qui m'ont fait l'honneur d'êêtre rapporteurs de cette thèse.

Je tiens également à remercier messieurs Habib Mehrez, chargé de recherche CNRS au LIP6, et Luc Burgun, directeur du département Recherche et Développement de Mentor Graphics Corp. Meta Systems Div., qui ont bien voulu accepter de juger ce travail en participant à mon jury de thèse.

J'exprime ma vive gratitude à monsieur Gérard Morisset pour m'avoir accueilli à Meta Systems et à mademoiselle Edmée Legai et monsieur Eric Hochapfel pour m'avoir intégré dans leurs équipes.

Je remercie messieurs Christophe Fouassier, Eric Hochapfel, Cyril Quennesson et Behzad Safari pour leurs importants contributions et leurs conseils judicieux grâce auxquels ce travail a pu aboutir.

Je remercie également messieurs Gérald Couaillier, Pascal Danois et Grégory Leclercq pour leur gentillesse et leurs talents linguistiques. De nombreux documents auraient été bien mal écrits sans leur aide.

Mes remerciements s'adressent aussi à tous les employées de la division Meta Systems de Mentor Graphics Corp. pour les marques de sympathie qu'ils n'ont cessé de me témoigner.

Je remercie également tous les membres du département ASIM du LIP6, pour leurs gentillesse et leur humour.

A tous ceux qui m'ont aidé de près ou de loin, à un moment ou à un autre, dans la préparation de ce travail, je tiens à adresser mes remerciements les plus sincères.

Résumé

Les évolutions technologiques récentes ont fait apparaître pour le développement de grands systèmes électroniques, des architectures à plusieurs niveaux de hiérarchie. Le partitionnement sur chacun des niveaux d'une telle architecture, est une des étapes lors de la réalisation d'un *design*. Les principales applications qui motivent notre étude se situent dans le domaine du prototypage à l'aide de circuits programmables, ou dans le domaine de l'émulation. Ainsi, l'architecture de l'émulateur CELARO de Mentor Graphics nous a servi de modèle d'études.

Les approches connues à ce jour utilisent toutes les mêmes algorithmes et optimisent le résultat de partitionnement sur chacun des niveaux de la hiérarchie sans prendre en compte l'interdépendance des résultats entre les différents niveaux. C'est précisément dans ce contexte que se place cette thèse, dans laquelle nous proposons une méthodologie permettant d'améliorer le résultat global du partitionnement en terme de ressources matérielles utilisées et en terme de temps de calcul.

La méthodologie est basée sur une étude bibliographique approfondie qui a permis de sélectionner les algorithmes les plus susceptibles d'apporter les améliorations voulues. L'analyse des performances des différents algorithmes retenus est effectuée à l'aide d'un ensemble de résultats expérimentaux obtenus par l'application des algorithmes choisis sur un groupe représentatif de *netlists*. Grâce à cette analyse, nous avons identifié les divers avantages et inconvénients de chacun de ces algorithmes, et pu les modifier en vue d'une meilleure adaptation á leur usage dans le cadre d'une architecture hiérarchique.

Selon les performances des algorithmes, ceux-ci ont été combinés afin de déterminer plusieurs outils de partitionnement dédiés à chaque niveau hiérarchique. Une comparaison expérimentale systématique de ces outils à l'aide d'un banc de test permet de déterminer les approches performantes. L'enchaînement des différents outils de partitionnement a également une influence sur le résultat. Une analyse expérimentale des différentes combinaisons possibles nous a conduit à en identifier un petit nombre particulièrement performants.

A partir de cette sélection, nous avons effectué une étude expérimentale comparative systématique sur un ensemble de *netlists* de grandes tailles, industrielles ou générées. Les résultats obtenus sur les 3 premiers niveaux de l'émulateur CELARO montrent que la mise en œuvre de l'ensemble des outils et de la méthodologie élaborés au cours de cette thèse conduit à des améliorations très significatives en terme de volume de matériel nécessaire : en moyenne un gain évoluant entre 15 et 38%. Parallèlement, le temps de calcul a été diminué entre 80 et 85% pour les mêmes *netlists*.

Mots clés : Partitionnement, Architecture hiérarchique, Générateur de netlists aléatoires, Circuit programmable

Abstract

The increasing size of large electronic systems has recently led to their implementation on architectures with several levels of hierarchy. The partitioning of a design into each of the hierarchical levels is one step towards its implementation on such an architecture. The main applications that motivate our research lie in the domain of rapid prototyping and hardware emulation. Therefore, we validated our approach on the hierarchical architecture of the CELARO emulator from Mentor Graphics.

Todays partitioning techniques for hierarchical architectures all use the same algorithms on the different levels and optimize the partitioning results on each level of the hierarchy without taking into account the dependencies between these results. This is the context of our research. A method is proposed that improves the global partitioning result in terms of both hardware resources needed for the implementation of a design, and CPU time.

This method is based on an extensive study of existing algorithms which permitted the identification of those that are well-suited for the partitioning of very large netlists on hierarchical architectures. The analysis of the performances of these algorithms is based on a set of experimental results that have been obtained by applying the chosen algorithms on a certain number of representative netlists. With this analysis, we identified various advantages and disadvantages of the existing algorithms, and were able to improve them in order to produce a better adaptation on the hierarchical structure of the architecture.

Depending on the performances of the different algorithms, they were combined in order to form partitioning tools dedicated to each level of the hierarchy. An experimental comparison of these tools allowed the identification of several approaches of interest. The scheduling of these approaches also has an influence on the result. An experimental analysis of various combinations finally led to a small number of approaches that were particularly performant.

This set of approaches was the basis for an extensive experimental, and comparative study on a set of very large, real, industrial netlists and very large, randomly generated netlists. The results obtained for the first three levels of the CELARO emulator show that the tools and the method developed during this work lead to significant improvements in terms of required hardware resources : we achieved an average gain of 15 to 38%. In addition, the execution time decreased between 80 and 85% for the partitioning of the same netlists.

Keywords : Partitioning, Hierarchical architecture, Benchmark generator, Programmable circuit

Table des matières

Liste des figures

Liste des tableaux

Chapitre 1

Introduction

Le partitionnement de *netlists* représentées par des graphes ou des hypergraphes est un problème désormais classique en conception de circuits ou d'architectures matérielles. De très nombreuses publications lui ont été consacrées. Cependant, les évolutions technologiques récentes ont fait apparaître, pour le développement de grands systèmes, des architectures à plusieurs niveaux de hiérarchie qui conduisent à reconsidérer la problématique du partitionnement afin de prendre en compte les contraintes liées à la multiplicité de niveaux hiérarchiques. Le partitionnement de très grandes *netlists* sur architectures hiérarchiques multi-niveaux est l'objectif principal de ce travail.
Ce sujet a été très peu abordé dans la littérature (cf. notre analyse bibliographique dans chapitre 2.3.7) et, à notre connaissance, le travail présenté ici est la première étude approfondie ayant abouti à la définition d'une méthodologie complète validée expérimentalement sur une architecture-cible concrète à 4 niveaux de hiérarchie.

Les différentes techniques de partitionnement existantes poursuivent toutes l'objectif d'optimiser un certain critère sur un seul niveau hiérarchique. La particularité du partitionnement pour une architecture composée de plusieurs niveaux hiérarchiques est l'interdépendance des résultats de partitionnement sur les différents niveaux : dans le cadre d'un partitionnement qui monte dans la hiérarchie à partir du niveau de base (i.e. à partir du plus petit élément), un partitionnement optimisé sur un niveau n peut mener à un plus mauvais résultat en terme de nombre de ressources utilisées sur un niveau hiérarchique supérieur $(n+p)$ qu'un partitionnement moins optimal au niveau n.
Dans le cadre d'un partitionnement qui descend dans la hiérarchie à partir du niveau *top*, un partitionnement optimisé sur un niveau n peut mener à un partitionnement non-faisable en terme de respect des contraintes de cardinalité maximale ou de coupure sur un niveau hiérarchique inférieur $(n-p)$. Dans ce cas, le partitionnement du niveau n doit être reconsidéré.

L'application principale qui motive l'étude de tels problèmes se situe dans le domaine du prototypage à l'aide de circuits programmables ou dans le domaine de l'émulation. Ainsi, l'architecture de l'émulateur CELARO de Mentor Graphics nous a servi de modèle d'études. Les particularités de l'émulateur ainsi que celles du partitionnement d'une *netlist* en vue de son implémentation sur cet émulateur sont présentées dans ce chapitre introductif.

1.1 Les architectures hiérarchiques

1.1.1 Composition d'architectures hiérarchiques

Une architecture hiérarchique est composée de différents niveaux de *hardware*. Elle peut être modélisée par un arbre où chaque étage correspond à un niveau hiérarchique. Chaque élément d'un certain niveau de la hiérarchie est composé uniquement d'éléments se trouvant sur des niveaux inférieurs.
Nous trouvons ces architectures hiérarchiques surtout dans les grands systèmes électroniques tels que les émulateurs [But91] [Men99a], mais aussi à une échelle plus réduite dans les MCM (*multi-chip modules*) où il y a plusieurs circuits sur un même substrat [Rie95], ce qui mène à une architecture à deux niveaux hiérarchiques.
L'essentiel de notre travail vise à définir une méthodologie générale applicable à une grande variété d'architectures hiérarchiques. Cette méthodologie sera validée expérimentalement sur une architecture réelle (à 4 niveaux), celle de l'émulateur CELARO de la société Mentor Graphics Corp., Meta Systems Division.

1.1.2 Exemple de l'architecture de l'émulateur CELARO

L'émulation est "la simulation du fonctionnement d'un ordinateur sur un autre, généralement plus puissant" (Le Petit Larousse 1995, page 383).
L'émulateur CELARO que nous décrivons ici est un ordinateur au sens large du terme, c'est–à–dire "une machine automatique de traitement de l'information, obéissant à des programmes formés par des suites d'opérations arithmétiques et logiques" (Le Petit Larousse 1995, page 722). Plus particulièrement, il s'agit d'un système matériel reconfigurable composé d'un ensemble de cartes contenant des circuits programmables reliés entre eux par des interconnexions programmables. Ce système permet la vérification fonctionnelle d'un *design*, avant la production d'un prototype du circuit intégré correspondant, par la reproduction des fonctionnalités du *design* par des circuits programmables. Traditionnellement, cette vérification est effectuée à l'aide de simulateurs logiciels, où les fonctionnalités du *design* sont reproduites par un programme.

La simulation logicielle d'un *design* nécessite l'exécution séquentielle des différents modèles qui le constituent afin de déterminer le comportement de chacun d'entre eux. Le temps de simulation d'un *design* augmente alors de manière significative avec sa taille, puisque le comportement de chacun des modèles est évalué de manière séquentielle et les modifications du comportement d'un modèle doivent être prises en compte par tous les modèles qui en dépendent.

L'émulation représente une approche plus rapide pour la vérification d'un *design*: son comportement est modélisé par un ensemble d'éléments logiques reconfigurables reliés entre eux, ce qui permet l'exécution des différents modèles du *design* **en parallèle**. Le processus de vérification est ainsi accéléré d'un facteur 10^3 à 10^6 par rapport à la simulation logicielle, selon le *design* à tester et le niveau d'abstraction utilisé.

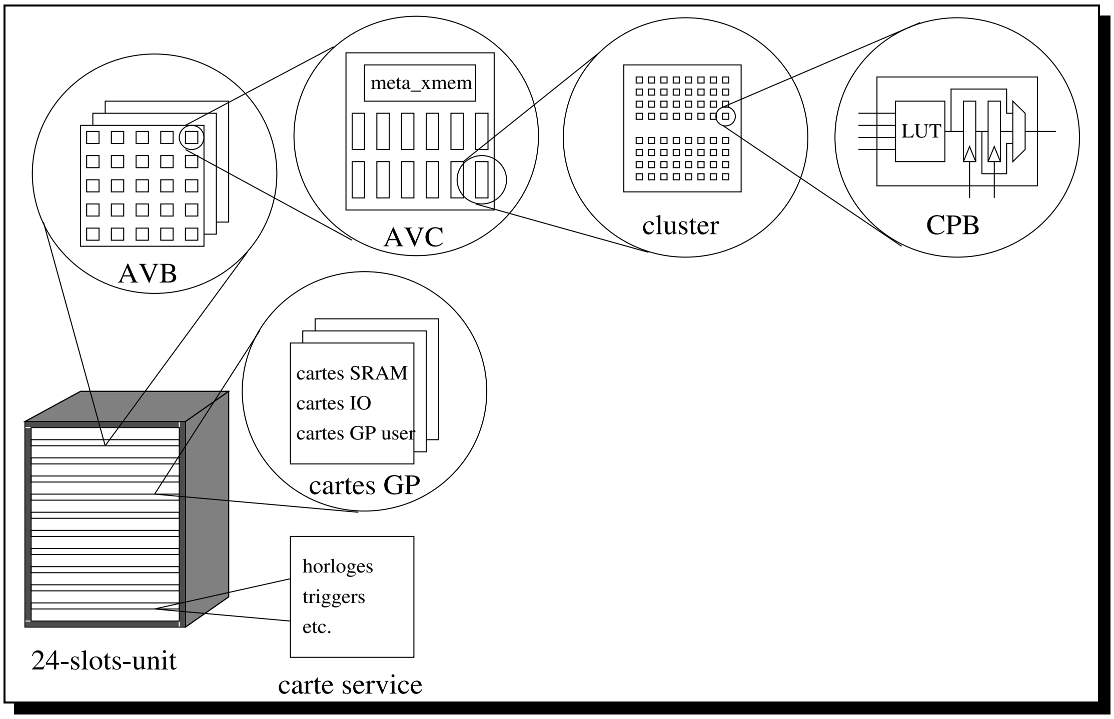


Figure 1.1: *Architecture de l'émulateur CELARO*

1.1.3 Architecture de l'émulateur CELARO

L'élément de base de l'émulateur CELARO est le CPB (*configurable physical block*, voir glossaire). Il s'agit d'une cellule programmable contenant une LUT (*look up table*) et deux registres *latch* permettant ainsi la réalisation de toute fonction booléenne à 4 entrées et une sortie pouvant être éventuellement mémorisée.
Quelques dizaines de CPBs sont regroupés dans un *cluster*.Un tel *cluster* contient autant de connecteurs d'entrée et de sortie qu'il contient de CPBs. Une bonne dizaine de ces *clusters*, soit quelques centaines de CPBs, se trouvent regroupés dans chaque circuit AVC[1] avec une mémoire meta_xmem intégrée (voir glossaire). Un AVC a entre 100 et 200 connecteurs d'entrée/sortie. Une carte AVB[2] a plus de 1000 connecteurs d'entrée/sortie et contient entre 40 et 50 circuits AVCs.
Ces cartes sont regroupées avec d'autres cartes, comme par exemple des cartes mémoires contenant des meta_sram_banks, dans des unités de 20 à 30 cartes. Selon la configuration de la machine, un émulateur CELARO contient un, deux, quatre ou huit de ces unités. L'architecture de l'émulateur CELARO est illustrée à l'aide d'un exemple d'une machine contenant une de ces unités dans la figure 1.1.

1.1.4 Implémentation d'un design sur l'émulateur

La vérification d'un *design* à l'aide de l'émulateur est basée sur l'implémentation dans l'émulateur d'un modèle équivalent au comportement logique du *design*. L'implémentation

[1] *Accelerated Verification Circuit*, voir glossaire
[2] *Accelerated Verification Board*, voir glossaire

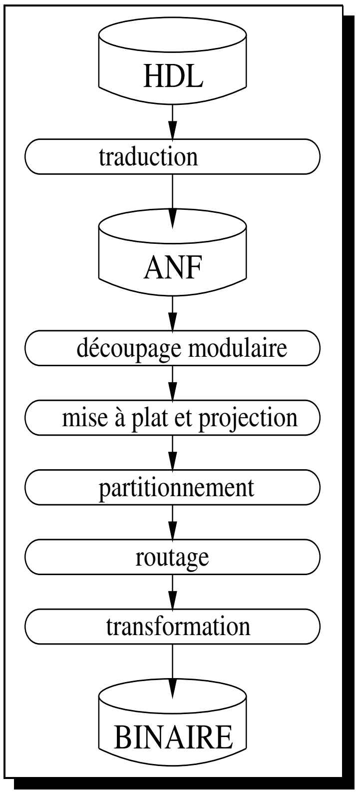


Figure 1.2: *Compilation d'un design*

est réalisée en créant une description binaire du *design* à partir d'une description dans un langage *hardware* HDL. Cette réalisation se fait en plusieurs phases.
Dans un premier temps, la description en langage *hardware* est traduite en un format de description de *netlist* appelé ANF, interne à Mentor Graphics.
Cette description est ensuite découpée en modules et instances de modules. Chaque instance est mise à plat (*flattening*)[3]. Chaque instance contient donc des éléments d'un seul niveau hiérarchique. Ces éléments sont projetés sur des CPBs (*mapping*)[4]. Les *netlists* de CPBs créées ainsi sont partitionnées en *clusters*, AVCs, AVBs et *24-slot-units*, puis les interconnexions entre les cellules communiquant entre elles sont routées[5]. Une illustration de ce processus est présentée dans la figure ci-contre.
Le partitionnement est donc utilisé lors de l'implémentation d'un *design* sur l'émulateur.

[3] On appelle "mise à plat" ou *flattening* la suppression de la notion de hiérarchie par le remplacement des éléments hiérarchiques du niveau n par leur équivalent au niveau $(n+1)$.

[4] On appelle "projection" ou *mapping* le regroupement de fonctions logiques et leur transformation en fonctions pouvant être réalisées par l'architecture cible. Dans le cadre de l'émulateur CELARO, ces éléments sont des fonctions logiques booléennes à 4 entrées et une sortie réalisées par des CPBs.

[5] L'interconnexion des différentes cellules d'une partition nécessite la présence de ressources matérielles spécifiques. On appelle "routage" la programmation de ces ressources.

1.2 Outils de partitionnement existants pour l'émulateur CELARO

1.2.1 Objectifs du partitionnement

L'objectif principal du partitionnement lors de la compilation d'une *netlist* est de minimiser le nombre de cartes AVBs nécessaires pour réaliser l'intégralité de la *netlist*. Le processus du partitionnement doit s'achever dans un temps de calcul raisonnable et il doit être reproductible pour permettre de retrouver des résultats équivalents lors d'une recompilation de la *netlist*.
Un objectif secondaire est l'amélioration de la fréquence d'émulation.
A partir de ces objectifs, nous pouvons définir des objectifs intermédiaires comme par exemple minimiser le nombre d'AVCs et de *clusters* tout en minimisant le nombre moyen de leurs connecteurs, ou diminuer le temps de calcul sur chacun des trois niveaux hiérarchiques, c'est–à–dire du partitionnement en *clusters*, en AVCs et en AVBs.

Le routage à partir d'une partition donnée n'est pas un problème simple et l'intégration des contraintes de routage dans les outils de partitionnement est une tâche particulièrement complexe. On se contente donc de vérifier la routabilité d'une partition juste après sa création. Les *clusters* sont traités de manière différente : ils sont réellement routés juste après leur création.
Une partition créée est alors réalisable sur l'émulateur, si elle respecte les contraintes de cardinalité[6] et de coupure[7] et si l'évaluation de la routabilité fournit des résultats satisfaisants.

1.2.2 Quatre outils, deux algorithmes

Actuellement, le partitionnement de la chaîne de compilation de l'émulateur CELARO est divisé en quatre outils, un pour chaque niveau hiérarchique de l'architecture :

- *partCluster* réalise le partitionnement en *clusters* physiques.
- *partMeta* effectue le partitionnement en AVCs. Chaque AVC contient entre 10 et 20 *clusters* physiques et une mémoire multi-port.
- *partBoard* effectue le partitionnement en AVBs.
- *partRack* effectue le placement des AVBs et d'autres cartes (cartes mémoire, cartes d'entrée/sortie) dans les emplacements de cartes (*slots*) disponibles de l'émulateur.

partRack est un outil spécifique dépendant de la configuration de l'émulateur. Ce niveau de partitionnement n'est pas traité dans ce travail. Néanmoins, le placement des cartes dans l'émulateur a une certaine influence sur la fréquence de l'émulation et nécessite donc une attention particulière.

[6] Nous appelons la cardinalité d'une partition la somme des poids des cellules qui la constituent.
[7] Nous appelons la coupure d'une partition la somme des poids des *nets* ayant en même temps des connecteurs à l'intérieur et à l'extérieur de la partition.

Les trois autres outils (*partCluster*, *partMeta* et *partBoard*) sont réalisés en utilisant deux algorithmes de voisinage appelés *bisect*[8] et *multisect*[9] rappelés en introduction des chapitres 4 et 5. Ces deux algorithmes travaillent en trois phases successives : lors de la première phase, l'algorithme *bisect* est appliqué récursivement afin de diviser la cardinalité des partitions à traiter (dichotomies successives). Cette phase s'arrête quand la première feuille de l'arbre ainsi créé respecte une certaine cardinalité maximale. Les autres feuilles de l'arbre sont traitées ultérieurement avec les cellules non-partitionnées lors des deuxième et troisième phases.
La deuxième phase du partitionnement s'applique sur chacune des feuilles de l'arbre créé lors de la première phase. Lors de cette deuxième phase, l'algorithme *bisect* effectue de nouveau un bipartitionnement récursif. Cette fois-ci, l'algorithme s'arrête si les feuilles de l'arbre respectent une certaine cardinalité maximale, inférieure à la cardinalité de l'élément physique du niveau hiérarchique actuel et si elles ont une coupure inférieure à cet élément physique. Le résultat de cette phase est donc un *clustering* des partitions créées lors de la première phase.
Ces *clusters* sont regroupés lors de la troisième phase afin de former des partitions physiques. L'objectif de ce regroupement est de minimiser le nombre de partitions physiques tout en respectant leurs contraintes de cardinalité et de coupure. L'algorithme utilisé lors de cette phase est *multisect*.

1.2.3 Avantages et inconvénients

Les avantages sont la rapidité de l'algorithme *bisect* utilisé pour réaliser le principe de dichotomies successives et la qualité des partitions physiques obtenues avec l'algorithme *multisect* qui a été développé pour le partitionnement de l'émulateur CELARO.

Par contre, les inconvénients des outils existants sont encore nombreux :

- Nombre d'AVBs élevé :
  L'inconvénient majeur est le résultat du partitionnement en terme de nombre d'AVBs nécessaires pour implémenter un *design*. Le point critique étant le partitionnement en AVCs comme démontré dans section 8.3.

- Temps de calcul important :
  L'algorithme *multisect* a une complexité temporelle élevée, comme constaté dans chapitre 5.1.1, ce qui mène à un temps de calcul important.

- Fréquence de l'émulation non-satisfaisante :
  Les algorithmes actuels appartiennent à la famille des algorithmes de voisinage. Leur objectif est la minimisation de la coupure entre partitions tout en minimisant le nombre de partitions. L'optimisation de la fréquence n'est pas un objectif de ces algorithmes. Pourtant, c'est justement une fréquence élevée qui justifie l'utilisation de l'émulation à la place de la simulation logicielle pour la vérification d'un *design*.

[8] *bisect* est un algorithme de bipartitionnement visant à minimiser le critère du *ratio-cut* entre les deux sous-ensembles de la partition.
[9] *multisect* est un algorithme de multipartitionnement visant à minimiser le nombre de sous-ensembles de la partition tout en respectant entre autres les contraintes de cardinalité maximale et de coupure.

1.3 Besoins du partitionnement

1.3.1 Améliorer le taux de remplissage

Le découpage en AVCs a été identifié comme étant la phase critique du partitionnement. Pour améliorer les résultats de cette étape, nous avons étudié des algorithmes visant à minimiser la coupure, afin d'augmenter l'utilisation des cellules dans les AVCs et de réduire le nombre moyen de connecteurs qu'ils utilisent, tout en conservant approximativement le même nombre d'AVCs.

1.3.2 Améliorer le temps de calcul

Le partitionnement en AVCs constitue actuellement 90% du temps de calcul. Puisque l'algorithme *multisect* a une complexité élevée, nous avons étudié d'autres algorithmes de multipartitionnement, susceptibles de le remplacer lors de la troisième phase du partitionnement en AVCs.

1.3.3 Améliorer la fréquence du design

La fréquence de l'émulation dépend du placement des différentes cellules interconnectées dans l'émulateur ainsi que du routage effectué. Un moyen d'améliorer la fréquence d'un *design* est donc la prise en compte des chemins critiques lors du partitionnement. Nous avons alors étudié des algorithmes permettant d'optimiser la fréquence de l'émulation.

1.4 Comment satisfaire ces besoins : Les objectifs et contributions de ce travail

L'objectif de cette thèse est la définition d'une méthodologie et la mise au point d'outils associés pour résoudre efficacement le partitionnement pour des architectures multi-niveaux. Nous décrivons ci-dessous les principales étapes de notre travail.

1.4.1 Revue de la littérature

Une étude bibliographique approfondie nous a permis de sélectionner les algorithmes les plus susceptibles d'apporter les améliorations voulues. Cette étude est présentée dans le chapitre 2.

Le partitionnement pour architectures hiérarchiques a été très peu étudié jusqu'ici, et les publications disponibles sur le sujet (cf. chapitre 2.3.7 pour une analyse) ne semblent pas être allées jusqu'à la mise en œuvre d'une véritable approche multi-niveaux.
A notre connaissance, le travail présenté ici est donc le premier à définir une méthodologie d'approche jusqu'à une mise en œuvre effective et une expérimentation comparative complète.

1.4.2 Constitution d'un banc de test : générateur de très grandes netlists hétérogènes

L'analyse des performances de différents algorithmes est effectuée par un test expérimental : les algorithmes sont appliqués à un banc de test contenant un ensemble représentatif de *netlists*.
Nous avons constitué un tel banc de test composé de *netlists* aléatoires ayant le même comportement lors du partitionnement que des *designs* industriels implémentés habituellement sur l'émulateur CELARO. Une étude des *benchmarks* et des générateurs existants a révélé qu'aucun d'entre eux ne répond complètement à ces critères. Nous avons donc développé notre propre générateur de *netlists* aléatoires. Ce générateur, présenté dans le chapitre 3, a fait l'objet d'une publication internationale (cf. [Pis99a]).

1.4.3 Amélioration et évaluation expérimentale des différents algorithmes

L'évaluation expérimentale des algorithmes existants et des algorithmes sélectionnés lors de l'étude bibliographique a été effectuée en utilisant des *netlists* industrielles et des *netlists* aléatoires. Les résultats de ces tests nous ont permis d'identifier les avantages et inconvénients de chaque algorithme pour le partitionnement dans le cadre d'une architecture multi-niveaux. L'amélioration et l'étude des performances des algorithmes est présentée dans les chapitres 4 à 7.
Nos principales contributions à l'amélioration des algorithmes de base étudiés sont :

- dans le chapitre 4 (consacré au bipartitionnement par une approche de voisinage) :
  - le développement d'un algorithme de *clustering* en prétraitement de l'algorithme bisect,
  - l'analyse de l'influence de la détermination du bipartitionnement initial sur le résultat du partitionnement,
  - l'introduction d'une nouvelle fonction de coût permettant de déterminer un bipartitionnement équilibré sans pour autant dégrader la coupure,
  - l'ensemble des résultats obtenus a permis de développer un nouvel algorithme (bisect*) améliorant de manière significative les performances de bisect.
- dans le chapitre 5 (consacré au multipartitionnement par une approche de voisinage) :
  - l'analyse de différentes fonctions de coût permettant de minimiser le nombre de partitions nécessaires à l'implémentation d'une *netlist* tout en minimisant le nombre moyen de *nets* coupés afin d'améliorer le partitionnement sur un niveau supérieur de l'architecture hiérarchique,
  - deux fonctions de coûts se sont révélées particulièrement performantes, conduisant à deux versions améliorées de l'algorithme multisect (multisect*, multisect**).
- dans le chapitre 6 (consacré au bipartitionnement par calcul du flot maximal) :
  - le développement d'un algorithme permettant d'optimiser le choix des sommets source et puits,

  - le développement d'un algorithme effectuant le marquage directement sur l'hypergraphe lors du calcul du flot maximal. Le travail concernant cet algorithme a été soumis à publication (cf. [Pis99b]).
  - ces améliorations ont conduit à un nouvel algorithme de bipartitionnement par calcul du flot maximal appelé hyperFBB.

- dans le chapitre 7 (consacré au multipartitionnement par construction et découpage d'un ordonnancement de sommets) :
  - l'adaptation des algorithmes "window" et "DP-RP" aux besoins du partitionnement pour architectures hiérarchiques,
  - l'analyse de l'influence de la taille de la fenêtre sur le résultat de partitionnement,
  - l'analyse de l'influence de la fonction d'attraction sur le résultat de partitionnement
  - les résultats ont permis de développer un algorithme particulièrement adapté aux architectures multi-niveaux, appelé MDOC.

1.4.4 Intégration des algorithmes dans les outils de partitionnement

Jusqu'ici, les différentes publications sur le partitionnement pour architectures hiérarchiques appliquent le même algorithme de partitionnement sur chacun des niveaux. Or, sur les différents niveaux nous avons des contraintes différentes et ainsi des objectifs différents. L'utilisation d'un même et unique algorithme est dans ce cas sous-optimal et l'utilisation de différents algorithmes ou combinaisons d'algorithmes devrait être envisagée. C'est cette idée nouvelle qui est développée dans le chapitre 8.

1.4.5 Etude sur l'enchaînement des outils de partitionnement

L'enchaînement des différents outils de partitionnement a également une influence sur le résultat. Par exemple, pour l'émulateur CELARO, le partitionnement existant utilise uniquement l'enchaînement partMeta avec un partCluster intégré suivi de partBoard et de partRack. D'autres enchaînements nous ont semblé intéressants et mériter une analyse expérimentale approfondie. Cette analyse est développée dans les sections 8.2.5 et 8.2.6, et a conduit à identifier un petit nombre d'enchaînements particulièrement performants.

1.4.6 Vérification expérimentale

A partir des meilleurs enchaînements identifiés précédemment, nous avons effectué une étude expérimentale comparative systématique sur un ensemble de *netlists* industrielles et de *netlists* générées de grande taille (cf. chapitre 8 section 8.3).
Les résultats obtenus sur les 3 premiers niveaux de l'émulateur CELARO montrent que la mise en œuvre de l'ensemble des outils et de la méthodologie élaborés au cours de cette thèse conduit à des améliorations très significatives (15 - 38% en moyenne) en terme de volume de *hardware* nécessaire.

1.4.7 Bilan et perspectives

Le chapitre 9 résume l'ensemble des résultats de ce travail et suggère un certain nombre de perspectives pour des recherches futures dans le domaine du partitionnement de très grandes *netlists* sur architectures hiérarchiques multi-niveaux.

Chapitre 2

Partitionnement de graphes et d'hypergraphes : état de l'art

2.1 Introduction

2.1.1 Définition du partitionnement

Le partitionnement est une technique permettant de diviser un circuit ou un système en au moins deux parties de plus petite taille pour diminuer sa complexité. En effet, lors de la conception d'un circuit, celui-ci est souvent découpé en modules afin de permettre de développer séparément des différentes parties du circuit (principe du "diviser pour régner"). A cause de la croissance exponentiellement rapide de la taille des circuits VLSI[1], le partitionnement est devenu de plus en plus important dans le domaine du développement, de l'implémentation et de la validation de circuits.

2.1.2 Applications du partitionnement

Les applications du partitionnement sont nombreuses. Rien que dans le domaine de la microélectronique, nous pouvons citer [Alp95a], [Joh96] :

- Réalisation de *multi-chip modules* (MCM) : Les MCMs ont été développés pour améliorer la performance d'un circuit comme par exemple son taux d'intégration et sa fréquence par rapport aux cartes de circuits imprimés (*printed circuit boards - PCB*). Pour cela, le circuit doit être découpé en plusieurs modules en respectant un nombre maximal de connexions entre *chips* ainsi que l'emplacement fixe des connecteurs des différents modules sur le substrat [Rie95].
- *Floorplanning* : Une *netlist* est découpée en blocs en respectant des contraintes de taille et d'entrées/sorties. La tâche du *floorplanning* consiste à déterminer les positions des blocs et les positions de leurs pins afin d'optimiser la taille du circuit et de respecter les contraintes de fréquence. Ainsi la qualité du *floorplanning* dépend de la qualité du partitionnement. Un mauvais partitionnement avec beaucoup d'interconnexions entre les blocs rend le *floorplanning* plus difficile et augmente souvent la taille du circuit.

[1] En 1965, Moore a constaté que la taille des circuits double tous les 18 mois à 24 mois. Cette règle est connue sous le nom "Loi de Moore".

- Simulation distribuée : Les *designs* sont souvent testés en utilisant un simulateur logique. A cause de la taille croissante des *designs*, qui implique une augmentation de la place mémoire occupée et du temps utilisé par les simulateurs, il est indispensable de paralléliser la simulation. En particulier, Cong et co-auteurs [Con94a] ont proposé de partitionner le design et de simuler les différentes partitions en parallèle sur une station de travail. Guettaf [Gue98] parallélise également le processus de la simulation. Il propose de partitionner le design et d'utiliser un réseau de stations de travail. Dans les deux cas, l'objectif du partitionnement est de minimiser la communication entre les différentes partitions.

- Emulation et prototypage rapide : La vérification d'un *design* peut être accélérée par l'utilisation d'un émulateur. Dans ce cas, comme dans le cas du prototypage rapide, le *design* doit être implémenté sur un ensemble de circuits programmables interconnectés. Il est donc nécessaire de partitionner le *design* en sous-ensembles qui peuvent être implémentés sur des circuits programmables. Ces sous-ensembles doivent respecter des contraintes de taille et d'entrées/sorties d'un circuit programmable. De plus, le nombre de sous-ensembles est limité au nombre de circuits programmables disponibles.

Selon les applications, les *netlists* à partitionner contiennent des éléments différents : une *netlist* à partitionner pour être réalisée par un MCM est constituée de blocs de cellules, tandis qu'une *netlist* à partitionner pour l'implémentation sur un émulateur contient des LUT (*look-up table*) et des éléments séquentiels.

La description d'un algorithme d'une manière générique nécessite de faire abstraction du cadre de son application. Cette abstraction est réalisée par l'utilisation de modèles communs pour les différentes applications. Cette modélisation est présentée dans la section 2.2 et utilisée dans la section 2.3 pour décrire les différentes méthodes de partitionnement.

Certains des algorithmes ont fait l'objet de comparaisons expérimentales présentées dans la section 2.4.

Ces comparaisons, comme les validations de la plupart des algorithmes présentés section 2.3, ont été effectuées sur des problèmes test qui sont décrits dans la section 2.5.

2.2 Modélisation d'une netlist

Dans la littérature, différentes modélisations d'une *netlist* ont été proposées. Une description détaillée peut être trouvée dans [Gon95][Len90].
Le choix de la modélisation est souvent conditionné par l'objectif du partitionnement. La modélisation la plus réaliste est la représentation d'une *netlist* par un hypergraphe orienté.

2.2.1 Modélisation par un hypergraphe

Définition 2.1 Hypergraphe orienté

Un hypergraphe orienté $H = (V, E)$ est déterminé par la donnée:

- *d'un ensemble V dont les éléments sont appelés les* sommets; *$|V|$ est le nombre d'éléments dans V;*
- *d'un ensemble E dont les éléments $e_i \in E$ sont des couples de sous-ensembles de sommets appelés* hyperarcs *:*

$$e_i = (\{v_j^-\}, \{v_j^+\}) \;\; avec \;\; \{v_j^-\} \subseteq V, \{v_j^-\} \neq \emptyset, \{v_j^+\} \subseteq V, \{v_j^+\} \neq \emptyset, v_j^+ \cap v_j^- = \emptyset.$$

$|E|$ est le nombre d'éléments dans E.
$\{v_j^-\}$ sont les extrémités initiales de e_i et $\{v_j^+\}$ les extrémités terminales de e_i.

Un certain nombre d'algorithmes de partitionnement n'exploitent pas l'orientation des hyperarcs. Dans ce cas, il suffit de représenter la *netlist* par un hypergraphe non-orienté.

Définition 2.2 Hypergraphe non-orienté

Un hypergraphe non-orienté $H = (V, E)$ est déterminé par la donnée :

- *d'un ensemble V dont les éléments sont appelés les* sommets; *$|V|$ est le nombre d'éléments dans V;*
- *d'un ensemble E dont les éléments $e_i \in E$ sont des sous-ensembles de sommets appelés* hyperarêtes *:*

$$e_i = \subseteq V.$$

$|E|$ est le nombre d'éléments dans E.
$\{v_j\}$ sont les sommets incidents à e_i.

Exemple 2.1 Hypergraphe

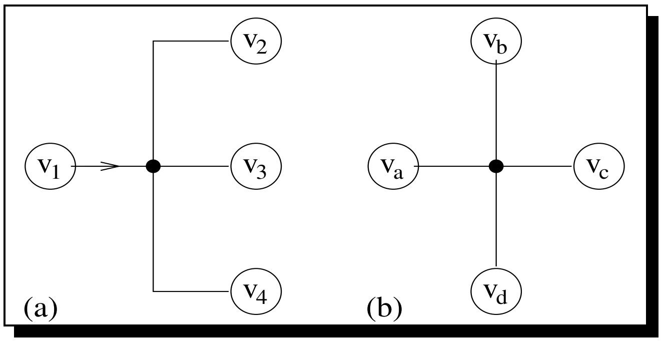


Figure 2.1: *Hypergraphe orienté et non-orienté*

2.1.(a) L'ensemble des sommets $V = \{v_1, v_2, v_3, v_4\}$ est connecté par un hyperarc $e_i = \{\{v_1\}, \{v_2, v_3, v_4\}\}$. L'origine de l'hyperarc est en v_1, ses extrémités terminales sont les sommets v_2, v_3, v_4.
2.1.(b) L'ensemble des sommets $V = \{v_a, v_b, v_c, v_d\}$ est connecté par une hyperarête $e_i = \{v_a, v_b, v_c, v_d\}$. Une représentation graphique de l'hypergraphe orienté et de l'hypergraphe non-orienté est illustrée par la figure ci-contre[2].

Dans le cas où tous les hyperarcs ou hyperarêtes d'un hypergraphe contiennent au plus deux sommets, on parle d'un graphe.

[2] Une représentation graphique alternative [Alp95a][Gon95] met un cercle autour des sommets inclus dans l'hyperarête.

2.2.2 Modélisation par un graphe

Définition 2.3 Graphe orienté

Un graphe orienté $G = (V, E)$ est déterminé par la donnée :

- *d'un ensemble V dont les éléments sont appelés les* sommets. *$|V|$ est le nombre d'éléments dans V;*
- *d'un ensemble E dont les éléments $e_i \in E$ sont des couples de sommets appelés* arcs

$$e_i = (v_j, v_k) \text{ avec } v_j \in V, v_k \in V.$$

$|E|$ est le nombre d'arcs dans E.
v_j est l'extrémité initiale de e_i et v_k l'extrémité terminale de e_i.

Dans le cas où toutes les hyperarêtes de l'hypergraphe contiennent au plus deux sommets, on parle d'un graphe non-orienté.

Définition 2.4 Graphe non-orienté

Un graphe non-orienté $G = (V, E)$ est déterminé par la donnée :

- *d'un ensemble V dont les éléments sont appelés les* sommets. *$|V|$ est le nombre d'éléments dans V;*
- *d'un ensemble E dont les éléments $e_i \in E$ sont des couples de sommets appelés* arêtes

$$e_i = (v_j, v_k) \text{ avec } v_j \in V, v_k \in V.$$

$|E|$ est le nombre d'arêtes dans E.

Exemple 2.2 Graphe

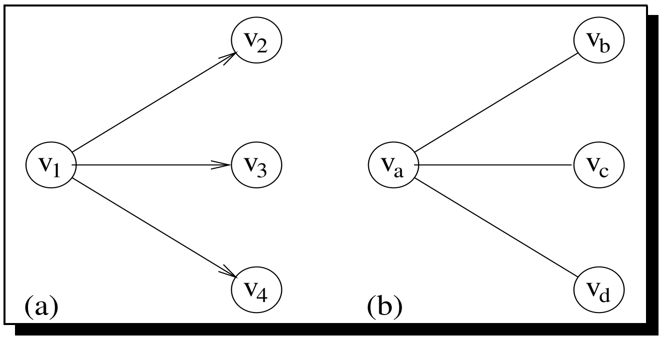


Figure 2.2: *Graphe orienté et non-orienté*

2.2.(a) L'ensemble des sommets $V = \{v_1, v_2, v_3, v_4\}$ est connecté par trois arcs $e_1 = (v_1, v_2)$, $e_2 = (v_1, v_3)$ et $e_3 = (v_1, v_4)$. L'origine des arcs est en v_1, les extrémités terminales sont les sommets v_2, v_3, v_4.
2.2.(b) L'ensemble des sommets $V = \{v_a, v_b, v_c, v_d\}$ est connecté par trois arêtes $e_1 = (v_a, v_b)$, $e_2 = (v_a, v_c)$ et $e_3 = (v_a, v_d)$. La représentation graphique est donnée dans la figurei ci-contre.

L'hypergraphe est la représentation naturelle d'une *netlist*, mais certains algorithmes s'appliquent uniquement aux graphes. C'est le cas des algorithmes utilisant la méthode spectrale qui est basée sur la représentation de la *netlist* par une matrice d'adjacence (cf.

section 2.3.3) ou encore pour les méthodes utilisant le théorème du *max-flow-min-cut* pour déterminer la coupure minimale entre deux partitions par le calcul du flot maximal dans un graphe (cf. section 2.3.2). Il est donc nécessaire de transformer la *netlist*, ou l'hypergraphe représentant cette *netlist*, en un graphe.

2.2.3 Transformation d'un hypergraphe en graphe

Plusieurs méthodes ont été proposées. La plus simple consiste à remplacer toutes les hyperarêtes par une clique : Une hyperarête connectant n sommets $\{v_1, v_2, \ldots, v_n\}$ est remplacée par $n(n-1)/2$ arêtes connectant chaque paire de sommets $\forall_{0 \leq i,j \leq n}(v_i, v_j)$.
Cette transformation a deux inconvénients :

1. La matrice d'adjacence du graphe peut être pleine, bien que la matrice représentant la *netlist* soit creuse.

2. Il n'existe pas de système de poids sur les arêtes permettant de déterminer deux sous-ensembles de sommets ayant la même coupure dans l'hypergraphe et dans le graphe correspondant. Ceci a été prouvé dans [Ihl93].

Exemple 2.3 Modèle de clique
Soit $e = \{v_1, v_2, v_3, v_4\}$ *une hyperarête connectant les quatre sommets* v_1, v_2, v_3 *et* v_4. *Dans le modèle clique, elle est remplacée par les 6 arêtes ayant chacune deux connecteurs* $(v_1, v_2), (v_1, v_3), (v_1, v_4), (v_2, v_3), (v_2, v_4), (v_3, v_4)$ *comme représenté dans la figure 2.3.*

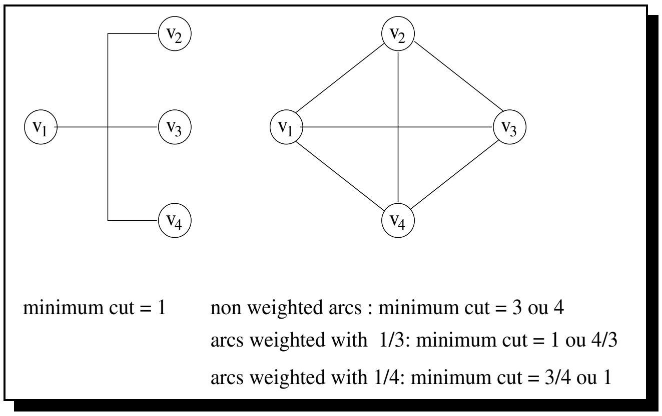


Figure 2.3: *Transformation par le modèle clique*

La coupure séparant deux partitions quelconques dans l'hypergraphe est toujours égale à 1. Le bipartitionnement du graphe varie suivant le poids des arêtes :

- *Soit un poids des arêtes de* 1. *Le bipartitionnement* $\{v_1, v_2\}, \{v_3, v_4\}$ *a une coupure de 4.*

- *Soit un poids des arêtes de* $1/|e| = 0.25$. *Le bipartitionnement* $\{v_1, v_2\}, \{v_3, v_4\}$ *a une coupure de 1. Le bipartitionnement* $\{v_1\}, \{v_2, v_3, v_4\}$, *par contre, a une coupure de 0.75.*

- *Soit un poids des arêtes de* $1/(|e|-1) = 0.33$. *Le bipartitionnement* $\{v_1, v_2\}, \{v_3, v_4\}$ *a une coupure de 1.33, tandis que le bipartitionnement* $\{v_1\}, \{v_2, v_3, v_4\}$ *a une coupure de 1.*

Une autre transformation de base, le modèle en étoile, consiste à remplacer l'hyperarête par un sommet factice et des arêtes entre ce sommet et chaque sommet de l'hyperarête. On obtient donc $|V'| = |V| + |E|$ sommets et $|E'| = f|E|$ arêtes où f est le *fan* moyen des hyperarêtes.
Cette transformation a deux inconvénients :

1. L'ajout d'un sommet factice fait accroître la taille du graphe. Si on considère que chaque cellule d'une *netlist* pilote un *net*, on a autant de sommets que d'hyperarcs dans l'hypergraphe représentant cette *netlist*. Par la transformation suivant le modèle en étoile, on obtient donc environ deux fois plus de sommets dans le graphe que dans l'hypergraphe.

2. Une coupure séparant deux sous-ensembles de sommets dans le graphe ne correspond jamais à la coupure séparant les mêmes sous-ensembles de sommets dans l'hypergraphe, si les poids sur des arêtes doivent être positifs. Ceci a été prouvé dans [Ihl93].

Exemple 2.4 Modèle en étoile
Soit $e = \{v_1, v_2, v_3, v_4\}$ *une hyperarête connectant les quatre sommets* v_1, v_2, v_3 *et* v_4. *Dans le modèle en étoile, elle est remplacée par le sommet factice* v_d *et les 4 arêtes* $(v_1, v_d), (v_2, v_d), (v_3, v_d), (v_4, v_d)$ *comme représenté dans la figure 2.4.*

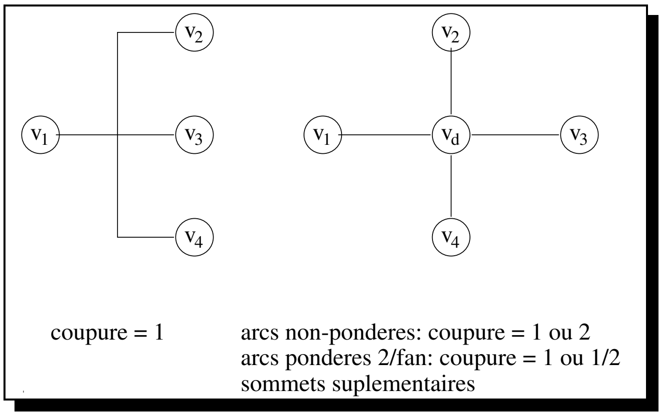


Figure 2.4: *Transformation par le modèle en étoile*

La coupure séparant deux partitions dans l'hypergraphe est toujours égale à 1. Le bipartitionnement du graphe varie suivant le poids des arêtes :

- *Soit un poids des arêtes de* 1. *Le bipartitionnement* $\{v_1, v_2\}, \{v_3, v_4\}$ *a une coupure de 2, tandis que le bipartitionnement* $\{v_1\}, \{v_2, v_3, v_4\}$ *a une coupure de 1.*

- *Soit un poids des arêtes de* $2/|e| = 0.5$. *Le bipartitionnement* $\{v_1, v_2\}, \{v_3, v_4\}$ *a une coupure de 1. Le bipartitionnement* $\{v_1\}, \{v_2, v_3, v_4\}$ *par contre a une coupure de 0.5.*

Après avoir décrit les différents domaines d'application du partitionnement ainsi que les différentes formes de représentation d'une *netlist*, nous allons voir maintenant les différentes méthodes du partitionnement.

2.3 Méthodes de partitionnement

Les méthodes de partitionnement peuvent être classées de différentes manières. Il est possible de faire un classement suivant le domaine d'application ou suivant le type d'algorithmes. Johannes [Joh96] combine ces deux classements. Il différencie le bipartitionnement, le multipartitionnement, le partitionnement avec duplication des cellules et le partitionnement avec des objectifs autres que la coupure, tels que optimiser la performance d'un circuit ou minimiser sa taille.
Alpert [Alp95a] a choisi un classement suivant les propriétés des algorithmes. Il distingue des algorithmes de voisinage, des algorithmes basés sur une représentation géométrique particulière, des algorithmes transformant le problème de partitionnement en un problème combinatoire et des algorithmes de *clustering*.
Sherwani [She93] distingue les partitionnements selon qu'ils s'appliquent au niveau système, au niveau carte ou au niveau circuit. Les algorithmes utilisés peuvent être classés en algorithmes constructifs ou itératifs, déterministes ou probabilistes ainsi que les algorithmes basés sur la migration de groupes entre partitions, les algorithmes évolutionnaires ou autres.

Nous nous sommes intéressés au partitionnement qui optimise le résultat de l'implémentation d'une *netlist* sur une architecture hiérarchique avec des contraintes de taille et de coupure sur chaque niveau de la hiérarchie. L'émulateur CELARO de Mentor Graphics est un exemple pour une telle architecture. L'application concrète des résultats de nos recherches à cet émulateur sera l'objet du chapitre 8. Notre approche consiste à établir un ensemble d'algorithmes de partitionnement où chacun de ces algorithmes est utilisé au mieux connaissant ses avantages et inconvénients. Cet ensemble d'algorithmes a été constitué à l'aide d'une étude bibliographique approfondie. L'objectif étant de choisir le meilleur algorithme de chaque type afin de diversifier le traitement des *netlists*. Nous adoptons donc pour cette étude bibliographique une classification selon le type d'algorithmes :

- Partitionnement "*top-down*", différencié en :
  - partitionnement par algorithmes de voisinage,
  - bipartitionnement par exploitation des flots dans un graphe,
  - partitionnement basé sur le calcul des valeurs et vecteurs propres.
- "*Bottom-up clustering*".
- Duplication de logique.
- Méthodes composites.

2.3.1 Partitionnement "top-down" par algorithmes de voisinage

Les différents algorithmes de voisinages ont en commun de partir d'une solution initiale, puis de chercher une solution voisine meilleure dans l'ensemble des solutions jusqu'à trouver une solution dans un minimum local. Ces algorithmes sont présentés dans les sections 2.3.1.1, 2.3.1.2 et 2.3.1.3. Les algorithmes de voisinage présentés section 2.3.1.4 proposent différentes techniques pour éviter les minima locaux.

2.3.1.1 Algorithmes de bipartitionnement

Kernighan et Lin [Ker70] ont développé un algorithme déterminant la coupure minimale du bipartitionnement d'un graphe[3] en deux partitions de taille équilibrée (bissection). L'algorithme effectue des échanges de sommets entre deux partitions déterminées aléatoirement. Un mécanisme de passes consécutives permet d'éviter certains minima locaux au début du traitement. Chaque passe commence avec la meilleure solution déterminée lors de la passe précédente. Pendant une passe, on échange les deux sommets avec le plus grand gain. Après l'échange, les deux sommets sont verrouillés, ce qui évite des boucles infinies, puis le gain des sommets non verrouillés est mis à jour.
En détail, l'algorithme procède de la manière suivante :

1. Soit $G = (V, E)$ le graphe à partitionner. Détermination d'une solution initiale, c'est-à-dire d'une bissection $(X, \bar{X})$ initiale $X \cup \bar{X} = G$, $X \cap \bar{X} = \emptyset$, $|X| \leq |\bar{X}| \leq (|X| + 1)$.

2. Calcul du coût de déplacement d'un sommet v_i (v_j) de la partition X $(\bar{X})$ dans la partition $\bar{X}$ (X), partition, $\Delta(v_i)$, pour tous les sommets $v_i \in G$ non verrouillés. Pour $v_i \in X$:
   $\Delta(v_i) =$ nombre de connexions de v_i avec des sommets appartenant à la partition X
   $-$ nombre de connexions avec des sommets externes à la partition X

3. Pour toutes les paires de sommets (v_i, v_j) avec $v_i \in X$ et $v_j \in \bar{X}$, calculer le coût d'échanges :

$$\text{coût} = \begin{cases} \Delta(v_i) + \Delta(v_j) & \text{si } v_i \text{ et } v_j \text{ ne sont pas connectés} \\ \Delta(v_i) + \Delta(v_j) + 2n & \text{si } v_i \text{ et } v_j \text{ sont connectés par n arêtes} \end{cases}$$

4. Tant qu'il y a des sommets non verrouillés :
   - Effectuer le meilleur échange, c'est-à-dire l'échange des deux sommets qui engendrent un coût minimal.
   - Verrouiller ces sommets après l'échange.
   - Si la coupure entre les deux partitions a diminué grâce à cet échange, sauvegarder cette configuration.
   - Mettre à jour les sommets non verrouillés connectés à v_i et v_j.
   - Déterminer le coût d'échanges en se basant sur les sommets non verrouillés.

5. Si la coupure déterminée est inférieure à la coupure déterminée préalablement, sauvegarder cette solution comme meilleure solution, puis déverrouiller tous les sommets et retourner en 2).
   Sinon, restaurer la solution sauvegardée lors de la dernière passe, fin de l'algorithme.

[3] L'algorithme n'est pas applicable à un hypergraphe à cause de la fonction de coût utilisée.

Les inconvénients majeurs de cet algorithme sont sa complexité en $O(|V|^3)$ et la restriction de son application sur les graphes.

Fiduccia et Mattheyses [Fid82] ont développé un algorithme proche de celui de Kernighan et Lin. Leur algorithme s'applique aux hypergraphes et effectue des déplacements de sommets d'une partition vers l'autre au lieu des échanges de sommets. La détermination des gains à l'aide des *nets* critiques[4] accélère le calcul du gain d'un sommet. L'organisation des sommets dans une liste des gains permet de réduire la complexité de l'algorithme à $O(|V|)$. Enfin, la suppression de la contrainte d'équilibre entre les deux partitions permet de déterminer des coupures plus petites :

1. Déterminer un bipartitionnement initial, qui n'est pas nécessairement équilibré, mais qui respecte des restrictions sur la taille des deux partitions.

2. Déverrouiller tous les sommets dans les deux partitions après avoir déterminé leurs gains.

3. Création d'une double liste chaînée de gains comme présentée dans la figure 2.5. Le double chaînage facilite l'enlèvement et le replacement de cellules dans la liste des gains.

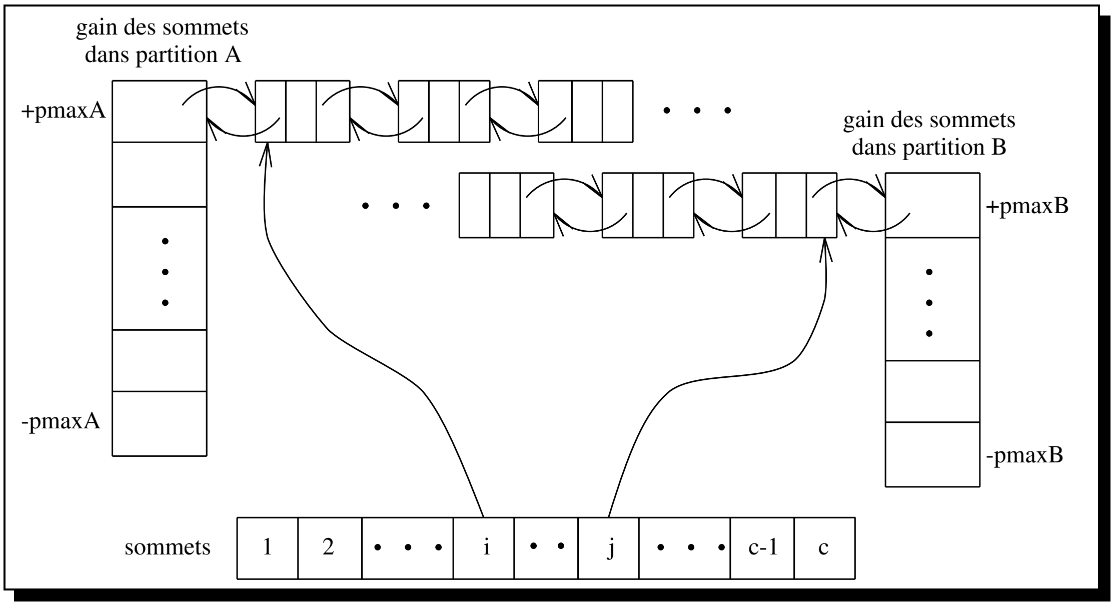


Figure 2.5: *Double liste chaînée*

4. Choisir le sommet avec le plus grand gain et le mettre dans l'autre partition si cela est conforme aux restrictions de la taille des deux partitions.

5. Mise à jour du gain des sommets non verrouillés.

[4] On appelle *net* critique un *net* ayant un sommet qui changerait la coupure si on le passait d'une partition dans l'autre. Un *net* est donc critique si tous les sommets appartenant à ce *net* se trouvent dans la même partition ou si un sommet se trouve seul dans une des partitions.

6. Répéter les points 3 à 5 jusqu'à ce que tous les sommets soient verrouillés.
7. Tant que le meilleur résultat de la passe actuelle est meilleur que celui de la passe précédente, répéter les points 2 à 6.

Krishnamurthy [Kri84] a proposé certaines améliorations de l'algorithme de Fiduccia-Mattheyses :

- Afin de prévoir les futurs mouvements de sommets, Krishnamurthy propose de ne pas restreindre le calcul du gain des sommets sur des *nets* critiques, mais de prendre également en compte les *nets* avec plus d'un sommet dans une des deux partitions. Ainsi, on obtient un vecteur de gain pour chaque sommet, où le premier élément contient le nombre de *nets* critiques connectés au sommet, le deuxième élément le nombre de *nets* ayant deux sommets dans une des deux partitions, etc. Ceci permet de prévoir quel *net* devient critique après le mouvement actuel. Krishnamurthy utilise le vecteur des gains pour choisir le sommet qui va effectuer le mouvement dans le cas où il y a plusieurs sommets avec le même gain maximal. Ce phénomène est observé fréquemment dans le cas où tous les sommets ont le même nombre de connecteurs, comme c'est le cas pour les CLBs des FPGAs de Xilinx [Xil92] ou des CPBs de l'émulateur CELARO [Men99a].
- Les *nets* ayant au moins un sommet verrouillé dans chacune des deux partitions ne peuvent pas disparaître de la coupure. Krishnamurthy parle de *nets* verrouillés. Il propose d'arrêter une passe quand le nombre de *nets* verrouillés dépasse le nombre de *nets* coupés par la meilleure solution déterminée durant cette passe, la coupure ne pouvant être améliorée.
- Afin de sortir d'un minimum local, Krishnamurthy propose d'utiliser un mécanisme présenté par Kernighan et Lin [Ker70] : Les deux partitions sont bipartitionnées et chaque sous-partition de la première partition est fusionnée avec une sous-partition de la deuxième partition. Ensuite, tout le processus de détermination d'une coupure minimale est répété jusqu'à l'obtention d'une nouvelle coupure minimale. Tant que la coupure minimale diminue entre deux phases d'amélioration, on continue le traitement.

Hagen et co-auteurs [Hag95b] se sont inspirés des travaux de Krishnamurthy pour examiner l'influence du choix du sommet d'échanges sur le résultat de la coupure minimale dans le cas où plusieurs sommets ont le même gain maximal.
En effet, un tel choix semble nécessaire : Hagen et co-auteurs [Hag95b] ont constaté lors du bipartitionnement d'une *netlist* de 833 cellules que 10 à 40 cellules en moyenne ont le même gain maximal.
Les expériences menées montrent que le choix implicite de Fiduccia et Mattheyses [Fid82] d'utiliser une liste des gains dans laquelle le dernier sommet inséré est le premier sommet enlevé (*LIFO*) fournit de meilleurs résultats en terme de coupure minimale que l'organisation des sommets dans un ordre aléatoire ou par une *FIFO*.
Pour résoudre le problème du choix entre plusieurs sommets de même gain maximal, Hagen et co-auteurs [Hag95b] proposent un algorithme basé sur celui de Krishnamurthy [Kri84]: Un terme respectant l'attraction effectuée par un sommet déjà verrouillé est pris en compte

pendant le calcul du vecteur des gains. Ainsi, la coupure d'une bipartition baisse sur l'ensemble des *netlists* de *benchs*. La complexité de l'algorithme est de $O(kp|V|)$ où k est la taille du vecteur des gains et p le nombre moyen de *pins* par sommet. Elle est donc la même que celle de l'algorithme de Krishnamurthy.

Dutt et Deng ont proposé dans [Dut96a] une autre variante de l'algorithme de Fiduccia et Mattheyses [Fid82] qui utilise une fonction de gain probabiliste :
en partant d'un bipartitionnement initial tiré aléatoirement, le gain initial g associé à chaque sommet v_i est calculé en dépendance de la probabilité pour un *net* participant actuellement à la coupure d'appartenir toujours à la coupure après un déplacement de cellules: soit $v_i \in V_1$ une cellule connectée au *net* n_i et n_i un *net* ayant des cellules dans les deux partitions. Soit $c(n_i)$ la capacité du *net* n_i et $p(n_{i1})$ la probabilité de n_i de disparaître de la coupure en plaçant toutes les cellules connectées à n_i dans la partition V_2 sous condition que v_i a changé de partition. De plus, soit $p(n_{i2})$ la probabilité de n_i de disparaître de la coupure en plaçant toutes les cellules connectées à n_i dans la partition V_1 sous condition que v_i n'a pas changé de partition, alors

$$g(v_i) = \sum_{\forall i} c(n_i)(p(n_{i1}) - p(n_{i2}))$$

Ensuite, la probabilité $p(v_i)$ pour chaque sommet v_i de passer dans l'autre bipartition est calculé en dépendance de son gain $g(v_i)$.
En partant de cette probabilité, le gain $g(v_i)$ de chaque sommet v_i est calculé de nouveau. Cette itération (calcul du gain, calcul de la probabilité) est effectuée plusieurs fois afin d'obtenir des informations sur les mouvements futurs.
Ensuite, on effectue réellement les meilleurs échanges si la restriction des tailles des partitions est respectée. Les sommets déplacés sont verrouillés et les gains des sommets non – verrouillés et la coupure entre les deux partitions sont mis à jour.
Ceci est répété jusque tous les sommets sont verrouillés ou qu'aucune passe ne peut être effectué à cause du critère de cardinalité des deux partitions.
Finalement, le bipartitionnement qui mène à la plus petite coupure est conservé.
Soit p le nombre moyen de *pins* par sommets et $|V|$ le nombre de sommets. La complexité temporelle de l'algorithme est de $O(p|V|\log|V|)$. La complexité en place mémoire est de $O(p)$.

Dans [Dut96b], Dutt et Deng ont présenté un autre algorithme de voisinage. Cet algorithme est motivé par les hypothèses que des *netlists* réelles sont des aggrégations de petits groupes de cellules fortement connexes (*clusters*) et que la probabilité pour la coupure initiale déterminée aléatoirement de passer à travers de ces groupes est relativement importante[5].
L'algorithme favorise le déplacement d'un *cluster* entier afin qu'il soit entièrement dans une des deux partitions.
En général, la coupures augmente avec chaque mouvement après avoir déplacé un *cluster* entier. Il est donc plus efficace de commencer à déplacer un autre *cluster* au lieu de continuer à déplacer des cellules dans le voisinage du *cluster* déplacé. Pour cela, l'algorithme est le suivant :
Si après un certain nombre de déplacements, la coupure n'a toujours pas diminuée, on estime avoir déplacé un *cluster* entier. L'algorithme restaure donc la meilleure solution

[5] Cette probabilité diminue avec la croissance de la cardinalité de la *netlist* et la diminution de la cardinalité des groupes.

déterminée, sélectionne la cellule avec le meilleur gain global entre les cellules libres[6] et recommence l'étape de mouvement jusqu'à ce que toutes les cellules soient verrouillées.
Les expériences ont été effectuées en utilisant les *benchmarks* de ACM/SIGDA (Partitioning93) contenant entre 619 et 2844 cellules. L'équilibre des partitions a été fixé entre 45% et 55%. Les résultats tiennent uniquement compte de la coupure entre les deux partitions. Les résultats expérimentaux montrent en moyenne une amélioration de 45.5% de l'algorithme par rapport à l'algorithme de Krishnamurthy [Kri84]
Comparé avec des partitionneurs comme MELO de Alpert et Yao [Alp95b] et PARABOLI de Riess et al. [Rie94] la coupure a diminuée de 8.8% à 17.5% par rapport à PARABOLI et de 23% par rapport à MELO.
Le temps de calcul est plus petit que les algorithmes MELO et PARABOLI (facteur 1.2 - 3.3), mais un peu plus lent que l'algorithme FM [Fid82].

Souvent, il n'est pas nécessaire d'obtenir une bissection, c'est-à-dire deux partitions de même taille. Néanmoins, la suppression de cette contrainte mène souvent à des partitions fortement déséquilibrées. Les algorithmes présentés jusqu'ici imposent pour éviter ce phénomène un seuil, c'est–à–dire une cardinalité minimale pour une des partitions. Cette mesure garantie un certain équilibre des deux partitions. Elle risque par contre de passer à côté d'une meilleure solution ne respectant pas les seuils imposés, sans pour autant être déséquilibrée. Pour éviter ce problème, Wei et Cheng [Che91], [Wei91] ont introduit le *ratio-cut* comme fonction de coût.

Définition 2.5 *Ratio-cut*

Etant donné un hypergraphe $H = (V, E)$ avec un ensemble de sommets V et un ensemble d'arêtes E.
Soit c_{ij} la capacité d'une arête entre les sommets i et j.
$(X, \bar{X})$ est appelé une coupure séparant un ensemble de sommets X de son complément $\bar{X} = V - X$. La capacité de cette coupure est :

$$C_{X\bar{X}} = \sum_{i \in X} \sum_{j \in \bar{X}} c_{ij}$$

Le ratio-cut est défini par :

$$R_{X\bar{X}} = \frac{C_{X\bar{X}}}{|X||\bar{X}|}$$

où $|X|$ et $|\bar{X}|$ sont les cardinalités des sous-ensembles X et $\bar{X}$.

L'objectif consiste donc à minimiser cette fonction de coût, c'est-à-dire minimiser la coupure en favorisant en même temps un bipartitionnement équilibré :

$$\forall X, X \subset V, X \neq \emptyset, \bar{X} \neq \emptyset, \min(R_{X\bar{X}})$$

Wei et Cheng ont proposé une heuristique rapide qui a une complexité $O(p)$ avec p le nombre de connecteurs dans un module. Cet algorithme se décompose en 3 phases [Wei91] :

1. Détermination d'une solution initiale.

[6] Il s'agit de cellules non-verrouillées et non-connectées aux cellules verrouillées.

2. Déplacement du sommet avec le plus grand gain de la première partition vers la deuxième partition jusqu'à ce que tous les sommets soient déplacés. Le meilleur bipartitionnement déterminé est restauré et le déplacement recommence. Cette fois ci de la deuxième vers la première partition. Cette phase s'arrête quand l'algorithme se trouve dans un minimum local

3. Déplacement du sommet avec le plus grand gain des deux partitions pour améliorer davantage le critère du *ratio-cut*.

2.3.1.2 Algorithmes de multipartitionnement

L'objectif du partitionnement est rarement le découpage d'un hypergraphe ou d'un graphe en deux sous-ensembles, mais plutôt en un certain nombre de sous-ensembles. Le but est donc d'effectuer un multipartitionnement et non un bipartitionnement.

Dans cette optique, Sanchis [San89] a élargie l'approche de Krishnamurthy [Kri84] au multipartitionnement. L'objectif de son algorithme est le partitionnement d'une *netlist* en un nombre fixe de sous-ensembles tel que la coupure, c'est-à-dire le nombre de *nets* avec des connecteurs dans au moins deux différents sous-ensembles, soit minimale. Le cœur de l'algorithme est le calcul d'un vecteur de gains pour chacune des cellules. La nouveauté par rapport à Krishnamurthy est que le calcul des gains doit prendre en compte la distribution des connecteurs des *nets* sur k sous-ensembles.
La complexité de l'algorithme est de $O(lmk(\log(k) + pl))$ par passe, où k est le nombre de sous-ensembles, p le degré maximal d'une cellule, l le nombre de niveaux de gains utilisés et m la somme des nombres de connecteurs sur l'ensemble des cellules.
L'algorithme a été validé en utilisant des *netlists* générées aléatoirement : A partir d'un nombre de cellules, d'un nombre de *nets* et d'un *fanout* moyen des *nets*[7], toutes les cellules et tous les *nets* sont créés. Puis, les *nets* sont connectés aléatoirement avec les cellules.
Si l'algorithme est soumis à des contraintes de temps de calcul, l'application récursive de l'algorithme de Krishnamurthy donne souvent de meilleurs résultats en terme de nombre de *nets* coupés. Par contre, les résultats de l'algorithme de Sanchis s'améliorent avec un temps de calcul croissant, permettant une augmentation du nombre de niveaux de gains utilisés.

Néanmoins, le gain en terme de nombre de *nets* coupés est obtenu au détriment du temps de calcul.
Cong, Li et Bagrodia [Con94a] proposent d'effectuer d'abord un prétraitement sous forme d'un algorithme de *clustering* et d'appliquer ensuite une variante de l'algorithme de Sanchis [San89] pour obtenir un multipartitionnement. Ainsi, le traitement devient plus rapide, mais il gagne aussi en nombre de *nets* coupés. L'algorithme de *clustering* forme des cônes de logique combinatoire où chaque racine d'un cône est soit une sortie primaire, soit une cellule pilotant un *net* de *fanout* supérieur à 1.
Ainsi, le nombre de *nets* coupés a pu être réduit en moyenne de 50% pour un découpage en 4 partitions et de 33% pour un découpage en 8 partitions des circuits du *benchmark* ISCAS85 par rapport à l'algorithme de Sanchis [San89].

[7] Des *nets* avec un *fanout* moyen de 4 par exemple ont des *fanout* uniformement distribués entre 3,4 et 5.

2.3.1.3 Comparaison entre le multipartitionnement et le bipartitionnement récursif

L'application récursive de l'algorithme de bipartitionnement développé par Fiduccia et Mattheyses [Fid82] fournit de meilleurs résultats en terme de nombre de *nets* coupés que l'application d'un algorithme de multipartitionnement sur une *netlist* à plat [San89]. L'analyse faite par Cong et Lim dans [Con98] a permis d'identifier les causes de ce comportement qui est, à première vue, contre-intuitif : le multipartitionnement de la *netlist* à plat devrait permettre de déterminer de plus petites coupures entre les partitions puisque l'algorithme a "une vue globale" de la *netlist*, tandis que l'algorithme de bipartitionnement peut uniquement couper à l'intérieur d'une partition.
La réalité est bien différente : lors du multipartitionnement, de multiples cellules ont le même gain. Le mauvais choix mène dans ce cas rapidement dans un minimum local. Dans le cas du bipartitionnement récursif, le nombre de cellules ayant le même gain est limité dès la deuxième coupure. L'algorithme tend plus lentement vers un minimum local. Le résultat est une coupure totale entre les partitions nettement plus petite.
De plus, le multipartitionnement selon Sanchis [San89] a une complexité de mémoire de $O(N * K * (K-1))$ qui rend son application prohibitive pour de très grandes *netlists*.
Cong et Lim proposent une méthode pour éviter ces inconvénients : le principe consiste à de déterminer un multipartitionnement initial à l'aide d'un algorithme présenté dans [Con97], puis d'appliquer un algorithme de bipartitionnement sur chaque paire de partitions pour améliorer cette solution initiale.
L'approche permet de gagner en moyenne 86,2% par rapport à l'algorithme de Sanchis [San89] et de 17.3% par rapport à l'application récursive de l'algorithme de Fiduccia et Mattheyses [Fid82] en terme de nombre de *nets* coupés sur des *benchmarks* Partitioning93 [Koz91] et ISPD98 [Alp98].

2.3.1.4 Algorithmes permettant d'éviter les minima locaux

Les algorithmes décrits précédemment [Ker70] [Fid82][Kri84][San89][Con94a] partent d'une solution faisable et cherchent à l'améliorer cette solution par itérations successives. Ils convergent donc rapidement vers des minima locaux. Les algorithmes présentés dans la suite permettent d'éviter un certain nombre de ces minima locaux.

Le recuit simulé La méthode du recuit simulé (en anglais *simulated annealing*), présentée par Kirkpatrick, Gelatt et Vecchi [Kir83], permet d'éviter d'avantage des minima locaux que les algorithmes présentés jusqu'ici. Elle repose sur une analogie entre l'analyse statistique de certains processus physiques et le problème d'optimisation combinatoire :
L'implémentation de l'algorithme de Metropolis et co-auteurs [Met53] pour la simulation des différents états énergétiques lors du recuit d'un corps solide est applicable à d'autres problèmes d'optimisation combinatoire tels que le problème du voyageur de commerce, le placement, le routage ou le partitionnement. Pour ce faire, on a besoin d'une description du système, d'un générateur de "mouvements" aléatoires, d'une fonction d'objectif qui respecte certaines contraintes du système et d'une suite de températures du processus de recuit.
Soit k une solution du problème posé. Le recuit simulé choisit aléatoirement une solution voisine $k+1$ à cette solution. Si cette solution représente une amélioration, elle est retenue comme nouvelle solution. Mais, même si elle représente une détérioration de la

solution actuelle, elle est retenue avec une probabilité de $e^{\frac{-\Delta_k}{\Theta_k}}$ où Δ_k est la différence entre l'énergie de la solution voisine $k+1$ et la solution actuelle k et $\Theta_k > 0$ joue le rôle de la température[8].
La suite des solutions dépend donc de la suite des Θ_k : Si Θ_k tend rapidement vers 0, alors $\frac{-\Delta_k}{\Theta_k} \rightarrow -\infty$ et la probabilité de prendre une plus mauvaise solution tend vers 0. L'algorithme risque donc de s'arrêter rapidement dans un minimum local. Si par contre Θ_k tend trop lentement vers 0, alors la probabilité est grande de prendre une solution plus mauvaise. L'algorithme évite dans ce cas des minima locaux. Il risque par contre de mettre beaucoup d'itérations avant de s'arrêter.
Ceci a été illustré par un exemple : Avec une diminution lente de la température, le bipartitionnement d'un microprocesseur avec 5000 portes logiques et 200 connecteurs d'entrées/sorties donne deux circuits avec 353 et 321 connecteurs où 237 connecteurs correspondent aux *nets* coupés et le reste aux connecteurs externes. Pour une température 0, c'est à dire un refroidissement rapide, on obtient des circuits avec 677 et 730 connecteurs.

La méthode tabou La méthode dite recherche tabou (en anglais *tabu search*), proposée par Glover [Glo89], est une stratégie pour résoudre des problèmes d'optimisation combinatoire. C'est une procédure adaptative pouvant être utilisée en combinaison avec de nombreuses autres méthodes. Ainsi, ses applications sont multiples : de la théorie des graphes aux problèmes de programmation sur les entiers[9].
La recherche tabou effectue des mouvements d'une solution courante vers une solution voisine meilleure. Les k derniers mouvements sont maintenus dans une liste, appelée liste tabou. Les mouvements qui se trouvent dans la liste tabou sont interdits. Ainsi l'algorithme évite de boucler en restant dans une solution déjà déterminée et de rester près d'un minimum local.
Un élément essentiel de la méthode de recherche tabou est l'utilisation d'une fonction d'aspiration. Cette fonction dépend de la solution actuelle et du mouvement envisagé. Son rôle consiste à augmenter la flexibilité des mouvements en permettant des mouvements appartenant à la liste tabou sous condition qu'ils soient "assez bon".
L'introduction d'autres fonctions permet d'améliorer la recherche en l'intensifiant autour d'un minimum local ou en la diversifiant de manière globale afin de quitter une région autour d'un minimum local.
Les performances de la recherche tabou ont été examinées pour de nombreuses applications [Glo89]. Une comparaison des performances de l'algorithme avec celles du recuit simulé pour le problème du coloriage de graphes montre que la recherche tabou produit plus rapidement de meilleurs résultats.

Les algorithmes génétiques Les algorithmes génétiques sont des algorithmes d'optimisation, basés sur la théorie de la sélection naturelle dans l'évolution. Des mutations et des croisements entre membres d'une génération et le fait que des membres les plus adaptés, donc des meilleures solutions, produisent plus de descendants que des membres moins adaptés sont les éléments clés de l'évolution [Hol75].
Ainsi, l'algorithme commence avec une population initiale de solutions. Cette population évolue sur plusieurs générations. Les solutions de la génération actuelle sont remplacées par

[8] Dans un système physique, $\Theta = KT$ avec K la constante de Boltzmann et T la température en Kelvin.
[9] Une liste des applications plus détaillée se trouve dans [Alp95a],[Gon95].

ses descendants obtenus par croisement et mutation.
L'opérateur de croisement permet de mélanger partiellement deux solutions de la génération actuelle. Celui de mutation permet des perturbations aléatoires d'une seule solution.
Un schéma de remplacement décide finalement quel descendant est utilisé comme solution dans la génération suivante.
L'avantage des algorithmes génétiques est qu'ils évitent, comme le recuit simulé et la recherche tabou, de s'arrêter rapidement dans des minima locaux grâce à de multiples solutions de départ qui évoluent au cours des itérations.
Mais le choix des opérateurs de croisement, de mutations et du schéma de remplacement est critique pour la performance d'un tel algorithme. Une étude sur les différentes approches a été réalisée par Beasley et al. dans [Bea93a][Bea93b].

Bui et Moon ont présenté dans [Bui94][Bui98] un algorithme génétique pour le bipartitionnement d'un hypergraphe qui minimise le *ratio-cut* entre les deux partitions :
Leur algorithme part d'un ensemble de solutions initiales tirées aléatoirement qui est appelé population. Cette population évolue pendant plusieurs centaines de générations et le meilleur membre de la dernière génération est la solution retenue du problème posé.
A chaque génération, l'algorithme détermine deux membres de la population par un processus probabiliste. Chaque membre contient tous les sommets de l'hypergraphe qui sont affectés à une des deux partitions dans un certain ordre. Pour créer la descendance, les deux membres (solutions) choisis sont croisés (combinés). Pour ce faire, trois sous-ensembles contigus du premier membre sont concaténés avec trois sous-ensembles contigus du deuxième membre de la population pour créer un premier descendant et avec l'inversion des trois sous-ensembles contigus du deuxième membre pour créer un deuxième descendant.
Des mutations ont lieu pendant la création de la descendance. Chaque sommet change ainsi de partition avec une certaine probabilité.
Finalement, le meilleur des deux descendants est optimisé à l'aide d'une version adaptée de l'algorithme de Fiduccia et Mattheyses [Fid82] : Il effectue une seule passe et le nombre de sommets changeant de partition est limité afin d'accélérer l'algorithme et d'éviter de converger vers un minimum local.
Après la création de la descendance, l'algorithme décide si un membre de la génération actuelle est remplacé par cette descendance. Pour ce faire, l'algorithme compare le *ratio-cut* du meilleur descendant avec le parent le plus proche en terme de nombre de bits communs. Si le descendant a un meilleur *ratio-cut* que le parent, il le remplace, sinon il est supprimé.
A partir d'un certain point, ce schéma de remplacement change : Si le descendant n'a pas un meilleur *ratio-cut* que le parent le plus proche, il est comparé avec l'autre parent. S'il donne toujours un plus mauvais résultat que le parent, il est supprimé, sinon il remplace le parent.
Ce procédé sert à accélérer l'algorithme puisque de nombreuses tentatives de remplacement échouent avec le premier schéma de remplacement.
Enfin, un prétraitement fournit un ordonnancement initial des sommets. Cet ordonnancement a une certaine influence sur le résultat, même si son objectif n'est pas de fournir une bonne solution initiale. Celle-ci est toujours générée aléatoirement[10].
Les opérations de sélection, de croisement, de mutation et de remplacement ont toutes une

[10] La génération d'un "bon" ordonnancement à l'aide de l'algorithme WINDOW de Alpert et Kahng [Alp96] fournit un résultat plus mauvais que l'application de la recherche en profondeur d'abord présentée dans [Bui94]. De plus, le prétraitement ne conduit pas toujours à une meilleure solution du problème. C'est pour cela qu'il reste optionnel.

complexité de $O(|V|)$. L'algorithme de Fiduccia et Mattheyses [Fid82] utilisé pour améliorer le descendant d'une génération a une complexité de $O(|E|)$. La complexité de l'algorithme entier est donc de $O(k|E|)$ où k correspond au nombre de générations. Les expériences présentées dans l'article nécessitent entre 700 et 1600 générations pour aboutir à une bonne solution.
L'algorithme a été testé en utilisant 9 circuits du *benchmark* MCNC et deux circuits industriels. Les résultats ont été comparés avec ceux obtenus avec l'algorithme de Hagen et Kahng [Hag92a] et de Wei et Cheng [Wei91].
L'algorithme génétique de Bui et Moon fournit en moyenne des résultats meilleurs de 12.3% par rapport à l'algorithme de Wei et Cheng et de 29.5% par rapport à celui de Hagen et Kahng.
En même temps, il utilise 40% moins de temps CPU que l'algorithme de Hagen et Kahng, mais 4 fois plus que l'algorithme de Wei et Cheng pour les plus grands *benchs* utilisés (qui restent très modestes d'ailleurs - 12K modules).

Ramam et Patnaik ont présenté dans [Ram96] un algorithme génétique et un algorithme de mutations pour le partitionnement en *multichip modules* (MCM). Ces algorithmes tiennent compte de quatre contraintes simultanément:

- Le nombre maximal de connecteurs de chaque partition fixé par le nombre d'entrée/sorties dans le *slot* du MCM substrat,
- la surface maximal de chaque partition,
- la dissipation de la chaleur de chacune des partitions,
- le délai maximal entre deux partitions.

Ainsi la fonction d'aptitude (angl.:*fitness function*) est composée de quatre fonctions de coût qui tiennent compte de ces quatre contraintes. Elle a la forme

$$F(x) = \frac{A}{B + C_A(x) + C_H(x) + C_P(x) + C_T(x)}$$

où A et B sont des constantes qui sont déterminées par des expériences et C_A, C_H, C_P et C_T des fonction de contraintes de la surface, de la dissipation de chaleur, du nombre de connecteurs et du délai.
Le schéma de sélection des membres qui vont se reproduire combine deux schémas classiques de sélection : Le schéma qui favorise les membres les plus aptes de la génération actuelle (angl.: *stochastic remainder selection scheme*) et le schéma de sélection aléatoire (angl.: *roulette wheel selection scheme*). Ainsi, le schéma de sélection évite de converger rapidement vers un minimum local, comme ce serait le cas avec l'utilisation du premier schéma seul, et de chercher longtemps sans converger, comme ce serait le cas avec l'utilisation du deuxième schéma seul.
Les membres choisis ainsi sont croisés pour créer de nouvelles solutions. Pour cela, Ramam et Patnaik utilisent un opérateur de croisement adaptatif, où la probabilité de croisement est moins grande pour les membres aptes et plus grande pour les membres moins aptes.
L'opérateur de mutation utilisé par Ramam et Patnaik suit les mêmes principes: un membre moins apte a une plus grande probabilité de subir une mutation qu'un membre apte.
L'algorithme a été comparé avec celui de Fiduccia et Mattheyses et le recuit simulé sur 5

circuit benchmark. Les résultats observés sont meilleurs[11].
Ramam et Patnaik ont également présenté un algorithme de mutations. Puisque celui-ci n'effectue pas de croisement, il converge plus rapidement. L'inconvénient de cet algorithme est le temps de calcul important : le bipartitionnement de 49 composants avec 143 *nets* dure 133s, quand les contraintes de délai ne sont pas prises en compte et plus d'une heure, quand ces contraintes sont respectées.

Les algorithmes de mutation Saab et Rao [Saa90] ont développé une heuristique pour déterminer la bissection d'un graphe non-orienté. Cette heuristique n'a pas d'opérateur de croisement comme un algorithme génétique typique, mais un opérateur de mutation plus complexe. Il s'agit donc d'un algorithme de mutations et non d'un algorithme génétique.
L'heuristique part d'une solution initiale, obtenue par construction itérative des deux partitions. Un schéma de classification estime pour chaque sommet des deux partitions, s'il est "bon" ou "mauvais". Un sommet est classifié "mauvais", si la somme des coûts des arêtes internes divisée par la somme des coûts des arêtes externes est inférieur à un nombre tiré aléatoirement dans l'intervalle $[0;1]$. Autrement, il est "bon". Pour éviter des échanges défavorables, comme par exemple la déclaration "mauvais" de tous les sommets d'une partitions, le nombre maximal de sommets "mauvais" a été limité à un quart des sommets.
Soit $|X|$ et $|Y|$ le nombre de sommets jugés "mauvais" dans partition 0 et partition 1. Si $|X| \leq |Y|$, alors les $|X|$ sommets "mauvais" de partition 0 sont échangés avec un ensemble $Z \subseteq Y$, $|Z| = |X|$ de sommets de partition 1. Sinon, les $|Y|$ sommets "mauvais" de partition 1 sont échangés avec un ensemble $Z \subseteq X$, $|Z| = |Y|$ de sommets de partition 0.
La mutation consiste donc à échanger des sous-ensembles de même cardinalité dans les deux partitions.
Les résultats expérimentaux sur des petits *benchmarks* de moins que 200 cellules montrent la même qualité de bissection que l'algorithme de Kernighan et Lin [Ker70] pour un meilleur temps de CPU.

2.3.1.5 Conclusion

Les algorithmes de voisinage sont les algorithmes les plus utilisés dans le domaine du partitionnement. Ceci est du à leur complexité en temps réduite et leur simplicité d'implémentation.
Les algorithmes permettant d'éviter certains minima locaux sont moins intuitifs. L'amélioration de la qualité de la solution se fait au détriment du temps de calcul qui devient beaucoup trop important pour des applications telles que l'émulation. La différence entre la qualité des solutions n'est pas assez importante pour justifier leur utilisation.

2.3.2 Bipartitionnement par exploitation des flots dans un graphe

2.3.2.1 Description du principe

Les algorithmes de partitionnement proposés jusqu'ici sont des heuristiques permettant de déterminer un minimum local du nombre de *nets* coupés entre les partitions. La qualité du partitionnement dépend fortement de la solution initiale et de la rapidité de convergence de

[11] D'ailleurs, les résultats sont difficilement comparables puisque FM et le recuit simulé n'ont pas les mêmes objectifs que l'algorithme génétique présenté ici.

l'algorithme.
Les algorithmes de bipartitionnement basés sur le calcul du flot maximal dans un graphe surmontent ces problèmes en permettant de déterminer la coupure minimale globale entre deux partitions. La première étape consiste à transformer la *netlist*, ou son modèle en forme d'hypergraphe orienté, en un graphe orienté. Plusieurs méthodes de transformation ont été présentées dans la section 2.2.
La deuxième étape consiste à déterminer le flot maximal dans le graphe orienté.

Définition 2.6 *Flot dans un graphe orienté*

Soit $G = (V, E)$ un graphe orienté avec des arcs pondérés et (v_i, v_j) un tel arc de capacité c_{ij}.
Soit $s \in V$ le sommet de départ du flot et $t \in V$ le sommet d'arrivée. On appelle s source et t puits. Le flot entre source s et puits t a les propriétés suivantes :

- *La somme des flux entrant dans un sommet est égale à la somme des flux sortant de ce sommet : $\forall v_j \in V - \{s, t\} : \sum_i \varphi(v_i, v_j) = \sum_k \varphi(v_j, v_k)$.*
- *Le flot sur un arc est supérieur ou égal à 0 et inférieur ou égal à la capacité de l'arc: $\forall (v_i, v_j) \in E, 0 \leq \varphi(v_i, v_j) \leq c_{ij}$.*

Définition 2.7 *Coupe entre les sommets source et puits*
Soit $(X, \bar{X})$ un bipartitionnement de G avec $s \in X, X \subset V$ et $t \in \bar{X}, \bar{X} = V \backslash X$. On appelle coupe séparant les sommets source s et puits t un ensemble d'arcs $(v_i, v_j) \in E$ tels que $v_i \in X$ et $v_j \in \bar{X}$. La capacité $C_{X,\bar{X}}$ de cette coupe est la somme des capacité des arcs dans la coupe : $C_{X,\bar{X}} = \sum_{(v_i,v_j) \in E, v_i \in X, v_j \in \bar{X}} c_{ij}$.

La coupe entre les sommets source et puits définie ci-dessus se distingue de la coupure dans un graphe, car elle est constituée uniquement d'arcs sortant de X.
Ford et Fulkerson ont présenté dans [For62] le théorème faisant la correspondance entre le flot maximal et la coupe minimale :

Théorème 2.1 *Max-flow min-cut theorem:*
Etant donné un flot maximal φ_{max} dans un graphe $G(V, E)$ orienté avec V l'ensemble des sommets et E l'ensemble des arcs. Soit $X = \{v_i \in V :$ il existe un chemin augmentant de la source s à v_i dans $G\}$ et $\bar{X} = V \backslash X$. Alors, $(X, \bar{X})$ est une coupe de capacité minimale égale à $|\varphi_{max}|$ et φ_{max} sature tous les arcs de X à $\bar{X}$.

Une fois le flot maximal déterminé, il faut alors marquer tous les sommets qu'on peut atteindre à partir du sommet source à travers des arcs non-saturés comme appartenant à la partition X et tous les autres sommets comme appartenant à la partition $\bar{X}$, pour déterminer la coupe minimale séparant les sommets source et puits.

2.3.2.2 L'algorithme de Ford et Fulkerson

Ford et Fulkerson ont présenté dans [For62] un algorithme pour calculer le flot maximal entre deux sommets source et puits tirés aléatoirement. Il procède de la manière suivante :

1. Détermination d'un flot initial, compatible avec les contraintes de capacité sur les arcs (Par exemple $(0, 0, \ldots, 0)$).

2. Construction du graphe d'écart $\bar{G}$:
   Pour tous les arcs (v_i, v_j) de G, on fait correspondre au plus deux arcs :

   (v_i, v_j) si le flot $\varphi(v_i, v_j)$ sur cet arc est inférieur à sa capacité $c_i j$
   (v_j, v_i) si le flot $\varphi(v_i, v_j)$ sur cet arc supérieur à 0

   Les capacités sur les arcs sont les suivantes :

$$
\begin{array}{ll}
c_{(v_i,v_j)} = c_i j - \varphi(v_i, v_j) & \text{si l'arc existe} \\
c_{(v_j,v_i)} = \varphi(v_i, v_j) & \text{si l'arc existe}
\end{array}
$$

3. Itération k : Recherche d'un chemin de s à t dans le graphe d'écart $\bar{G}$. S'il n'en existe pas, fin du programme. Le flot de l'itération courante est le flot maximal. Si ce chemin existe, on l'appelle P^k.

4. Soit ϵ^k la capacité du chemin P^k sur le graphe d'écart. Définir le flot φ^{k+1} par :

$$
\begin{array}{ll}
\varphi^{k+1} = \varphi^k + \epsilon^k & \text{si } (v_i, v_j) \in P^k \\
\varphi^{k+1} = \varphi^k - \epsilon^k & \text{si } (v_j, v_i) \in P^k \\
\varphi^{k+1}_{max} = \varphi^k_{max} + \epsilon^k &
\end{array}
$$

   $k = k + 1$ et retourner en 3.

L'algorithme de calcul du flot maximal est un algorithme coûteux en temps de calcul à cause de sa complexité en $O(|V|^3)$ où $|V|$ correspond au nombre de sommets dans le graphe.

2.3.2.3 Amélioration de l'algorithme de Ford et Fulkerson

De nombreux algorithmes de calcul du flot maximal avec une complexité inférieure ont été présentés depuis la publication de Ford et Fulkerson [For62] :
Ford et Fulkerson ont cherché des chemins augmentants quelconques entre les sommets source et puits.
Edmonds et Karp [Edm72] ont démontré que la complexité de l'algorithme de marquage peut être réduite à $O(|V||E|^2)$, si le marquage se fait sur le plus court chemin entre s et t.
L'algorithme de Dinic [Din70] permet de réduire davantage la complexité de l'algorithme de marquage à $O(|V|^2|E|)$ en procédant d'abord à la détermination de "couches" de sommets qui ont la même distance du sommet source en terme de nombre d'arcs. Après la détermination de ces "couches", l'algorithme procède au marquage entre chaque paire voisine de "couches" de sommets en commençant avec la source. Ainsi, l'algorithme de marquage détermine plusieurs chemins augmentants simultanément.
Goldberg et Tarjan [Gol88] ont présenté un algorithme basé sur le concept du *preflow*. Un *preflow* est un flot qui peut être plus grand à l'entrée d'un sommet qu'à sa sortie. L'idée est de conserver un *preflow* maximal dans le graphe et de le propager vers un autre sommet en direction du sommet puits. Cette technique effectue un parcours en largeur d'abord et utilise ainsi le plus court chemin pour propager le flot de la source vers le puits.
Si le puits ne peut pas être atteint à partir d'un sommet, l'excès du flot dans le sommet, c'est-à-dire tout le flot qui entre dans le sommet, est renvoyé en direction de la source. L'algorithme se termine quand tous les sommets ont un excès de zéro. Dans ce cas, le *preflow* devient le flot maximal entre les sommets source et puits.

La complexité de l'algorithme pour calculer le flot maximal dans un graphe dense de $|V|$ sommets est en $O(|V|^3)$. A l'aide de l'utilisation d'une structure de données basée sur des arbres dynamiques (angl. *dynamic tree data structure*), la complexité peut être réduite à $O(|V||E|\log(|V|^2/|E|))$ pour un graphe de $|V|$ sommets et $|E|$ arcs.
Pour un aperçu plus détaillé des méthodes de calcul du flot maximal dans un graphe, voir [Gol88][Gon95].

2.3.2.4 Avantages et inconvénients

Tous les algorithmes de partitionnement basés sur le calcul du flot maximal dans un graphe orienté permettent de déterminer exactement la coupe minimale. Leurs inconvénients sont:

- La cardinalité des partitions obtenues n'est pas très équilibrée. En effet, l'objectif de ces algorithmes n'est pas de séparer deux partitions de même taille, mais de minimiser le nombre d'interconnexions entre elles.
- L'application d'un tel algorithme pour déterminer le bipartitionnement d'une *netlist* nécessite la transformation de l'hypergraphe modélisant la *netlist* en un graphe orienté. A cause de cette transformation et à cause du fait que l'algorithme détermine la coupe minimale, la coupure entre les deux partitions n'est probablement pas la coupure minimale de la *netlist* (voir aussi définition 2.7 dans la section 2.3.2).

Pendant longtemps, les méthodes pour déterminer une coupe minimale par le calcul d'un flot maximal n'ont pas été appliquées dans le domaine du partitionnement à cause de ces inconvénients. Depuis la fin des années 80, son application est devenue plus populaire, grâce à son utilisation pour la duplication de logique. Récemment, plusieurs publications dans le domaine du partititionnement ont surmonté les problèmes relevés.

2.3.2.5 Optimisation

Hu et Moerder [Hu85] utilisent la transformation suivant le modèle en étoile pour construire un graphe non-orienté à partir de l'hypergraphe modélisant un circuit : Un sommet factice est créé pour chaque hyperarête de l'hypergraphe et chaque sommet incident à cette hyperarête est connecté avec le sommet factice par une arête.
Une technique développée par Lawler [Law76] permet ensuite de transformer le graphe non-orienté $G = (V, E)$ en un graphe orienté $G' = (V', E')$, sur lequel il est possible d'appliquer l'algorithme de calcul du flot maximal :

- Pour chaque sommet $v_i \in V$ non factice, il faut créer deux sommets v_{i1} et v_{i2} dans V' ainsi que deux arcs (v_{i1}, v_{i2}) et (v_{i2}, v_{i1}) de capacité infinie dans E'.
- Pour chaque sommet factice $v_{di} \in V$, il faut créer deux sommets v_{di1} et v_{di2} dans V' ainsi que deux arcs (v_{di1}, v_{di2}) et (v_{di2}, v_{di1}) de capacité un dans E'.
- Pour chaque arête $(v_j, v_{di}) \in E$, il faut créer deux arcs de capacité infinie entre les sommets (v_{di2}, v_{j1}) et les sommets (v_{j2}, v_{di1}) dans E'.

On obtient donc $|V'| = 2|V| + 2|E|$ sommets et $|E'| = 2|V| + (2f + 1)|E|$ arêtes où f est le nombre de connecteurs (*fan*) moyen des hyperarêtes.

Exemple 2.5 Modèle de Lawler
Etant donné le modèle en étoile d'un hypergraphe avec un sommet factice v_d ainsi que les 4 arêtes $(v_1, v_d), (v_2, v_d), (v_3, v_d), (v_4, v_d)$. La transformation suivant le modèle de Lawler fournit le graphe représenté figure 2.6. L'application d'un algorithme de calcul du flot

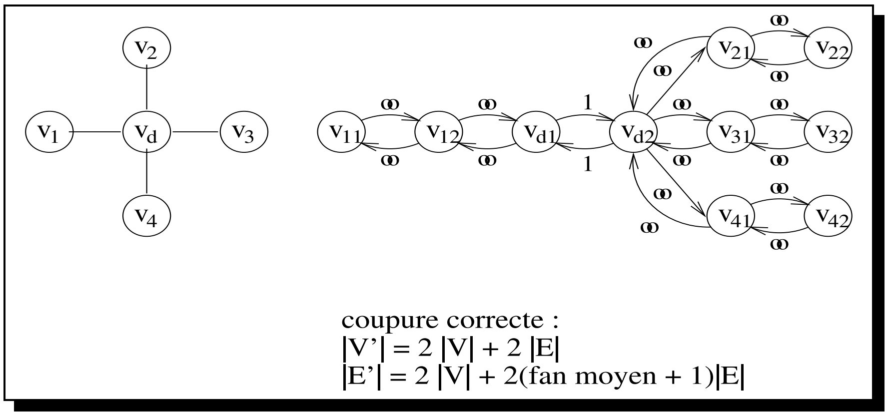


Figure 2.6: *Transformation par le modèle de Lawler*

maximal sature toujours les arcs entre les deux sommets factices, puisque la capacité des autres arcs est infinie. La coupure minimale séparant deux sous-ensembles contient donc uniquement des arcs entre deux sommets factices. Puisque la capacité d'un tel arc est égale à un, on déduit que la coupure est aussi de un, suivant le théorème du max-flow min-cut [For62]. L'application de cette transformation n'introduit donc pas d'erreur dans le calcul de la coupure.

Une simplification de la transformation de Lawler [Law76] a été proposée par Yang et Wong [Yan94][Yan96]. Ils proposent la transformation suivante pour passer du graphe non-orienté $G = (V, E)$ vers un graphe orienté $G' = (V', E')$, où uniquement les sommets factices seront dédoublés :

- Pour chaque sommet factice $v_{di} \in V$, il faut créer deux sommets v_{di1} et v_{di2} dans V' ainsi que deux arcs (v_{di1}, v_{di2}) et (v_{di2}, v_{di1}) de capacité un dans E'.
- L'arc $(v_j, v_{di}) \in E$ est remplacé par deux arcs de capacité infinie entre les sommets (v_{di2}, v_j) et les sommets (v_j, v_{di1}) dans E'.
- Pour chaque arc $(v_{di}, v_j) \in E$, il faut créer deux arcs de capacité infinie entre les sommets (v_{di2}, v_j) et les sommets (v_j, v_{di1}) dans E'.

On obtient donc $|V'| = |V| + 2|E|$ sommets et $|E'| = (2f + 1)|E|$ arêtes où f est le *fan* moyen des hyperarêtes.

Exemple 2.6 Modèle de Yang et Wong
Etant donné le modèle en étoile d'un hypergraphe orienté avec un sommet factice v_d ainsi que les 4 arcs $(v_1, v_d), (v_d, v_2), (v_d, v_3), (v_d, v_4)$. La transformation suivant le modèle de Yang et Wong fournit le graphe représenté dans la figure 2.7. L'application d'un algorithme de calcul du flot maximal fournit pour cette transformation le même flot maximal et ainsi le même résultat en nombre d'arcs coupés en utilisant moins de sommets et d'arcs que le modèle de Lawler.

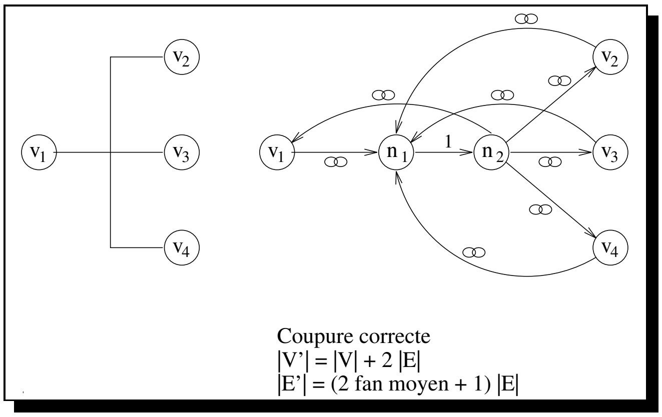


Figure 2.7: *Transformation par le modèle de Yang et Wong*

A partir de cette représentation de l'hypergraphe modélisant une *netlist*, Yang et Wong [Yan94][Yan96] ont développé un algorithme basé sur le calcul du flot maximal suivant l'algorithme de Ford et Fulkerson [For62], permettant de déterminer une coupure minimale menant à un bipartitionnement équilibré de la *netlist*. L'idée consiste à déterminer itérativement une coupure minimale dans le graphe. Si cette coupure mène à deux partitions équilibrées, l'algorithme s'arrête. Sinon, la plus petite des deux partitions est transformée en un seul sommet, un sommet voisin est fusionné avec ce sommet, puis une nouvelle coupure est déterminée.
Soit $H = (V, E)$ l'hypergraphe modélisant une *netlist*. La complexité de l'algorithme est dominée par la complexité de l'algorithme de marquage du graphe lors de la détermination du flot maximal. Cette complexité est en $O(|E|^2)$ pour l'algorithme présenté dans [Yan94][Yan96].

Li, Lillis et Cheng [Li95] ont observé une sensibilité de l'algorithme de Yang et Wong [Yan94] par rapport au choix des sommets source et puits et par rapport au choix du sommet à fusionner avec la plus petite des deux partitions lors du calcul récursif du flot maximal.
Ceci est observé lors du partitionnement répétitif d'une *netlist* avec différents sommets source et puits et différentes méthodes d'agrégation.
Par rapport à l'algorithme de Yang et Wong [Yan94], Li, Lillis et Cheng [Li95] proposent :

- Déterminer un ordonnancement initial qui minimise la distance linéaire des interconnexions à l'aide de la méthode spectrale, puis fusionner plusieurs sommets sur les deux extrémités de cet ordonnancement et utiliser ces sommets fusionnés comme source et puits pour accélérer le temps de calcul.

- Choisir le sommet à fusionner avec la plus petite des deux partitions à l'aide de l'ordonnancement.

- Des coupures minimales de même grandeur sont déterminées par une recherche en largeur d'abord (BFS - *Breadth First Search*). Cet algorithme a une complexité en $O(|E|)$.

- Le flot maximal est déterminé sur un graphe obtenu à partir de l'hypergraphe par une transformation suivant le modèle en étoile. L'algorithme de calcul du flot maximal est celui de Goldberg et Tarjan [Gol88].

Li, Lillis et Cheng [Li95] apportent ainsi une amélioration de 14.3% en moyenne par rapport à l'algorithme FBB [Yan94] en terme de nombre de *nets* coupés sur les circuits du *benchmark* Partitioning93 de MCNC [Koz91].

Liu et Wong [Liu97][Liu98a] présentent avec l'algorithme FBB-MW une extension de l'algorithme FBB de Yang et Wong [Yan94][Yan96]. Celui-ci détermine un multipartitionnement en respectant des contraintes de cardinalité et de nombre maximal de connecteurs. Les améliorations par rapport à l'algorithme FBB sont :
Après le calcul du flot maximal, il y a en général plusieurs coupures minimales possibles. En choisissant la coupure la plus appropriée, c'est-à-dire la plus équilibrée, le nombre d'itérations peut décroître, ce qui peut diminuer le temps de calcul. L'heuristique utilisée pour déterminer la coupure la plus appropriée est la suivante :

1. Chercher à partir du puits t. La partition X_0 est déterminée par :
   $X_0 = \{v \mid \exists \text{ un chemin augmentant de } v \text{ vers } t\}$

2. Chercher à partir de la source s. La partition X_1 est déterminée par :
   $X_1 = \{v \mid \exists \text{ un chemin augmentant de } s \text{ vers } v\}$
   $i = 2$

3. Sélectionner un sommet v incident à $X_j, 1 \leq j < i$
   $X_i = \{u \mid \exists \text{ un chemin augmentant de } v \text{ vers } u \text{ et } u \text{ n'est pas marqué}\}$

4. $i = i + 1$.
   Si tous les sommets ont été marqués, poursuivre en 5
   sinon, poursuivre en 3.

5. Sélectionner k tel que $\max(\sum_{i=1}^{k} \text{area}(X_i)) \leq A_{max}$
   Retourner $X = \bigcup_{i=1}^{k} X_i$

FBB minimise uniquement la coupure introduite par le bipartitionnement entre deux partitions, mais ne tient pas compte de la coupure avec d'autres partitions ou avec l'extérieur. Afin de pouvoir effectuer un multipartitionnement qui respecte un nombre maximal de connecteurs, il faut tenir compte de tous les connecteurs externes d'une partition, c'est-à-dire des connecteurs d'entrées/sorties du graphe ainsi que des connecteurs introduits par la coupure avec d'autres partitions déjà créées. L'algorithme FBB-MW modélise ces connecteurs dans le graphe avant de calculer le flot maximal par une transformation du graphe G en un graphe G' [Liu97][Liu98a] :

1. Tous les sommets du graphe G sont des sommets dans G'.

2. Pour chaque *net* venant d'une autre partition et connecté à plusieurs sommets dans G :

   - Ajouter un sommet factice.
   - Ajouter un arc de capacité infinie de chaque sommet incident à ce *net* vers le sommet factice.
   - Ajouter un arc de capacité 1 du sommet factice vers le sommet puits t.

3. Pour chaque sommet d'entrée/sortie[12] ou pour chaque *net* venant d'une autre partition et connecté à un seul sommet dans G :

   - Ajouter un arc de capacité 1 du sommet incident à ce *net* ou du sommet d'entrée/sortie vers le sommet puits t.

Ainsi, il est assuré que la coupure minimale déterminée à l'aide du calcul du flot maximal corresponde au nombre total de connecteurs des deux partitions.
Le multipartitionnement est finalement obtenu par un bipartitionnement récursif du graphe G'.

2.3.2.6 Conclusion

Récemment, le calcul du flot maximal a été utilisé en combinaison avec d'autres objectifs que la minimisation du nombre de *nets* coupés afin de tenir compte de l'architecture plus complexe des FPGAs. Ainsi, il a été utilisé dans [Liu98b] pour vérifier la validité d'un partitionnement dans des FPGAs contenant des ressources hétérogènes et il a été appliqué aux problèmes du partitionnement pour des FPGAs *time-multiplexed* [Liu98c][Liu99].
Le partitionnement pour l'émulateur CELARO de Mentor Graphics a comme premier objectif de minimiser le nombre de cartes utilisées pour implémenter une *netlist*. La coupure au niveau des AVCs s'est montrée comme étant le point critique de tout le partitionnement. Un algorithme capable de déterminer la coupure minimale est donc d'un grand intérêt. Pour cela, nous avons conçu et implémenté un algorithme basé sur le calcul du flot maximal (cf. chapitre 6.).

2.3.3 Partitionnement basé sur le placement de sommets

2.3.3.1 Description du principe

Les algorithmes de partitionnement de ce type ont en commun d'utiliser une représentation géométrique de la *netlist* : à partir d'une description matricielle d'un graphe modélisant la *netlist*, ils déterminent un ordonnancement unidimensionnel ou une représentation vectorielle dans l'espace à n-dimensions.

2.3.3.2 Placement avec une contrainte quadratique

Hall présente dans [Hal70] une solution pour le problème du placement des sommets d'un graphe dans un espace euclidien de dimension n. Il montre que cette représentation du graphe peut être utilisée pour identifier des *clusters* dans le graphe :
Soit $G = (V, E)$ un graphe contenant $|V|$ sommets et $|E|$ arêtes. Soit c_{ij} le nombre d'arêtes entre sommet v_i et sommet v_j et A la matrice d'adjacence décrivant le graphe avec:

$$A = (c_{ij}) \begin{cases} = 0 & \text{si } v_i = v_j \\ \geq 0 & \text{si } v_i \neq v_j \end{cases}$$

[12] On considère que les entrées/sorties primaires sont modélisées par des sommets. Sinon, il n'est pas nécessaire de faire la distinction entre *nets* venant d'une autre partition et entrées/sorties primaires.

Soit x_i la coordonnée X du sommet v_i.
Le problème unidimensionnel consiste à placer les $|V|$ sommets du graphe sur l'abscisse x tels que la somme des carrés des distances pondérées avec le nombre d'arêtes soit minimale. Autrement dit : il faut déterminer le vecteur $X^T = (x_1, x_2, \dots, x_i, \dots, x_n)$ qui est la solution de

$$\begin{aligned} \min z &= \min \left(\frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} (x_i - x_j)^2 c_{ij} \right) \\ \text{tel que} \quad & X^T X = 1 \\ & \exists (i,j) x_i \neq x_j \end{aligned}$$

Le critère d'optimisation, c'est–à–dire la fonction de coût, est la somme pondérée des distances au carré. Le carré des distances est utilisé pour éviter une distance négative. La condition $X^T X = 1$ permet d'éviter de déterminer une multitude de solutions dépendantes linéairement et de trouver la solution triviale $X = 0$. La condition $\exists (i,j) x_i \neq x_j$ évite que tous les sommets soient placés au même endroit. En effet, il y a au moins deux sommets avec une coordonnée différente.
Etant donnée la matrice d'adjacence $A = (c_{ij})$ et la matrice D :

$$D = (d_{ij}) = \begin{cases} 0 & \text{si } i \neq j \\ \sum_{i=1}^{n} c_{ij} = \sum_{j=1}^{n} c_{ij} & \text{si } i = j \end{cases}$$

d_{ii} est égale à la somme de tous les poids de l'ensemble des connexions entre le sommet v_i et tous les autres sommets du graphe. C'est–à–dire d_{ii} est la somme des poids des *nets* incidents au sommet v_i.
On appelle matrice de disconnexion ou matrice laplacienne du graphe la matrice B :

$$B = D - A$$

Le problème consistant à minimiser les distances pondérées entre sommets peut être transformé comme suit :

$$\begin{aligned} z &= \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} (x_i - x_j)^2 c_{ij} \\ &= \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} (x_i^2 - 2x_i x_j + x_j^2) c_{ij} \\ &= \frac{1}{2} (\sum_{i=1}^{n} x_i^2 c_{i*} - 2 \sum_{i=1}^{n} \sum_{j=1}^{n} x_i x_j c_{ij} + \sum j = 1^n x_j^2 c_{*j}) \\ &= \sum_{i=1}^{n} x_i^2 c_{i*} - \sum_{i=1}^{n} \sum_{j=1}^{n} x_i x_j c_{ij} \\ &= X^T B X \end{aligned}$$

Ces transformations sont possibles car A est symétrique. Nous avons donc $c_{i*} = c_{*j}$. Le problème se transforme donc en un problème de programmation quadratique :

$$\begin{aligned} \min z &= \min X^T B X \\ \text{tel que} \quad & X^T X = 1 \end{aligned}$$

En introduisant le multipliant de Lagrange λ, on peut écrire :

$$L = X^T BX - \lambda(X^T X - 1)$$

Pour minimiser L, on met la première dérivée de L, L', égale à 0:

$$\begin{aligned} L' = 2BX - 2\lambda X &= 0 \\ (B - \lambda I)X &= 0 \\ X^T(B - \lambda I)X &= 0 \\ X^T(BX - \lambda X) &= 0 \\ X^T BX - \lambda X^T X &= 0 \\ \lambda &= X^T BX \end{aligned}$$

La solution du problème est donnée par $X = \mu$, vecteur propre de B et λ la valeur propre correspondante. Il faut donc trouver la deuxième plus petite valeur propre λ_2 et le vecteur propre correspondant μ_2 (la plus petite valeur propre a comme vecteur propre correspondant un vecteur dans lequel tous les éléments sont égaux).
Le premier élément de ce vecteur μ_{21} correspond à l'endroit où est placé le sommet 1, le deuxième élément l'emplacement du sommet 2, etc.
L'extension du problème unidimensionnel au problème n-dimensionnel nécessite le calcul des $(n+1)$ plus petits vecteurs propres $(\mu_1, \mu_2, \ldots, \mu_{n+1})$. Le sommet 1 du graphe est placé aux coordonnés $(\mu_{21}, \mu_{31}, \ldots, \mu_{n+1,1})$, le sommet 2 aux coordonnés $(\mu_{22}, \mu_{32}, \ldots, \mu_{n+1,2})$, etc.
Le placement unidimensionnel peut être utilisé pour déterminer un multipartitionnement en coupant cet ordonnancement aux endroits où il y a une faible concentration de sommets. Cette formule est très vague. En effet, Hall [Hal70] n'a pas décrit comment un tel multipartitionnement peut être déterminé.

Hagen et Kahng utilisent le placement unidimensionnel obtenu avec l'algorithme de Hall [Hal70] pour déterminer le bipartitionnement d'un hypergraphe qui minimise le *ratio-cut* [Hag91] [Hag92a].
L'hypergraphe est transformé en graphe en remplaçant chaque hyperarête avec k *pins* par une clique où chaque arête est pondérée avec un facteur $1/(k-1)$. L'application de l'algorithme de Hall [Hal70] sur ce graphe fournit un ordonnancement linéaire des sommets. En parcourant cet ordonnancement et en calculant à chaque fois le *ratio-cut* entre les sommets ayant un indice inférieur et les sommets ayant un indice supérieur ou égal à l'indice actuel, il est possible de déterminer le bipartitionnement qui minimise le *ratio-cut*.
Des résultats expérimentaux obtenus par l'application de l'algorithme sur des *benchmarks* du MCNC montrent une amélioration du *ratio-cut* de 9% en moyenne par rapport à l'algorithme de Wei et Cheng [Wei91] pour un temps CPU multiplié par 4.
Un temps CPU relativement élevé est d'ailleurs une des caractéristiques de ce type d'algorithme.

2.3.3.3 Placement avec une contrainte linéaire

L'objectif commun des algorithmes de placement de sommets est de minimiser la longueur totale des interconnexions entre sommets et non de minimiser la longueur quadratique.

C'est dans cette optique, que Riess, Doll et Johannes [Rie94] ont développé un algorithme de bipartitionnement qui minimise le critère du *ratio-cut* lors du découpage d'un ordonnancement unidimensionnel. Cet ordonnancement est obtenu à l'aide d'un outil de placement présenté dans [Kle91][Sig91]. Contrairement à Hall [Hal70] et Hagen, Kahng [Hag91], l'outil de placement employé minimise la somme des distances des interconnexions entre sommets avec une contrainte linéaire :

$$\begin{aligned} \min z &= \min\left(\sum_{i=1}^{n}\sum_{j=1}^{n}|x_i - x_j|c_{ij}\right) \\ \text{tel que} & \quad \sum_{i=1}^{n} x_i = f \end{aligned}$$

Ceci peut être transformé en un problème quadratique:

$$\begin{aligned} \min & \quad \sum_{i=1}^{n}\sum_{j=1}^{n} g_{ij}(x_i - x_j)^2 \\ \text{tel que} & \quad \sum_{i=1}^{n} x_i = f \end{aligned}$$

avec $g_{ij} = \frac{c_{ij}}{|x_i - x_j|}$. Ce problème est résolu à l'aide des méthodes itératives présentées dans [Sig91].
Les performances de l'algorithme ont été comparées avec celles de l'algorithme de Hagen et Kahng [Hag91][Hag92a] à l'aide du *benchmark* de l'ACM/SIGDA [Koz91]. La coupure a pu être améliorée en moyenne de 40.9%, le *ratio-cut* de 47.9%. au prix d'un temps de calcul plus important.

Li, Lillis, Liu et Cheng [Li96] ont généralisé la fonction de coût en introduisant un exposant constant α avec $1 \leq \alpha \leq 2$:

$$\begin{aligned} \min z &= \min\left(\sum_{i=1}^{n}\sum_{j=1}^{n}(x_i - x_j)^{\alpha}c_{ij}\right) \\ \text{tel que} & \quad \sum_{i=1}^{n} x_i = f \end{aligned}$$

La transformation de ce problème en un problème quadratique mène à:

$$\begin{aligned} \min & \quad \sum_{i=1}^{n}\sum_{j=1}^{n} g_{ij}(x_i - x_j)^2 \\ t.q. & \quad \sum_{i=1}^{n} x_i = f \end{aligned}$$

avec $g_{ij} = \frac{c_{ij}}{(x_i - x_j)^{2-\alpha}}$. Ce problème peut être résolu par des méthodes itératives [Sig91]. Les résultats expérimentaux obtenus à l'aide des circuits *benchs* de MCNC montrent que le placement linéaire a son minimum en terme de longueur totale des interconnexions pour $\alpha = 1,2$.

Une approche complètement différente à été choisie par Alpert et Yao. Ils ont démontré dans [Alp95b] que le problème consistant à partitionner un ensemble de sommets en k partitions qui minimisent la coupure est identique au problème consistant à déterminer un partitionnement $S_1, S_2, \ldots, S_k$ des d vecteurs propres $\mu_i, 0 \leq i \leq d$ de la matrice laplacienne $Q = D - A$ tel que les k partitions maximisent

$$\sum_{h=1}^{k} \parallel Y_h \parallel^2 \text{ où } Y_h = \sum_{\mu_i \in S_h} \mu_i$$

Ils proposent un algorithme qui détermine un ordonnancement linéaire des sommets à partir de la matrice des vecteurs propres : en commençant avec le vecteur propre qui a la plus grande norme, on choisit à chaque itération le "meilleur" vecteur μ_i non encore traité. Ce vecteur additionné aux vecteurs traités $\mu_j \in S$ maximise la norme de la somme: max $\parallel \sum_{\mu_j \in S} \mu_j + \mu_i \parallel$.
Pour découper l'ordonnancement, Alpert et Yao utilisent l'algorithme DP-RP développé par Alpert et Kahng [Alp95c]. Cet algorithme découpe un ordonnancement linéaire des sommets en un nombre minimal de partitions constituées de sous-ensembles contigus. Ces partitions respectent des contraintes de cardinalité et de coupure. L'algorithme sera présenté plus en détail dans le chapitre 7.

Zien, Schlag et Chan [Zie96] ont posé le problème du multipartitionnement qui minimise la somme du *scaled-cost* sur l'ensemble des partitions en respectant la taille des sommets: soit k le nombre de partitions, E_h la somme des poids des arêtes ayant exactement un sommet à l'intérieur de la partition h, $\parallel P_h \parallel$ la somme des cardinalités de tous les sommets à l'intérieur de la partition h, M la matrice diagonale où m_{ii} est la cardinalité du sommet i et Y la matrice d'affectation qui indique, si oui ou non, un sommet appartient à une partition. Y est de dimension $n \times k$. $y_{ih} = 1$ si le sommet i appartient à la partition h et $y_{ih} = 0$ dans les autres cas. Si R est la matrice d'affectation pondérée avec $r_{ih} = \frac{y_{ih}}{\sqrt{P_h}}$, alors

$$R^T Q R = \sum_{h=1}^{k} \frac{1}{2} \left(\sum_{i=1}^{n} \sum_{j=1}^{n} (r_{ih} - r_{jh})^2 c_{ij} \right) = \sum_{h=1}^{k} \frac{E_h}{\parallel P_h \parallel}$$

correspond à la somme du *scaled-cost* sur l'ensemble des partitions.
Minimiser le *scaled-cost* consiste à résoudre le problème suivant:

$$\begin{array}{rl} \min & R^T Q R \\ \text{tel que} & R^T M R = I \\ & R \in \mathcal{D}_{n,k} \end{array}$$

Actuellement, on ne connaît pas un algorithme de complexité polynômial pouvant résoudre ce problème. Néanmoins, il peut être relaxé, ce qui mène à un problème avec une solution polynomiale :

$$\begin{array}{rl} \min & X^T Q X \\ \text{tel que} & X^T M X = I \end{array}$$

Celui-ci peut être transformé en un problème classique de détermination de valeurs propres d'une matrice :
Soit $M = S^T S$, $Q' = {S^{-1}}^T Q S^{-1}$ et $X' = SX$. On obtient donc après une transformation :

$$\begin{array}{ll} \min & X'^T Q' X' \\ \text{tel que} & X'^T X' = I \end{array}$$

Ce problème consiste donc à déterminer les vecteurs propres de Q'.

2.3.3.4 Avantages et inconvénients

Tous ces algorithmes partent d'une description matricielle de la *netlist*, ce qui implique une transformation de l'hypergraphe, modèle naturel d'une *netlist* en graphe (cf. section 2.2). Les inconvénients d'une telle transformation dans le contexte de ces algorithmes sont :

- L'hypergraphe est transformé en graphe en remplaçant chaque hyperarête par une clique, ce qui modifie la coupure.
- Selon le nombre de *pins* sur les hyperarêtes, la matrice décrivant la *netlist* peut être une matrice pleine bien que le nombre d'hyperarêtes ne soit pas très élevé.

De plus, les temps de calcul rapportés dans les différentes publications sont importants pour les *netlists* utilisées qui sont pourtant de petite dimension.

2.3.3.5 Algorithmes de recherches locales

Alpert et Kahng contournent ces problèmes en développant dans [Alp96] un algorithme qui détermine un ordonnancement par une recherche locale. En commençant avec un sommet quelconque, le prochain sommet à ordonné est celui qui optimise une certaine fonction d'attraction.
Alpert et Kahng proposent les fonctions d'attraction suivantes :

- choisir le voisin du dernier sommet ordonné, ce qui correspond à effectuer une exploration du graphe en profondeur *DFS - depth first search*,
- effectuer une exploration du graphe en largeur (*BFS - breadth first search*),
- choisir le sommet qui a le plus de *nets* en commun avec les sommets déjà rangés (*Max Adjacency Ordering*),
- choisir le sommet qui a le plus de sommets déjà ordonnés comme voisins (*Min Perimeter (King) Ordering*),
- choisir le sommet qui maximise le nombre de *nets* absorbés en cas de groupement (*Absorption Objective*)
- choisir le sommet qui minimise le nombre de sommets ordonnés divisé par la coupure (*Scaled Cost Objective*). Si tous les sommets ont une cardinalité égale à 1, le *Scaled Cost Objective* est équivalent à l'objectif de minimiser de la coupure (*Min Cut Objective*).

Cette implémentation a une complexité de $O(|V|^2)$. La complexité peut être réduite à $O(|V|\log|V|)$, où $|V|$ est le nombre de sommets du graphe.
Afin de ne pas être obligé de traiter tous les sommets ordonnés, seuls les sommets se trouvant à l'intérieur d'une fenêtre déterminée par la variable W exercent une attraction aux sommets non-ordonnés.
En appliquant l'algorithme DP-RP développé par Alpert et Kahng [Alp95c], l'ordonnancement est coupé en un nombre minimal d'ensembles contigus de sommets respectant des contraintes de cardinalité et de coupure et formant ainsi un multipartitionnement de l'hypergraphe.

2.3.3.6 Conclusion

La recherche locale est une approche permettant d'exploiter les connexions locales dans un *design*. L'algorithme de [Alp96] est en plus relativement rapide par rapport aux algorithmes exploitant la méthode spectrale. En combinaison avec l'algorithme DP-RP, il permet d'effectuer un multipartitionnement qui minimise le nombre de partitions tout en respectant des contraintes de cardinalité et de coupure. Il a donc été retenu pour une évaluation plus détaillé qui est présentée dans le chapitre 7.

2.3.4 Clustering

2.3.4.1 Clustering versus multipartitionnement

Alpert et Kahng définissent dans [Alp95a] le *clustering* comme étant un multipartitionnement avec un nombre de partitions important par rapport à la taille de la *netlist* à traiter.
Nous préférons une autre définition : nous parlons de partitionnement, quand l'algorithme a une vue d'ensemble sur la *netlist* et de *clustering*, quand l'algorithme a une vue restreinte sur la *netlist*. Pour cela, nous appelons le partitionnement une approche "*top-down*" et le *clustering* une approche *bottom-up*.

2.3.4.2 Algorithmes basés sur les déplacements aléatoires

Hagen et Kahng [Hag92b] ont déterminé des *clusters* à l'aide des déplacements aléatoires (*random walks*) dans des hypergraphes modélisant des *netlists*.
Un déplacement aléatoire est un processus stochastique à temps discret par lequel on se déplace d'un sommet v_i à un sommet voisin v_j choisi aléatoirement entre tous les sommets voisins du sommet v_i.
L'heuristique proposée est la suivante: On effectue des déplacements aléatoires pour tous les sommets v_i de l'hypergraphe H. Un tel déplacement aléatoire se termine dès qu'il forme un cycle. Tous les sommets visités au cours de ce cycle font parti de l'ensemble $C(v_i)$. Le nombre de fois un sommet v_j a été parcouru pendant un déplacement aléatoire partant de v_i est sauvegardé dans un tableau.
Un critère appelé ressemblance (*sameness*) est calculé à partir des valeurs dans ce tableau pour toutes les paires de sommets de l'hypergraphe. Deux sommets sont regroupés dans un *cluster*, si leur ressemblance est positive.

2.3.4.3 Algorithmes pour minimiser le délai

La caractéristique de ces algorithmes est le regroupement de cellules au long des chemins de logique combinatoire afin de minimiser le délai dans un *design*.
Rajaraman et Wong [Raj93] ont proposé un tel algorithme qui vise à minimiser le délai entre différents circuits d'un *PCB*.
La *netlist* est modélisée par un graphe acyclique orienté. L'algorithme se compose de deux phases :

1. Dans la première phase, chaque sommet v_i du graphe obtient un marquage qui dépend du marquage des sommets prédécents v_j de v_i et qui reflète le délai entre la sortie de v_i et une entrée primaire.

2. Dans la deuxième phase, des *clusters* sont formés en utilisant le marquage effectué lors de la première phase. En commençant avec les sorties primaires et en respectant un nombre maximal de sommets par *cluster*, les sommets ayant le marquage le plus proche des sommets du *cluster* sont réunis avec celui-ci. Si le sommet choisi fait déjà parti d'un autre *cluster*, il est dupliqué.

Brasen et Saucier présentent dans [Bra94] [Bra95] un algorithme de partitionnement en FPGAs. Chacune des partitions déterminées respecte une cardinalité maximale et un nombre de connecteurs maximal. L'objectif de cet algorithme est de minimiser le délai maximal de la *netlist* implémentée sur une carte de FPGAs et de maximiser en même temps le *scaled-cost*[13] afin de minimiser le nombre de FPGAs utilisés. L'algorithme procède de la manière suivante :

- Tous les chemins combinatoires sont identifiés et rangés dans l'ordre décroissant de leur délai associé.

- L'identification des cellules v_i ayant une sortie primaire ou pilotant une cellule séquentielle. Marquer v_i et toutes les cellules se trouvant sur un chemin combinatoire qui se termine en v_i avec une certaine couleur. Ainsi, on obtient des cônes de logique combinatoire. Les intersections de ces cônes, c'est-à-dire les cellules ayant la même combinaison de coloriage, forment des *clusters*.

- L'évaluation du délai des chemins combinatoires en fonction du nombre de *clusters* traversés. Si le délai d'un de ces chemins est supérieur au délai maximal autorisé, l'algorithme effectue un regroupement des *clusters* traversés, tant que le regroupement respecte les contraintes de cardinalité et de nombre de connecteurs.

- Regroupement des *clusters* qui maximisent le critère du *scaled-cost* en respectant les contraintes posées : pour chaque *cluster*, le coût est déterminé en tant que rapport du nombre de cellules à l'intérieur du *cluster* divisé par le nombre d'entrées/sorties de ce *cluster*:
$$\text{Cost}_i = \frac{\text{Size}_i}{\text{I/O}_i}$$

[13] Le *scaled-cost* est le rapport entre la cardinalité et le nombre de connecteurs d'une partition.

Le *cluster* i sera regroupé avec le *cluster* j, si le coût de ce regroupement est maximal.

$$\forall(i,j) \quad \max\,(\mathrm{Cost}_{i,j}) = \max\left(\frac{\mathrm{Size}_i + \mathrm{Size}_j}{\mathrm{IO}_i + \mathrm{IO}_j - \mathrm{IO}_{ij}}\right)$$

Il faut donc calculer le coût de chaque regroupement possible et choisir celui qui est le plus bénéfique. Les éléments séquentiels sont ensuite regroupés avec les *clusters* afin de minimiser le coût total.

Cet algorithme nous semble particulièrement intéressant, puisqu'il vise à minimiser la longueur du chemin critique. Nous verrons au chapitre 8 (section B) par contre que son application dans le cadre de l'émulateur CELARO ne fournit pas les résultats espérés.

L'algorithme présenté par Yang et Wong [Yan95a] poursuit les mêmes objectifs que celui de Brasen et Saucier [Bra94][Bra95] : minimiser le délai tel que les contraintes de cardinalité et de nombre de connecteurs soient respectées.
L'algorithme procède en deux phases appelées marquage et *clustering* : lors du marquage, un cône de logique combinatoire est associé à chacune des cellules v_i. Ces cônes minimisent le délai en sortie de la cellule v_i tout en respectant les contraintes posées. Pendant la deuxième phase, les *clusters* sont formés selon le marquage effectué en commençant avec les sorties primaires et jusqu'à ce que toutes les cellules soient traitées.
Les résultats expérimentaux obtenus dans [Yan95a] sur les circuits du *benchmark* ISCAS85 montrent que le délai a pu être réduit d'un tiers par rapport à l'algorithme de Fiduccia et Mattheyses [Fid82] en utilisant le même nombre de ressources.

2.3.4.4 Algorithmes pour minimiser le nombre de partitions

L'algorithme de Fang et Wu [Fan97] [Fan98] a pour l'objectif de minimiser le nombre de FPGAs nécessaires pour implémenter un *design* tout en minimisant la fréquence de fonctionnement de ce *design* après son implémentation. Cet algorithme de partitionnement consiste de trois phases et s'appuie sur l'exploitation de la hiérarchie du *design*:

1. *Functional-cluster formation* : Le *design* est synthétisé en conservant la structure modulaire. Ainsi, le *design* se présente sous forme d'un arbre contenant plusieurs niveaux hiérarchiques, dont le *top-level*, les modules, les *netlists* correspondantes et finalement les *clusters* fonctionnels.

2. *Slack computation* : Une analyse des délais sur les chemins combinatoires est effectuée afin de déterminer les délais entre *clusters* fonctionnels.

3. *Set-covering with functional replication* : L'objectif de cette phase est l'implémentation du *design* dans un nombre minimal de FPGAs en minimisant simultanément le délai du chemin critique. Ceci est réalisé à l'aide d'une fonction de coût qui contient

   (a) un terme qui prend en compte la connectique entre un sommet et les sommets appartenant déjà à la partition.

   (b) un terme qui détermine la qualité de la partition. Il s'agit du ratio entre le nombre de CLBs et le nombre de *pins* d'entrée et de sortie.

   (c) un terme qui prend en compte l'évolution du délai.

> En commençant avec les sommets du niveau "modules" et en progressant vers un niveau de granularité plus fin, on essaie de couvrir le niveau actuel à l'aide d'un seul FPGA. Si aucune des feuilles (*clusters* fonctionnels) peut être affectée à un FPGA quelconque, un algorithme de duplication fonctionnelle [Kuz94b] est appliqué.

L'algorithme a une complexité de $O(|V|^2 * m)$ avec $|V|$ le nombre de sommet de l'hypergraphe et m le nombre de modules.

2.3.4.5 Algorithmes de préparation pour le partitionnement

Une autre application du *clustering* est la préparation d'une *netlist* au partitionnement en regroupant un petit nombre de cellules. L'intérêt d'une telle méthode a été démontré par Cong et Smith dans [Con93].
Roy et Sechen [Roy93] proposent une méthode de ce type pour accélérer l'algorithme de recuit simulé [Kir83].
Dans la première phase, les *clusters* naturels du circuit sont déterminés : Les sommets connectés par des arêtes d'un poids supérieur à un certain seuil sont regroupées dans un *cluster*. Après avoir traité tous les sommets, le seuil est actualisé. Ces deux étapes sont répétées jusqu'on ne trouve plus d'arêtes avec un poids supérieur au seuil.
L'objectif de la deuxième phase est de créer des *clusters* de taille uniforme. Ceci se fait essentiellement par:

- La suppression de *nets* ayant 2 *pins* par un regroupement des sommets connectés par ce *net*.
- La réduction du *fanout* des *nets*.

Song, Shen et Zhuang [Son98] proposent également un algorithme de *clustering* destiné à la préparation d'une *netlist* pour le partitionnement.
Soit $|V|$ le nombre de noeuds du graphe. L'algorithme présenté permet de calculer les connectivités entre les $\frac{|V|(|V|-1)}{2}$ paires de sommets. Au lieu de calculer les $\frac{|V|(|V|-1)}{2}$ paires d'une manière séquentielle, les paires de sommets sont divisés en $|V|-1$ groupes formant ainsi un arbre : {[1,2], [1,3], ...[1,$|V|$]}, {[2,3] ...[2,$|V|$]}, ...,{[$|V|$-1,$|V|$]}. La recherche dans cet arbre s'effectue une seule fois pour chaque groupe. On parcours donc le graphe $|V|-1$ fois au lieu de $\frac{|V|(|V|-1)}{2}$ fois. De plus chaque groupe est indépendant des autres. La connectivité de chacun des groupes peut donc être calculée parallèlement sur des machines différentes. Puis, les paires de sommets avec la plus grande connectivité sont regroupés dans un *cluster*.
La complexité de l'algorithme est $O(L|V|M^{k+1})$ où L est le nombre d'itérations du processus de *clustering*, M le degré moyen d'un sommet et k l'ordre de la connectivité générale.

2.3.5 Duplication de parties du design

2.3.5.1 Motivation

Le partitionnement en FPGAs est caractérisé par un faible taux de remplissage des circuits par rapport au taux d'occupation de leurs connecteurs d'entrées/sorties. Cette caractéristique a motivé le développement des algorithmes de duplication, qui renoncent à la contrainte que les partitions doivent être disjointes. En effet, le principe de ces algorithmes consiste à dédoubler des parties de logiques dans différents FPGAs chaque fois que cela apparaît bénéfique pour diminuer la coupure.

2.3.5.2 Objectif de la duplication

L'objectif de la duplication est une baisse du nombre de connecteurs des partitions. Cette baisse facilite le routage entre partitions et permet, en relâchant les contraintes de coupure, un taux de remplissage plus élevé.
La duplication de sommets est également appliquée dans des algorithmes qui ont pour objectif de minimiser le délai d'un *design*. Le délai entre deux circuits est plus grand que le délai entre deux portes logiques d'un même circuit. Un *net* pilotant des portes logiques dans différentes partitions appartient à la coupure entre ces partitions. La duplication de la porte logique pilotant ce *net* enlève celui-ci de l'ensemble des *nets* coupés. Le chemin auquel appartient le *net* ne traverse plus les bornes d'un circuit. Le délai est donc inférieur lorsque la porte est dupliquée.

2.3.5.3 Principe de la duplication

La duplication dans la partition j d'une cellule v_k faisant parti de la partition i est illustrée à l'aide d'un exemple :

Exemple 2.7 Duplication d'un sommet dans un graphe orienté
Soit $G = (V, E)$ *un graphe orienté avec l'ensemble des sommets* $V = \{v_0, v_1, \ldots, v_7\}$ *et* P_1, P_2 *un bipartitionnement de* G *avec les partitions* $P_1 = \{v_0, \ldots, v_4\}$ *et* $P_2 = \{v_5, v_6, v_7\}$, *comme présenté dans la figure 2.8 [Hwa95].*

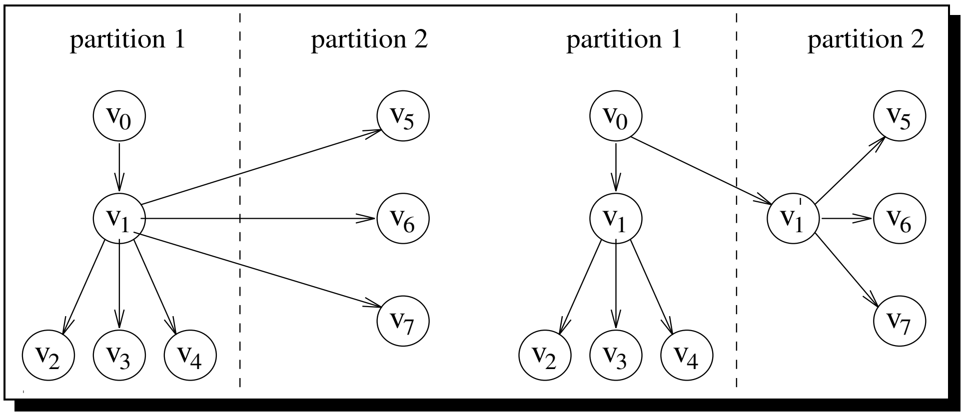


Figure 2.8: *Exemple d'un sommet dupliqué dans un graphe orienté*

La duplication du sommet v_1 *dans la partition 2 est effectuée en trois étapes :*

- *On ajoute le sommet* v_1' *à l'ensemble des sommets dans la partition 2 :* $P_2 = P_2 \cup v_1'$.
- *Pour chaque arc à l'entrée de* v_1 *venant d'un sommet* $v_i \in P_1$, *on ajoute un arc de* v_i *vers* v_1'. *Ici, il faut donc ajouter l'arc* (v_0, v_1').

- *Chaque arc sortant de v_1 allant vers un sommet $v_j \in P_2$ est remplacé par un arc sortant de v'_1 allant vers le sommet v_j. Ici, il faut donc remplacer les arcs $(v_1, v_5), (v_1, v_6), (v_1, v_7)$ par les arcs $(v'_1, v_5), (v'_1, v_6), (v'_1, v_7)$.*

Grâce à la duplication du sommet v_1, la coupure entre partition 1 et partition 2 a pu être réduite de 3 arcs à 1 arc.

La duplication d'une cellule agit donc sur la coupure comme suit :
Elle enlève de la coupure des *nets* en sortie de la cellule dupliquée allant vers des cellules qui se trouvent dans la partition où la duplication a eu lieu. Elle introduit dans la coupure les *nets* d'entrée de la cellule dupliquée venant d'une autre cellule dans la partition d'origine. Dans le cas d'une *netlist* réelle contenant des *nets* de *multi-fanout*, les *nets* enlevés de la coupure sont des *nets* en sortie de la cellule dupliquée allant vers des cellules qui se trouvent uniquement dans la partition d'origine et dans la partition où la duplication a eu lieu. Les *nets* introduits par la duplication sont dans ce cas les *nets* en entrée de la cellule dupliquée connectés à des cellules qui se trouvent uniquement dans la partition d'origine. Un exemple de duplication d'un sommet dans un hypergraphe est présenté dans la figure 2.9.

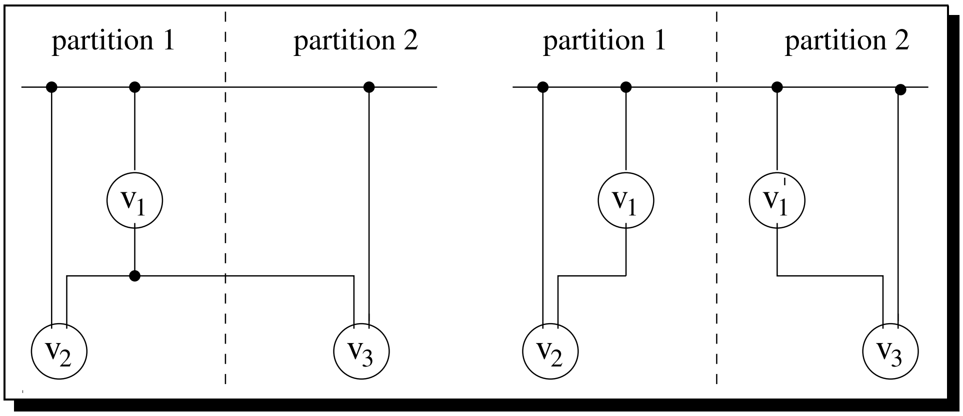


Figure 2.9: *Exemple d'un sommet dupliqué dans un hypergraphe*

2.3.5.4 Algorithmes de duplication de sommets

Kring et Newton [Kri91] ont développé un algorithme pour le bipartitionnement d'un hypergraphe. Cet algorithme introduit la technique de duplication de sommets dans l'algorithme de Fiduccia-Mattheyses [Fid82].
Soit (A, B) un bipartitionnement de l'hypergraphe $H = (V, E)$. Un sommet peut appartenir soit à la partition A soit à la partition B soit, dans le cas d'une duplication, aux deux partitions simultanément, donc à AB.
Trois types de déplacements sont donc possibles:

1. Le mouvement : $A \to B$ et $B \to A$.
2. La duplication : $A \to AB$ et $B \to AB$.
3. L'inverse de la duplication : $AB \to A$ et $AB \to B$.

Par conséquent, les *nets* peuvent avoir 8 états: $A, A_A, B, B_B, AB, AB_A, AB_B$ et AB_{AB}. Le corps du nom du *net* indique le nom de la partitions à laquelle il appartient et l'indice la partition dans laquelle il est critique.
Il est primordial de limiter le nombre de sommets dupliqués. Pour ce faire, Kring et Newton [Kri91] proposent deux stratégies. La première stratégie consiste à favoriser des déplacements de type 3 (inverse de la duplication) : dès qu'un sommet dupliqué a un gain dépassant un certain seuil pour un déplacement de type 3, ce déplacement est effectué, même si le gain du sommet n'est pas maximal. La deuxième stratégie consiste à définir un gain minimal à partir duquel les déplacements de type 2 (duplication) sont autorisés.
L'algorithme emploie la même structure de données et les mêmes mécanismes de sélection que l'algorithme de Fiduccia et Mattheyses [Fid82]. Sa complexité est donc de $O(|E|)$.

Hwang et El Gamal [Hwa95] ont également présenté un algorithme de duplication de sommets. Cet algorithme est basé sur le calcul du flot maximal dans un graphe. Il permet de déterminer l'ensemble des cellules à dupliquer pour minimiser la coupure entre k partitions issues d'un multipartitionnement. L'algorithme ne peut toutefois pas assurer que le nombre de cellules à dupliquer soit inférieur à une valeur maximale. En effet, le problème de duplication de cellules en respectant une taille maximale et un nombre maximal de connecteurs pour chaque partition est un problème NP-complet.
Il est donc nécessaire de définir une heuristique, capable de traiter ce problème. Hwang et El Gamal proposent d'utiliser une variante de l'algorithme de Fiduccia et Mattheyses (FM) [Fid82] qui tient compte de l'orientation des *nets*[14]. Les différences par rapport à l'algorithme décrit dans [Fid82] sont :

- Le calcul du gain est modifié afin de tenir compte de la direction de la duplication.

- Les contraintes sont appliquées uniquement à la partition dans laquelle la duplication a lieu.

Cet algorithme (FM-BiRep) est la base de deux heuristiques pour le multipartitionnement: la première heuristique applique simplement FM-BiRep à chaque paire de partitions. La deuxième heuristique (MC-Rep) applique d'abord l'algorithme qui détermine l'ensemble de cellules à dupliquer à chaque paire de partitions. Si la solution déterminée est applicable, c'est-à-dire la duplication respecte les contraintes, les cellules concernées sont dupliquées.
Les deux heuristiques font une seule passe sur toutes les partitions. L'heuristique FM-BiRep a une complexité de $O(|V||E|)$, tandis que l'heuristique MC-Rep a une complexité $O(k|V||E|\log(\frac{|V|^2}{|E|}))$.
L'algorithme a été testé à l'aide du *benchmark* Partitioning93 [Koz91], contenant 534 à 33553 portes, et de dix *netlists* industrielles contenant 453 à 2013 portes. Ces *netlists* ont été partitionnées en FPGAs, puis la coupure modifiée par les algorithmes FM-BiRep et MC-Rep. Le nombre de connecteurs a pu être diminué en moyenne de 15% pour FM-BiRep et de 18% pour MC-Rep par rapport à la coupure initiale obtenue avec l'algorithme de Fiduccia et Mattheyses [Fid82]. En même temps, le taux de remplissage moyen des des circuits est passé de 42% à 54%.
Concernant le temps de calcul, FM-BiRep est beaucoup plus rapide que MC-Rep, particulièrement sur les grands *benchs*.

[14] L'algorithme de Fiduccia et Mattheyses traite des hypergraphes non-orientés.

Pour pouvoir appliquer leur heuristique à des hypergraphes, Hwang et El Gamal [Hwa95] ont transformé chaque hyperarc de multiple *fanout* en un arbre, comme illustré dans la figure 2.10.

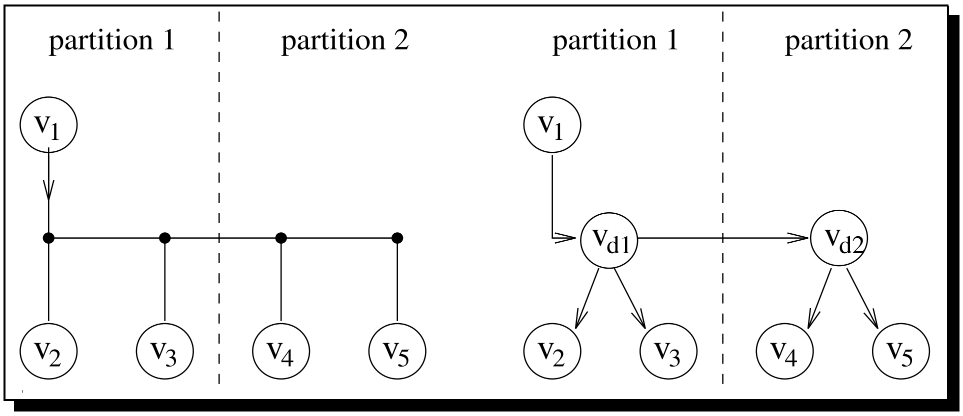


Figure 2.10: *Transformation d'un net de multiple fanout en un arbre*

La détermination des sommets à dupliquer dans chacune des partitions permet ensuite de minimiser la somme des connecteurs en entrée de ces partitions ($\min \sum_i |\text{in}(V_i)|$).
Yang et Wong se sont intéressés dans [Yan95b] à minimiser le nombre d'hyperarcs coupés. Ils considèrent que chaque hyperarc est piloté par exactement un sommet. Le moyen permettant de minimiser le nombre d'hyperarcs coupés est donc la duplication d'un ensemble de sommets de la partition V_i dans $\bar{V_i}$ tels que le nombre de connecteurs en sortie de la partition V_i soit minimal :
Chaque partition de l'hypergraphe est d'abord transformée en un réseau de duplication :

- Le sommet source est connecté à chaque entrée primaire de V_i par un arc de capacité infinie.
- Chaque hyperarc est remplacé par un sommet factice, un arc de capacité 1 entre le *driver* de l'hyperarc et le sommet factice ainsi que des arcs de capacité infinie entre le sommet factice et des sommets pilotés par l'hyperarc.

Après avoir calculé le flot maximal [Yan94] dans le réseau de duplication, l'ensemble des sommets à dupliquer est déterminé de la manière suivante pour chaque partition i :

- Soit T l'ensemble de sommets sur un chemin augmentant vers le sommet puits t et $S = V' \backslash T$
- Soit $W_i = T \backslash (\{v \in \bar{V_i} \cap v \in e | e \subset V_i \cap e \subset \bar{V_i}\} \cup \{t\})$.
- Pour tous les $j = 1, \ldots, k, j \neq i$:
  $W_{ij} = \{w \in W_i | w$ peut être atteint à partir de V_j uniquement à travers des sommets dans $W_i\}$.

L'algorithme a une complexité de $O(knP)$ où k est le nombre de partitions, n le nombre d'hyperarcs et P le nombre de connecteurs dans l'hypergraphe.
La duplication de sommets avec contraintes de taille et de connecteurs est un problème

NP-complet comme présenté par Hwang et El Gamal [Hwa95]. L'algorithme proposée par Yang et Wong applique récursivement l'algorithme de bipartitionnement proposé dans [Yan94][Yan96] pour réduire la taille des ensembles de sommets à dupliquer.
L'algorithme a été testé à l'aide du *benchmark* Partitioning93 [Koz91]. Une comparaison avec l'algorithme proposé par Hwang et El Gamal [Hwa95] montre que l'algorithme coupe 57% de *nets* en moins. Le temps de calcul sur une station de travail SPARC10 passe de 2h30 à 3 minutes pour un *bench* de 30K portes.

2.3.5.5 Algorithmes de duplication fonctionnelle

Un autre type de duplication, la duplication fonctionnelle, a été introduite par Kuznar, Brglez et Zajc [Kuz94a]. Cette duplication est intégrée dans un algorithme de multipartitionnement de *netlists* pour l'implémentation dans des FPGAs de type hétérogène. Son objectif est de minimiser le coût total des FPGAs nécessaires pour cette implémentation ainsi que le nombre d'interconnexions entre les différents FPGAs. L'algorithme est basé sur la duplication fonctionnelle (*functional replication*) de certaines parties de la *netlist*. Cette technique nécessite des connaissances sur la fonctionnalité de la cellule dupliquée : uniquement les *nets* fonctionnels sont réellement connectés.

Exemple 2.8 Duplication fonctionnelle
Le principe de la duplication est présenté dans la figure 2.11.

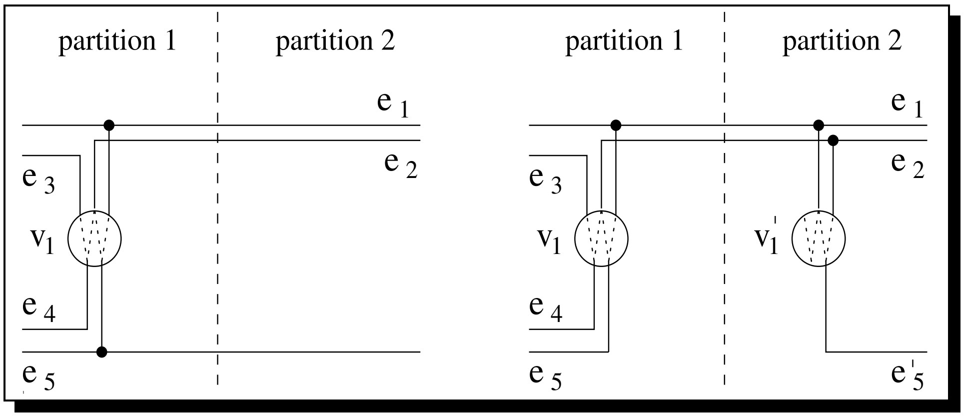


Figure 2.11: *Transformation d'un net de multiple fanout en un arbre*

Lors de la duplication du sommet v_1, uniquement, les nets desquels dépend le net e'_5 sont réellement connectés au sommet dupliqué v'_1, c'est-à-dire e_1 et e_2. Ceci nécessite évidemment une connaissance sur les dépendances entre nets de sortie et nets d'entrée du sommet dupliqué.

Les connecteurs et le statut des *nets* correspondants sont décrits par des vecteurs binaires. Si un *net* est critique, l'élément correspondant dans le vecteur est égal à 1, sinon, il est égal à 0.
A partir de ces vecteurs, on peut facilement calculer le gain d'un simple déplacement d'une cellule, le gain d'une duplication traditionnelle et le gain d'une duplication fonctionnelle. Un mécanisme de mise à jour assure la rapidité de l'algorithme.
Les performances de l'algorithme ont été examinées en utilisant les *netlists* du *benchmark* Partitioning93 [Koz91]. Suivant le seuil du potentiel de duplication choisi, entre 0% et 19.8%

des cellules ont été dupliquées. Les résultats du multipartitionnement de ces *netlists* ont été comparés avec l'algorithme présenté dans [Kuz93] : pour un coût comparable, l'utilisation des IOBs[15] a pu être réduite de 77% à 67%. Par contre, le temps de calcul est très long : le multipartitionnement d'une *netlist* de 2904 CLBs dure plus d'une heure.
L'inconvénient majeur de cette méthode est qu'elle est efficace uniquement dans le cas où la logique à dédoubler possède plusieurs sorties. Ceci est vrai pour les CLBs de la famille XC30xx de Xilinx [Xil92], mais pas pour les CPBs utilisés dans les circuits programmables de l'émulateur CELARO. Une utilisation de cet algorithme n'est donc pas envisageable au niveau partitionnement de CPBs.

2.3.5.6 Augmentation de la fréquence

La duplication de sommets a été appliquée jusqu'ici pour minimiser la coupure entre partitions. Elle peut également être utilisée pour augmenter la fréquence d'un *design*. Ainsi, Liu *et al.* présentent dans [Liu95a] un algorithme de partitionnement qui utilise des méthodes de *retiming*[16] et de duplication de sommets pour obtenir de bonnes performances en terme de fréquence de fonctionnement.
L'approche est basée sur la construction d'un réseau de duplication contenant le graphe G ainsi qu'une copie du graphe G' dans laquelle tous les arcs sont inversés. Chaque sommet du graphe G est connecté avec sa copie par un arc de capacité infinie. Enfin, deux sommets appelés *super-source* et *super-puits* sont créés et connectés aux sommets source et puits du graphe et de sa copie par des arcs de capacité infinie.

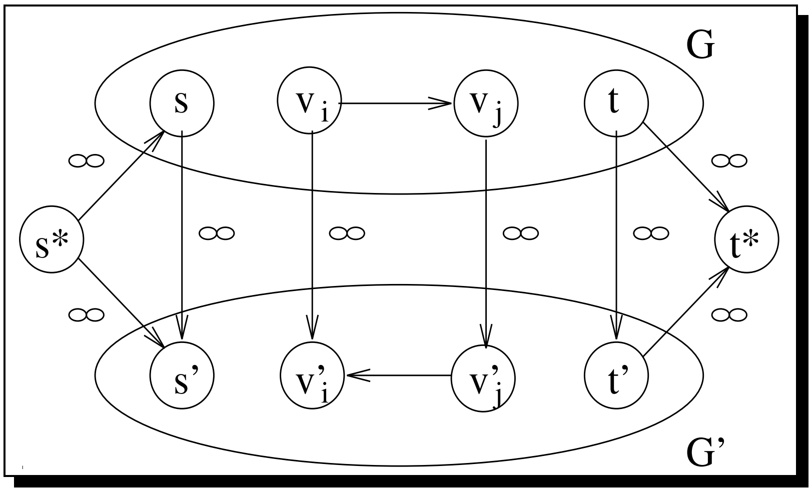


Figure 2.12: *Réseau de duplication*

L'algorithme modélise le problème de duplication en vue d'une amélioration temporelle par un problème de programmation en entiers. La relaxation du problème en entier fournit une borne inférieure de la qualité de la solution.
La dualité de la relaxation du programme a induit le concept d'une fonction de pénalité

[15] *Input Output Block*, voir glossaire

[16] Soit $G = (V, E)$ un *data-flow graph*. Le *retiming* spécifie une transformation du graphe d'origine en un graphe ayant des registres ajoutés et enlevés. Leiserson *et al.* [Lei91] ont proposé un algorithme utilisant cette transformation pour optimiser la fréquence. L'algorithme a une complexité de $O(|V||E|\log(|V|)$ où $|V|$ est le nombre de sommets et $|E|$ le nombre d'arcs.

pour les capacités des arcs : à chaque itération, l'algorithme identifie les boucles[17] qui ne respectent pas les contraintes de délais. La fonction de pénalité ajoute des poids sur les arcs des boucles afin d'éviter de couper ces boucles. L'itération s'arrête quand une solution est trouvée ou quand le nombre maximal d'itérations est atteint.
Des expériences ont été effectuées en utilisant sept circuits industriels contenant entre 3383 et 12717 cellules. Le nombre de cellules dupliquées a été limité à 20% du nombre total de cellules. Les performances de l'algorithme ont été mesurées en terme de nombre de *nets* coupés et de longueur du chemin critique lors d'un bipartitionnement. Ces performances ont été comparées avec celles obtenues à l'aide de l'algorithme de Fiduccia et Mattheyses [Fid82] (FM) et l'algorithme de Liu et co-auteurs [Liu94] (LAMP). L'algorithme de duplication fournit en moyenne une coupure de 22.2% inférieure à celle obtenue avec FM et de 23.8% inférieure à celle obtenue avec LAMP. En même temps, la période d'un cycle d'horloge est réduite en moyenne de 27.6% pour FM et de 3.4% pour LAMP.
Pour les sept circuits, le nombre de sommets dupliqué est compris entre 12.5% et 20%.

Mak et Wong [Mak96] ont développé un algorithme de duplication de sommets basé sur le calcul du flot maximal dans un graphe. Le graphe d'origine est transformé en un réseau de duplication comme présenté dans [Liu95a]. Soit $(A, \bar{A})$ un bipartitionnement du réseau de duplication qui coupe l'ensemble des sommets du graphe en partitions $(X, \bar{X})$ et l'ensemble des sommets de la copie du graphe G' en partitions $(Y', \bar{Y'})$.
Il a été démontré dans [Liu95a] que $S = X$, $T = \{v | v' \in \bar{Y'}\}$ et $R = V - S - T$ est une solution optimale du problème de duplication de sommets dans un graphe.
Il faut donc dupliquer tous les sommets v pour lesquels $v' \in A$ et $v \in \bar{A}$. Pour ce faire, Mak et Wong [Mak96] ont proposé la construction d'un nouveau réseau N_∞ après avoir déterminé le flot maximal dans le réseau de duplication N. N_∞ contient les mêmes sommets que N et pour chaque arc (v_i, v_j) dans N, on construit :

- un arc (v_i, v_j) de capacité infinie dans N_∞, si le flot sur cet arc dans N est inférieur à sa capacité.
- un arc (v_j, v_i) de capacité infinie dans N_∞, si le flot sur cet arc dans N est supérieur à 0.

Le nouveau réseau N_∞ a la même coupure minimale que le réseau de duplication N.
En ajoutant pour chaque sommet v_i dans le G des arcs (v_i', v_i) de capacité correspondante à la cardinalité du sommet v_i dans le réseau N_∞, il est possible de déterminer la cardinalité de tous les sommets à dupliquer par la coupure minimale correspondant au flot maximal dans N_∞.
La complexité de l'algorithme est dominé par la complexité de l'algorithme de calcul du flot maximal de Goldberg et Tarjan [Gol88]. Il a une complexité en $O(|V||E| \log \frac{|V|^2}{|E|})$.
L'algorithme proposé par Mak et Wong permet également de réduire la cardinalité de l'ensemble de sommets à dupliquer déterminé par une autre heuristique de duplication comme par exemple celle de Kring et Newton [Kri91] ou celle de Hwang et El Gamal [Hwa95].
Pour ce faire, les auteurs considèrent l'ensemble de sommets de la partition à laquelle appartient le sommet source et l'ensemble de sommets de la partition à laquelle appartient le sommet puits comme deux ensembles initiaux qu'il faut départager. L'application

[17] Les boucles résultent de la transformation du graphe d'origine en un graphe de duplication.

de l'algorithme présenté ci-dessus permet finalement de déterminer l'ensemble minimal de sommets à dupliquer pour obtenir la coupure minimale.
L'algorithme a été testé à l'aide du *benchmark* ISCAS85.
Par rapport à l'algorithme de Liu et co-auteurs [Liu95a] qui duplique entre 21% et 34% du circuit, l'algorithme de Mak et Wong [Mak96] duplique seulement entre 5% et 15% du circuit.

Un autre algorithme poursuivant le même objectif : minimiser le délai d'une *netlist* implémentée sur des FPGAs, a été présenté par Sawkar et Thomas dans [Saw95].
Leur approche consiste à construire des cônes logiques à partir des sorties primaires et des cellules pilotant des *nets* de multiple sortance (angl.: *fanout*). vers les entrées primaires. Les cônes construits ainsi sont regroupés dans des partitions en deux étapes :

- La première phase traite les cônes logiques qui respectent des contraintes de cardinalité et d'entrées/sorties : Le coût de chaque cône est calculé en fonction de son gain pour la cardinalité et les entrées/sorties de la partition en formation. Le cône avec le plus meilleur coût est fusionné avec la partition actuelle et les cônes impactés par cette fusion sont mis à jour. La duplication de parties de la *netlist* appartenant à plusieurs cônes permet de réduire la coupure entre partitions et de garder l'ensemble d'un cône dans une partition.

- La deuxième phase du regroupement traite des cônes non-faisables. Ces cônes sont bipartitionnés récursivement en assurant qu'à chaque fois au moins une des deux partitions soit faisable. Ensuite, les partitions obtenues sont traitées comme les cônes faisables.

L'algorithme a une complexité de $O(|V|^2)$. Sa performance a été évaluée en utilisant les *benchmarks* de MCNC[18]. Les résultats en terme de nombre de FPGAs utilisés et de *nets* entre partitions ont été comparés avec un algorithme de bipartitionnement récursif basé sur le recuit simulé.
L'algorithme de Sawkar et Thomas [Saw95] utilise en moyenne 23% de FPGAs en moins et 16% de connecteurs en moins pour les *netlists* examinées. Le partitionnement de 4370 LUTs en 14 FPGAs met environ 20 minutes sur une station de travail DEC 3100.

2.3.5.7 Avantages et inconvénients

La duplication de cellules permet de réduire la coupure entre partitions ce qui facilite leur routage. En même temps, la probabilité de couper un chemin combinatoire baisse ce qui augmente la fréquence. Cet effet est accentué par l'utilisation d'un algorithme qui partitionne avec comme objectif de minimiser les délais.

La duplication de cellules a aussi des inconvénients :

- Elle est appliquée en post-traitement, ce qui augmente le temps de calcul et la mémoire utilisée.

- Le nombre de cellules dupliquées est difficilement contrôlable ce qui risque d'augmenter de manière significative la taille de la *netlist*.

[18] Le nombre de CLBs diverge encore par rapport à [Liu98a].

Un autre problème est directement lié à l'émulation : La création de *nets* portants le même nom nécessite la conservation de toutes les informations concernant la duplication afin de pouvoir identifier chaque signal lors du deboguage de la *netlist* avec l'émulateur.

2.3.5.8 Conclusion

Nous avons analysé dans ce paragraphe un certain nombre d'algorithmes de partitionnement poursuivant divers objectifs, mais utilisant toujours la même méthode pour y parvenir : la duplication de sommets.
L'avantage de cette méthode est qu'elle rend le routage plus facile par une réduction de la coupure entre partitions. Par contre, en raison des inconvénients présentés dans section 2.3.5.7, notamment celui de la difficulté de la mise en œuvre dans le cadre de l'émulation, elle n'a pas été retenue dans la suite de ce travail.

2.3.6 Algorithmes composites

L'émulation de très grands *designs* contenant plusieurs millions de portes logiques nécessite l'implémentation de la *netlist* correspondante sur des circuits programmables. Une des phases de l'implémentation est le partitionnement de la *netlist* en ensembles de cellules respectant la cardinalité et le nombre de connecteurs de ces circuits programmables.
La plupart des algorithmes présentés jusqu'ici ne sont pas en mesure de répondre à ces conditions. En effet, soit ils ont une trop grande complexité de calcul, ce qui mène à un temps d'exécution trop important, soit ils fournissent une solution médiocre en terme de nombre de partitions créées.
Récemment, plusieurs publications se sont consacrées avec succès à ce problème. Elles ont en commun le découpage du partitionnement en plusieurs phases et la combinaison de différents algorithmes appliqués pendant chacune des phases.

Riess et Schoene décrivent dans [Rie95] un algorithme composé de trois phases pour le partitionnement des MCMs *multichip modules*. Ces trois phases sont répétées jusqu'à ce que le nombre de partitions obtenues soit égal au nombre de *chips* de l'architecture cible.

- Calculer un placement bidimensionnel pour toutes les cellules. Ce placement initial est déterminé à l'aide d'une fonction quadratique introduite par Hall [Hal70] ($\min X^TQX$ où Q est la laplacienne du graphe représentant la *netlist*). La fonction a été modifiée pour prendre en compte les particularité des *multichip modules*, comme par exemple les *pads* sur le bord du substrat.

- Après le placement initial, chaque partition contenant un nombre de cellules trop important pour un *chip* est bipartitionnée en utilisant une variante du critère du *ratio-cut* qui tient compte des particularités du partitionnement des MCMs.

- On améliore itérativement le partitionnement provisoire déterminé pendant la phase précédente à l'aide d'une méthode utilisant l'heuristique du *ratio-cut* [Wei91] avec une fonction de coût appropriée.

La performance de l'algorithme est testée sur des *benchs* contenant entre 600 et 100000 cellules. Le partitionnement du plus grand circuit en 16 sous-ensembles dure 14 heures sur une station de travail DEC 3000 Modèle 600 AXP, ce qui n'est pas compatible avec les objectifs de notre travail.

Karypis, Aggarwal, Kumar et Shekhar [KaA97] ont présenté le partitionneur hMETIS qui est considéré par beaucoup de chercheurs comme le meilleur partionneur actuellement disponible. Plusieurs articles [KaK98] [KaK99][19] ont suivi la première publication introduisant des nouvelles fonctionnalités.
hMETIS consiste en trois phases :
Une première phase pendant laquelle des cellules de l'hypergraphe modélisant la *netlist* sont regroupées dans des macrocellules. Cette opération est répétée jusquà ce que l'hypergraphe soit suffisamment petit.
Dans la deuxième phase, plusieurs bissections sont effectuées en utilisant soit une coupure équilibrée tirée au hasard, soit un algorithme de croissance autour d'un germe. Ensuite, cette solution initiale est améliorée à l'aide d'un algorithme de bissection, comme par exemple celui de Kernighan-Lin [Ker70] ou de Fiduccia-Mattheyses [Fid82].
La troisième phase consiste à retrouver les cellules d'origine à partir des macrocellules tout en améliorant le partitionnement de l'hypergraphe simplifié. Pour ce faire, l'algorithme procède en étapes de reconstitution. A chaque étape, l'algorithme de bissection est appliqué à l'hypergraphe de cette étape afin d'améliorer le nombre de coupures.
Les résultats présentés montrent que l'algorithme hMETIS coupe en moyenne 4% à 23% moins d'hyperarêtes que d'autres algorithmes et qu'il produit ces résultats en 4 à 15 fois moins de temps CPU.
L'algorithme est en premier lieu destiné au placement de cellules, puisqu'il permet de fixer le nombre de partitions que l'on veut créer. Il ne permet pas de fixer des contraintes de cardinalité et de coupure maximales, ce qui est nécessaire pour le partitionnement en circuits programmables. Néanmoins, il est envisageable d'adapter cet algorithme afin qu'il respecte ces contraintes ou de l'utiliser dans une autre application comme par exemple le découpage de la *netlist* en sous-*netlists* qui sont traitées séparément. Par contre, il semble que la description de l'algorithme dans [KaA97], [KaK98] et [KaK99] ne suit pas la méthodologie décrit dans [Cal99a], puisqu'il n'est pas possible de reproduire le comportement de hMETIS par l'implémentation de l'algorithme décrit dans ces publication [Cal99c]. Pour ces raisons, il n'a pas retenu pour une évaluation ultérieure.

2.3.7 Partitionnement pour plusieurs niveaux hiérarchiques

Butts *et al.* [But91] semblent les premiers à avoir proposé une méthode pour le partitionnement sur plusieurs niveaux hiérarchiques: des circuits sont sur des cartes qui sont dans des boîtes qui sont placées dans des *racks*. Il y a donc des contraintes de cardinalité et de nombre d'interconnexions sur plusieurs niveaux. L'objectif consiste à partitionner une *netlist* de portes logiques ou CLBs[20] en un nombre minimal de partitions respectant les contraintes et permettant l'implémentation de la *netlist* à l'aide une telle hiérarchie. Pour cela, le même algorithme de partitionnement basé sur l'algorithme de Palesko et Akers [Pal83] est appliqué sur les différents niveaux: cet algorithme place d'abord toutes les cellules (portes logiques ou CLBs) dans une partition. Ensuite, un germe est sélectionné aléatoirement, puis d'autres cellules ayant le plus grand gain sont assemblées avec ce germe afin de former une nouvelle partition. Le gain est calculé en fonction de la modification de la coupure suite au

[19] Pour plus d'information, voir aussi *http://www.cs.umn.edu/ karypis.*
[20] Configurable Logic Block

placement d'une cellule dans la nouvelle partition. Pour cela, l'algorithme utilise la notion de *net* critique présentée dans [Fid82]. En cas d'égalité de gains entre deux cellules, celle avec le plus grand nombre de connecteurs est choisie et placée dans la partition. Après ce placement, le gain des cellules ayant des *nets* en commun avec la cellule placée est mis à jour.
Le préplacement individuel de certaines cellules dans les partitions, c'est-à-dire le choix judicieux d'une certaine cellule comme germe, permet d'améliorer d'autres critères tels que la fréquence du *design* après son implémentation.
Après chaque placement de cellule dans la partition actuelle, la cardinalité et la coupure de la partition sont mises à jour. Si la cardinalité et la coupure respectent les contraintes physiques, la partition est déclarée valide. Si seule la cardinalité respecte les contraintes, la partition est déclarée semi-valide. Si la cardinalité n'est plus respectée, la partition est déclarée non-valide. Le placement de cellules dans la partition actuelle est arrêté quand la partition devient non-valide. Dans ce cas, la dernière partition déclarée valide est restaurée. Le résultat du partitionnement est l'affectation de trois chiffres à chacune des cellules indiquant le *rack*, la carte et le circuit dans lequel la cellule est implémentée.

Butts *et al.* utilisent le même partitionnement sur les trois niveaux hiérarchiques, circuits, cartes et *racks*. C'est–à–dire, ils optimisent uniquement le partitionnement pour chacun des niveaux hiérarchiques indépendamment des autres niveaux. En particulier, ils n'étudient pas l'interdépendance entre les résultats de partitionnement sur les différents niveaux et ne fournissent pas de résultats expérimentaux. Contrairement à cette approche, nous définissons une méthodologie complète pour déterminer le meilleur partitionnement possible pour une architecture composée de plusieurs niveaux hiérarchiques. De plus, notre approche est validée expérimentalement sur une architecture cible concrète composé de 4 niveaux hiérarchiques.

Chou *et al.* [Cho94] proposent une méthode de partitionnement à plusieurs niveaux: d'abord une variante de l'algorithme *ratio-cut* [Wei91], le *local ratio-cut clustering scheme*, est appliqué à la *netlist* afin de réduire la complexité.
Ensuite, un algorithme qui s'inspire du paradigme de Espresso II [Bra85], le *set covering partitioning* est appliqué sur les sous-graphes créés. Etant donné un circuit C et la cardinalité approximative d'un *cluster* c. Le *local ratio-cut clustering* consiste à extraire un souscircuit avec $|C'| = c$ cellules, à déterminer toutes les cellules voisines aux cellules dans $C', N(C')$ et à appliquer l'algorithme du *ratio-cut* sur cet ensemble de cellules. On obtient comme résultat un *cluster* δ avec $|\delta| \leq c$. L'algorithme *Set Covering Partitioning* effectue un partitionnement en 4 étapes appelées *Initcover, Irredundant, Reduce* et *Expand.* Il se base sur des *clusters* déterminés lors de l'étape précédente.
Initcover crée des FPGAs potentiels à partir de ces *clusters* à l'aide d'une variante de l'algorithme de Fiduccia et Mattheyses [Fid82]. Pour éviter qu'un *cluster* fasse parti de trop de FPGAs potentiels en même temps, des pénalités exponentielles sont introduites pour ces *clusters.*
L'objectif de *Irredundant* est de conserver un ensemble minimal de FPGAs potentiels. Il faut donc déterminer les candidats qui sont essentiels pour que chaque *cluster* fasse parti d'au moins d'un FPGA potentiel. Si après cette étape, il reste encore des FPGAs potentiels, ceux-ci sont redondants et peuvent être enlevés.
Reduce génère un maximum de petits FPGAs potentiels en affectant les *clusters* dans les zones recouvrantes à un des FPGAs potentiels.

Dans *Expand*, un coût est associé avec chaque FPGA potentiel. Ce coût est le *scaled cost* (la coupure divisée par la cardinalité du FPGA potentiel [Zie96]). Ensuite, l'algorithme de Fiduccia et Mattheyses [Fid82] est appliqué à l'ensemble des FPGAs potentiels afin de regrouper ceux qui respectent après leur regroupement les contraintes de cardinalité et de coupure maximale d'un FPGA.
Les résultats expérimentaux obtenus sur trois *netlists* avec 38K, 49K et 160K portes logiques montrent une meilleure performance par rapport à l'application récursive de l'algorithme de Fiduccia et Mattheyses [Fid82]. Notamment pour la plus grande *netlist*, le temps de calcul pour le partitionnement en FPGAs de 2700 portes logiques avec 184 connecteurs d'entrée/sorties a été divisé par 5 et le taux de remplissage, c'est-à-dire le quotient entre le nombre de portes logiques de la *netlist* et le nombre de portes logiques disponibles, augmenté de 10%, passant de 22.5% à 31.7%.

L'algorithme présenté par Behrens, Harbich et Barke [Beh96a] est spécialement dédié à l'émulation : étant donné la cardinalité $s_{\max}$ et le nombre de connecteurs $i_{\max}$ d'un circuit programmable, l'objectif du partitionnement est de minimiser le nombre de circuits programmables nécessaires pour implémenter un certain *design*.
L'approche choisie par Behrens *et al.* consiste à utiliser la hiérarchie naturelle du *design*. Pour cela, ils introduisent la notion "de cellule hiérarchique" ou "d'élément hiérarchique" H. Il s'agit d'une cellule sur un certain niveau hiérarchique du *design*. Ces cellules hiérarchiques H peuvent être classées en cellules faisables H_{feasible}, cellules presque faisables $H_{\text{nearfeasible}}$ et cellules non-faisables $H_{\text{unfeasible}}$, avec

$$\begin{array}{rcl} s(H_{\text{feasible}}) \leq s_{\max} & , & i(H_{\text{feasible}}) \leq i_{\max} \\ s(H_{\text{nearfeasible}}) = s_{\max} + \epsilon & , & i(H_{\text{nearfeasible}}) \leq i_{\max} \\ s(H_{\text{nearfeasible}}) \leq s_{\max} & , & i(H_{\text{nearfeasible}}) = i_{\max} + \epsilon \\ s(H_{\text{unfeasible}}) > s_{\max} & , & i(H_{\text{unfeasible}}) > i_{\max} \end{array}$$

Elles peuvent également être qualifiées suivant le nombre de cellules hiérarchiques n_H qu'elles contiennent en cellules purements hiérarchiques H_{HHH}, cellules plutôt hiérarchiques H_{HHL}, cellules plutôt mises à plat H_{HLL} et cellules complètement mises à plat H_{LLL}.
Après avoir identifié ces cellules dans le *design*, plusieurs techniques sont appliquées :

- Un regroupement local de cellules faisables sur le même niveau hiérarchique tant que les contraintes de cardinalité et de nombre de connecteurs sont respectées. Un tel ensemble de cellules qui respecte ces contraintes est aussi appelé *cluster*.

- Un regroupement local étendu sur plusieurs niveaux hiérarchique si cette hiérarchie est profonde et très régulière. Le critère est le nombre d'instances d'une cellule sur les différents niveaux.

- Une migration de cellules vers des *clusters* déjà déterminés. Dans une première phase, uniquement des cellules sont distribuées aux *clusters*, qui n'augmentent pas la coupure du *cluster* respectif. Dans un deuxième temps toutes les cellules restantes sont distribuées tant que les contraintes imposées aux *clusters* sont vérifiées.

- L'extraction d'éléments hiérarchiques dans les cellules *nearfeasible* jusqu'à ce qu'elles deviennent *feasible*. Les éléments extraits sont eux aussi *feasible*, mais de mauvaise qualité. Afin d'améliorer leur qualité et la qualité d'autres éléments, un dernier pas est effectué, pendant lequel plusieurs éléments faisables peuvent être regroupés.

- Les cellules restantes sont regroupées dans une partition, puis l'algorithme de Sanchis [San89] est appliqué à cette partition.

L'algorithme a été testé à l'aide de circuits industriels ayant jusqu'à 143 000 portes logiques équivalentes. Les résultats expérimentaux montrent une amélioration jusquà 66% en terme de nombre de circuits programmables nécessaires à l'implémentation d'un *design* par rapport à l'algorithme de Fiduccia et Mattheyses [Fid82].
Néanmoins, il est difficile de tirer une conclusion :

1. L'algorithme est comparé uniquement avec l'algorithme de Fiduccia et Mattheyses. Celui-ci poursuit l'objectif de minimiser la coupure entre deux partitions, mais pas de minimiser le nombre de partitions lors d'un bipartitionnement récursif. Il n'est donc pas étonnant que l'algorithme de Behrens *et al.* nécessite moins de circuits programmables pour l'implémentation d'un *design*.
2. La comparaison de l'algorithme est effectuée sur un *benchmark* de *netlists* industrielles qui n'apparaissent dans aucune autre publication. Les résultats sont donc impossibles à comparer avec les résultats obtenus pour d'autres algorithmes.

2.4 Etudes comparatives de différents algorithmes de partitionnement

2.4.1 Objectif d'une telle étude

Notre objectif est l'amélioration de la performance des outils de partitionnement pour la compilation sur une architecture hiérarchique de grande capacité, plus particulièrement pour celle de l'émulateur CELARO. Pour cela, nous avons recherché dans la littérature les comparaisons de différents algorithmes de partitionnement, analysé différentes combinaisons de ces algorithmes, puis combiné ces algorithmes dans les outils afin d'optimiser le résultat du partitionnement.
Des comparaisons systématiques de différentes techniques de bipartitionnement ont été présentés dans [Hau97] et [Che97].

2.4.2 L'étude comparative du bipartitionnement par algorithmes de voisinage

Hauck et Borriello examinent dans [Hau97] différentes techniques du bipartitionnement[21] progressant d'une solution à une autre solution. Leur approche consiste à choisir un ensemble des meilleurs techniques de bipartitionnement selon leur expérience personnelle. Ces techniques sont intégrées au fur et à mesure dans un algorithme de base afin d'estimer le changement de la performance de l'algorithme à la suite de cette modification.
L'algorithme de base (appelé KLFM pour Kernighan–Lin Fiduccia–Mattheyses) est celui

[21] Hauck et Borriello examinent des techniques de bissection puisque les deux partitions doivent être équilibrées.

de Fiduccia et Mattheyses [Fid82]. Il peut être modifié et complété de plusieurs façons : l'application de l'algorithme avant ou après le *mapping* technologique, le regroupement de cellules *clustering* avant le bipartitionnement, la création d'un bipartitionnement initial de manière aléatoire ou suivant une approche plus sophistiquée, la modification de la fonction de coût, l'ajout de gains de niveaux supérieurs pour prévoir des mouvements futurs, la modification de l'équilibre entre les deux partitions ou encore des *runs* multiples pour élargir l'espace de recherche et augmenter ainsi la probabilité d'éviter des minima locaux.
L'évaluation expérimentale des algorithmes a été effectuée à l'aide des circuits du *benchmark* Partitioning93 [Koz91]. Leur taille varie entre 3225 et 25589 portes logiques.
Le *mapping* des portes logiques sur une certaine technologie (FPGAs) regroupent en général plusieurs portes logiques afin de former un CLB. Le bipartitionnement avant ce *mapping* fournit de meilleurs résultats en terme de coupure que le bipartitionnement après le *mapping*. Hauck et Borriello expliquent ce phénomène par le regroupement de cellules lors du *mapping* qui mène à un circuit avec une coupure minimale intrinsèque plus élevée.
Le *clustering* apporte deux avantages :

- Prise en compte des connexions locales avant le découpage global.

- Augmentation de la performance de KLFM grâce à une augmentation du nombre de *pins* par cellule. Il a été démontré par Goldberg et Burstein [Gol83] que KLFM fournit les meilleurs résultats, si les cellules du circuits ont en moyenne au moins six connecteurs. Dans les circuits réels par contre, chaque cellule a en moyenne environ 2.8 à 3.5 connecteurs.

Plusieurs méthodes de *clustering* ont été présentées. La méthode la plus performante poursuit trois objectif : l'absorption d'un grand nombre de *nets*, le regroupement de cellules de petite cardinalité et le regroupement de cellules ayant un grand nombre de *nets* en commun [Sch72].
Après avoir effectué le *clustering*, le circuit est partitionné. Le résultat de ce partitionnement est utilisé comme partitionnement initial pour le bipartitionnement sur le circuit à plat.
Il y a plusieurs approches pour revenir sur le *clustering* (*unclustering*). La meilleur performance est réalisée par le *unclustering* itératif. Il s'agit d'une méthode qui enlève progressivement des *clusters* hiérarchiques formés auparavant.
Le bipartitionnement initial peut également être construit selon de différentes méthodes, comme par exemple par une construction des deux partitions à partir de deux germes, par un bipartitionnement aléatoire, la recherche en largeur ou en profondeur d'abord à partir d'un germe, ou encore par les méthodes spectrales. Les résultats expérimentaux montrent que le bipartitionnement initial aléatoire fournit les meilleurs résultats, sans doute parce que les autres méthodes convergent plus rapidement vers un minimum local.
Le résultat en terme de coupure décroît avec l'augmentation du nombre d'essais, puisqu'on explore davantage l'espace de solutions ce qui augmente la probabilité d'éviter des minima locaux. Le gain décroît avec le nombre de *runs*.
La prise en compte de gains de niveaux supérieurs a été introduite par Krishnamurthy [Kri84]. Une évaluation de la performance de l'algorithme en dépendance du nombre de niveaux montre une décroissance de l'amélioration en terme de coupure avec le nombre de niveaux utilisés. Un bon compromis entre temps de calcul et coupure est atteint pour le niveau 3. Au delà, les gains ne sont pas assez importants pour justifier le temps de calcul

supplémentaire.
Le prochain aspect analysé est l'influence de l'organisation des listes de gains sur la coupure du bipartitionnement. Comme présenté par Hagen, Huang et Kahng [Hag95b], l'organisation des listes de gains en LIFO donne de meilleurs résultats. Hagen *et al.* expliquent ce phénomène par le déplacement consécutif de cellules interconnectées qui est le résultat d'une telle gestion des gains. En effet, les cellules en tête de liste sont toujours les cellules dernièrement mis à jour.
Le partitionnement dual concerne le déplacement de *nets* au lieu de sommets. A chaque déplacement, un *net* est enlevé de la coupure, ce qui peut introduire d'autres *nets* dans la coupure. Le partitionnement utilisant cette méthode, effectue d'abord une passe normale et ensuite une passe duale. L'algorithme s'arrête quand deux passes consécutives n'ont pas pu améliorer la coupure. Un tel algorithme a été présenté par Cong *et al.* dans [Con96].
Une comparaison des résultats de partitionnement avec et sans le partitionnement dual montre l'inefficacité de cette approche En effet, le nombre de *nets* coupés ne varie pas beaucoup entre les deux approches, tandis que le temps de calcul est multiplié par un facteur 20.
Au lieu de fixer la taille minimale d'une partition à 49% de la taille totale du circuit, il est possible de relâcher cette contrainte à par exemple 40%. Hauck et Borriello constatent qu'il est préférable pour un multipartitionnement en plusieurs circuits de garder des contraintes de 49%. En effet, dans le cas spécial décrit dans l'article où une *netlist* est partitionnée en 32 circuits sans limite de nombre de connecteurs, il est préférable d'effectuer une suite de bissections récursives afin d'obtenir une distribution équilibrée des cellules sur les différents circuits.

2.4.3 L'étude comparative des algorithmes qui minimisent la coupure d'un bipartitionnement

Chekuri *et. al* ont présenté dans [Che97] une comparaison expérimentale des performances en terme de temps de calcul de cinq algorithmes qui minimisent la coupure lors du bipartitionnement d'un graphe non-pondéré et non-orienté.
Les cinq algorithmes comparés sont des implémentations efficaces des algorithmes présentés dans [Hao92], [Kar93], [Kar96], [Nag92] et [Nag94].
Les algorithmes ont été améliorés en utilisant des structures de données efficaces ainsi qu'en implémentant une variante des heuristiques de Padberg et Rinaldi [Pad90] comme prétraitement. Ces heuristiques contractent certains arcs du graphe pendant le calcul du flot maximal dans l'algorithme de Gomory et Hu [Gom61].
Les algorithmes ont été testés à l'aide d'un ensemble de graphes générés avec NOIGEN et avec REGGEN.
NOIGEN est un générateur qui a été présenté dans [Nag94]. Il génère des graphes aléatoires contenant des composants fortement connexes.
REGGEN est un générateur qui produit des unions de cycles aléatoires. le nombre de cycles aléatoires. Ces cycles ne contiennent pas forcement des arcs disjoints. La variante de l'algorithme de Hao et Orlin [Hao92] représente la plus robuste implémentation.
La variante de l'algorithme de Nagamochi, Ono et Ibaraki [Nag94] est en moyenne la deuxième plus rapide implémentation réalisée. Sur les graphes générés avec NOIGEN, elle est même la plus performante.
L'implémentation de l'algorithme de Karger et Stein [Kar93] et celle de l'algorithme de

Karger [Kar96] sont moins performantes que les autres algorithmes implémentés. Elles sont par contre plus performantes que les implémentations d'origines.
Néanmoins, l'étude montre deux perspectives pour les prochaines recherches :

- Un algorithme est plus robuste que les autres. C'est-à-dire, il montre en moyenne une meilleure performance en temps de calcul sur tous les types de graphe sur lesquels il est appliqué. On pourrait donc conclure qu'un algorithme est bon sur tous les types de graphes, sans pour autant être toujours le meilleur.

- Sur un certain type de graphe, un autre algorithme fournit toujours de meilleurs résultats. On pourrait en conclure que pour certains types de graphes, il y a réellement un algorithme spécialisé, qui fournit toujours de meilleurs résultats que d'autres algorithmes.

Ces interprétations sont à relativiser, puisque les graphes sur lesquels les algorithmes sont appliqués sont des graphes totalement aléatoires. De plus, les auteurs ne mentionnent rien sur l'équilibre des deux partitions déterminées.

2.4.4 Conclusion

Les résultats présentés dans [Hau97] et [Che97] constituent de bonnes bases pour des comparaisons systématiques. Il y a par contre plusieurs points qui nous empêchent de tirer des conclusions directes de ces résultats pour le partitionnement d'une architecture hiérarchique :

- Les expériences ont été effectuées sur des circuits générés. Chekuri *et al.* utilisent des circuits générés aléatoirement. Nous allons voir dans chapitre 3 que de tels circuits ne permettent pas d'évaluer correctement les performances d'un algorithme de partitionnement. Dans le cas du *benchmark* Partitioning93 utilisé par Hauck et Borriello, il s'agit de circuits de très petite taille ($<$ 26000 portes logiques). Nous nous intéressons au partitionnement de très grandes *netlists* (jusqu'à 10M portes logiques). Des résultats, obtenus sur des *netlists* de petites tailles avec une structure qui ne correspond pas à celle de circuits réels, ne sont donc pas très significatifs.

- Les deux articles examinent des algorithmes de bipartitionnement qui poursuivent l'objectif de minimiser la coupure entre les deux partitions sans pour autant respecter des contraintes de cardinalité et de coupure maximale. Le partitionnement pour une architecture hiérarchique nécessite par contre un multipartitionnement respectant justement des contraintes de cardinalité et de coupure. En effet, à la fin du partitionnement, chaque partition doit correspondre à un élément hiérarchique, l'objectif étant de minimiser le nombre d'éléments nécessaire pour l'implémentation d'un *design* tout en assurant la routabilité des partitions obtenues et en respectant un certain temps de compilation. Ce ne sont pas les mêmes objectifs poursuivis par les algorithmes analysés dans [Che97] [Hau97]. Chekuri *et al.* et Hauck et Borriello examinent uniquement les performances des algorithmes pour un seul bipartitionnement en négligeant les conséquences du résultat de celui-ci pour la suite dans le cas d'un bipartitionnement récursif.

- Enfin, les deux articles effectuent des comparaisons d'algorithmes d'un seul type. Hauck et Borriello comparent des techniques pour les algorithmes de voisinage progressant d'une solution à une autre, tandis que Chekuri *et al.* analysent surtout des algorithmes basés sur le calcul du flot maximal dans un graphe. Notre objectif consiste à analyser les performances de différents algorithmes de partitionnement et de les combiner au mieux pour optimiser le résultat global du partitionnement sur une architecture hiérarchique.

Suite à l'analyse des deux études comparatives d'algorithmes de partitionnement présentées dans cette section nous avons constaté le besoin d'un banc de test réaliste. L'examen des *benchmarks* existants ainsi que des *netlists* créées avec des générateurs existants est présenté dans la prochaine section.

2.5 Bancs de test disponibles pour l'évaluation des algorithmes de partitionnement : état de l'art

L'évaluation des performances de différents outils de partitionnement se base généralement sur la comparaison des résultats expérimentaux obtenus avec des exemples de référence: un certain nombre de ces exemples de référence est sélectionné afin de former un banc de test (angl.: *benchmark*). L'évaluation des performances des outils de partitionnement est d'autant plus significative que les exemples de références du banc de test reflètent les caractéristiques des applications réelles.

Dans la littérature, il existe différents *benchmarks* et divers générateurs de *netlists* pouvant servir comme exemples de référence. Dans cette section nous décrivons ces *benchmarks* et ces *générateurs* et analysons, si les *netlists* correspondantes peuvent être utilisées pour évaluer les performances des algorithmes de partitionnement pour une architecture hiérarchique.

2.5.1 Benchmarks existants

2.5.1.1 ISCAS85, ISCAS89, Partitioning93

ISCAS85, *ISCAS89* et *Partitioning93* sont des *benchmarks* maintenus et distribués par le CBL[22] [Koz91]. *Partitioning93*, le plus récent des ces *benchmarks* est particulièrement intéressant pour la validation et le test des outils de partitionnement.
Il s'agit d'un ensemble de 37 *netlists* disponibles comme *netlists* de portes logiques (15 à 25589 portes) ou comme *netlists* de CLBs[23] (2 à 2901 CLBs).

L'avantage principal de ces *benchmarks* est qu'ils sont utilisés dans de nombreuses publications. Les raisons suivantes nous ont amenées par contre à constater que ces *benchmarks* ne possèdent pas toutes les caractéristiques dont nous avons besoin :

- Ils n'ont pas la cardinalité des *netlists* visées. En particulier, ils ne sont pas de taille comparable aux *netlists* implémentées habituellement sur l'émulateur CELARO. En effet, la plus grande *netlist* du *benchmark* *Partitioning93* contient 2904 CLBs après

[22] Collaborative Benchmarking Laboratory
[23] Un CLB (*configurable logic block*) est un groupement d'éléments logiques dans une architecture FPGA

l'implémentation dans des FPGAs XC3000 de Xilinx [Kuz93]. Une seule *netlist* des *benchmarks* possède plus de 26000 portes logiques [Rie94].

- Les résultats expérimentaux obtenus pour les différents outils de partitionnement sur ces *benchmarks* sont difficilement comparables. En effet, ayant été développés pour permettre la validation d'algorithmes de partitionnement pour le placement, il est souvent nécessaire de transformer leurs formats. Cette transformation peut entraîner une modification du contenu. Ainsi, il n'est pas rare de trouver un même *benchmark* possédant des propriétés différentes selon la publication [Alp95a].

2.5.1.2 ISPD98

Récemment, Alpert a décrit dans [Alp98] le *benchmark ISPD98*. Il contient 18 circuits de 13K à 210K cellules. Ces circuits sont présentés dans un format hypergraphe obtenu par une transformation de *designs* industriels d'IBM. Malheureusement, des informations importantes telles que la fonctionnalité des cellules, les informations de *timing* et de technologie ont été effacées par cette transformation qui apparemment a été nécessaire pour respecter la propriété industrielle.
A cause de cette transformation, les performances de certains algorithmes ne peuvent pas être étudiées :

- Les algorithmes basés sur la duplication fonctionnelle tels que celui présenté dans [Kuz94a] nécessitent de l'information sur la fonctionnalité des cellules.

- Les algorithmes ayant comme objectif de minimiser la longueur du chemin critique tels que celui présenté dans [Raj93] nécessitent des informations concernant les modules séquentiels et le délai des modules.

- Les algorithmes de partitionnement pour l'implémentation dans des FPGAs tels que [Fan97][Kru97], traitent des *netlists* de CLBs. Ces *netlists* sont caractérisées par l'uniformité de leurs cellules (CLBs), qui ont toutes le même nombre de connecteurs d'entrées/sorties. Les cellules des *netlists* du *benchmark ISPD98* n'ont pas cette caractéristique. Elles ne peuvent donc pas être utilisées dans leur format d'origine, mais doivent être transformées en *netlists* de CLBs. Puisque la fonctionnalité des différentes cellules n'est pas disponible, il est impossible de synthétiser une *netlist* de CLBs d'un circuit du *benchmark ISPD98*.

- Certains algorithmes de partitionnement exploitent l'orientation des *nets*, tel que les algorithmes construisant des structures de cônes logiques comme l'exemple présenté dans [Bra94]. Un certain nombre des connecteurs externes des circuits du *benchmark ISPD98* sont par contre bidirectionnels (jusqu'à 90% [Alp98]). Afin de pouvoir appliquer un tel algorithme, il est nécessaire de transformer ces connecteurs bidirectionnels en connecteurs unidirectionnels. La manière par laquelle cette transformation peut être effectuée sans pour autant modifier la structure de la *netlist* est un domaine ouvert à la recherche [Alp98].

De plus, un banc de test contenant des circuits de tailles fixes avec jusqu'à 210K cellules peut rapidement devenir obsolète dans le domaine de la microélectronique où nous observons une augmentation croissante des tailles de *netlists* : aujourd'hui déjà , nous traitons

des *netlists* contenant plusieurs millions de portes logiques.
Seuls les générateurs de *netlists* aléatoires offrent la possibilité de générer des *netlists* paramètrables en taille et en topologie.
De plus, ils évitent les problèmes de la propriété industrielle et permettent de disposer d'un grand nombre d'exemples afin de pouvoir comparer toutes les aspect des différentes méthodes de partitionnement et de couvrir un très large spectre d'applications réelles. Pour ces raisons, nous analysons maintenant des générateurs existants. Cette analyse est présentée dans la section suivante.

2.5.2 Générateurs existants

2.5.2.1 Générateur de Darnauer et Dai

Darnauer et Dai ont présenté dans [Dar96] une méthode pour générer des *netlists* aléatoires paramètrables respectant approximativement le paramètre de Rent [Lan71]. Il est possible de spécifier le nombre d'entrées/sorties, de cellules et de connecteurs par cellule. Le générateur tient compte des contraintes électriques d'un circuit réel. Par contre, il ne respecte pas sa structure hiérarchique. Nous montrerons dans le chapitre 3 que le respect de la structure hiérarchique lors de la génération est un critère essentiel pour la création de circuits semblables à des circuits réels.

2.5.2.2 Générateur de Ghosh et al.

Ghosh *et al.* [Gho98] ont développé un générateur basé sur l'analyse de circuits de référence: le générateur extrait la signature des interconnections d'un circuit pour définir une classe de circuits ayant les interconnexions semblables. Une perturbation induite dans le circuit de référence mène à un circuit de référence perturbé. La resynthèse de ce circuit perturbé conduit enfin à un certain nombre de mutants qui peuvent être utilisés comme banc de test. L'avantage de cette méthode est que les circuits ne sont pas complètement aléatoires.

2.5.2.3 Générateur de Hutton et al.

Introduction Hutton *et al.* [Hut96][Hut97] ont développé deux outils CIRC et GEN:
CIRC est un outil permettant d'analyser un circuit afin d'en extraire des paramètres statistiques.
GEN génère des *netlists* aléatoires à base de *LUTs*[24] en utilisant des paramètres caractéristiques spécifiés dans un fichier.
Il est possible d'analyser un circuit industriel à l'aide de CIRC et utiliser le résultat comme spécification pour le générateur GEN. On obtient ainsi un clône de la *netlist* d'origine, c'est-à-dire une *netlist* générée aléatoirement présentant les mêmes caractéristiques que le circuit modèle.
La comparaison de la *netlist* d'origine avec son clône a été adoptée par les auteurs pour valider leurs outils[Hut96, Hut97]. Il a également été adopté pour examiner l'aptitude des *netlists* aléatoires générées par GEN à qualifier les performances des outils de partitionnement. En effet, le clône doit être partitionné avec un nombre comparable de sous-ensembles part rapport à la *netlist* originale.

[24] Look Up Tables

Caractérisation de circuits avec CIRC La terminologie utilisée vient de la modélisation d'une *netlist* par un hypergraphe, où les cellules sont présentées par des sommets et les *nets* par des hyperarêtes.

Les caractéristiques extraites par CIRC lors de l'analyse d'une *netlist* et utilisées par GEN pour générer un clône sont :

- le nombre de *LUTs* de la *netlist*,
- le nombre d'entrées et de sorties primaires,
- la longueur de la chaîne longue de la *netlist*,
- les paramètres de forme du circuit, c'est-à-dire le vecteur contenant le nombre de cellules dans chaque couche logique, entrées et sorties primaires incluses,
- la distribution de la longueur des hyperarêtes, c'est-à-dire le nombre d'hyperarêtes avec une longueur 0, avec une longueur 1, etc. La longueur d'une hyperarête est le délai maximal entre deux cellules connectées par cette hyperarête,
- le *fanout* d'une cellule et la distribution de ce *fanout* sur l'ensemble des cellules.

Exemple 2.9 Illustration des caractéristiques retenues

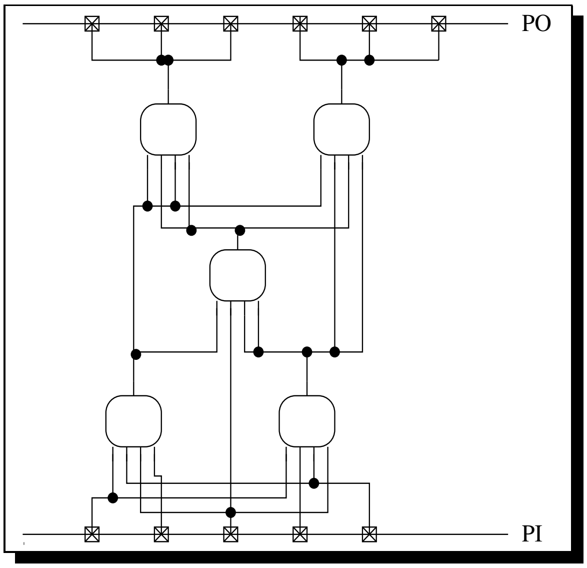


Figure 2.13: *Exemple d'une netlist*

La netlist présentée dans la figure 2.13 contient 5 LUTs, 5 entrées primaires et 6 sorties primaires. Sa chaîne critique a une longueur de 3. Sa forme est de [5, 2, 1, 2, 6], ce qui signifie qu'on dénombre 5 entrées primaires, 2 cellules avec au moins une connexion avec une entrée primaire, etc. La distribution des longueurs des hyperarêtes est [0, 7, 3] : 7 hyperarêtes ont un délai maximal de 1 et 3 hyperarêtes un délai maximal de 2. La distribution du fanout est [2, 2, 4, 2] : 2 nets ont un fanout de 1, 2 un fanout de 2, 4 un fanout de 3 et 2 un fanout de 4.

Génération de circuits avec GEN Lors de la génération d'une *netlist* aléatoire, l'outil GEN respecte les spécifications suivantes:

- le nombre de cellules,
- le nombre d'I/O,
- le délai,
- le nombre k d'entrées des *LUTs*,

- la fonction de forme,
- la distribution des longueurs des arêtes et des *fanouts*.

Validation de CIRC et GEN Hutton *et al.* [Hut96][Hut97] ont validé leurs travaux en comparant trois *netlists* : Une *netlist* du *benchmark* MCNC, son clône créé avec CIRC et GEN et une *netlist* de même taille générée aléatoirement. Des caractéristiques d'implémentation sur des FPGAs, notamment la longueur moyenne des fils routés sur des FPGAs et le nombre de pistes utilisées, ont servi comme critère de comparaison [Hut97]. Les résultats de l'implémentation du clône produit par CIRC et GEN sont proches de ceux de la *netlist* réelle, tandis que ceux de la *netlist* générée aléatoirement divergent beaucoup.

2.5.3 Conclusion

Nous avons discuté l'utilisation des *benchmarks* existants pour l'évaluation des performances de différents algorithmes de partitionnement. Pour plusieurs raisons, ces *benchmarks* n'ont pas les caractéristiques nécessaires pour une l'étude que nous nous proposons, ce qui nous a amené à examiner l'utilisation des générateurs de *netlists* aléatoires. Par contre, le générateur GEN présenté dans [Hut96] [Hut97] nous semble particulièrement intéressant. Une analyse plus approfondie de ce générateur est présenté dans chapitre 3 (section 3.1).

2.6 Conclusion

Nous avons étudié et analysé de nombreuses publications récentes dans le domaine du partitionnement. Aucun algorithme ne peut être directement appliqué pour partitionner une *netlist* destinée à une implémentation sur une architecture hiérarchique. Un algorithme seul ne semble donc pas pouvoir satisfaire les besoins du partitionnement pour une telle architecture.
Néanmoins, les conclusions de cette étude des algorithmes existants nous ont permis de choisir un ensemble d'algorithmes afin de former une bibliothèque (boîte à outils) d'algorithmes. Nous analysons les avantages et inconvénients de chacun des algorithmes pour les améliorer et les utiliser au mieux et dans un flot de partitionnement pour une architecture hiérarchique.

L'analyse des avantages et inconvénients de chacun des algorithmes est basée sur des résultats expérimentaux obtenus à la suite de l'application des algorithmes à des exemples de référence. Lors de la constitution d'un banc de test avec de tels exemples, nous avons constaté une carence dans ce domaine. En effet, les *benchmarks* existants ainsi que beaucoup de *netlists* créées avec des générateurs existants ne possèdent pas les caractéristiques nécessaires pour une telle étude. Seul le générateur GEN semblait satisfaire à nos besoins. L'étude concernant son utilisation présenté dans section 3.1 révélera par contre qu'il ne génère pas des *netlists* ayant les caractéristiques nécessaires.
Ceci nous a amené à développer notre propre générateur de *netlists* aléatoires qui est présenté dans le prochain chapitre.

Chapitre 3

PartGen : Un générateur de grandes netlists hétérogènes pour l'évaluation des algorithmes

L'évaluation des performances de différents algorithmes de partitionnement de *netlists* en vue de leur implémentation sur une architecture hiérarchique comme celle de l'émulateur CELARO est un des objectifs de ce travail. Cette évaluation des performances se base sur la comparaison des résultats expérimentaux obtenus avec des exemples de référence (angl.: *benchmark*).
Elle est d'autant plus significative que les *netlists* utilisées pour ce travail apparaissent dans des applications réelles.
Les *netlists* créées avec le générateur GEN présenté dans section 2.5.2.3 nous semblait posséder les caractéristiques nécessaires. Nous avons donc étudié leurs caractéristiques plus en détail.

3.1 Utilisation de GEN pour évaluer les performances de partitionnement

3.1.1 Etude expérimentale

Dans un premier temps, des translateurs bidirectionnels entre le langage *BLIF*[1] [Sen92] utilisé par CIRC et le langage *ANF* (cf. [Men99a] [Men99b]) ont été implémentés.
Afin de valider les translateurs, nous avons traduit la *netlist* originale d'un additionneur dans les deux directions. Le résultat des ces translations a été partitionné en *clusters* et comparé avec le partitionnement de l'original. Le nombre de *clusters* obtenus est identique dans les deux cas.

Dans un deuxième temps, nous avons examiné la possibilité d'utiliser les *netlists* générées par ces outils pour qualifier les performances d'un outil de partitionnement.
Les étapes du processus de validation sont illustrées dans la figure 3.1.

Une validation partielle de CIRC et GEN a été réalisée en prenant comme exemple un

[1] BLIF (Berkeley Logic Interchange Format) est un langage qui a été développé à l'université de Berkeley.

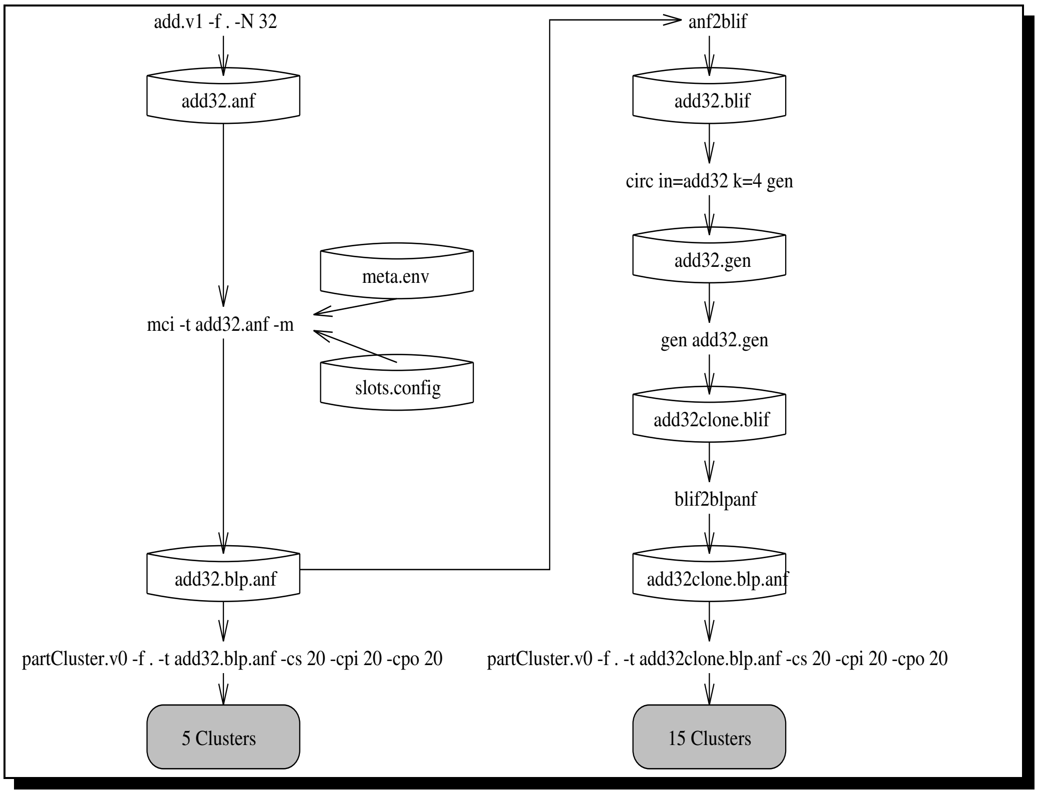


Figure 3.1: *Processus de validation de CIRC et GEN*

additionneur 32 bits obtenu à partir d'un générateur d'additionneurs paramétrable, interne à Mentor Graphics. Cet additionneur a été compilé afin d'obtenir une *netlist* de CPBs. Un clone de ce circuit a été généré en utilisant d'une part les translateurs développés et d'autre part les outils CIRC et GEN. Les deux *netlists*, l'original add32 et son clone add32clone, sont partitionnées en *clusters* logiques avec le même jeu de paramètres : La taille du *cluster* (cs) est limitée à 20 cellules, la coupure en entrée (cpi) et en sortie (cpo) à 20 connexions. Le même test a été fait avec un multiplieur 32 bits en utilisant un autre jeu de paramètres. Les différents résultats expérimentaux sont présentés dans le tableau 3.1.

Circuit	Nombre de cellules	cs	cpi	cpo	Nombre de *clusters* obtenus
add32	66	20	20	20	5
add32clone	62	20	20	20	15
mul32	2301	60	60	60	37
mul32clone	2168	60	60	60	75

Tableau 3.1: *Comparaison des résultats de partitionnement en clusters*

3.1.2 Analyse des résultats expérimentaux

Ils montrent que les *netlists* générées par CIRC et GEN ne peuvent pas être utilisées pour qualifier la performance des outils de partitionnement. En effet, les résultats du partitionnement en *clusters* varient de 103% en plus pour le multiplieur à 200% en plus pour l'additionneur entre la *netlist* originale et son clone.

Les paramètres pris en compte par le générateur GEN ne sont pas suffisants pour générer une *netlist* qui pourrait être utilisée pour qualifier la performance d'un outil de partitionnement. Pour illustrer ce fait, nous avons représenté graphiquement les deux *netlists* en utilisant l'outil AutoLogic II de Mentor Graphics : La *netlist* originale de l'additionneur 32 bits est très régulière (voir figure 3.2).

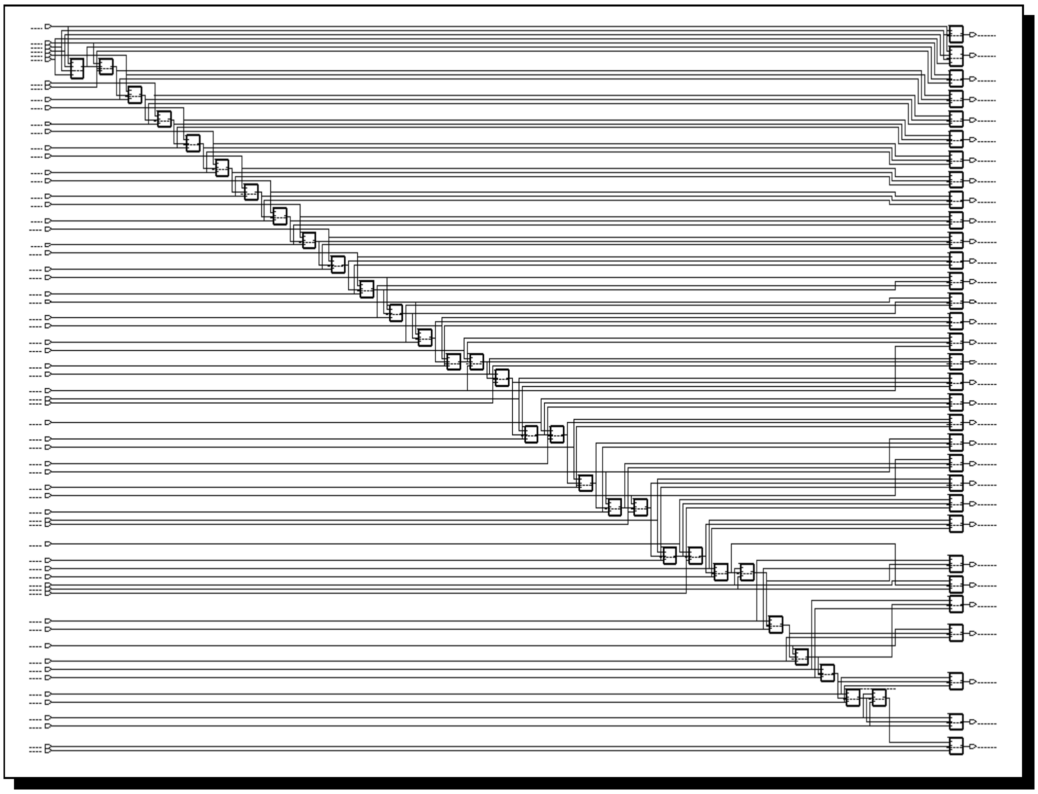


Figure 3.2: *Netlist d'un additionneur 32 bits*

L'outil de partitionnement coupe horizontalement en effectuant de très petites coupures. Les *clusters* peuvent être entièrement remplis tout en respectant leur coupure maximale. La *netlist* se partitionne donc en un petit nombre de *clusters*.
Le clone de la *netlist* originale est très irrégulier. Il est difficile de trouver des coupures franches. Par conséquent, tous les *clusters* sont saturés par leur coupure avant d'être entièrement remplis. La *netlist* se partitionne donc en un plus grand nombre de *clusters* comme nous le montre la figure 3.3.

3.1.3 Conclusion

L'analyse des *benchmarks* et des générateurs existants effectuée dans section 2.5 montre que les *netlists* des différents bancs de test et celles obtenues avec des générateurs existants n'ont pas les caractéristiques nécessaires pour évaluer les performances des algorithmes utilisés pour effectuer le partitionnement sur une architecture hiérarchique.
Afin de disposer d'un banc de test réaliste contenant de très grandes *netlists* semblables à

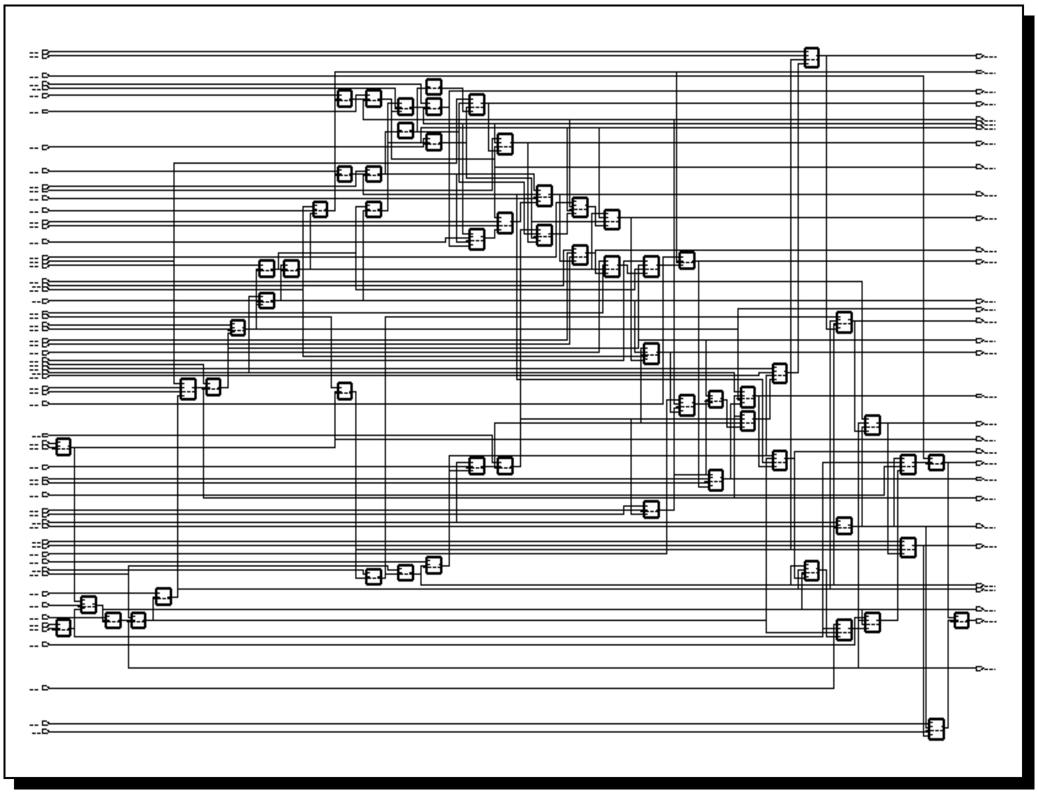


Figure 3.3: *Netlist du clone d'un additionneur 32 bits*

celles implémentées habituellement sur le type d'architecture hiérarchique visée nous avons développé un générateur de *netlists* aléatoires.

3.2 Besoins et spécifications

Nous avons vu que la comparaison des résultats expérimentaux obtenus avec des exemples de référence est d'autant plus significative que les *netlists* utilisées pour ce travail apparaissent dans des applications réelles. Les *netlists* de référence doivent donc être semblables à des très grandes *netlists* industrielles. En particulier, ces *netlists* doivent :

- être synthétisables en *netlists* de CPBs,
- être disponibles en grande quantité afin de permettre une évaluation représentative,
- avoir une cardinalité comparable aux *netlists* industrielles (entre 20K et 350K CPBs pour des *netlists* actuelles),
- avoir une structure hiérarchique semblable aux *netlists* industrielles pour permettre l'obtention de résultats comparables.

L'analyse des *benchmarks* et des générateurs existants effectuée dans section 2.5 montre que les *netlists* des différents bancs de test et celles obtenues avec des générateurs existants n'ont pas ces caractéristiques.

Il a donc été nécessaire de développer un générateur de *netlists* aléatoires mieux adapté à ce genre de problèmes avec l'objectif de pouvoir spécifier la composition de la *netlist* ainsi que sa taille de en terme de nombre de CPBs ou en terme de CPBs équivalents.

Ce générateur appelé PartGen est présenté en détail dans ce chapitre. La section suivante contient les résultats d'une analyse de *netlists* industrielles. Les résultats de cette analyse ont été pris en compte lors du développement du générateur. En particulier, nous avons identifié six types différents de *netlists*. Le générateur PartGen se compose donc de six sous-générateurs, un pour chaque type de *netlist* identifié. Ces sous-générateurs sont présentés dans les sections 3.4 à 3.9. L'intégration des différents sous-générateurs dans PartGen est présenté dans section 3.10. Les sections 3.11.2 à 3.11.4 contiennent la validation des différents sous-générateurs suivi d'une conclusion dans section 3.12.

3.3 Composition d'une netlist réelle

L'analyse fonctionnelle et statistique d'un nombre représentatif de *netlists* industrielles chez Mentor Graphics a fait apparaître six catégories :

1. Combinatoire irrégulière : Les *netlists* de ce type sont caractérisées par une très grande irrégularité. Elles sont souvent utilisées pour accomplir les fonctions de pilotage autour de grands blocs fonctionnels.

2. Opérateur : Il s'agit d'unités arithmétiques telles que les multiplieurs ou les additionneurs. Ces *netlists* sont très régulières, purement combinatoires et en général faciles à partitionner.

3. Mémoire isolée : Ce sont de grandes mémoires modélisées par des bancs de mémoires sur des cartes SRAM.

4. Mémoire intégrée : Petites mémoires modélisées par des *meta_xmem* qui devront être intégrées dans les AVCs .

5. Contrôleur : Les *netlists* de ce type contiennent des éléments combinatoires et des éléments séquentiels. Ils sont caractérisées par un rapport élevé du *fan* (nombre de connecteurs sur les *nets*) sur le nombre de cellules. Elles sont souvent irrégulières et, de ce fait, difficiles à partitionner.

6. Interconnexion : Les *netlists* de ce type sont utilisées pour connecter différents modules entre eux. Elles ne contiennent pratiquement pas de cellules.

Chaque circuit est composé d'un ou plusieurs de ces types de *netlist*.
Le but est de développer un générateur capable de créer des *netlists* comportant, ou non, plusieurs sous-*netlists* de différents types. Pour cela, il a été nécessaire de construire des générateurs dédiés à chaque type et un super-générateur intégrant ces différentes sous-*netlists* dans une seule *netlist*.

Les différents générateurs ainsi que leur intégration dans un super-générateur unique et paramétrable sont présentés dans les sections suivantes.

3.4 Générateur de combinatoire irrégulière

Pour obtenir des *netlists* contenant de la combinatoire irrégulière, nous nous appuyons sur le générateur GEN [Hut96] qui a été présenté dans section 2.5.2.3. Nous utilisons GEN dans son plus simple mode qui nécessite uniquement la spécification du nombre de connecteurs d'entrée des *look up-tables* (LUTs) et le nombre de LUTs à générer. Les *netlists* de combinatoire irrégulière générées par PartGen contiennent uniquement des LUTs à quatre entrées et une sortie. Le nombre de modules des *netlists* ainsi que leur cardinalité est paramétrable.

3.5 Générateur d'opérateurs

Pour obtenir des *netlists* d'opérateurs, nous utilisons les générateurs de multiplieurs et de compteurs de Mentor Graphics. Ces générateurs se paramètrent selon la taille des opérandes.
Notre objectif consiste à réaliser un générateur de *netlists* de taille donnée. Cela implique l'étude de la relation entre la taille de l'opérande et la taille de la *netlist* générée, définie par le nombre de CPBs.
Pour le multiplieur, nous avons déterminé la relation suivante :

$$\text{taille\_multiplieur} = 2.08 * \text{taille\_opérande}^2 \tag{3.1}$$

Pour le compteur, la dépendance entre la cardinalité et la taille de l'opérande est :

$$\text{taille\_compteur} = 4 * \text{taille\_opérande} \tag{3.2}$$

PartGen génère soit un multiplieur, si le nombre de *netlists* d'opérateur à générer est égal à un, soit un compteur et le reste des *netlists* en multiplieurs, si ce nombre est supérieur à un. Tous les multiplieurs sont de taille égale. Le compteur sert à ajuster la taille de la *netlist* d'opérateurs. En effet, la dépendance quadratique entre le nombre d'éléments générés et la taille de l'opérande du multiplieur fait que la taille spécifiée de la *netlist* d'opérateurs est difficile à respecter uniquement en utilisant des multiplieurs : Un multiplieur 62x62 bits contient 7967 cellules, tandis qu'un multiplieur 64x64 bits en contient 8475.
Le nombre de modules générés ainsi que leurs cardinalités sont paramétrables.

3.6 Générateur de mémoire isolée

Les modèles de mémoires isolées sont créés en utilisant le générateur sram.v0 de Mentor Graphics dans un environnement dont le seul paramètre est le nombre de K mots. Puisque l'objectif fixé consiste à développer un générateur paramétrable en nombre de CPBs ou en nombre de CPBs équivalents, il faut établir une relation entre le nombre de cellules et la taille de la mémoire : Un K mot de 32 bits de mémoire est équivalent à 4.125 CPBs.

PartGen permet de générer un certain nombre de *netlists* de mémoire isolée de taille égale. Le nombre de modules générés ainsi que leurs cardinalités sont paramétrables.

3.7 Générateur de mémoire intégrée

Les mémoires intégrées sont des petites mémoires réalisées par des *meta_xmem*. Pour créer ce type de mémoire, nous utilisons le générateur xmem.v0 de Mentor Graphics intégré dans un environnement paramétrable. Celui-ci est défini par le nombre de mots. Il faut donc également établir une relation entre le nombre de cellules et la taille de la mémoire :
Un mot de 16 bits de mémoire *meta_xmem* est équivalent à 3 CPBs.

PartGen permet de générer un certain nombre de *netlists* de mémoire intégrée de taille égale.
Le nombre de modules générés ainsi que leurs cardinalités sont paramétrables.

3.8 Générateur de contrôleur

3.8.1 Introduction

Le contrôleur est le cinquième type de *netlist* identifié.
Pour développer un générateur de contrôleurs, nous utilisons les données obtenues lors de l'analyse d'une *netlist* industrielle[2]. L'objectif est de pouvoir générer des *netlists* aléatoires qui ont les mêmes caractéristiques que le modèle de référence et qui se partitionnent avec un résultat comparable à celui-ci.

3.8.2 Caractéristiques et paramètres retenus

Les caractéristiques suivantes ont été extraites lors de l'analyse de la *netlist* réelle :

- La *netlist* contient 48836 cellules dont 69% de CPBs combinatoires et 31% de CPBs configurés en *flip-flop* (FF).
- 66.6% des CPBs combinatoires ont quatre entrées, 22.8% ont trois entrées, 9.9% ont deux entrées et 0.7% ont une entrée.
- La totalité des CPBs configurés en FF ont une entrée.
- La *netlist* contient 49819 *nets* dont 1307 (soit environ 3%) avec une connexion extérieure.
- Le nombre d'entrées primaires (PIs) est du même ordre de grandeur que le nombre de sorties primaires (POs).
- Le *fan* moyen des *nets* est de 4.22. La distribution sur les *nets* est présentée dans le tableau 3.2.

  De plus, il y a deux *nets* d'horloge avec plus de 6500 connexions (soit environ avec 1/7 des cellules), qui ne figurent pas dans le tableau.

[2] L'analyse d'autres *netlists* de contrôleur industrielles montre que la *netlist* sélectionnée est représentative de ce type de *netlists*

fan	nombre de *nets*	pourcentage du nombre total de *nets*
2	34767	69.7%
3	5268	10.6%
4	4099	8.2%
5	1560	3.1%
6	795	1.6%
7	524	1.1%
8	347	0.7%
9	332	0.7%
10	195	0.4%
]10;20]	903	1.8%
]20;100]	977	2.0%
]100;1000]	45	0.1%
]1000;2000]	5	0.01%

Tableau 3.2: *Distribution du fan des nets*

- La chaîne critique compte 38 cellules. Sa longueur est indépendante de la taille du contrôleur. Elle contraint la fréquence maximale du *design* une fois implémenté sur l'architecture hiérarchique.

Afin de simplifier la génération de la *netlist*, de respecter des contraintes imposées par la chaîne de compilation et de conserver le caractère aléatoire de la construction, le générateur respecte les règles suivantes :

- Le nombre de FFs est de 30 à 35% du nombre d'éléments.
- Le nombre de LUTs est égal au nombre de CPBs de la *netlist* moins le nombre de FFs.
- 1.5% du nombre d'éléments sont des entrées primaires, 1.5% sont des sorties primaires.
- Le nombre de *nets* est égal au nombre d'équipotentielles, c'est-à-dire à la somme des nombres de LUTs, de FFs et d'entrées primaires.
- Tous les LUTs utilisent l'entrée $I1$, 99.3% l'entrée $I2$, 89.4% l'entrée $I3$ et 66.6% l'entrée $I4$.
- Les deux *nets* d'horloge sont remplacés par un seul *net* qui est connecté à toutes les connexions d'horloge des FFs pour des raisons de simplification.
- La chaîne critique a été fixée arbitrairement à 50 cellules combinatoires parce que sa longueur est indépendante de la taille du contrôleur et qu'elle contraint uniquement la fréquence de la *netlist* implémentée sur l'architecture hiérarchique visée.

Les résultats expérimentaux obtenus lors de l'analyse du générateur GEN (cf. section 2.5.2.3) montrent que les données statistiques sont insuffisantes pour construire une *netlist* qui se partitionne d'une façon analogue à une *netlist* réelle. En effet, la structure hiérarchique du modèle et une certaine localité des interconnexions doivent également être prises en compte.

La hiérarchie de la *netlist* réelle a été étudiée à l'aide de l'algorithme de bipartitionnement décrit dans section 4.1.1. Cet algorithme coupe une *netlist* en deux sous-ensembles en minimisant le critère du *ratio-cut* [Wei91]. L'application récursive de cet algorithme à la *netlist* réelle fournit une structure arborescente.
Cette structure est reproduite lors de la génération par une construction récursive commençant avec les macro-contrôleurs, correspondants aux feuilles de l'arbre et remontant les branches jusqu'à arriver à la racine. La décomposition hiérarchique d'une *netlist* de contrôleur est illustrée par la figure 3.4.

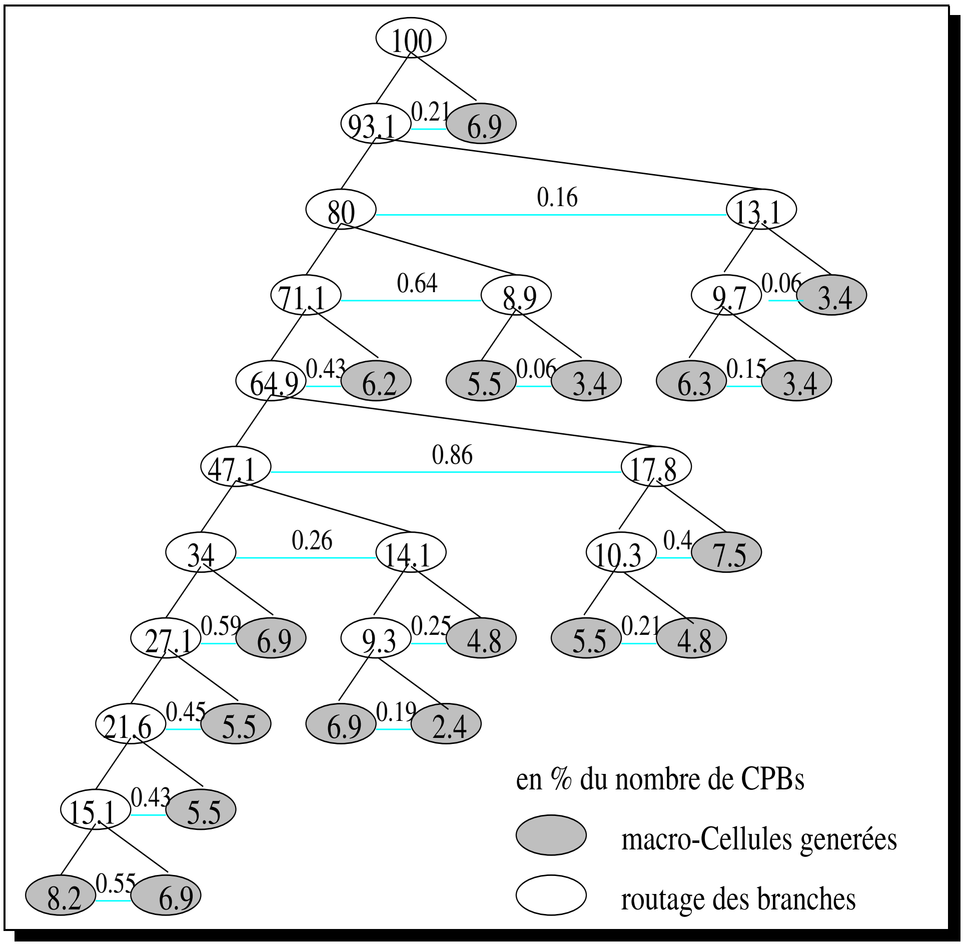


Figure 3.4: *Représentation de la construction hiérarchique*

3.8.3 Contraintes de construction d'un macro-contrôleur

Les *nets* avec un *fan* supérieur à quatre ont une grande probabilité d'être coupés par le partitionnement. Par contre, environ 90% des *nets* des *netlists* analysées ont un *fan* inférieur ou égal à quatre.
Pour conserver la localité des interconnexions à l'intérieur des macro-contrôleurs, nous procédons comme suit : nous commençons avec un ordonnancement aléatoire des cellules.

La mesure de proximité pour deux cellules v_i et v_j est la distance de ces cellules dans l'ordonnancement, c'est–à–dire $|i-j|$. Le voisin v_j d'une cellule v_i est déterminé en fonction du *fan* du *net* avec lequel ils sont interconnectés. Dans le cas d'un *net* de *fan* inférieur ou égal à quatre, nous sélectionnons le voisin v_j de v_i avec une probabilité décroissante selon leur distance $|i-j|$. Dans PartGen, nous utilisons une gaussienne, comme illustré dans figure 3.5.

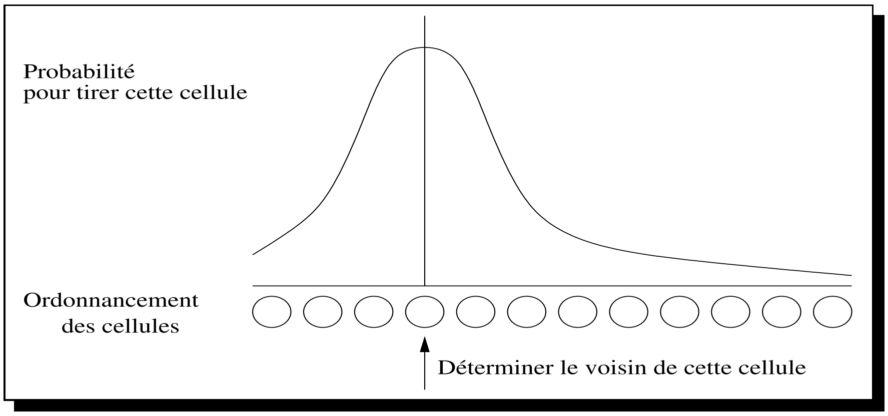


Figure 3.5: *Détermination du voisin dans le cas d'un net de faible fan*

Dans le cas d'un *net* de *fan* supérieur à quatre, nous sélectionnons le voisin par un tirage aléatoire équiprobable comme présenté dans figure 3.6.

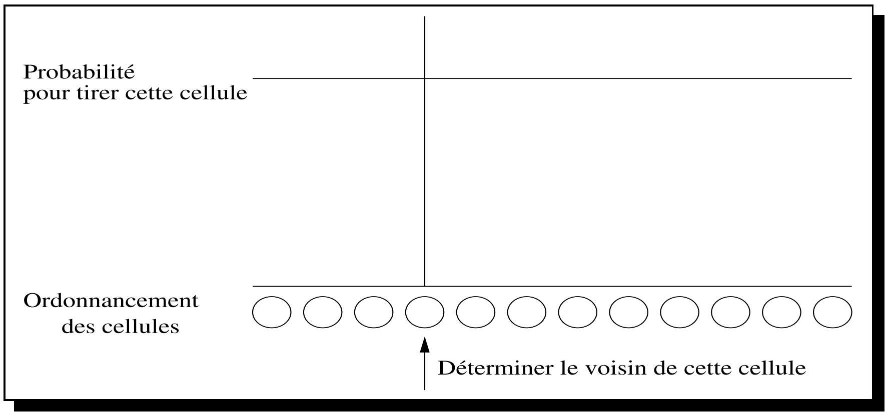


Figure 3.6: *Détermination du voisin dans le cas d'un net de fort fan*

Afin de respecter les règles élémentaires de l'électronique et d'obtenir une *netlist* proche de la réalité, certaines contraintes complémentaires ont été imposées pour la construction des macro-contrôleurs:

- Les connexions directes entre entrées et sorties primaires (PI-PO) et entre *flip-flops* (FF-FF) sont rarement utilisées dans des *netlists* réelles. Elles ne sont donc pas autorisées.

- Le nombre de connecteurs nécessaires est la somme des PIs, des POs, des entrées et sorties des FFs, et des entrées et sorties des CPBs. Le nombre de connecteurs disponibles est la somme des connecteurs de tous les *nets*. Celui-ci doit être supérieur ou égal au nombre de connecteurs nécessaires.

- Chaque *net* doit avoir un seul *driver*, afin d'éviter des court-circuits.
- Chaque entrée des LUTs, FFs, PIs et POs utilisée doit être branchée sur exactement un *net*. Ceci évite d'avoir des connecteurs en l'air et des court-circuits.
- Chaque *net* a au plus un connecteur appartenant à la même cellule. En particulier, les boucles sur la même cellule sont interdites.
- Il faut respecter la longueur de la chaîne critique.
- Les connexions doivent être de nature aléatoire.
- Les *nets* de *fan* inférieur ou égal à quatre sont connectés localement.
- Les *nets* de *fan* supérieur à quatre effectuent des connexions globales.
- La construction doit se faire le plus rapidement possible.

3.8.4 Construction de la netlist aléatoire d'un macro-contrôleur

Dans un premier temps, tous les éléments de la *netlist* aléatoire (LUTs, FFs, PIs et POs) sont générés et chaînés dans des listes, une pour chacun des types. Un tableau de pointeurs sur les différents éléments de ces listes facilite la sélection aléatoire.

Dans un deuxième temps, les *nets* sont générés et chaînés dans une liste. Un certain *fan* est attribué aléatoirement à chacun d'entre eux. La distribution des *fans* sur les *nets* correspond à celle de la *netlist* réelle.

A la fin de la deuxième phase, tous les éléments de la *netlist* ont été générés. Ils sont sauvegardés dans quatre listes chaînées et accessibles par quatre tableaux de pointeurs.

Il faut maintenant connecter tous les éléments suivant les règles de routage décrites ci-dessus.
D'abord, chaque *net* est branché sur un connecteur d'une cellule pour éviter des *nets* ou des cellules non fonctionnels. Cette opération est réalisable, car le nombre de sorties primaires est égal au nombre d'entrées primaires. On a donc autant de *nets* que d'éléments à piloter.

La connectique est terminée par un parcours en profondeur de la *netlist*. L'organigramme présenté dans la figure 3.7 illustre ce processus.

Pour chaque cellule, en commençant par les sorties primaires, on teste les connecteurs en entrée :

- Si ce connecteur n'est pas encore branché sur un *net*, on en cherche un qui ne respecte pas les contraintes de routage présentées précédemment.
- Si ce *net* n'a pas encore de *driver*, on cherche un élément susceptible de le piloter, c'est-à-dire qui respecte les contraintes définies ci-dessus. En particulier, celui-ci est déterminé en local par un tirage aléatoire pondéré pour les *nets* d'un *fan* inférieur à quatre et en global par un tirage aléatoire équiprobable pour les *nets* d'un *fan* supérieur ou égal à quatre.

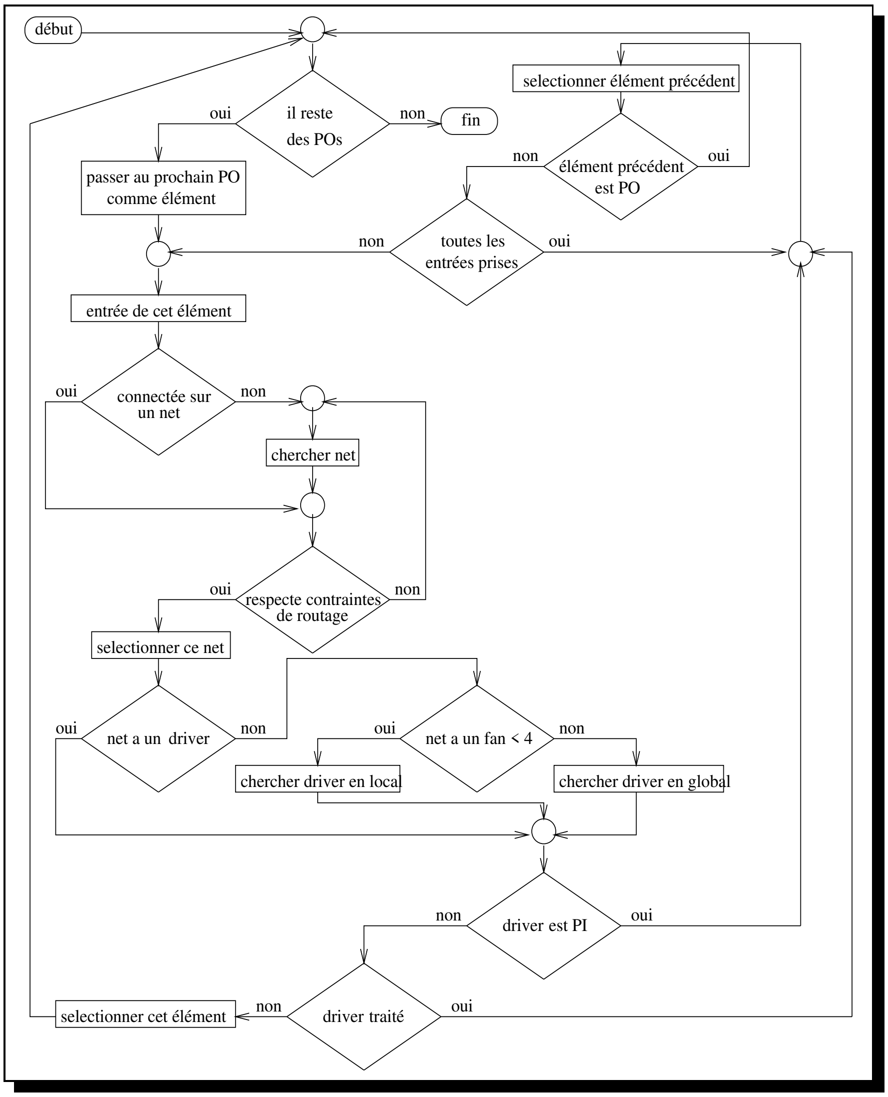


Figure 3.7: *Processus de connexion*

- Si ce *driver* n'est ni une entrée primaire ni un élément déjà traité [3], on teste récursivement toutes ses entrées.

Le choix d'un *net* se fait par un tirage aléatoire. Après avoir choisi un certain *net*, on vérifie qu'il respecte les contraintes de routage. Notamment, on teste s'il lui reste encore des connecteurs libres, s'il ne connecte pas une FF avec une autre, un PO avec un PI, ou un élément avec lui-même. De plus, on vérifie si, suite à cette connexion, la longueur des chemins auxquels il appartient, n'excède pas la longueur maximale autorisée.
Si le *net* choisi respecte ces contraintes, le choix aléatoire est retenu, sinon, on tire aléatoirement un autre *net*. Ce processus est répété tant que le *net* choisi ne respecte pas toutes les contraintes. Si au bout d'un certain nombre de tirages, aucun *net* adéquat n'a pu être identifié, le processus de tirage est interrompu. La liste chaînée des *nets* est alors parcourue et le premier *net* qui vérifie toutes les contraintes est retenu.

[3] Détection de cycles

Le choix d'un *driver* se fait également par un tirage aléatoire.
Dans certains cas, ce choix peut être restreint comme illustré dans le tableau 3.3.

type d'éléments	conditions supplémentaires	type du *driver*
LUT	longueur maximale atteinte	FF, PI
LUT	—	LUT, FF, PI
FF	—	LUT, PI
PO	—	LUT, FF

Tableau 3.3: *Choix théoriquement possible*

Après avoir déterminé le type du *driver*, un élément est tiré aléatoirement. Suivant le *fan* du *net* à piloter, ce tirage se fait soit de façon pondérée soit de façon équilibrée. Un tirage aléatoire pondéré favorise une connexion locale, tandis qu'un tirage équilibré tend plutôt à fournir une connexion globale. Si l'élément déterminé ainsi ne pilote pas encore un *net*, il est retenu. Sinon, on tire un autre élément du même type. Ce processus est répété tant que l'élément déterminé pilote un *net*.
Si au bout d'un certain nombre d'itérations, on n'a pas pu identifier un élément adéquat, le processus de tirage aléatoire est interrompu. La recherche continue avec le parcours de la liste chaînée des éléments. Pour effectuer une connexion locale, ce parcours commence avec le voisin de l'élément à piloter. Autrement, il commence au début de la liste chaînée. Le premier élément qui ne pilote pas de *net* est retenu. Si le parcours de la liste n'a pas permis de déterminer un élément de ce type, on essaie de trouver un élément d'un autre type en respectant les contraintes mentionnées dans le tableau 3.3.

La construction d'un macro-contrôleur est présentée à l'aide de l'exemple de la génération d'une *netlist* contenant 13 cellules.

Exemple 3.1 Construction d'un macro-contrôleur contenant 13 cellules
La construction d'une netlist avec 13 cellules a été choisie comme exemple. Ce cas n'est pas très réaliste, mais il illustre bien le processus de la construction.
Pour une netlist de taille 13, on trouve neuf LUTs, dont cinq LUTs avec quatre entrées, trois LUTs avec trois entrées et un LUT avec une entrée, quatre FFs, une PI[4] *et une PO. Les éléments créés sont représentés dans la figure 3.8.*

Pour correspondre au nombre d'éléments déterminé préalablement, il faut générer 14 nets (9 LUTs + 4 FFs + 1 PI = 14 équipotentielles) possédant ensemble au moins 49 connecteurs. Suivant le profil de fan défini ci-dessus, la distribution obtenue est présentée dans le tableau 3.4.

fan	*nombre de nets*	*nombre de connecteurs*
2	*12*	*24*
3	*1*	*3*
4	*1*	*4*

Tableau 3.4: *Distribution du fan sur les nets*

Soit un total de 14 nets avec 31 connecteurs.
Le nombre de connecteurs sur les nets (31) étant inférieur au nombre de connecteurs

[4] Il existe aussi une deuxième entrée primaire qui alimente le *net* d'horloge et qui est connectée à toutes les entrées horloge des FFs

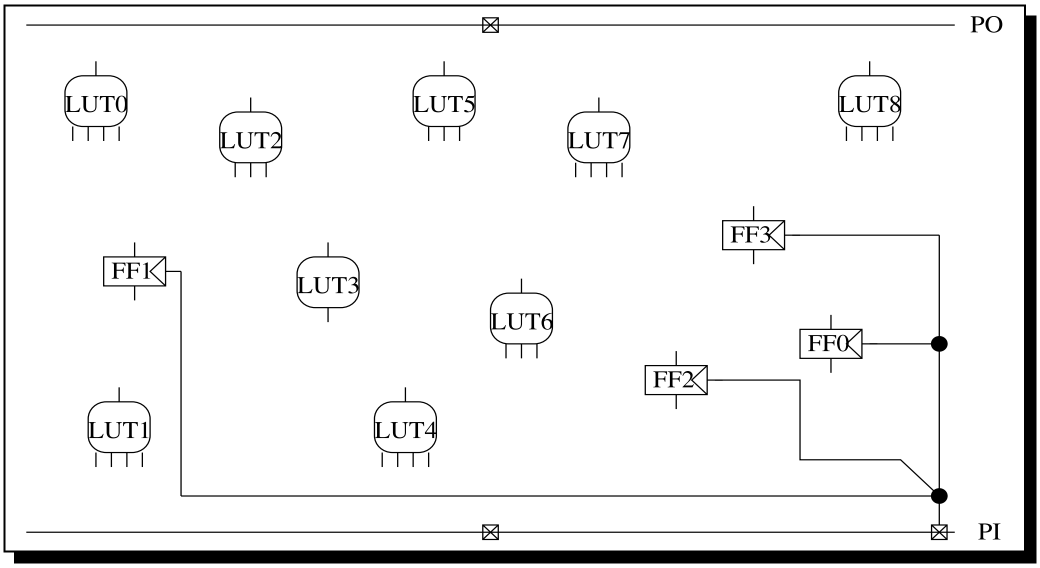


Figure 3.8: *Phase 1 : Représentation des éléments*

nécessaires (49), le programme s'arrête avec un message d'erreur adéquat. Cet effet se produit sur des netlists de petite taille ($<$ 200 *cellules). Cela est dû à la division modulo lors de la détermination de la distribution du fan. L'effet n'a pas été observé pour des macro-contrôleurs avec plus de 300 cellules, ce qui correspond à un contrôleur d'environ 12000 cellules. Ce générateur étant conçu pour créer de grandes netlists de taille supérieure à 10000 cellules, il n'est pas nécessaire de corriger cet effet.*
Pour pouvoir poursuivre cet exemple, deux nets d'un fan de 2 ont été remplacés par deux nets d'un fan de 11, afin d'atteindre les 49 connecteurs nécessaires. Ceci influence le fan moyen et n'est donc pas acceptable pour le processus de génération réel. Pour notre exemple, cette astuce n'influence pas la façon de construire la netlist. Elle permet, par contre, d'illustrer la construction à l'aide d'un exemple simple et de taille réduite.
Un fan est affecté à chaque net. Le résultat de cette phase est une liste chaînée de tous les nets, où chacun a un fan déterminé aléatoirement[5] *et où la distribution du fan correspond à celle représentée dans le tableau 3.4. Cette deuxième phase de construction est illustrée par la figure 3.9.*

L'application du branchement de chaque net sur un connecteur d'une cellule est illustrée par la figure 3.10

La phase finale du processus de la construction est illustrée par la figure 3.11. Nous effectuons cette construction en supposant que la longueur du chemin critique est limitée à trois.

La construction de la *netlist* se fait par un processus aléatoire. Il est possible de ne pas aboutir à sa construction. Trois cas différents ont été observés pour la génération des macro-contrôleurs de petites tailles :

- Lors du routage des éléments de la *netlist*, aucun *net* compatible avec les contraintes de routage ne peut être trouvé.

[5] Les *nets* ne sont pas triés suivant leur *fan* dans la liste chaînée

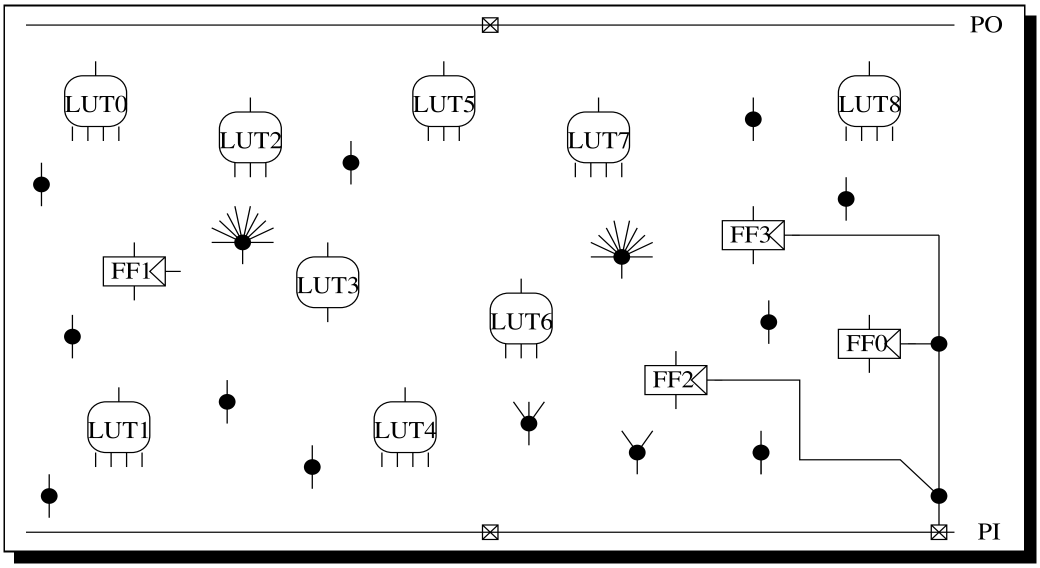


Figure 3.9: *Phase 2 : Création des nets*

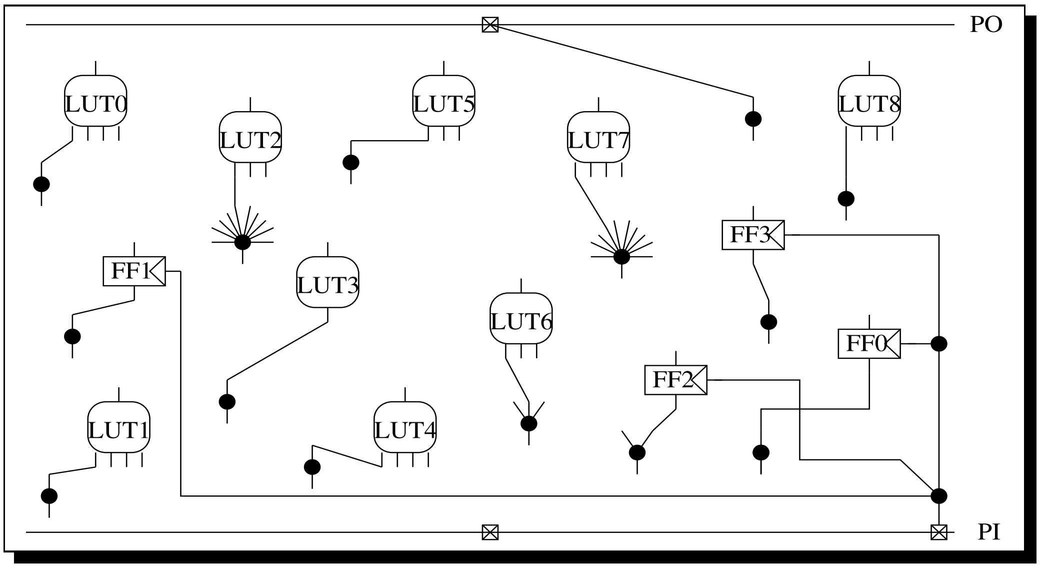


Figure 3.10: *Phase 3 : Connexion initiale des nets*

- Lors du routage des éléments de la *netlist*, aucun élément compatible avec les contraintes du tableau 3.3 ne peut être trouvé.

- Le routage n'a pas connecté tous les éléments générés.

Les deux premiers cas sont réellement bloquant. La *netlist* du macro-contrôleur ne peut pas être générée. Le troisième cas n'est pas bloquant. La *netlist* est générée sans les éléments non-connectés.

Exemple 3.2 Suppression des cellules non-connectées
Après avoir construit la connectique de la netlist du macro-contrôleur, nous pouvons mettre en évidence une erreur: Les éléments FF0, FF1, FF3, LUT0, LUT1, LUT3, LUT4, LUT5

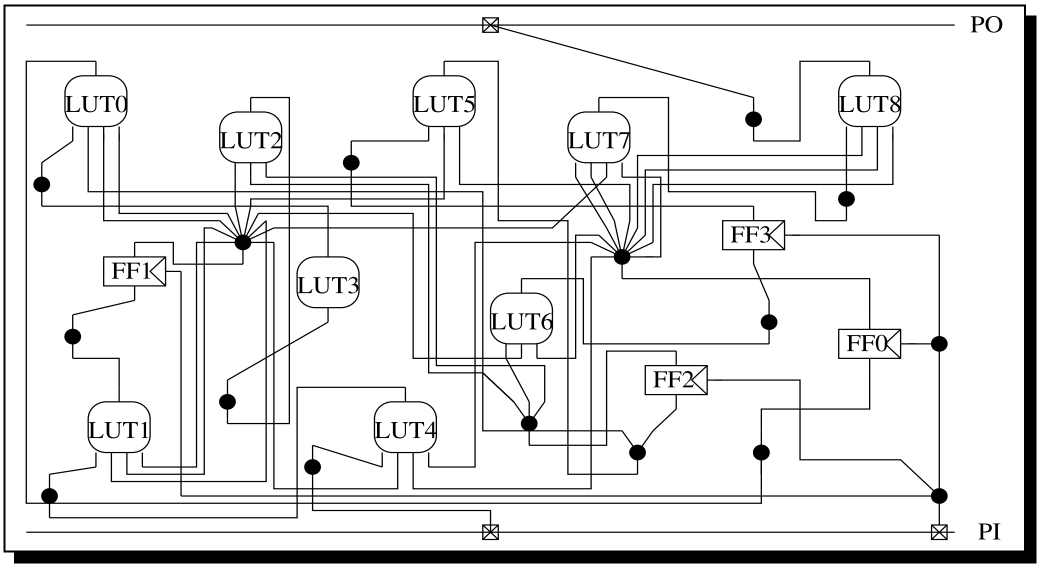


Figure 3.11: *Phase 4 : Construction de la netlist d'un macro-contrôleur*

et LUT8 ne sont pas connectés (voir figure 3.12). Pour contourner cette erreur, les éléments non-connectés sont supprimés.

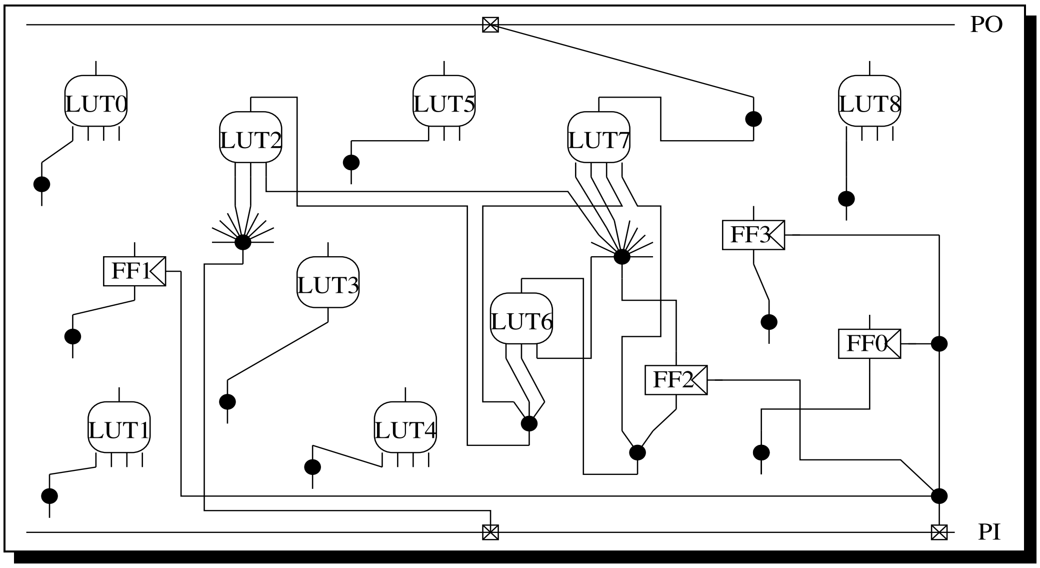


Figure 3.12: *Erreur : Eléments non-connectés*

Dans le cas le pire, la suppression des éléments non-connectés amène à construire des *netlists* vides. Il est donc nécessaire de vérifier le résultat du processus de génération. Un résumé est donné dans le tableau 3.5.

A partir d'une taille de 600 cellules, la *netlist* d'un macro-contrôleur est construite correctement, or le générateur sera utilisé pour générer des contrôleurs d'une certaine taille. Les résultats présentés dans le tableau 3.5 sont donc tout à fait acceptables.

CPBs	essais	échecs	éléments non-connectés	temps CPU
300	10	4	70%-90% des FFs	0.05 secondes
600	10	2	6%-13% des FFs	0.1 secondes
700	10	0	5%-12% des FFs	0.2 secondes
1000	10	0	3%-10% des FFs	0.3 secondes
2000	10	0	1%-2% des FFs	0.8 secondes
10000	5	0	—	12 secondes
20000	5	0	—	40 secondes

Tableau 3.5: *Construction d'une netlist de macro-contrôleur*

3.8.5 Assemblage des macro-contrôleurs

Les *netlists* des macro-contrôleurs, générées dans l'étape précédente, sont maintenant interconnectées selon la hiérarchie présentée dans figure 3.4 pour former un seul contrôleur.

Les résultats expérimentaux sont présentés dans le tableau 3.6. Ils montrent que la génération est très fiable et extrêmement rapide pour des *netlists* de grandes tailles : Un contrôleur de 300K CPBs (équivalent à 1.5M portes) est généré en neuf minutes seulement. Néanmoins, la génération d'un contrôleur de 20K CPBs échoue à deux reprises. Ceci provient d'un problème de construction d'un des 18 macro-contrôleurs. Cependant, puisque le temps de CPU du processus de génération est très bref, il n'est pas très coûteux de recommencer la génération.

nombre de CPBs	nombre d'essais	nombre d'échecs	temps CPU
20000	20	2	6 secondes
50000	10	0	16 secondes
100000	10	0	62 secondes
200000	10	0	4 minutes
300000	5	0	9 minutes

Tableau 3.6: *Génération d'une netlist de contrôleur*

3.8.6 Conclusion

Le générateur de contrôleur permet de développer de grandes *netlists* ayant les mêmes caractéristiques qu'un contrôleur industriel concernant :

- le nombre de *nets*, d'entrées primaires et de sorties primaires par rapport au nombre de cellules,
- le *fan* moyen,
- la distribution du *fan* sur les différents *nets*,
- le nombre de FFs par rapport au nombre de LUTs,
- la distribution du nombre d'entrées utilisées par les LUTs,
- la longueur du chemin critique,

- la localité des interconnexions,
- la hiérarchie de la *netlist*.

La construction se fait aléatoirement dans un délai très bref pour des *netlists* ayant jusqu'à 100K cellules et dans un délai raisonnable pour des *netlists* de plus grande taille.
De plus, des essais ont démontré que la construction réussit dans la plupart des cas (voir tableau 3.6).

Afin de valider le générateur développé, il reste à vérifier que la *netlist* générée se partitionne de la même façon que la *netlist* du contrôleur industriel. Ceci est présenté dans section 3.11.3.

3.9 Générateur d'interconnexions

L'objectif de PartGen est la génération d'une *netlist* contenant différents modules ou instances créés à l'aide des générateurs décrits ci-dessus. Il est donc nécessaire de connecter aléatoirement les différentes *netlists* (modules) générées précédemment afin d'obtenir une seule *netlist*.

D'abord, tous les modules sont parcourus, afin de déterminer le nombre total d'entrées primaires (PI), de sorties primaires (PO) et de connecteurs d'horloge. Ensuite, il faut déterminer aléatoirement le nombre d'entrées primaires de la *netlist* PI_{real}. Ce nombre peut théoriquement varier entre 1 et la somme des entrées primaires des modules[6] PI_{max} :

$$1 \leq PI_{real} \leq PI_{max} \quad \text{avec} \quad PI_{max} = \sum_{\forall \text{modules}} PI$$

Dans notre application, la borne supérieure a été fixée à la somme des entrées primaires des modules divisée par le nombre de modules afin de conserver une certaine profondeur du graphe à ce niveau de la construction :

$$PI_{real} = \text{rand}[1; \frac{PI_{max}}{\text{modules}}]$$

Ce choix limite celui du nombre de sorties primaires. Pour pouvoir connecter tous les connecteurs de sortie des modules, il faut disposer d'un nombre minimal de sorties primaires:

$$PO_{min} \leq PO_{real} \leq PO_{max} \quad \text{avec} \quad PO_{max} = \sum_{\forall \text{modules}} PO$$

En effet, PO_{real} doit être supérieur à la somme des sorties primaires des modules moins le nombre maximal de connecteurs libres en entrée de celles-ci. Ce nombre correspond à la différence entre le nombre maximal d'entrées primaires et le nombre réellement retenu :

$$PO_{min} = PO_{max} - (PI_{max} - PI_{real})$$

Après avoir déterminé ainsi l'interface de la *netlist*, il faut instancier chaque module avec des connexions entre ses propres entrées et sorties primaires et les entrées et sorties primaires de la *netlist* ainsi que celles des autres modules.

[6] Si un connecteur d'horloge a été détecté dans un des modules, la *netlist* aura aussi un connecteur d'horloge, qui sera relié à tous les connecteurs d'horloge.

Le routage doit évidemment respecter les règles élémentaires de l'électronique. Pour cela certaines contraintes ont été imposées.

- Le nombre de *nets* est égal au nombre d'équipotentielles, c'est-à-dire à la somme des sorties primaires des modules et des entrées primaires de la *netlist*.
- Chaque *net* doit avoir exactement un *driver*, afin d'éviter des court-circuits et des éléments en l'air.
- Chaque connecteur des modules doit être branché à exactement un *net*, pour éviter des connecteurs en l'air et des court-circuits.
- Il faut éviter les boucles sur le même module.
- Les connexions doivent être de nature aléatoire.
- La construction doit se faire le plus vite possible.

3.10 Intégration des sous-générateurs dans PartGen

Afin de faciliter l'utilisation du générateur, les différents sous-générateurs développés ont été intégrés dans un environnement commun.
La commande suivante :

```
PartGen <bench_name> <bench_size>
            <percent_op> <nbSubgraphs_op>
            <percent_sram> <nbSubgraphs_sram>
            <percent_xmem> <nbSubgraphs_xmem>
            <percent_glue> <nbSubgraphs_glue>
            <percent_ctrl> <nbSubgraphs_ctrl>
```

permet de générer une *netlist* nommé *bench_name* contenant *bench_size* cellules[7] qui est composée :

- de *percent_op* pour cent d'opérateurs, divisés en *nbSubgraphs_op netlists* indépendantes,
- de *percent_sram* pour cent de mémoire isolée, divisés en *nbSubgraphs_sram netlists* indépendantes,
- de *percent_xmem* pour cent de mémoire intégrée dans les AVCs, divisés en *nbSubgraphs_xmem netlists* indépendantes,
- de *percent_glue* pour cent de combinatoire irrégulière, divisés en *nbSubgraphs_glue netlists* indépendantes,
- de *percent_ctrl* pour cent de contrôleur, divisés en *nbSubgraphs_ctrl netlists* indépendantes.

[7] Pour les sous-graphes en *meta_sram_bank* et en *meta_xmem*, on obtient la taille correspondante en place occupée

Par contre nous n'avons pas de contrôle direct sur le nombre de connecteurs d'entrées/sorties créés. En effet, le nombre de connecteurs des instances de type opérateur, mémoire intégrée, mémoire isolée et combinatoire irrégulière est déterminé par le sous-générateur en question. Le nombre de connecteurs des instances de type contrôleur est fixé par rapport à la cardinalité de ces instances. Le nombre de connecteurs d'une telle instance correspond donc approximativement au nombre de connecteurs d'une *netlist* de contrôleur industrielle pour des instances contenant de 10K à 150K CPBs.
Chaque partie de la *netlist* générée avec PartGen respecte ainsi les relations entre cardinalité et nombre de connecteurs des *netlists* réelles. Nous supposons donc que le paramètre de Rent [Lan71] est respecté à l'intérieur de la *netlist*.
Quant au nombre de connecteurs externes de la *netlist*, celui-ci a rarement une influence sur le résultat du partitionnement. Le fait de ne pas pouvoir le contrôler n'est donc pas très important.

3.11 Résultats expérimentaux

3.11.1 Introduction

L'objectif du générateur PartGen est de créer des *netlists* aléatoires pouvant servir de *benchmark* pour le partitionnement. Il faut donc vérifier si les *netlists* générées possèdent les caractéristiques nécessaires. En effet, une telle *netlist* doit se partitionner de façon analogue à une *netlist* réelle.

La validation de PartGen a été effectuée sur un serveur SUN Enterprise 5000 avec 8 CPUs Ultra Sparc II, 167 MHz et 4Go RAM.
Le critère de validation est le taux moyen de remplissage des FPGAs nécessaires pour l'implémentation de la *netlist* entière et le taux moyen d'utilisation des connecteurs de ces FPGAs.

Définition 3.1 *Taux moyen de remplissage, taux moyen d'utilisation de connecteurs:*
Soit n le nombre de FPGAs nécessaires à l'implémentation d'une netlist de $|V|$ CLBs. Soit k le nombre de CLBs et c le nombre de connecteurs disponibles dans chaque FPGA. Si $|E|$ est la somme des connecteurs externes utilisés sur les n FPGAs, alors $F = |V|/(n \times k)$ est le taux moyen de remplissage des FPGAs et $P = |E|/(n \times c)$ leurs taux moyen d'utilisation de connecteurs.

Le partitionnement est effectué en respectant les contraintes de deux FPGAs différents de la famille 3000 de Xilinx : le XC3064 avec 224 CLBs et 120 connecteurs d'entrées/sorties et le XC3090 avec 320 CLBs et 144 connecteurs d'entrées/sorties [Xil92].

L'outil de partitionnement employé pour les tests effectue un bipartitionnement récursif de la *netlist*. Le partitionnement se termine quand toutes les partitions respectent les contraintes de cardinalité et de coupure des FPGAs sélectionnés. L'algorithme utilisé dans l'outil de partitionnement est celui décrit dans section 4.2.3, une variante de l'algorithme *ratio-cut* présenté par Wei et Cheng dans [Wei91].

La validation des générateurs existants, c'est-à-dire du générateur d'opérateurs, de mémoire isolée et de mémoire intégrée, est inutile, car ils créent des *netlists* fonctionnelles.
La validation des autres générateurs est présentée par la suite.

3.11.2 Test du générateur de combinatoire irrégulière

Le générateur de combinatoire irrégulière crée des *netlists* qui ne possèdent pas de caractéristiques particulières et qui sont par conséquent difficilement partitionnables. Pour le démontrer, nous avons comparé le partitionnement d'une telle *netlist* avec celui obtenu pour une *netlist* de type opérateur de même taille. Cette dernière étant très régulière est facile à partitionner comme nous avons constaté dans section 3.3.
Les résultats expérimentaux sont présentés dans le tableau 3.7.

	Combinatoire irrégulière				Opérateur				
FPGA	#CLBs	F_irr	P_irr	T_cpu	#CLBs	F_op	P_op	T_cpu	F_irr / F_op
XC3064	4417	16.2%	73.2%	12 s	5298	48.3%	64.4%	9 s	0.34
XC3090	4417	12.5%	66.5%	12 s	5298	44.7%	69.7%	8 s	0.28
XC3064	9103	17.7%	72.3%	43 s	10218	45.2%	66.4%	23 s	0.39
XC3090	9103	13.9%	67.1%	43 s	10218	42.0%	69.8%	21 s	0.33
XC3064	25595	12.7%	63.2%	281 s	24858	48.7%	71.4%	121 s	0.26
XC3090	25595	13.4%	76.5%	297 s	24858	40.7%	69.6%	109 s	0.33

Tableau 3.7: *Test du générateur de netlist de combinatoire irrégulière*

Les résultats montrent que la *netlist* de combinatoire irrégulière est beaucoup plus difficile à partitionner. En effet, pour le même taux d'utilisation des *pins* (P_irr, P_op), le rapport (F_{irr}/F_{op}) entre les taux de remplissage F_{irr} (obtenu pour les *netlists* de combinatoire irrégulière) et F_{op} (obtenu pour les *netlists* d'opérateurs) est dans l'intervalle [0.26;0.39] pour des FPGAs XC3064 et dans l'intervalle [0.28;0.33] pour des FPGAs XC3090. L'outil de partitionnement nécessite donc deux à trois fois plus de FPGAs pour implémenter une *netlist* de combinatoire irrégulière que pour une *netlist* d'opérateur. Ce facteur est réaliste aux vues des résultats de partitionnement obtenus pour ce type de *netlist* dans des *designs* industriels. Ces observations confirment notre approche consistant à utiliser GEN pour générer des *netlists* de combinatoire irrégulière.

3.11.3 Validation du générateur de contrôleur

Pour valider le générateur de *netlists* de type contrôleur, nous avons comparé les résultats de partitionnement en terme de taux de remplissage moyen (F) et taux d'utilisation des connecteurs (P) obtenus pour des *netlists* générées avec PartGen avec ceux obtenus pour des *netlists* de contrôleurs industriels. Nous avons également comparé nos résultats avec ceux obtenus pour des *netlists* clone des *netlists* industrielles générées avec GEN[8] [Hut96] et avec ceux obtenus pour le partitionnement de *netlists* de même cardinalité générées aléatoirement sans aucune information statistique ni hiérarchique.

Les cardinalités des *netlists* industrielles sont 16893 CLBs (indust_1), 20718 CLBs (indust_2) et 22578 CLBs (indust_3). Cinq *netlists* différentes de taille égale ont été générées avec PartGen et avec GEN pour chaque *netlist* industrielle. Ces *netlists* ont été partitionnées afin de vérifier la sensibilité des deux générateurs par rapport au processus de génération aléatoire. Nous présentons ici uniquement le meilleur (best) et le plus mauvais (worst) résultat obtenu.

[8] L'outil CIRC permet l'extraction d'informations statistiques de la *netlist* industrielle. Le générateur GEN tient compte de ces informations lors du processus de génération. Le résultat est une *netlist* clone qui est supposée reproduire les mêmes informations statistiques que la *netlist* d'origine.

Nous utilisons deux autres algorithmes de partitionnement, en plus de l'algorithme de bipartitionnement récursif (RBA) [Wei91], pour valider le générateur de contrôleur. Le premier est l'application récursive d'une variante de l'algorithme FBB [Yan96] décrit dans 6.2.1 avec un critère d'équilibre de 25% des CLBs dans une des deux partitions. Le deuxième est l'algorithme WINDOW et DPRP décrit dans chapitre 7.
Les résultats expérimentaux sont présentés dans le tableau 3.8 et illustrés dans la figure 3.13.

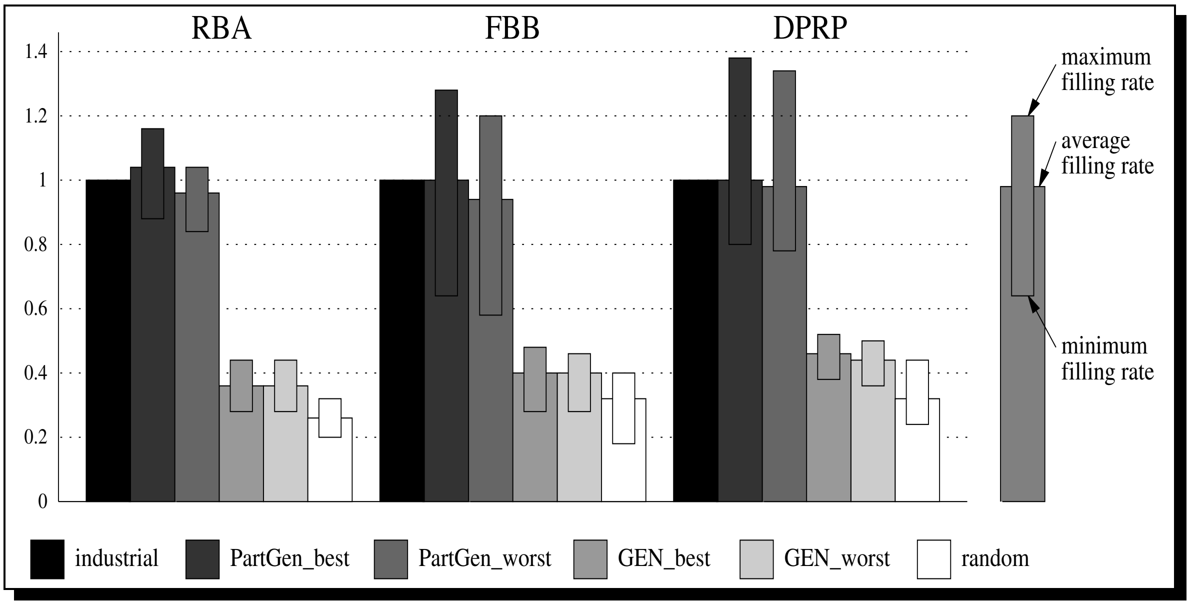


Figure 3.13: *Représentation graphique des taux de remplissage moyen*

	RBA				FBB				DPRP			
Netlist	F	F/Fo	P	Tcpu	F	F/Fo	P	Tcpu	F	F/Fo	P	Tcpu
indust_1	44.1%	1	61.7%	122s	32.5%	1	66.5%	12min	62.3%	1	89.6%	350s
PartGen_best	50.5%	1.15	73.5%	111s	35.7%	1.10	65.7%	38min	52.2%	0.84	95.0%	345s
PartGen_worst	45.5%	1.03	68.3%	126s	33.9%	1.04	64.9%	30min	49.7%	0.80	96.3%	358s
GEN_best	19.3%	0.44	72.6%	177s	15.5%	0.48	62.6%	72min	28.5%	0.46	98.2%	555s
GEN_worst	18.9%	0.43	69.9%	170s	15.1%	0.46	62.5%	64min	27.3%	0.44	97.9%	561s
random_1	14.3%	0.32	74.5%	322s	11.0%	0.34	63.2%	141min	18.2%	0.29	97.9%	993s
indust_2	52.2%	1	62.4%	188s	52.8%	1	64.8%	61min	67.5%	1	91.2%	562s
PartGen_best	45.2%	0.87	71.5%	194s	33.3%	0.63	65.4%	48min	55.0%	0.81	96.7%	555s
PartGen_worst	43.4%	0.83	71.8%	218s	30.8%	0.58	62.3%	69min	52.5%	0.78	96.8%	588s
GEN_best	15.3%	0.29	63.6%	382s	14.6%	0.28	70.6%	107min	24.7%	0.37	96.9%	1018s
GEN_worst	14.7%	0.28	62.1%	388s	14.1%	0.27	68.7%	113min	24.2%	0.36	97.3%	1020s
random_2	10.4%	0.20	60.5%	615s	9.6%	0.18	62.7%	239min	16.2%	0.24	97.8%	1749s
indust_3	42.2%	1	72.1%	245s	26.4%	1	66.9%	38min	41.0%	1	89.6%	1407s
PartGen_best	44.9%	1.06	72.9%	302s	33.6%	1.27	65.0%	183min	56.5%	1.38	96.1%	1391s
PartGen_worst	42.0%	1.00	68.4%	295s	31.9%	1.21	63.7%	249min	54.8%	1.34	95.6%	1470s
GEN_best	15.2%	0.36	73.8%	539s	12.5%	0.47	63.8%	283min	21.5%	0.52	97.6%	1749s
GEN_worst	14.6%	0.35	71.1%	554s	12.2%	0.46	62.3%	284min	20.7%	0.50	98.7%	1722s
random_3	11.6%	0.27	63.3%	627s	10.6%	0.40	63.8%	454min	17.9%	0.44	97.6%	1980s

Tableau 3.8: *Validation du générateur de netlists de contrôleur*

Les résultats expérimentaux montrent que le partitionnement des *netlists* générées avec PartGen fournit un taux de remplissage moyen comparable à celui des *netlists* industrielles pour les trois algorithmes de partitionnement. En même temps, le taux de remplissage obtenu pour les *netlists* générées avec GEN et celui obtenu pour les *netlists* générées de manière totalement aléatoire diffère sensiblement.
En effet, le rapport (F / F_o) entre les taux de remplissage moyen des FPGAs XC3064 des *netlists* générées F et celui des *netlists* industrielles F_o est proche de 1 pour les *netlists*

générées avec PartGen. Le meilleur résultat a été obtenu pour l'algorithme RBA : le taux de remplissage moyen diverge de moins de 17% de celui des *netlists* industrielles. FBB et DPRP obtiennent de bons résultats pour *netlist* indust_1 et des résultats satisfaisants pour les deux autres *netlists*. Les plus mauvais résultats en terme de taux de remplissage moyen sont 0.58 (FBB) et 1.38 (DPRP) fois le taux de remplissage moyen de la *netlist* industrielle correspondante. En même temps, les taux de remplissage des *netlists* générées sans informations structurelles et topologiques et ceux obtenus pour les *netlists* générées avec GEN n'excèdent pas 0.44 (random) et 0.52 (GEN) fois le taux de remplissage moyen des *netlists* industrielles.
Notons également que le rapport (T_{pg} / T_o) entre le temps CPU du partitionnement des *netlists* générées avec PartGen T_{pg} et celui des *netlists* industrielles T_o est proche de 1 pour RBA et DPRP et n'excède 3.2 qu'une seule fois pour FBB. En même temps, le rapport (T_r / T_o) entre le temps CPU du partitionnement des *netlists* générées de manière totalement aléatoire T_r et celui des *netlists* industrielles n'est qu'une seule fois inférieure à 2.5.

Nous pouvons conclure alors que les *netlists* générées avec PartGen se partitionnent approximativement de la même manière que les *netlists* industrielles, ce qui valide l'utilisation de notre générateur de contrôleur pour créer des *netlists* utilisables pour l'évaluation des performances des différents algorithmes de partitionnement.

3.11.4 Test du générateur d'interconnexions

Le générateur d'interconnexions relie les modules générés avec les différents sous-générateurs. Il n'introduit pas de ressources physiques et ne peut pas créer de *netlists* sans ces sous-générateurs.
Les interconnexions créées dépendent de la nature des modules. Il n'y a pas d'interconnexions "typiques" entre modules. Pour ces raisons, le générateur d'interconnexions ne peut pas être validé séparément.
En effet, la variation du nombre d'entrées/sorties a une faible influence sur le résultat final. De plus, la plupart des algorithmes ne tient pas compte des connecteurs externes.
Néanmoins, l'influence des interconnexions externes sur le résultat du partitionnement reste un domaine ouvert à la recherche.

3.12 Conclusion

Nous avons constaté un besoin de *netlists* de grandes tailles pouvant servir comme *benchmark* pour qualifier les performances des outils de partitionnement.

Dans un premier temps, nous avons essayé d'utiliser des *benchmarks* disponibles ou des *netlists* créées avec des générateurs existants. Ces *netlists* ne se partitionnent pas de la même manière que des *netlists* réelles. Il était donc nécessaire de développer un générateur aléatoire mieux adapté.

L'analyse d'un nombre représentatif de *netlists* industrielles a fait apparaître six catégories. Notre générateur aléatoire est composé de plusieurs sous-générateurs; un pour chaque type de *netlist* identifié et un pour connecter les différents modules. Il permet ainsi de générer de grandes *netlists* hétérogènes comportant plusieurs sous-*netlist* de différents types.
Afin de générer ces dernières, nous nous basons sur des générateurs internes utilisés par

la société Mentor Graphics [Men99a] [Men99b] d'une part et sur nos propres générateurs d'autre part.
Pour faciliter leur utilisation, les différents générateurs ont été intégrés dans un environnement commun.

Les générateurs d'opérateur, de mémoire isolée et de mémoire intégrée créent des *netlists* fonctionnelles. Elles se partitionnent donc comme une *netlist* réelle.
Le générateur de combinatoire irrégulière se base sur le générateur GEN. Une comparaison avec des *netlists* de types opérateur montre que celles-ci se partitionnent plus facilement, ce qui confirme nos hypothèses.
Le générateur de contrôleur a été développé entièrement. Les résultats expérimentaux montrent qu'il est capable de générer des *netlist* aléatoires qui se comportent comme des *netlists* réelles lors du partitionnement.

Avec ces générateurs, nous disposons d'un outil permettant de créer de grandes *netlists* hétérogènes de composition et de taille arbitraire. Ces *netlists* seront utilisées lors des expériences effectuées pour l'évaluation des performances de différents algorithmes de partitionnement présentée dans les chapitres suivants.

Chapitre 4

Bipartitionnement par une approche de voisinage : algorithme bisect*

Dans ce chapitre nous analysons les performances d'un algorithme de bipartitionnement par approche de voisinage. Nous nous basons sur une première version de l'algorithme développée chez Mentor Graphics et appelée "bisect". L'étude des avantages et inconvénients de cet algorithmes nous a conduit à modifier l'algorithme de base menant à une version améliorée appelée "bisect*" (cf. section 4.2).
En particulier, nous avons apporté les améliorations suivantes :

- Développement d'un algorithme de *clustering* permettant de regrouper un certain nombre de cellules de la *netlist*. Cet algorithme est présenté dans la section 4.2.1.
- Proposition d'une autre méthode pour déterminer le bipartitionnement initial, présentée dans la section 4.2.2.
- Introduction d'une autre fonction de coût pour évaluer la qualité de la solution courante. Cette fonction est présentée dans la section 4.2.3.

4.1 Algorithme de base : "bisect"

4.1.1 Description de l'algorithme de base

L'algorithme de bipartitionnement existant appartient à la famille des algorithmes de voisinage. Il explore l'espace des solutions en passant d'une solution à une solution voisine. Il se termine quand il a atteint un minimum local.
L'algorithme implémenté est basé sur l'algorithme *ratio-cut* présenté par Wei et Cheng [Wei91]. Il s'agit d'un algorithme qui minimise le *ratio-cut* entre les deux sous-ensembles créés. L'algorithme effectue le bipartitionnement d'une *netlist* $N = (V, E)$ avec $|V|$ cellules et $|E|$ *nets* en 3 phases :

1. Déterminer un bipartitionnement initial.
   Un germe $v_i \in V$ est tiré aléatoirement. A partir de ce germe la *netlist* est explorée en

largeur, puis la cellule la plus distante de v_i, v_k, est déterminée comme second germe. Ensuite, deux partitions X et $\bar{X}$ sont créées avec $X = v_i$ et $\bar{X} = N - v_i$. Un gain est calculé pour chacune des cellules de la *netlist* à l'exception des deux germes v_i et v_k. Ce gain reflète la réduction du nombre de *nets* coupés entre les deux partitions dans le cas du déplacement d'une cellule de $\bar{X}$ vers X. Les cellules sont ensuite rangées dans des listes de gains comme proposé par Fiduccia et Mattheyses [Fid82]. Puis une par une, jusqu'il ne reste plus de cellules dans $\bar{X}$, la cellule $v_j \in \bar{X}$ avec le plus grand gain est sélectionnée et placée dans X. A chaque déplacement d'une cellule, les gains des cellules non-verrouillées et connectées à cette cellule ainsi que leurs rangements respectifs dans la liste des gains sont mis à jour. Ensuite, le nouveau *ratio-cut* entre les deux partitions est calculé et, dans le cas d'une amélioration, la configuration des deux partitions est sauvegardée. Ce procédé est répété en commençant avec $X = v_k$ et $\bar{X} = N - v_k$ Finalement, le bipartitionnement qui minimise le *ratio-cut* entre les deux partitions est restauré. Il servira par la suite de bipartitionnement initial.

2. Décalage itératif.
   Soit $(X, \bar{X})$ le bipartitionnement initial et rc_{old} le *ratio-cut* de cette solution. Un gain est calculé pour chacune des cellules de la *netlist*. Comme lors de la première étape, ce gain reflète la réduction du nombre de *nets* coupés entre les deux partitions dans le cas du déplacement d'une cellule de la partition actuelle vers l'autre partition. Les cellules sont ensuite rangées dans des listes de gains comme proposé par Fiduccia et Mattheyses [Fid82]. Puis une par une, jusqu'il ne reste plus de cellules dans $\bar{X}$, la cellule $v_j \in \bar{X}$ avec le plus grand gain est sélectionnée et placée dans X. A chaque déplacement d'une cellule, les gains des cellules non-verrouillées et connectées à cette cellule et leurs rangements respectifs dans la liste des gains sont mis à jour, puis le *ratio-cut* entre les deux partitions est calculé et, dans le cas d'une amélioration, la configuration des deux partitions est sauvegardée. Ensuite, le bipartitionnement qui minimise le *ratio-cut* entre les deux partitions rc_{new} est restauré. Si ce *ratio-cut* rc_{new} est inférieur à l'ancien *ratio-cut* ($rc_{new} < rc_{old}$), le traitement continue. Ce procédé est répété en déplaçant maintenant toutes les cellules qui sont actuellement dans X vers $\bar{X}$. Ces deux décalages sont répétés tant que la solution en terme de *ratio-cut* est améliorée c'est-à-dire $rc_{new} < rc_{old}$.

3. Echange de sommets :
   Soit $(X, \bar{X})$ le meilleur bipartitionnement déterminé lors de la phase précédente et rc_{old} le *ratio-cut* correspondant. Un gain est calculé pour chacune des cellules de la *netlist*. Comme lors de la première étape, ce gain reflète la réduction du nombre de *nets* coupés dans le cas du déplacement d'une cellule de la partition actuelle vers l'autre partition. Les cellules sont ensuite rangées dans des listes de gains comme proposé par Fiduccia et Mattheyses [Fid82]. Puis une par une, jusqu'il ne reste plus de cellules non-verrouillées, la cellule avec le meilleur gain dans une des deux partitions est sélectionnée et placée dans la partition opposée. A chaque déplacement d'une cellule, les gains des cellules non-verrouillées et connectées à cette cellule et leurs rangements respectifs dans la liste des gains sont mis à jour, puis le *ratio-cut* entre les deux partitions est calculé et, dans le cas d'une amélioration, la configuration des deux partitions est sauvegardée. Ensuite, le bipartitionnement qui minimise le *ratio-cut* entre les deux partitions est restauré. Tant que le *ratio-cut* déterminé pour la meilleure solution de cette passe rc_{new} est inférieur à l'ancien *ratio-cut* ($rc_{new} < rc_{old}$)

le traitement continue. Sinon, la partition obtenue lors de l'étape précédente est choisie en tant que résultat final et l'algorithme se termine.

Cet algorithme est de complexité $O(p|V|)$ avec p le nombre moyen de connecteurs par cellule dans la *netlist*. Dans le cas d'une *netlist* de CPBs, où chaque cellule a un au plus cinq connecteurs, nous avons une complexité linéaire en dépendance du nombre de cellules.

4.1.2 Avantages et inconvénients

Les grands avantages de cet algorithme sont sa rapidité grâce à sa complexité linéaire et sa capacité à déterminer des coupures naturelles d'une *netlist*. En effet, il n'est pas nécessaire, comme par exemple pour l'algorithme de Fiduccia et Mattheyses [Fid82], de spécifier une cardinalité maximale pour une des deux partitions. Les deux partitions créées sont en général naturellement équilibrées, grâce au critère du *ratio-cut*.

Néanmoins, nous avons pu observer qu'une des partitions était presque vide. Ceci se produit, quand la *netlists* est relativement connexe. Dans ce cas, l'algorithme n'est pas capable de déterminer un bipartitionnement équilibré avec une coupure assez petite pour que le *ratio-cut* correspondant soit inférieur au *ratio-cut* pour un bipartitionnement déséquilibré. Ce phénomène est illustré par l'exemple suivant.

Exemple 4.1 *Bipartitionnement déséquilibré :*
Soit $(X, \bar{X})$ *un bipartitionnement déséquilibré avec* $|X| = 4$, $|\bar{X}| = 40000$ *et* $c(X, \bar{X}) = 1$. *Le ratio-cut pour ce bipartitionnement est*

$$rc_1 = \frac{c}{|X| * |\bar{X}|} = \frac{1}{160000}$$

Soit $(Y, \bar{Y})$ *un bipartitionnement équilibré avec* $|Y| = 20000$, $|\bar{Y}| = 20000$ *et* $c(Y, \bar{Y}) = 2600$. *Le ratio-cut pour ce bipartitionnement est*

$$rc_2 = \frac{c}{|Y| * |\bar{Y}|} = \frac{1}{153846.15}$$

L'algorithme favorise donc la première solution, car $rc_1 < rc_2$. *Pour que l'algorithme choisisse la deuxième solution, il aurait fallu trouver une coupure* $c(Y, \bar{Y}) < 2500$. *Apparemment, l'algorithme n'a pas pu trouver une telle coupure.*

Lors de la première phase, l'algorithme détermine déjà un bipartitionnement initial d'une qualité acceptable qui dépendent des germes choisis. Nous avons observé que ce bipartitionnement initial mène rapidement, c'est-à-dire après deux ou trois itérations dans la deuxième et troisième phase, vers un minimum local. Ainsi, la qualité de la solution finale dépend fortement des deux sommets choisis pour construire le bipartitionnement initial. Il est donc raisonnable de supposer que cette solution est d'une qualité médiocre. Cette hypothèse a été confirmée par Hauck et Borriello [Hau97].

4.1.2.1 Bipartitionnement récursif

L'algorithme poursuit l'objectif de minimiser le nombre de *nets* coupés entre les deux partitions en favorisant des partitions de taille équilibrée. Il ne tient pas compte des connecteurs externes de l'ensemble de cellules à partitionner. Ces connexions externes sont

pourtant très importantes dans le cas d'un bipartitionnement récursif: les deux partitions issues d'un bipartitionnement sont à leur tour partitionnées récursivement tant que une des contraintes, comme par exemple des contraintes de cardinalité et de coupure ne sont pas respectées. Le résultat est un arbre de partitionnement où l'ensemble des feuilles constitue un multipartitionnement de la *netlist* d'origine. Lors d'un bipartitionnement, les *nets* connectant des cellules dans différentes branches de l'arbre sont comptabilisées en tant que connecteurs externes des partitions respectives. Si ces connecteurs externes ne sont pas distribués équitablement entre les deux partitions la profondeur de l'arbre augmente, ce qui entraîne l'augmentation du temps de calcul et du nombre de feuilles de l'arbre.
Nous allons illustré ce phénomène à l'aide d'un exemple.

Exemple 4.2 *Arbre de bipartitionnement :*
Soit $N = (V, E)$ une netlist contenant $|V| = 1000$ cellules et ayant 100 connecteurs externes. L'objectif consiste à partitionner cette netlist en un nombre minimal de partitions tels que chaque partition ne contienne pas plus de 250 cellules avec un maximum de 130 connecteurs externes. Un bipartitionnement récursif qui tient compte d'une distribution équilibrée des connecteurs externes à chaque bipartitionnement est présenté comme cas 1 de la figure 4.1. Le cas 2 de cette figure illustre un bipartitionnement récursif menant à des partitions avec les mêmes cardinalités et les mêmes coupures sans tenir compte d'une distribution équilibrée des connecteurs externes. L'arbre de partitionnement du cas 1 a une profondeur de 2, nécessite

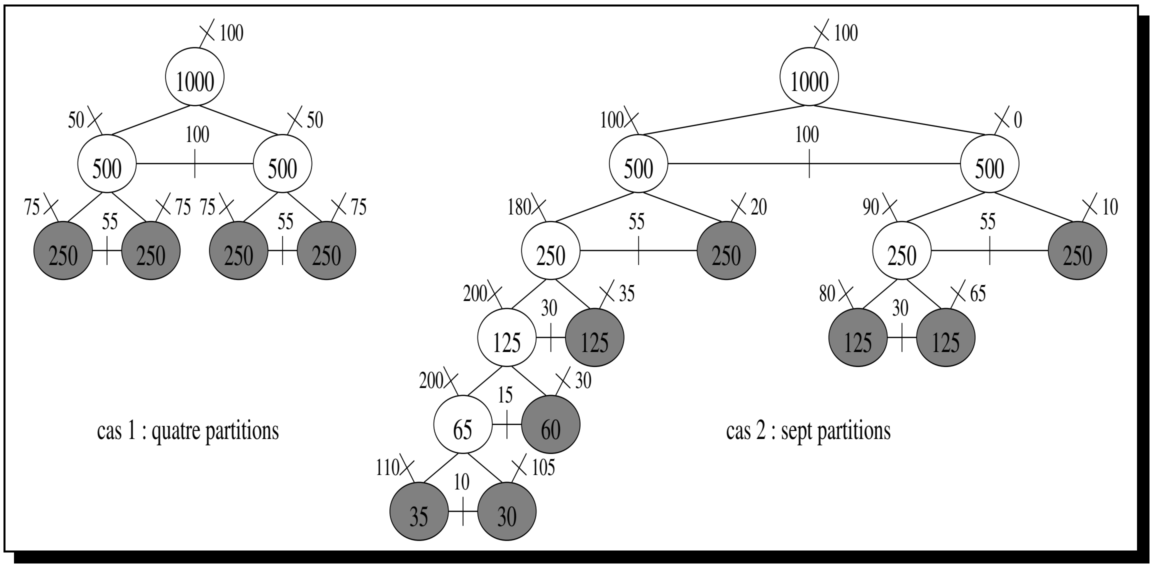


Figure 4.1: *Bipartitionnement sans et avec distribution équilibrée des connecteurs externes*

3 bipartitionnements et mène à 4 partitions, tandis que celui du cas 2 a une profondeur de 5, nécessite 7 bipartitionnement et mène à 7 partitions. Le simple fait de minimiser la coupure à chaque bipartitionnement sans tenir compte des connecteurs externes peut donc doubler le nombre de partitions nécessaires à l'implémentation d'une netlist.

Dans le cadre du bipartitionnement récursif avec contraintes de cardinalité et de coupure, l'objectif du bipartitionnement devrait donc consister à minimiser la coupure de chacune des deux partitions tout en distribuant équitablement les connecteurs externes.

4.2 Améliorations de l'algorithme de base

Nous présentons dans ce paragraphe nos diverses contributions visant à améliorer l'algorithme de base.
Suite à l'analyse des inconvénients, nous proposons des améliorations sous forme : d'un *clustering* initial, de différentes approches pour déterminer le bipartitionnement initial et de l'utilisation d'une fonction de coût différente. Ces améliorations sont présentées dans les trois sections suivantes.

4.2.1 Clustering initial

La première amélioration consiste à introduire un prétraitement qui est effectué une seule fois avant le bipartitionnement récursif. Par ce prétraitement nous fusionnons certaines cellules (*clustering*), ce qui mène à une *netlist* contenant moins de cellules et moins de *nets*. L'algorithme de bipartitionnement traite alors une *netlist* contenant moins de cellules avec un nombre moyen de connecteurs par cellule plus élevé. Le regroupement de deux cellules avec 5 connecteurs chacune et ayant 4 *nets* en commun par exemple mène à une cellule de cardinalité 2 avec 6 connecteurs.

Une des conséquences de ce prétraitement est une réduction potentielle du temps de calcul. Puisque l'algorithme de base a une complexité en $O(p|V|$, le temps de calcul est réduit, si le gain de performance obtenu grâce à une réduction du nombre de cellules n'est pas entièrement compensé par la perte de performance dû à l'augmentation du nombre moyen de connecteurs par cellule. Puisque nous veillons à éliminer au moins un *net* pour chaque regroupement de cellules, le nombre moyen de connecteurs par *net* croît moins vite que le nombre de cellules de la *netlist* ne décroît. Nous améliorons donc toujours le temps de calcul. Le taux de cette baisse du temps de calcul dépend de la relation entre la réduction du nombre de cellules et l'augmentation du nombre moyen de connecteurs par cellule.

Un CPB (cf. glossaire) peut réaliser toute fonction logique à quatre entrées et une sortie. Une *netlist* de CPBs contient donc des cellules ayant chacune cinq connecteurs. Le gain maximal du déplacement d'une cellule est donc 5, le minimum est -5. Il est évident que dans une *netlist* contenant plusieurs dizaines de milliers de cellules, beaucoup de cellules[1] ont le même gain.
La qualité du bipartitionnement dépend fortement du bon choix de la cellule à déplacer. Dans le cas où beaucoup de cellules ont le même gain, il est peu probable que le bon choix soit fait.
Après le *clustering*, les cellules ont un nombre moyen de connecteurs plus élevé. Si nous admettons que nous autorisons le regroupement de jusqu'à 10 cellules, le gain peut varier entre 40 et -40, dans le cas où les 10 cellules ont un seul *net* en commun. Il est évident que moins de cellules ont le même gain, plus on augmente la probabilité pour effectuer un déplacement de cellules intéressant et déterminer un meilleur bipartitionnement.

Après avoir présenté les raisons qui nous ont conduit à implémenter un prétraitement, nous allons voir maintenant les objectifs poursuivis par un tel procédé. En particulier, le *clustering* doit satisfaire les contraintes suivantes :

[1] en moyenne $\frac{\text{nombre de cellules}}{11}$ cellules

- Il doit être relativement rapide, pour ne pas dégrader le temps de calcul total. Puisque le *clustering* est effectué une seule fois, l'influence de son temps de calcul sur le temps de calcul total diminue dans le cas d'un bipartitionnement récursif avec le nombre de récursions.

- Il doit regrouper des cellules qui sont naturellement dans la même partition. En effet, le regroupement de cellules qui sont séparées par une coupure naturelle mène à une plus mauvaise coupure.

Pour répondre à ces conditions, nous avons implémenté l'algorithme suivant, qui est basé sur la distribution du nombre de connecteurs sur les *nets* dans les *netlists* réelles et l'influence de cette distribution sur le partitionnement : d'un côté les *nets* ayant beaucoup de connecteurs ont une plus grande probabilité d'être coupés que les *nets* avec peu de connecteurs. D'un autre côté, la grande majorité des *nets* d'une *netlist* de CPBs a peu de connecteurs. En effet, 70% des *nets* ont 2 connecteurs, 10% ont 3 connecteurs et 8% ont 4 connecteurs. Il faut donc favoriser le regroupement de cellules menant à une suppression de ces *nets*. Pour ce faire, l'algorithme opère comme suit :

1. Pour tous les *nets* ayant 2,3 où 4 connecteurs : calculer le gain du regroupement des cellules adjacentes à ce *net* en fonction de la réduction du nombre de connecteurs par ce regroupement. Ranger le *net* dans une liste de gains.

2. Parcourir la liste des gains en commençant avec le plus grand gain, puis regroupement des cellules adjacentes au *net* correspondant, si aucune d'entre elles n'est regroupée ailleurs.

3. Répétition des étapes 1) et 2).

4. Pour toutes les cellules non-fusionnées ayant au moins 4 *nets* en commun avec un *cluster* formé précédemment : fusion avec ce *cluster*, si sa cardinalité n'a pas dépassé un certain seuil. Cette étape est répétée une fois.

Par la première étape de l'algorithme nous garantissons de supprimer au moins un *net* par regroupement. La deuxième étape permet d'analyser et de classer des regroupements potentiels selon leur intérêt pour la diminution du nombre de connecteurs par *cluster*. Ces deux étapes sont répétées une fois pour augmenter la granularité. Elles ne sont pas répétées une troisième fois, puisque cela nous conduit à des *clusters* trop grands. En effet, le regroupement de quatre cellules qui peuvent être regroupées avec trois autres groupes de quatre cellules lors du deuxième regroupement mène déjà a un groupe contenant 16 cellules. Un troisième regroupement peut dans ce cas mener à un groupe composé de 64 cellules. Ceci réduit le nombre de solutions possibles sans pour autant apporter une amélioration pour l'algorithme de bipartitionnement.
La dernière étape sert à fusionner les cellules orphelines avec les *clusters* créés précédemment. Il est répété une fois pour permettre de fusionner des cellules qui deviennent fortement connexes avec des *clusters* après le regroupement des cellules orphelines lors de la première passe.

L'algorithme implémenté a une complexité de $O(|E|)$: la première étape parcourt tous les *nets* et deux fois toutes les cellules incidentes aux *nets* ayant moins de 5 connecteurs. Il est donc en $O(|E|)$. La deuxième étape parcourt encore une fois les mêmes *net*. Il est donc également en $O(|E|)$. La dernière étape parcourt toutes les cellules et analyse leurs connexités. Il est donc en $O(|V|)$. Puisque dans une *netlist* de CPBs, il y a plus de *nets* que de cellules, l'algorithme a une complexité en $O(|E|)$.

L'influence du *clustering* sur le résultat du bipartitionnement est présentée dans la section 4.3.2.

4.2.2 Bipartitionnement initial

La deuxième amélioration que nous avons apporté concerne le choix du bipartitionnement initial. Dans l'algorithme de base il est déterminé à partir de deux germes comme décrit dans section 4.1.1. Cette méthode mène rapidement vers un minimum local, comme nous avons observé dans la section 4.1.2.
Cette observation a été confirmée par Hauck et Borriello [Hau97]. Leur étude démontre que la meilleure solution d'un algorithme de voisinage est obtenue, si le bipartitionnement initial est déterminé aléatoirement. En nous appuyant sur cette étude, nous avons implémenté un algorithme déterminant un bipartitionnement initial d'une manière "aléatoire".

Le résultat d'une compilation dans le cadre de l'émulation doit être reproductible, c'est–à–dire deux compilations consécutives doivent aboutir au même résultat. Sinon, il est possible qu'une fois un *design* puisse être implémenté sur un certain nombre d'AVBs et qu'une autre fois ce même nombre d'AVBs soit insuffisant pour son implémentation. Il est donc impossible d'utiliser un bipartitionnement complètement aléatoire.

Une première approche consiste à parcourir la liste des cellules et à mettre toutes les cellules dans la première partition, tant que la moitié en terme de cardinalité n'est pas atteinte. Puisque le parseur du *design* remplit cette liste toujours dans le même ordre, il s'agit d'une approche qui assure la reproductibilité.
Par contre cette approche fournit déjà un bipartitionnement initial d'une certaine qualité: les cellules sont écrites dans le fichier lu par le parseur lors du *mapping* technologique qui traite les cellules dans un certain ordre.

Une deuxième approche consiste à parcourir la liste des cellules et à mettre une cellule sur deux dans la première partition et les autres cellules dans l'autre partition. Cette approche assure également la reproductibilité. Il peut par contre résulter en un bipartitionnement déséquilibré, si les cellules n'ont pas la même cardinalité.

L'utilisation de la première méthode apporte une amélioration par rapport à l'algorithme initial en terme de taux de remplissage et de temps de compilation. L'utilisation des deux approches consécutives en choisissant à chaque fois la meilleure des deux solutions mène à un temps de compilation important sans pour autant améliorer significativement le taux de remplissage.

Le détail des résultats expérimentaux ainsi que l'influence du bipartitionnement initial sur le résultat final est présentée dans la section 4.3.2.

4.2.3 Fonction de coût

Une autre amélioration apportée consiste à modifier la fonction de coût. Nous avons vu dans la section 4.1.2 que le critère du *ratio-cut* détermine quelquefois un bipartitionnement

très déséquilibré ce qui mène à des partitions de petite cardinalité et dans le cas d'un bipartitionnement récursif à un grand nombre de partitions. Ceci implique un nombre élevé d'applications de l'algorithme de bipartitionnement, et ainsi un temps de calcul élevé.
Une des possibilités pour obtenir un bipartitionnement équilibré consiste à fixer un seuil pour l'équilibre comme proposé par Fiduccia et Mattheyses [Fid82]. Cette stratégie empêche par contre l'identification d'un bipartitionnement naturel, comme nous avons remarqué dans section 4.1.2.
Pour éviter ces problèmes, nous avons implémenté et testé une autre fonction de coût.

Cette fonction de coût (fc) accentue davantage l'équilibre des deux partitions. Elle est basée sur le *ratio-cut* avec un dénominateur modifié : au lieu de multiplier la cardinalité des deux partitions, nous multiplions la racine carrée de la plus grande cardinalité des deux partitions avec le carré de la plus petite cardinalité.

$$fc = \begin{cases} \frac{c(X,\bar{X})}{\sqrt{X}\bar{X}^2} & \text{si } X > \bar{X} \\ \frac{c(X,\bar{X})}{\sqrt{\bar{X}}X^2} & \text{sinon} \end{cases}$$

Cette fonction de coût permet d'effectuer un bipartitionnement équilibré tout en conservant la qualité de la coupure. Lors du bipartitionnement récursif, le nombre de partitions résultant est donc plus petit que le nombre de partitions obtenu avec la fonction du *ratio-cut*. Ainsi, l'arbre de bipartitionnement est moins profond ce qui mène à un plus petit temps de calcul sans détérioration notable des valeurs de coupures obtenues. Les résultats expérimentaux confirmant ce type de comportement sont présentés dans la section 4.3.2.

4.3 Evaluation expérimentale

Les avantages et inconvénients de chacune des améliorations apportées ont été déterminées à l'aide d'un bipartitionnement récursif en ensembles de cellules respectant les contraintes de cardinalité et de coupure de FPGAs XC3064 (224 cellules, 120 I/O), de *clusters* dans l'AVC (quelques dizaines de cellules, et deux fois plus d'I/O), d'AVCs (entre 700 et 800 cellules et entre 100 et 200 I/O) et d'AVBs(40 à 50 cellules, plus de 1000 I/O).

4.3.1 Description des benchmarks

Comme *benchmark*, nous avons choisi 10 *netlists* industrielles et 10 *netlists* générées avec le générateur PartGen [Pis99a]. Les 10 *netlists* générées comportent 7 *netlists* de CPBs monotype et 3 *netlists* d'AVCs multi-type (peu de sommets avec beaucoup de connecteurs) [2].
Les 10 *netlists* industrielles se composent de 5 *netlists* de CPBs de type contrôleur et de 3 *netlists* d'AVCs d'un microprocesseur avec un nombre de connecteurs par *net* qui varie[3] de 2.55 à 10.45.
Les caractéristiques sont présentées en détail dans le tableau 4.1.

[2] A priori, cela n'a pas de sens de parler de type de *netlist* lorsqu'il s'agit d'une *netlist* d'AVCs.
[3] La modification du *fanout* moyen de la *netlist* entraîne également le changement de son type. D'ailleurs, le type n'est pas significatif lorsqu'il s'agit d'une *netlist* d'AVCs.

Circuit	type de circuit	#CPBs	#AVCs	#XMEM	# SRAM	#nets	#pins par net
glue5K	irrégulier	5000	—	—	—	5008	4.35
op5K	régulier	5327	—	—	—	5327	3.96
ctrl15K	contrôleur	13779	—	—	—	13779	3.61
glue15K	irrégulier	15000	—	—	—	15008	4.87
op15K	régulier	15000	—	—	—	15354	3.98
ctrl35K	contrôleur	35025	—	—	—	35025	3.64
ctrl50K	contrôleur	50478	—	—	—	50478	3.63
mix50K3	mixte	—	191	—	4	10390	2.76
mix50K4	mixte	—	181	—	4	9634	2.67
mix55K5	mixte	—	199	—	5	11013	2.79
industA	mixte	6756	—	16	16	7564	4.49
industB	data path	15075	—	—	—	15333	4.14
industC	contrôleur	16893	—	—	—	17101	4.20
industD	contrôleur	20780	—	—	—	21680	4.05
industE	contrôleur	22578	—	—	—	23807	4.23
industF	contrôleur	22716	—	—	—	31049	3.02
industG	contrôleur	48900	—	64	—	49820	4.54
industK	microprocesseur	—	2123	—	—	98914	2.55
industL	microprocesseur	—	2123	—	—	98914	5.45
industM	microprocesseur	—	2123	—	—	98914	10.45

Tableau 4.1: *Caractéristiques des netlists utilisées comme benchmark*

4.3.2 Présentation de la performance de l'algorithme bisect

L'influence des différentes modifications apportées sur la performance de l'algorithme *bisect* est présentée dans cette section. En particulier, nous avons comparé :

- la fonction de coût permettant un bipartitionnement équilibré comme décrit dans la section 4.2.3 (*balanced*),
- le bipartitionnement initial déterminé aléatoirement décrit dans la section 4.2.2 en combinaison avec le bipartitionnement aléatoire (*initAlea*),
- le bipartitionnement initial déterminé aléatoirement deux fois avec sélection de la meilleure solution comme décrit dans section 4.2.2 en combinaison avec le bipartitionnement équilibré (*2InitAlea*),
- le regroupement initial de cellules décrit dans section 4.2.1 avec le bipartitionnement équilibré (*initClustering*),
- le regroupement initial de cellules suivi du bipartitionnement initial aléatoire en combinaison avec le bipartitionnement équilibré (*initAleaClust*).

Les résultats expérimentaux obtenus pour ces modifications ont été comparés avec ceux obtenus pour l'algorithme de base (*bisect*), décrit dans section 4.1.1.
Les résultats expérimentaux sont regroupés en fonction des types de contraintes de partitionnement prises en compte (*cluster*, AVC, XC3064 et AVB). Pour chaque contrainte de partitionnement, nous présentons deux figures. La première figure contient l'influence de la modification de l'algorithme sur le nombre moyen de partitions nécessaires pour l'implémentation des *netlists*. La deuxième illustre l'influence des modification apportées sur le temps de calcul. Dans les deux cas, nous avons distingué les résultats obtenus à l'aide des *netlists* générées de ceux obtenus à l'aide des *netlists* industrielles.
Puisque nous avons d'abord modifié la fonction de coût et ensuite amélioré davantage l'algorithme en utilisant toujours cette même fonction de coût, nous basons notre comparaison sur les résultats expérimentaux obtenus avec celle-ci : les résultats obtenus pour les modifications appelées *initAlea*, *2InitAlea*, *initClustering* et *initAleaClust* sont normalisés par rapport aux résultats obtenus pour la modification de la fonction de coût, *balanced*.

Les résultats expérimentaux sont présentés dans les figures 4.2 à 4.9 (voir aussi les résultats expérimentaux en annexe).

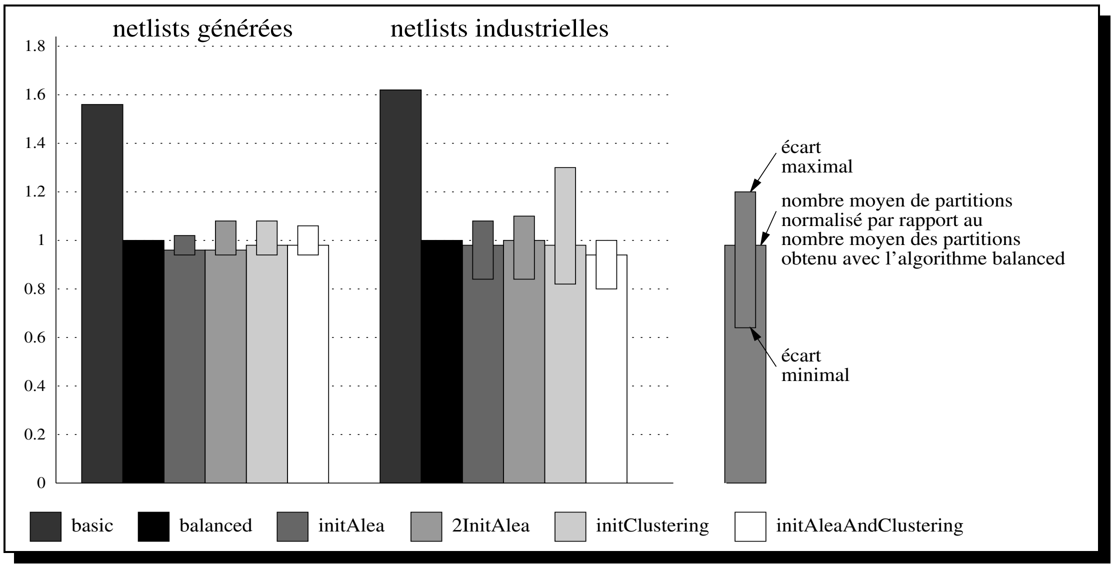


Figure 4.2: *Nombre moyen de clusters créés*

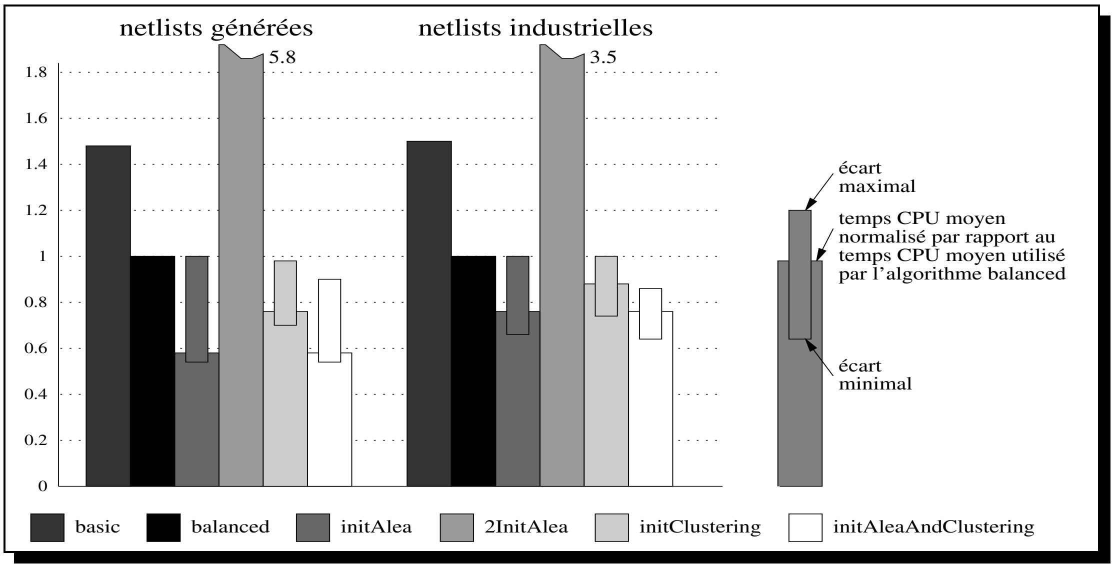


Figure 4.3: *Temps de calcul moyen pour le partitionnement en clusters*

Le nombre élevé de partitions obtenu pour l'algorithme de base est dû à l'utilisation du *ratio–cut* comme fonction de coût qui détermine souvent un bipartitionnement très déséquilibré menant à des partitions de petite cardinalité. Le changement de cette fonction de coût apporte donc immédiatement une amélioration en terme de nombre de partitions obtenu. Les performances de l'algorithme ont été améliorées davantage par l'introduction d'un regroupement initial des cellules ainsi qu'un bipartitionnement initial aléatoire.

Ces améliorations ont un faible impact au niveau du nombre de partitions nécessaires à l'implémentation d'une *netlist* par rapport à l'algorithme *balanced*. Cet impact varie selon

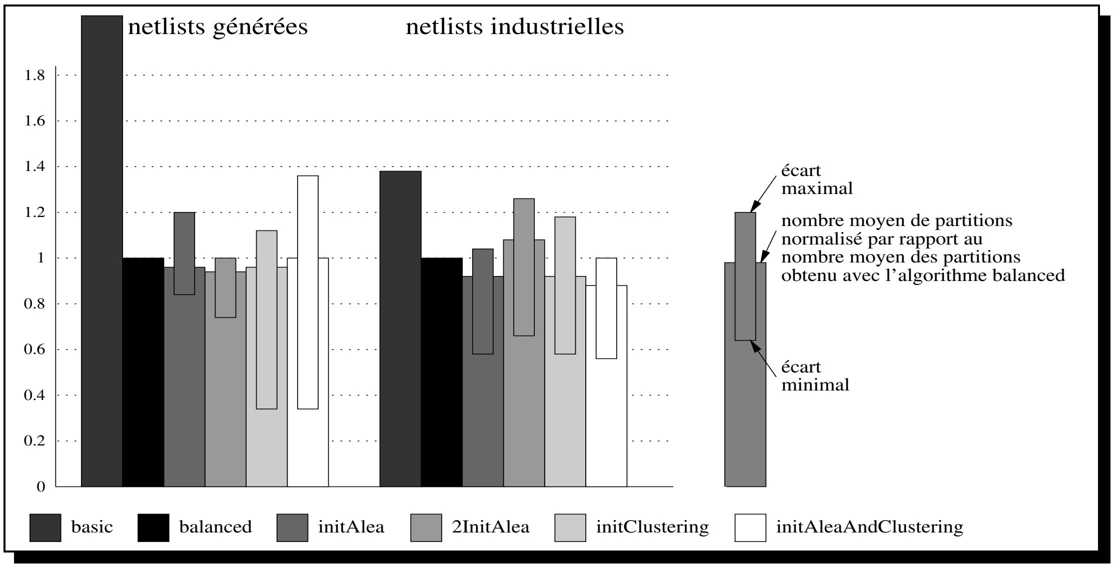


Figure 4.4: *Nombre moyen d'AVCs créés*

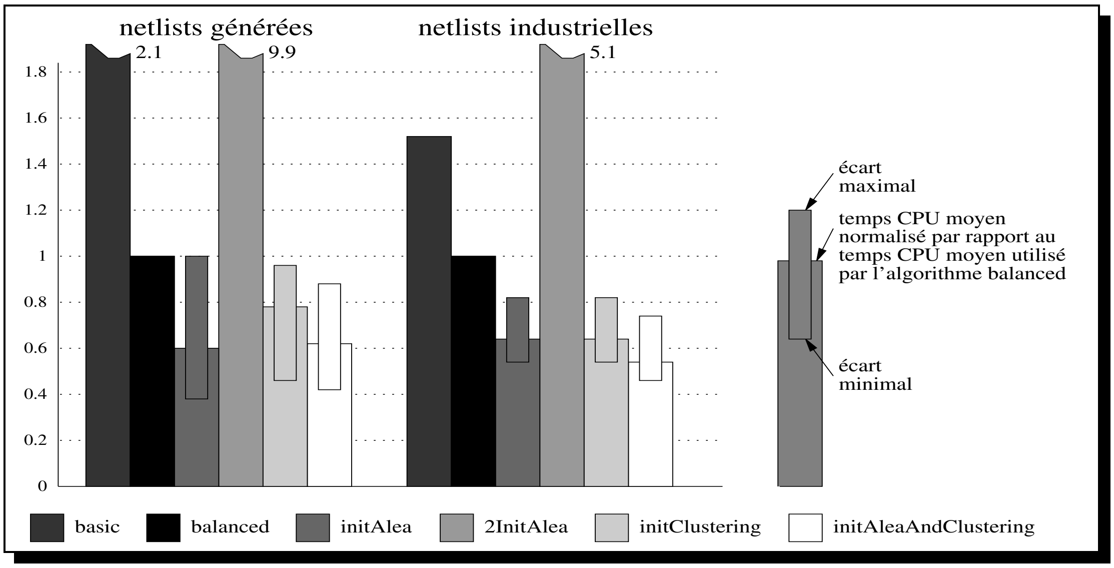


Figure 4.5: *Temps de calcul moyen pour le partitionnement en AVCs*

les contraintes de partitionnement entre 0 et 10% pour le partitionnement en *clusters*, AVCs et FPGAs XC3064. Il est quasiment inexistant pour le partitionnement en AVBs.
Néanmoins, les améliorations ont une grande importance pour le temps de calcul lors du partitionnement en *clusters*, AVCs et FPGAs XC3064 qui est réduit entre 10 et 50% pour *initAlea*, *initClustering* et *initAleaClust*. L'initialisation aléatoire répétée (*2InitAlea*) par contre nécessite un multiple de ce temps de calcul, ce qui la rend incompatible avec nos besoins.

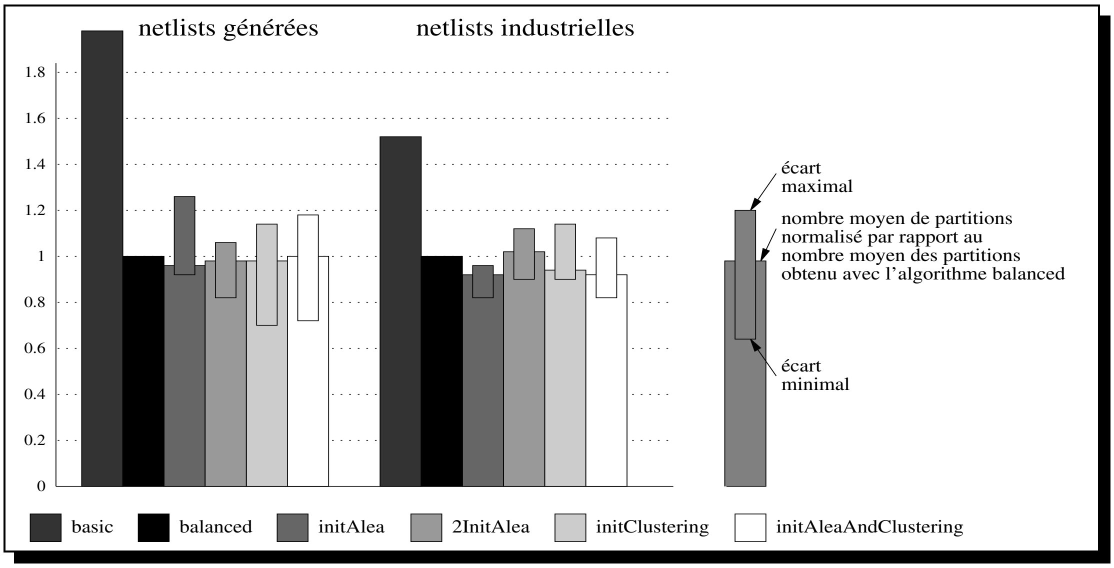


Figure 4.6: *Nombre moyen de FPGAs XC3064 créés*

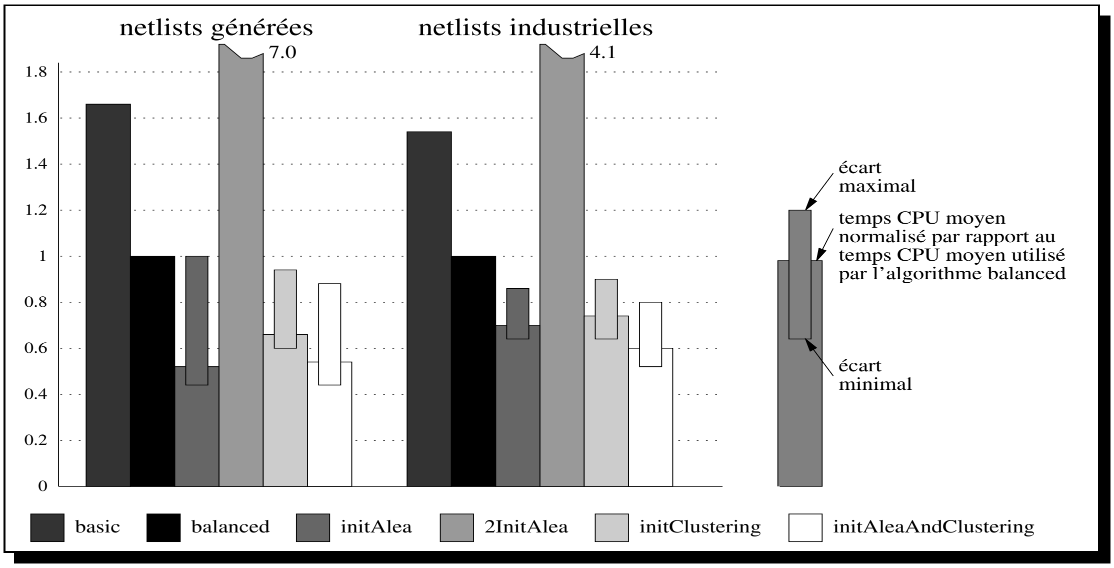


Figure 4.7: *Temps de calcul moyen pour le partitionnement en FPGAs XC3064*

4.4 Conclusions

Dans ce chapitre, nous avons étudié l'algorithme de base pour le bipartitionnement équilibré (bisect) en analysant ses avantages et inconvénients. Nous avons constaté que :

- la fonction de coût utilisée (*ratio-cut*) détermine fréquemment des partitions de cardinalité très déséquilibrée ce qui mène dans le cas du bipartitionnement récursif à un grand nombre de partitions,

- l'uniformité des *netlists* de CPBs contenant uniquement des cellules de quatre entrées et d'une sortie résulte en un grand nombre de cellules ayant le même gain lors de

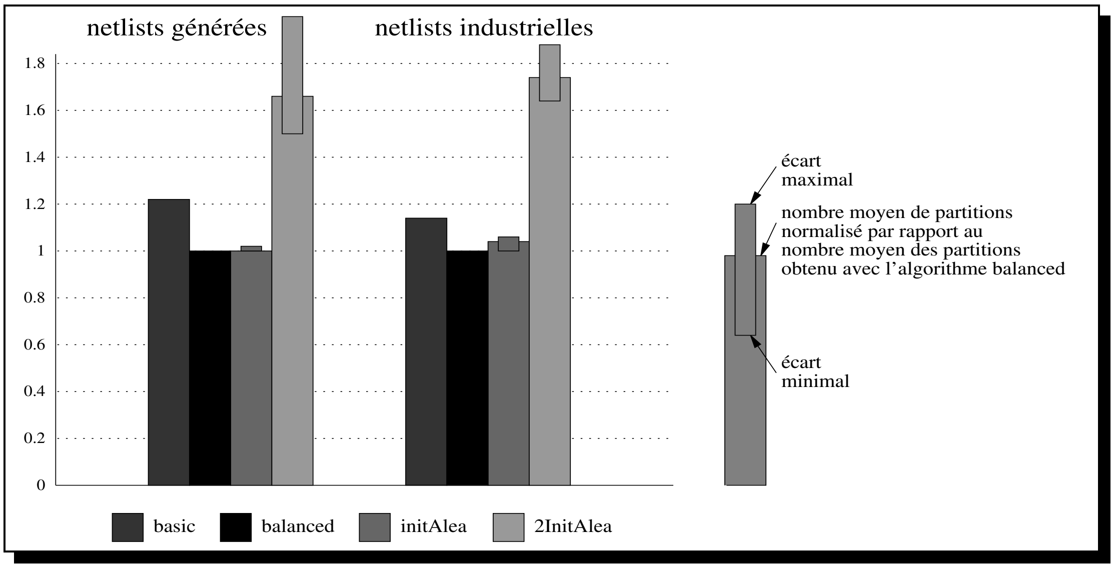


Figure 4.8: *Nombre moyen d'AVBs créés*

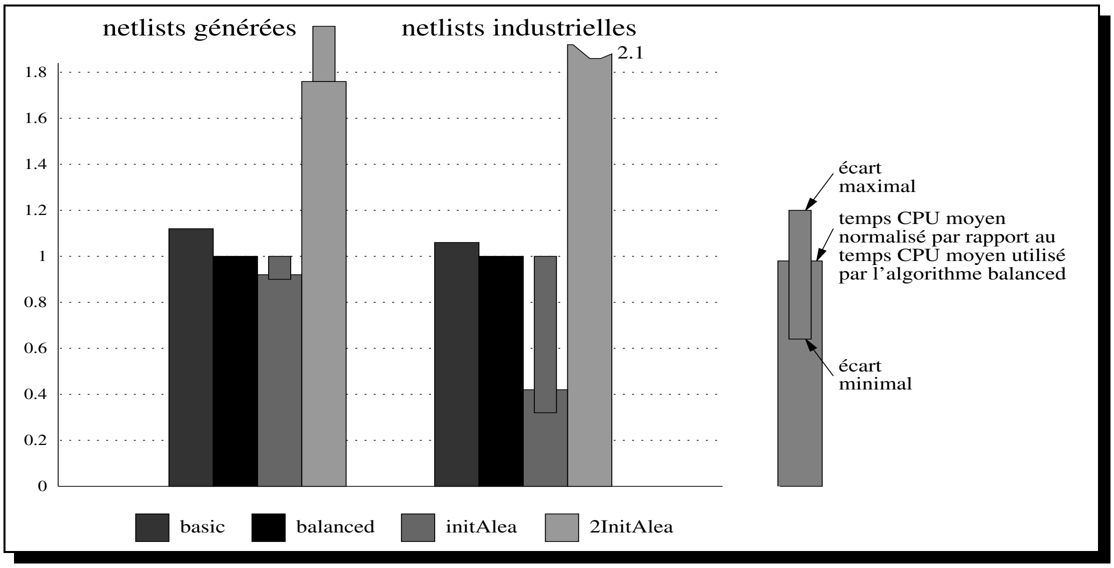


Figure 4.9: *Temps de calcul moyen pour le partitionnement en AVBs*

la sélection de la cellule à déplacer d'une partition dans l'autre. Ainsi la probabilité est très grande d'effectuer un mauvais choix et de déterminer un mauvais bipartitionnement.

- la façon de déterminer le bipartitionnement initial mène rapidement à un minimum local d'une qualité médiocre et ainsi à un grand nombre de partitions,

- l'objectif poursuivi par l'algorithme de base, c'est–à–dire déterminer un bipartitionnement qui minimise le nombre de *nets* coupés entre les deux partitions, ne correspond pas à l'objectif visé dans le cas du bipartitionnement récursif pour minimiser le nombre de partitions, c'est–à–dire déterminer un bipartitionnement qui minimise le nombre de connecteurs des deux partitions tout en équilibrant leur cardinalité et leur nombre de connecteurs.

Pour éviter ces inconvénients, nous avons proposé dans ce chapitre plusieurs améliorations de l'algorithme de base :

- Le regroupement de cellules en tant que prétraitement au bipartitionnement récursif permet de réduire le nombre de cellules à traiter et de diversifier le nombre de connecteurs sur les cellules, donc les gains des cellules.
- L'introduction d'un bipartitionnement initial aléatoire pour diminuer la probabilité d'être proche d'un minimum local.
- L'utilisation d'une autre fonction de coût permettant d'effectuer un bipartitionnement équilibré tout en conservant la qualité de la coupure.

Les meilleures performances en termes de nombre de partitions obtenu ainsi qu'en temps de CPU ont été déterminées pour la variante intégrant le regroupement initial suivi du bipartitionnement initial aléatoire en combinaison avec la fonction de coût permettant de déterminer les partitions de cardinalité équilibrées : le nombre de partitions a pu être réduit en moyenne jusqu'à 2% pour les *netlists* générées et de 5 à 12% pour les *netlists* industrielles. En même temps, le temps de CPU est réduit en moyenne de 39 à 47% pour les *netlists* générées et de 25 à 45% pour les *netlists* industrielles.

Cette version de l'algorithme, appelée bisect*, a été retenue pour les expériences suivantes.

Chapitre 5

Multipartitionnement par une approche de voisinage : algorithme multisect*

L'algorithme de multipartitionnement de base ("multisect") a été entièrement développé chez Mentor Graphics. Il appartient, comme l'algorithme présenté dans le chapitre précédent, à la famille des algorithmes de voisinage. L'analyse des performances de cet algorithme est le sujet de ce chapitre. L'étude de ses avantages et inconvénients nous a conduit à une version améliorée de l'algorithme appelée "multisect*" (cf. section 5.2).
Nos principales contributions sont la modification de la fonction de coût et de la fonction de calcul des gains des déplacements de cellules.

5.1 Algorithme de base : "multisect"

5.1.1 Description de l'algorithme de base

L'objectif de l'algorithme multisect est d'atteindre un certain taux de remplissage moyen des partitions, c'est–à–dire partitionner un ensemble de cellules en un certain nombre de partitions de telle sorte que chaque partition contienne en moyenne un certain nombre de cellules tout en respectant les contraintes de cardinalité maximale et de coupure.
Soit $\mathrm{cut}(P^i)$ la coupure de la partition i et cutMax la coupure maximale d'une partition. Le coût d'un partitionnement en k partitions est :

$$cost(P^k) = \sum_{i=1}^{k} \begin{cases} \mathrm{cut}(P^i) - \mathrm{cutMax} & \text{si } (\mathrm{cut}(P^i) > \mathrm{cutMax}) \\ 0 & \text{sinon} \end{cases}$$

Soit tdr le taux de remplissage moyen ciblé des partitions, U la cardinalité maximale de ces partitions, $|V| = n$ le nombre de cellules à partitionner et s leur poids moyen. L'algorithme effectue alors un multipartitionnement en trois phases :

1. Détermination du nombre minimal de partitions k.
   Le nombre minimal de partitions correspond à la somme des cardinalités des cellules divisée par la cardinalité maximale d'une partition : $k = \lceil s * n/U \rceil$.

2. Placement initial.
   Les cellules sont placées dans les k partitions en respectant la cardinalité maximale de chacune des partitions. Si une cellule ne peut plus être placée dans aucune des partitions, une nouvelle partition est créée $(k = k + 1)$. Les partitions valides à ce moment sont conservées. Les cellules faisant parties des partitions non-valides sont redistribuées sur les $(k = 1)$ partitions.
   Après la phase du placement initial, toutes les cellules ont été placées dans une partition et chacune de ces partitions respecte la contrainte de cardinalité maximale.

3. Itération – Déplacement de sommets :
   Le coût de la solution actuelle P^k, $\text{cost}(P^k)$ est une fonction du nombre de connecteurs de chaque partition. Cette solution ainsi que son coût sont sauvegardés : $P_s^k = P_{min}^k = P^k$ et $\text{cost}_s(P^k) = \text{cost}_{min}(P^k) = \text{cost}(P^k)$.
   Ensuite toutes les cellules sont déverrouillées. Puis, un gain en fonction du nombre de *nets* coupés est calculé pour chacune des cellules de la *netlist* et ceci pour chaque partition : supposons que la cellule v_j se trouve actuellement dans la partition i avec $1 \leq i \leq k$. Nous avons donc un gain qui reflète la coupure actuelle de la partition i ainsi que la modification de la coupure de cette partition dans le cas de l'enlèvement de la cellule v_j ($gainFrom(v_j)$) et $k-1$ gains qui reflètent la coupure actuelle des partitions m et la modification de la coupure de ces partitions m avec $\forall_{1 \leq m \leq k} m \neq i$ dans le cas du placement de v_j dans la partition m ($gainTo(v_j)$). Le gain du déplacement d'une cellule $gain(v_j)$ est la somme du gain du retrait de sa partition d'origine $\text{gain}_e(v_j)$ plus le plus grand gain engendré par l'ajout de ce sommet dans une partition $\text{gain}_p(v_j)$. Chacun des deux gains $\text{gain}_{ep}(v_j)$ se calcule en fonction du gain respectif ($\text{gainR}(v_j)$ représente soit $\text{gainTo}(v_j)$, soit $\text{gainFrom}(v_j)$) de la même manière :

$$\text{gain}_{ep}(v_j) = \begin{cases} 0 & \text{si } (\text{cut}_{ep} < \text{ cutMax}) \ \wedge \ (\text{cut}_{ep} - \text{ gainR}(v_j) < \text{ cutMax}) \\ \text{cut}_{ep} - \text{ cutMax} & \text{si } (\text{cut}_{ep} > \text{ cutMax}) \ \wedge \ (\text{cut}_{ep} - \text{ gainR}(v_j) < \text{ cutMax}) \\ 4 * \text{gainR}(v_j) & \text{sinon} \end{cases}$$

   Puis une par une, jusqu'à ce que toutes les cellules soient verrouillées, la cellule v_j avec le plus grand gain est sélectionnée et placée dans la partition pour laquelle le gain est maximal. A chaque déplacement d'une cellule, les gains des cellules non-verrouillées sont mis à jour.
   Ensuite, le coût de cette solution $\text{cost}(P^k)$ est calculé. Ce coût correspond à la somme des dépassements de la coupure maximale des différentes partitions. Dans le cas d'une amélioration $(\text{cost}(P^k) < \text{cost}_{min}(P^k)$, la solution est conservée $(P_{min}^k = P^k$, $\text{cost}_{min}(P^k) = \text{cost}(P^k))$.

4. Condition d'arrêt de l'itération :
   Si la meilleure solution déterminée lors de cette passe est meilleure que la solution de la passe précédente $(\text{cost}_{min}(P^k) < \text{cost}_s(P^k))$, $P^k = P_{min}^k$ et retourner en 3).
   Sinon, $P^k = P_s^k$ continuer avec 5).

5. Condition d'arrêt de l'algorithme :
   Si le taux de remplissage moyen des partitions est supérieur au taux de remplissage visé ($tdr_{real} > tdr$) et toutes les partitions sont valides[1] ou si $tdr_{real} \leq tdr$ et au moins une des partitions est valide, continuer avec 6).
   Sinon, ajouter une partition $(k \leftarrow k + 1)$ et continuer avec 3).

[1] Une partition est valide si elle respecte les contraintes de cardinalité maximale et de coupure et si elle est routable (cf. 1).

6. Fin de l'algorithme :
   Le multipartitionnement P^k est la solution finale. Des cellules se trouvant dans des partitions non-valides[2] sont traitées ultérieurement avec des cellules d'un autre ensemble non-partitionné.

Le calcul des gains d'enlèvement et de placement de chaque cellule a une complexité en $O(p|V|)$ avec p le nombre moyen de connecteurs par cellule dans la *netlist*. A chaque passe, toutes les cellules sont déplacées. La complexité d'une passe est donc en $O(p|V|^2)$.
Soit b le nombre de passes effectuées. Il est difficile de prédire la valeur de b. Généralement, le traitement s'arrête dans un minimum local après 5 à 10 passes. b est donc de l'ordre de 5 à 10. Théoriquement, l'algorithme peut effectuer toutes les combinaisons de placement des n cellules dans les k partitions disponibles ce qui mène à une complexité exponentielle. Supposons pour le calcul de la complexité que le nombre de passes est égal à b, sans oublier que b peut croître de façon exponentielle avec la taille de l'ensemble de cellules à partitionner. De plus, le nombre de partitions déterminé au début peut être insuffisant. Dans le cas le plus défavorable, le partitionnement initial permet de placer toutes les cellules dans le nombre minimal de partitions, mais le critère du point 5) n'est jamais valide. Nous ajoutons donc à chaque fois une partition tant que le taux de remplissage moyen ciblé n'est pas atteint. Au total, nous ajoutons alors $k' = k((1 - tdr)/tdr)$ partitions. La complexité de l'algorithme multisect est donc en $O(kbp|V|^2)$ où b peut être exponentiel en $|V|$.

5.1.2 Avantages et inconvénients

L'inconvénient principal de l'algorithme est justement cette complexité élevée. En effet, l'application de l'algorithme sur un grand nombre de cellules pour un partitionnement en un grand nombre de partitions, c'est–à–dire en partitions de faibles cardinalités par rapport à la cardinalité de l'ensemble de cellules, est prohibitif : la première passe d'un partitionnement de 15000 CPBs en partitions respectant des contraintes des FPGAs XC3064 (224 cellules, 120I/O) n'était pas terminée au bout de 200 heures de temps de calcul sur un serveur SUN Enterprise 5000 contenant 12 processeurs[3] Ultra Sparc I et 4 Go de RAM et uniquement 38% d'une *netlist* de 6724 CPBs ont été partitionnés en AVCs au bout de 230 heures.

Un autre problème a été constaté concernant la fonction de coût d'un multipartitionnement. Le coût est la somme des dépassements de coupures des différentes partitions. Ainsi le coût est minimal, c'est–à–dire zéro, si toutes les partitions sont valides. Par contre, une solution avec plusieurs partitions non-valides peut être meilleure qu'une solution avec une partition non-valide :

Exemple 5.1 Qualité d'un multipartitionnement :
Soient P_1^5 *et* P_2^5 *deux solutions d'un partitionnement en 5 sous-ensembles avec une coupure maximale permise de* 160. *Soient* 163, 163, 163, 160 *et* 160 *les coupures respectives des cinq partitions de* P_1^5 *et* 90, 90, 90, 90 *et* 170 *celles de* P_2^5. *Les coûts sont dans les deux cas :*

$$\begin{aligned} cost(P_1^5) &= 3+3+3=9 \\ cost(P_2^5) &= 10 \end{aligned}$$

La première solution avec trois partitions non-valides, mais un dépassement de la coupure

[2] Une partition est non-valide si elle ne respecte pas au moins une des trois contraintes de validité.
[3] Le process a été exécuté sur un des 12 processeurs.

de 9 est considérée comme étant meilleure que la deuxième solution avec une partition non-valide, mais un dépassement de la coupure de 10.

Dès lors que toutes les partitions sont valides, la fonction de coût ne fait pas de différence entre les deux solutions. Or dans le cas d'un partitionnement pour une architecture composée de plusieurs niveaux hiérarchiques, comme par exemple pour l'émulateur CELARO, il y a bien une différence de qualité : le nombre d'AVBs nécessaires pour réaliser une *netlist* d'AVCs dépend entre autre du nombre moyen de connecteurs par AVC qui dépend du partitionnement de CPBs en AVCs.

De plus, la fonction du gain utilisée pour sélectionner la cellule menant au meilleur déplacement choisit cette cellule selon des critères qui ne sont pas les mêmes que ceux qui déterminent la qualité du multipartitionnement.

L'avantage de cet algorithme est le bon remplissage des partitions. Appliqué sur de petits ensembles de cellules, il permet de déterminer un très bon multipartitionnement en terme de nombre de partitions nécessaires à l'implémentation d'une *netlist* et ceci dans un temps de calcul raisonnable.

5.2 Améliorations de l'algorithme de base

5.2.1 Fonctions de coût

L'amélioration principale que nous avons apportée à l'algorithme de base concerne la fonction de coût et la fonction de calcul des gains. Nous avons constaté dans la section 5.1.2 des problèmes au niveau de la fonction de coût déterminant la qualité d'une solution lors d'une passe. Cette fonction minimise la somme des dépassements de la coupure maximale des différentes partitions, ce qui n'est pas très efficace dans le cas d'un partitionnement sur plusieurs niveaux hiérarchiques. De plus, elle ne poursuit pas le même objectif que la fonction de gain utilisée pour déterminer le meilleur déplacement d'une cellule.

Nous avons donc modifié la fonction de coût et celle des gains pour améliorer le nombre moyen de connecteurs des partitions au-delà de la contrainte de coupure maximale tout en minimisant le nombre de partitions nécessaires pour implémenter une *netlist*. De plus, nous déterminons la meilleure cellule à déplacer selon une fonction de gain qui poursuit les mêmes objectifs que la fonction de coût. Nous avons implémenté et testé trois combinaisons différentes de fonctions de coût et de gain :

1. La première fonction de coût (cost1), pour déterminer la qualité d'une solution, est la même que celle utilisée dans l'algorithme de base. Elle a pour l'objectif de maximiser le nombre de partitions valides en terme de coupure. Ainsi, tout sous-ensemble non-valide, c'est–à–dire tout dépassement de la coupure maximale d'un sous-ensemble physique, contribue au coût, ce qui n'est pas le cas des partitions valides : Soit $\mathrm{cut}(P^i)$ la coupure de la partition i et cutMax la coupure maximale d'une partition. La fonction de coût d'un partitionnement en k partitions est :

$$cost1(P^k) = \sum_{i=1}^{k} \begin{cases} \mathrm{cut}(P^i) - \mathrm{cutMax} & \text{si } (\mathrm{cut}(P^i) > \mathrm{cutMax}) \\ 0 & \text{sinon} \end{cases}$$

La fonction de gain déterminant le meilleur déplacement a été modifiée afin de poursuivre le même objectif que la fonction de coût déterminant la qualité d'une solution. Ainsi, le gain du déplacement d'une cellule est égal à 0 si le sous-ensemble concerné par ce déplacement était valide avant le déplacement et s'il l'est toujours après. Le fait de ne pas tenir compte du gain au-delà de la coupure maximale d'un sous-ensemble physique mène en moyenne à des sous-ensembles avec des coupures proches de la coupure maximale. Soit v_i une cellule se trouvant actuellement dans la partition P^j, alors le gain pour le déplacement de cette cellule $\text{gain}(v_i)$ est la somme du gain de la suppression de la cellule de la partition j $\text{gain}_e(v_i)$ et du gain maximal du placement dans une autre partition $\text{gain}_p(v_i)$. Chacun des deux gains $\text{gain}_{ep}(v_i)$ se calcule en fonction du gain respectif ($\text{gainR}(v_i)$ représente soit $\text{gainTo}(v_i)$, soit $\text{gainFrom}(v_i)$) de la même manière :

$$\text{gain}_{ep}(v_i) = \begin{cases} 0 & \text{si } (\text{cut}_{ep} < \text{ cutMax}) \ \wedge \ (\text{cut}_{ep} - \text{ gainR}(v_i) < \text{ cutMax}) \\ \text{gainR}(v_i) & \text{si } (\text{cut}_{ep} > \text{ cutMax}) \ \wedge \ (\text{cut}_{ep} - \text{ gainR}(v_i) > \text{ cutMax}) \\ \text{cut}_{ep} - \text{ cutMax} & \text{sinon} \end{cases}$$

2. La deuxième combinaison fonction de coût - fonction de gain (cost2) utilise la même fonction de coût que celle utilisée précédemment (cost1). C'est la fonction de gain déterminant le meilleur déplacement qui a été changée. Nous favorisons toujours les déplacements de cellules qui rapprochent la coupure des différents sous-ensembles de la coupure maximale. Par contre, nous tenons aussi désormais compte des changements de coupures au-delà de la validité des sous-ensembles. Nous espérons par cette mesure pouvoir diminuer la coupure moyenne des sous-ensembles afin de faciliter le partitionnement sur les niveaux supérieurs de l'architecture hiérarchique.

$$\text{gain}_{ep}(v_i) = \begin{cases} \text{gainR}(v_i) & \text{si } (\text{cut}_{ep} < \text{ cutMax}) \ \wedge \ (\text{cut}_{ep} - \text{ gainR}(v_i) < \text{ cutMax}) \\ 4(\text{cut}_{ep} - \text{ cutMax }) & \text{si } (\text{cut}_{ep} > \text{ cutMax}) \ \wedge \ (\text{cut}_{ep} - \text{ gainR}(v_i) < \text{ cutMax}) \\ 4(\text{gainR}(v_i)) & \text{sinon} \end{cases}$$

3. La troisième fonction de coût (cost3), pour déterminer la qualité d'une solution, minimise la somme du nombre de connecteurs des partitions. Par cette mesure, nous visons à minimiser la coupure moyenne sans pour autant chercher à rendre les sous-ensembles valides.

$$f(P^k) = \sum_{i=1}^{k} \text{ cut}(P^i)$$

La fonction de gain déterminant le meilleur déplacement sélectionne dans ce cas la cellule avec le plus grand gain. Il s'agit de la cellule qui minimise la somme des coupures et ainsi la coupure moyenne des sous-ensembles. Ainsi, nous espérons améliorer davantage la coupure moyenne des sous-ensembles sans pour autant augmenter leur nombre.

$$\text{gain}_{ep}(v_i) = \text{gainR}(v_i)$$

5.3 Evaluation expérimentale

5.3.1 Description des benchmarks

Nous avons sélectionné sept *netlists* générées dont deux de type combinatoire irrégulière et cinq de type opérateur pour évaluer la performance de l'algorithme multisect. Nous avons également choisit trois *netlists* du *benchmark* Partitioning93 présenté dans section 2.5.1.1. Les caractéristiques des dix *netlists* sont présentées dans le tableau 5.1.

Circuit	type de circuit	#CPBs	#nets	#pins par net
glue500	irrégulier	507	507	4.07
op500	régulier	618	618	3.87
s5378	benchmark MCNC	738	738	4.14
glue750	irrégulier	757	757	4.12
op750	régulier	932	932	3.90
s9234	benchmark MCNC	1031	1031	4.27
op1000	régulier	1113	1113	3.91
op1250	régulier	1523	1523	3.93
s13207	benchmark MCNC	1623	1623	4.13
op1500	régulier	1752	1752	3.93

Tableau 5.1: *Caractéristiques des netlists utilisées comme benchmark*

5.3.2 Présentation de la performance de l'algorithme multisect

Les performances de l'algorithme multisect ont été examinées en terme de nombre de partitions nécessaires à l'implémentation de la *netlist* (#XC3064), en terme de nombre moyen de connecteurs utilisés par partition (#pins) et en terme de temps de calcul (t_CPU). L'application des différentes versions de l'algorithme multisect aux *netlists* sélectionnées fournit les résultats présenté dans figure 5.1. Ces résultats sont normalisés par rapport aux résultats obtenus pour l'algorithme de base (voir aussi les résultats expérimentaux en annexe).

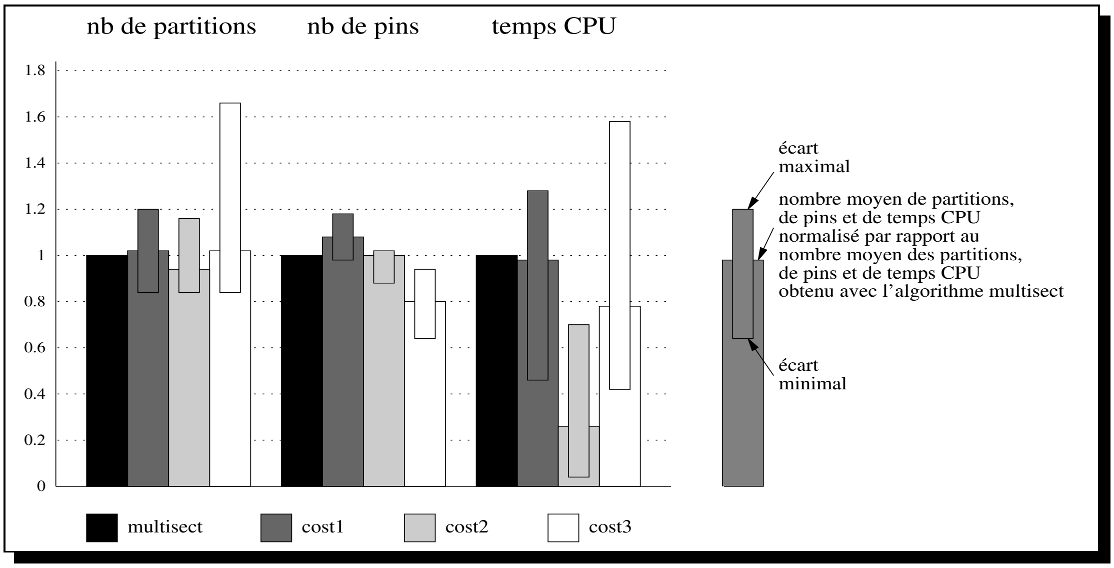


Figure 5.1: *Résultats du partitionnement en FPGAs XC3064*

Le résultat du partitionnement en terme de nombre de partitions nécessaires à l'implémentation des *netlists* est approximativement le même pour les trois fonctions de coût. Néanmoins, nous observons une grande amélioration au niveau du nombre moyen de connecteurs par partition et au niveau du temps de calcul : l'algorithme avec la fonction cost3 utilise en moyenne seulement 86 des 120 connecteurs disponibles soit 72% de la totalité, tandis que l'algorithme de base occupe en moyenne 108 connecteurs, soit 90% de la totalité des connecteurs disponibles. Simultanément, le temps de calcul diminue de 22%.
Le meilleur temps de calcul est réalisé par l'algorithme utilisant la combinaison cost2. Il divise par 4 le temps de calcul de l'algorithme de base. En même temps, le nombre moyen de partitions et de connecteurs est comparable entre ces deux versions.

Le nombre moyen de connecteurs par partition a une grande influence dans le cadre du partitionnement hiérarchique. Nous avons donc retenu l'algorithme avec la troisième combinaison fonction de coût – fonction de gain (cost3) pour effectuer un partitionnement *bottom-up*, c'est–̀-dire un partitionnement qui commence sur le plus bas niveau de l'architecture. Nous appelons cette version de l'algorithme multisect*.
Dans le cadre d'un partitionnement qui part du sommet de la hiérarchie, le nombre moyen de connecteurs par partition est sans importance à la condition qu'il n'y ait pas de contraintes pour le routage, ce qui a été supposé. Pour ce cas de figure, nous utilisons la combinaison cost2, puisqu'elle fournit un résultat comparable à celui de l'algorithme de base dans un temps plus court. Cette version de l'algorithme est appelée multisect**.

5.4 Conclusion

Le sujet de ce chapitre est l'étude de l'algorithme de base multisect. Nous avons constaté lors de cette étude que l'algorithme est très lent à cause d'une complexité temporelle élevée et que la fonction de coût utilisée ne permet pas de réduire le nombre moyen de connecteurs par partition au-delà de la contrainte de coupure maximale. Nous avons donc étudié l'utilisation d'autres fonctions de coût et d'autres fonctions pour calculer le gain.
Finalement, nous avons retenu deux combinaisons de fonction de coût – fonction de gain différentes :

- La première combinaison permet de réduire l'utilisation moyenne des connecteurs par partition de 90% à 72% dans un temps de calcul comparable tout en utilisant approximativement le même nombre de partitions. Ceci permet potentiellement de réduire le nombre de partitions sur le niveau hiérarchique supérieur dans le cadre d'un flot de partitionnement qui monte vers le sommet de la hiérarchie.

- Dans le cadre d'un partitionnement qui descend dans la hiérarchie à partir du sommet, il n'est pas nécessaire de minimiser le nombre moyen de connecteurs par partition. En effet, le partitionnement peut utiliser toutes les ressources disponibles sur le niveau hiérarchique inférieur, tant que ces ressources sont suffisantes. Pour cette application, nous avons donc retenu la combinaison nécessitant le moins de temps de calcul tout en conservant la qualité du multipartitionnement en terme de nombre de partitions créées.

Chapitre 6

Bipartitionnement par calcul du flot maximal dans un hypergraphe : algorithme hyperFBB

Dans ce chapitre nous analysons les performances d'un algorithme de bipartitionnement par calcul du flot maximal dans un hypergraphe. Nous nous basons sur un algorithme de base appelé "FBB" développée par Yang et Wong [Yan94] [Yan96]. L'étude des avantages et inconvénients de cet algorithmes nous a conduit à modifier l'algorithme de base menant à une version améliorée appelée "hyperFBB".
En particulier, nous avons apporté les améliorations suivantes :

- Proposition d'une autre méthode pour mieux choisir les sommets source et puits. Cet algorithme est présenté dans la section 6.2.1.
- Développement d'un algorithme permettant de déterminer le flot maximal directement sur l'hypergraphe, c'est–à–dire sans transformation d'hypergraphes en graphes. Cet algorithme a été soumis à une publication [Pis99b].

6.1 Algorithme de base : "FBB"

6.1.1 Description de l'algorithme de base

L'algorithme de base (FBB), présenté par Yang et Wong [Yan94][Yan96], est décrit dans section 2.3.2.5.

6.1.2 Avantages et inconvénients

L'avantage principal de cet algorithme est la détermination de la coupe minimale dans le graphe orienté séparant les sommets source et puits. Dans le cas du bipartitionnement d'un hypergraphe non-orienté et sous condition de l'application de la transformation d'hypergraphes en graphes proposée par Yang et Wong, la coupe minimale du graphe orienté correspond à la coupure minimale dans l'hypergraphe séparant les mêmes ensembles de sommets.

L'algorithme a aussi des inconvénients :

- Le choix des sommets source et puits est prépondérant pour la qualité du bipartitionnement. Yang et Wong effectuent un choix aléatoire. Cette méthode n'est pas envisageable pour le partitionnement en vue de l'application visé (l'émulateur CELARO). En effet, le résultat d'une compilation doit être reproductible. Il faut donc déterminer toujours les mêmes sommets initiaux.
  De plus, nous pensons obtenir des résultats meilleurs en terme de coupure par un choix plus judicieux. En effet, la qualité d'un bipartitionnement augmente avec la distance en terme de nombre de de *nets* sur le plus court chemin entre ces deux sommets.

- L'algorithme FBB a une complexité en $O(|E|^2)$[1]. Cette complexité est relativement élevée par rapport aux algorithmes de voisinages. Les temps de calcul risquent donc être importants pour de très grandes *netlists*.
  L'application de l'algorithme FBB sur un hypergraphe nécessite la transformation de cet hypergraphe en graphe orienté. La transformation proposée par Yang et Wong permet le calcul correct de la coupure. L'inconvénient de cette transformation est la création de sommets et d'arcs supplémentaires pour réaliser le graphe orienté nécessitant davantage de ressource machine en temps de calcul et en mémoire.

- Tant que le bipartitionnement correspondant à la coupe minimale obtenue n'est pas équilibré, Yang et Wong [Yan94][Yan96] proposent le procédé suivant : tous les sommets appartenant à la plus petite des deux partitions sont fusionnés en un sommet. Ensuite, un sommet voisin à celui-ci est sélectionné et également fusionné, puis le bipartitionnement est relancé. Le choix du sommet voisin se fait aléatoirement tant que l'hypergraphe est assez "gros". Sinon, tous les voisins sont examinés et celui qui minimise la coupure entre les deux partitions est sélectionné.
  Le seuil à partir duquel l'hypergraphe n'est plus considéré comme étant assez "gros" n'a pas été spécifié. Nous appliquons donc la recherche du sommet voisin qui minimise la coupure entre les deux partitions pour chaque fusion dans hyperFBB. Les résultats expérimentaux présentés dans section 6.3.2 montrent que l'augmentation du temps de calcul dû à cette modification est négligeable pour une cardinalité moyenne des *nets* inférieure à 5, ce qui est le cas pour les applications réelles.

- La signification du bipartitionnement équilibré n'a pas été détaillée. Une étude expérimentale présentée dans la section 6.3.2 montre que le bipartitionnement récursif utilisant l'algorithme hyperFBB fournit les meilleurs résultats en terme de nombre de partitions nécessaire pour implémenter une *netlist* quand ce seuil est fixé à environ 33%. En effet, un équilibre où les deux partitions doivent contenir chacune entre 45% - 55% des cellules de la *netlist* est trop restrictif. L'algorithme ne parvient pas à déterminer de coupures naturelles, il en résulte des coupures importantes. Un équilibre de 5% - 95% par contre n'est pas assez restrictif : à la fin du bipartitionnement récursif l'algorithme coupe en très petites partitions ce qui mène à un grand nombre de partitions presque vides. Le meilleur compromis est donc trouvé quand chacune des deux partitions contient entre 33% et 67% des cellules de la *netlist* à partitionner.

[1]Yang est Wong constatent dans [Yan94][Yan96] que leur algorithme a une complexité en $O(|V| \times |E|)$. Cette constatation est vraie, si $|V| < |E|$, ce qui n'est pas le cas dans nos applications. Pour le partitionnement d'AVCs en AVBs, nous avons même $|V| \ll |E|$.

6.2 Améliorations apportées : "hyperFBB"

Suite à l'analyse des inconvénients de l'algorithme FBB, nous proposons les améliorations suivantes présentées dans cette section :

- Proposition d'une méthode pour choisir les sommets source et puits aux extrémités du plus long court chemin dans l'hypergraphe.
- Développement d'un algorithme permettant de déterminer le flot maximal directement sur l'hypergraphe. Cet algorithme a été soumis à une publication [Pis99b].

6.2.1 Choix des sommets source et puits

L'algorithme choisissant les deux sommets source s et puits t détermine itérativement, à partir d'un sommet initial, le sommet le plus éloigné sur le plus court chemin.
Le choix du sommet initial est restreint aux sommets reliés à un connecteur externe. Si l'hypergraphe n'a pas de connexions vers l'extérieur, le sommet initial est le premier sommet dans la liste des sommets. S'il y a plusieurs sommets avec un connecteur externe, nous utilisons comme sommet initial le premier sommet dans la liste des sommets ayant un connecteur externe.

A partir du sommet initial le sommet le plus éloigné est cherché par une exploration en largeur. Ainsi nous assurons la détermination du sommet le plus éloigné sur le plus court chemin.
L'ensemble des *nets* utilisés pour cette exploration en largeur est de grande importance : si tous les *nets* sont utilisés, la distance entre les deux sommets est relativement courte (dans l'ordre de 4 ou 5) en raison des *nets* ayant un grand nombre de connecteurs et reliant ainsi un grand nombre de sommets. En utilisant uniquement les *nets* ayant au plus un certain nombre de connecteurs, la distance entre les deux sommets augmente. Pour différentes *netlists* industrielles, nous avons observé une augmentation continue de la distance jusqu'à atteindre un maximum pour un nombre de connecteurs d'environ 20. Si ce nombre baisse davantage, nous observons la création de parties déconnectées de la *netlists* menant à une distance plus courte. Pour cela, nous utilisons uniquement des *nets* ayant au plus 20 connecteurs pour déterminer le sommet le plus éloigné d'un sommet donné.

La complexité de l'algorithme implémenté pour déterminer les sommets source et puits est limitée par la complexité de l'algorithme d'exploration en largeur d'abord. Cet algorithme a une complexité en $O(|E|)$.

6.2.2 Marquage direct sur l'hypergraphe

6.2.2.1 Introduction

Dans toutes les approches proposées jusqu'ici ([Hu85] [Yan96] [Kuo97] [Liu98a], etc.) le calcul du flot maximal dans un hypergraphe se fait par transformation de cet hypergraphe en un graphe orienté, menant à une augmentation des besoins en ressources machine tant au niveau du temps de calcul qu'au niveau de l'espace mémoire occupé. Cette transformation devient inutile dans le cas du marquage des sommets directement sur l'hypergraphe.

Dans cette section, nous décrivons un algorithme pour déterminer le flot maximal directement sur l'hypergraphe. Cet algorithme utilise implicitement un graphe orienté obtenu par une transformation suivant le modèle introduit par Yang et Wong dans [Yan94][Yan96]. Par contre, le marquage des sommets et le partitionnement se font explicitement sur l'hypergraphe. Ainsi, nous parvenons à diminuer la complexité temporelle d'un facteur constant et la complexité en espace mémoire essentiellement d'un facteur $O(|V|)$ où $|V|$ est le nombre de sommets dans l'hypergraphe. Les résultats expérimentaux montrent une diminution du temps de calcul d'un facteur 0.53 en moyenne et une diminution de l'espace mémoire occupé d'un facteur 0.1 en moyenne.

6.2.2.2 L'algorithme de marquage

Soit $H = (V, E)$ un hypergraphe contenant $|V|$ sommets et $|E|$ hyperarêtes. Les sommets marqués sont organisés dans une structure FIFO (*first-in first-out*) afin de déterminer le chemin augmentant sur le plus court chemin. Si un tel chemin augmentant existe, les valeurs du flot sur les hyperarêtes au long de ce chemin sont mises à jour.

Chaque hyperarête $e_i \in E$ comporte deux marques indiquant les sommets v_j par lequel le flot entre dans l'hyperarête (flowFrom(e_i) $= v_j$) et le sommet v_k par lequel il sort de l'hyperarête (flowTo(e_i) $= v_k$). Etant donné que la capacité maximale des hyperarêtes est égale à 1, pour chaque hyperarête appartenant au support du flot il y a exactement un sommet par lequel le flot entre dans l'hyperarête et un autre par lequel il sort de l'hyperarête.

Chaque sommet $v_i \in V$ comporte trois marques. La première (label(v_i)) indique si le sommet a été marqué et mis dans la FIFO, la deuxième (scan(v_i)) montre si le sommet a été extrait de la FIFO et si tous ses voisins ont été marqués et la troisième (onEdge(v_i)) indique l'hyperarête par laquelle il a été marqué.

L'algorithme de marquage est composé de trois phases :

Marquage direct sur l'hypergraphe

1)	*Initialisation :*
	Pour toutes les hyperarêtes e_i faire :
	c(e_i) = 1, f(e_i) = 0, flowFrom(e_i) = NULL, flowTo(e_i) = NULL
	Pour tous les sommets v_i faire :
	label(v_i) = 0, scan(v_i) = 0, onEdge(v_i) = NULL
	Marquer sommet source s : label(s) = 1
	Mettre s dans la FIFO
2)	*Itération :*
	Tant qu'il y a de sommets dans la FIFO et sommet puits t est non-marqué (label(t) = 0) :
	Enlever v_i de la FIFO, scan(v_i) = 1
	Marquer tous les sommets voisins non-marqués et non traité v_k du sommet v_i (voir section 6.2.2.3)
	Mettre v_k dans la FIFO
3)	*Si label(t) != 0 alors :*
	Mettre à jour le flot sur l'hypergraphe (voir section 6.2.2.4)
	Retourner en 2)
	Sinon :
	Le flot est maximal.
	Fin de l'algorithme.

6.2.2.3 Marquage d'une hyperarête

Lors du marquage des sommets connectés à une hyperarête les différents cas pouvant se présenter sont indiqués dans la figure 6.1).

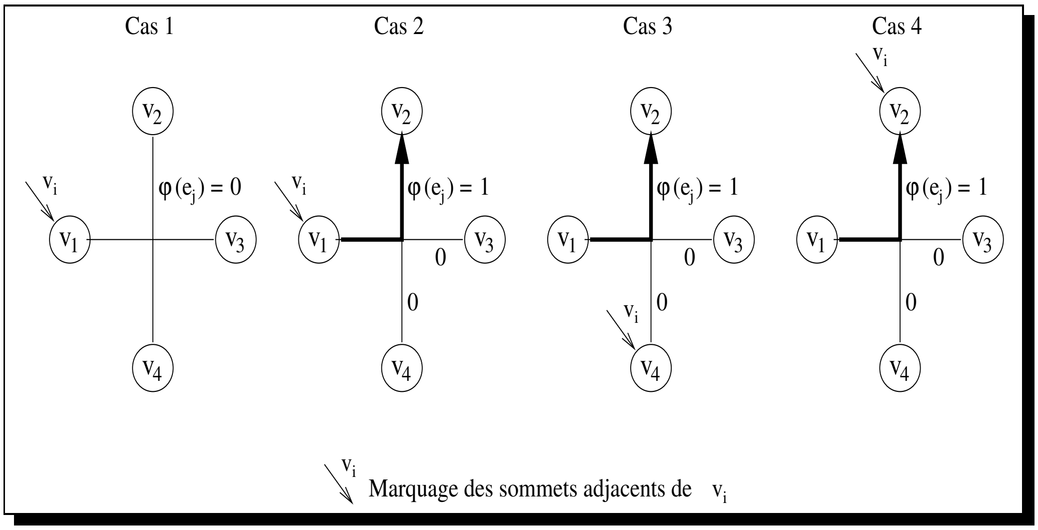


Figure 6.1: *Différents cas de figure du marquage d'une hyperarête*

Les règles de marquages sont les suivantes :
Marquage des hyperarêtes adjacentes à un sommet

Soit v_i le sommet qui vient d'être extrait de la FIFO.
Pour toutes les hyperarêtes e_j contenant sommet v_i ($v_i \in e_j$) faire :

- *Cas1 :* *S'il n'y a pas de flot sur l'hyperarête e_j ($f(e_j) = 0$) alors :*
  Pour tout $v_k \in e_j, v_k \neq v_i$,label(v_k) $= \infty$ faire :
  label(v_k) $= 1$
  fromVertex(v_k) $= v_i$
  onEdge(v_k) $= e_j$
- *Cas2 :* *Si $f(e_j) = 1$ et flowFrom(e_j) $= v_i$ alors :*
  Ne rien faire.
- *Cas3 :* *Si $f(e_j) = 1$ et flowFrom(e_j) $\neq v_i$ et flowTo(e_j) $\neq v_i$ alors :*
  Pour tout $v_k \in e_j, v_k =$ flowFrom(e_j)$, v_k \neq v_i$, label(v_k) $= \infty$ faire :
  label(v_k) $= 1$
  fromVertex(v_k) $= v_i$
  onEdge(v_k) $= e_j$
- *Cas4 :* *Si $f(e_j) = 1$ et flowFrom(e_j) $\neq v_i$ et flowTo(e_j) $= v_i$ alors :*
  Pour tout $v_k \in e_j, v_k \neq v_i$, label($v_k$) $= \infty$ faire :
  label(v_k) $= 1$
  fromVertex(v_k) $= v_i$
  onEdge(v_k) $= e_j$

6.2.2.4 Mis à jour du flot

L'algorithme de mise à jour du flot sur les arêtes s'effectue quand l'algorithme de marquage a atteint le sommet puits t. Les quatre cas possibles pour le changement des flots sur une hyperarête sont présentés dans la figure 6.2.

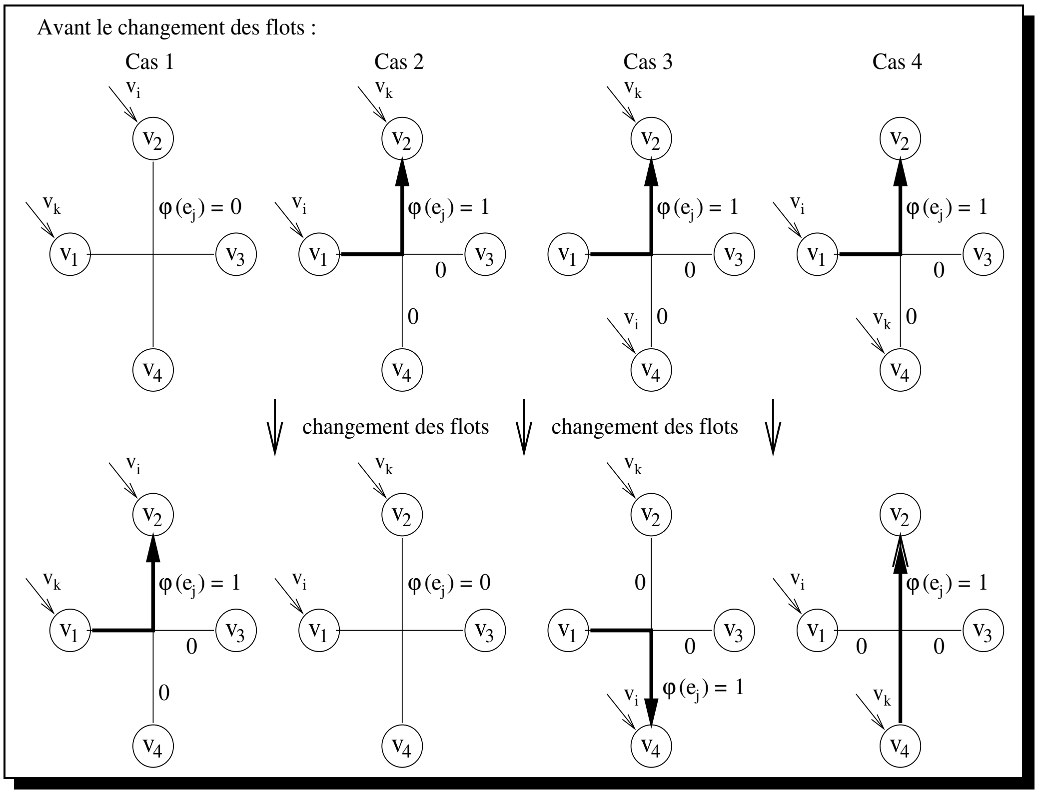


Figure 6.2: *Les quatre cas de changement des flots sur les hyperarêtes*

Soit v_i le sommet actuel, $e_j = \mathrm{onEdge}(v_i)$ l'hyperarête et $v_k = \mathrm{fromVertex}(v_i)$ le sommet à partir duquel v_i a été marqué.

Mis à jour du flot

- *Cas1 :* *Si* $f(e_j) = 0$ *alors :*
  - $f(e_j) = 1$
  - *flowFrom*$(e_j) = v_k$
  - *flowTo*$(e_j) = v_i$
- *Cas2 :* *Si* $f(e_j) = 1$, *flowFrom*$(e_j) = v_i$ *et flowTo*$(e_j) = v_k$ *alors :*
  - $f(e_j) = 0$
  - *flowTo*$(e_j) = NULL$
  - *flowTo*$(e_j) = NULL$
- *Cas3 :* *Si* $f(e_j) = 1$, *flowFrom*$(e_j) = v_d$ $(v_d \neq v_i, v_d \neq v_k)$, *et flowTo*$(e_j) = v_k$ *alors :*
  - *flowTo*$(e_j) = v_i$
- *Cas4 :* *Si* $f(e_j) = 1$, *flowTo*$(e_j) = v_d$ $(v_d \neq v_i, v_d \neq v_k)$ *et flowFrom*$(e_j) = v_i$ *alors :*
  - *flowFrom*$(e_j) = v_k$

6.2.2.5 Détermination de la coupure minimale

L'algorithme déterminant la coupure minimale utilise directement l'hypergraphe avec des flots sur les hyperarêtes obtenus lors du premier marquage qui n'atteint pas le sommet puits t : en partant du sommet source s, tous les sommets sur des chemins augmentants à partir de s sont marqués comme appartenant au même sous-ensemble de la partition que le sommet source.

6.2.2.6 Analyse de la complexité de l'algorithme

L'algorithme qui détermine le flot maximal entre les sommets source et puits cherche les plus courts chemins augmentants entre ces deux sommets. La recherche d'un tel chemin se fait par une exploration en largeur. Cet algorithme a une complexité temporelle de $O(|E|)$. L'exploration est effectuée tant qu'il y a des chemins augmentants entre les sommets source et puits. Dans un hypergraphe, il y a au plus $|E|$ chemins augmentant possibles. L'algorithme a donc une complexité temporelle de $O(|E|^2)$.
Celle de l'algorithme de Yang et Wong est de $O(|E|^2)$: dans [Yan96] Yang et Wong constatent une complexité de $O(|V||E|)$ basée sur l'hypothèse que $|E| < |V|$. Ceci est le cas si tous les connecteurs d'entrées/sorties sont modélisés par des sommets et si toutes les cellules pilotent au plus un *net*. Dans une *netlist* réelle, on peut plutôt supposer que chaque porte pilote au moins un *net*. Or, $|V| \leq |E|$. Dans le cas d'une *netlist* d'AVCs ou d'AVBs, nous avons même $|V| \ll |E|$. Nous constatons donc que la complexité temporelle de leur algorithme est $O(|E|^2)$.

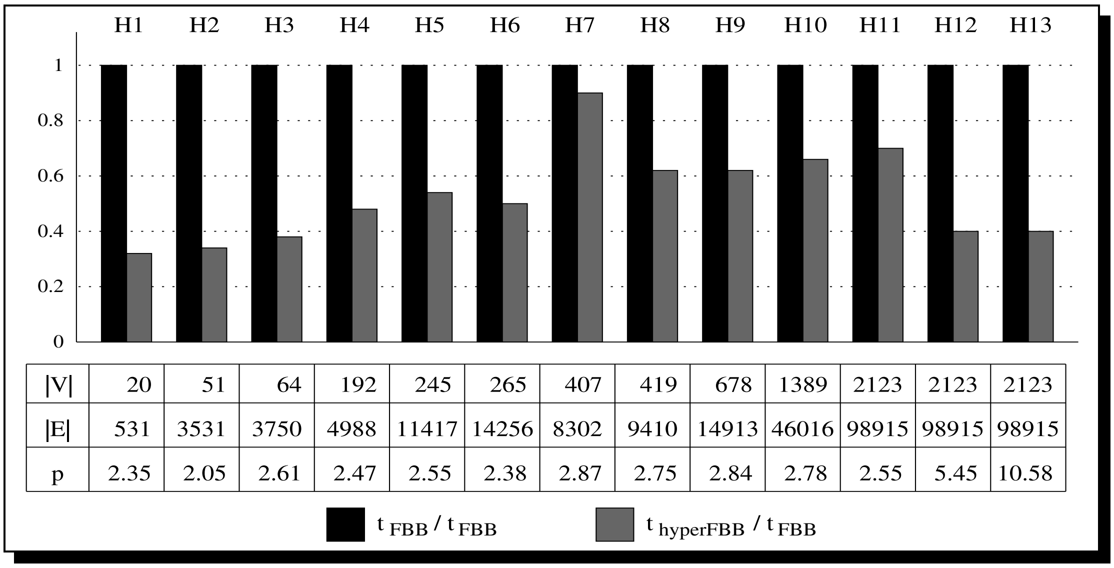
La complexité en espace mémoire occupé de notre algorithme de marquage correspond essentiellement à l'espace mémoire nécessaire pour contenir l'hypergraphe. Puisque l'hypergraphe est composé de V sommets et E hyperarêtes, nous avons donc une complexité en $O(|V| + |E|) = O(|E|)$.
L'algorithme de Yang et Wong [Yan96] nécessite de l'espace mémoire supplémentaire pour contenir le graphe orienté $G' = (V', E')$ composé de $|V'| = |V| + 2|E|$ sommets et de $|E'| = (2p+1)|E|$ arcs où p correspond à la cardinalité moyenne des hyperarêtes. Dans le pire cas, chaque hyperarête est connectée à $p = |V|$ sommets ce qui mène à une occupation de l'espace mémoire de $O(|V| + 2|E| + 2|V||E| + |E|) = O(|V||E|)$.
Notre algorithme améliore donc la complexité en espace mémoire occupé d'un facteur $O(|V|)$.

6.3 Evaluation expérimentale

6.3.1 Comparaison des performances de FBB et de hyperFBB

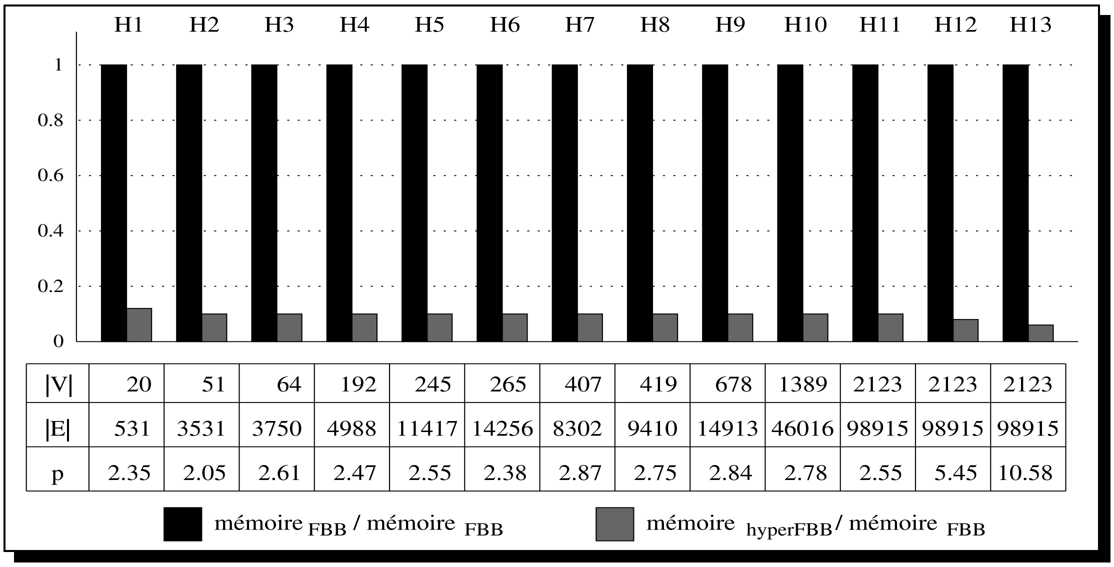
L'influence du marquage directement sur l'hypergraphe (hyperFBB) par rapport au marquage sur le graphe orienté obtenu à l'aide de la transformation décrit dans [Yan94][Yan96] (FBB) a été étudiée à l'aide des hypergraphes H1 à H11 modélisant des *netlists* industrielles et des *netlists* générées avec PartGen et des hypergraphes H12 et H13 obtenus par une modification de l'hypergraphe H11 visant à augmenter la cardinalité moyenne des *nets*. Ces hypergraphes contiennent $|V|$ sommets et $|E|$ hyperarêtes ayant en moyenne une cardinalité de p.
Nous avons comparé les performances des deux approches en terme de temps de calcul (t) et d'espace mémoire occupé (m).
Dans nos expériences, les deux algorithmes utilisent les mêmes sommets source et puits et trouvent les mêmes plus courts chemins augmentants dans l'hypergraphe ainsi que la même coupure minimale. Les résultats expérimentaux pour le temps de calcul sont présentés dans la figure 6.3, ceux pour l'espace mémoire occupé dans la figure 6.4 (voir aussi les résultats expérimentaux en annexe).

Ces résultats montrent que hyperFBB est meilleur que FBB en terme de temps de calcul et



|V|	20	51	64	192	245	265	407	419	678	1389	2123	2123	2123
|E|	531	3531	3750	4988	11417	14256	8302	9410	14913	46016	98915	98915	98915
p	2.35	2.05	2.61	2.47	2.55	2.38	2.87	2.75	2.84	2.78	2.55	5.45	10.58

Figure 6.3: *Comparaison des performances entre hyperFBB et FBB pour le temps de calcul*



|V|	20	51	64	192	245	265	407	419	678	1389	2123	2123	2123
|E|	531	3531	3750	4988	11417	14256	8302	9410	14913	46016	98915	98915	98915
p	2.35	2.05	2.61	2.47	2.55	2.38	2.87	2.75	2.84	2.78	2.55	5.45	10.58

Figure 6.4: *Comparaison des performances entre hyperFBB et FBB pour l'espace mémoire occupé*

d'espace mémoire occupé. En moyenne, hyperFBB utilise 53% du temps de calcul de FBB et le pourcentage dépasse 70% sur un exemple.
Concernant l'espace mémoire occupé, hyperFBB utilise seulement 1/10 de celui utilisé par FBB.

6.3.2 Performances de différentes versions de l'algorithme de partitionnement utilisant hyperFBB

Nous avons sélectionné 9 *netlists* industrielles et 7 *netlists* générées pour évaluer les performances des différentes versions de l'algorithme hyperFBB. Ces *netlists* sont présentées dans le tableau 6.1.

Circuit	type de circuit	#CPBs	#AVCs	#XMEM	# SRAM	#nets	#pins par net
glue5K	irrégulier	5000	—	—	—	5008	4.35
op5K	régulier	5327	—	—	—	5327	3.96
ctrl15K	contrôleur	13779	—	—	—	13779	3.61
glue15K	irrégulier	15000	—	—	—	15008	4.87
op15K	régulier	15000	—	—	—	15354	3.98
ctrl35K	contrôleur	35025	—	—	—	35025	3.64
ctrl50K	contrôleur	50478	—	—	—	50478	3.63
industB	data path	15075	—	—	—	15333	4.14
industC	contrôleur	16893	—	—	—	17101	4.20
industD	contrôleur	20780	—	—	—	21680	4.05
industE	contrôleur	22578	—	—	—	23807	4.23
industF	contrôleur	22716	—	—	—	31049	3.02
industG	contrôleur	48900	—	64	—	49820	4.54
industK	microprocesseur	—	2123	—	—	98914	2.55
industL	microprocesseur	—	2123	—	—	98914	5.45
industM	microprocesseur	—	2123	—	—	98914	10.45

Tableau 6.1: *Caractéristiques des netlists utilisées comme benchmark*

L'évaluation des performances des différentes améliorations apportées à l'algorithme hyperFBB est présentée dans cette section. En particulier, nous avons testé l'influence de la fusion du sommet voisin minimisant la coupure entre les deux partitions avec différentes contraintes d'équilibre par rapport à l'algorithme de base (hyperFBB avec une contrainte d'équilibre de 49% - 51%). Les différentes contraintes d'équilibre sont 20% - 80% (*min-cut 20-80*), 33% - 67% (*min-cut 33-67*) et 45% - 55% (*min-cut 45-55*). C'est–à–dire la plus petite des deux partitions ne doit pas contenir moins de 20% (resp. 33%, 45%) du nombre total de cellules et la plus grande des deux partitions pas plus de 80% (resp. 67%, 55%).
L'algorithme hyperFBB a été implémenté avec une contrainte d'équilibre de 45% - 55%. Il détermine les sommets source et puits comme décrit dans la section 6.2.1, effectue le marquage directement sur l'hypergraphe comme décrit dans la section 6.2.2 et sélectionne aléatoirement le sommet voisin à fusionner avec la plus petite des deux partitions.
Les résultats expérimentaux sont, comme pour les résultats de l'algorithme bisect, regroupés en fonction des types de contraintes de partitionnement (*cluster*, AVC, XC3064 et AVB). Pour chaque contrainte nous présentons une figure pour montrer l'amélioration moyenne en terme de nombre de partitions nécessaires pour l'implémentation de la *netlist* en distinguant les résultats obtenus pour les *netlists* générées de ceux obtenus pour les *netlists* industrielles. Les résultats obtenus pour les différentes améliorations apportées sont normalisés par rapport aux résultats de hyperFBB avec une contrainte d'équilibre de 49% - 51%. Ils sont présentés dans les figures 6.5 à 6.8 (voir aussi les résultats expérimentaux en annexe C.3.1).

Lors du bipartitionnement minimisant la coupure entre les deux partitions, nous avons souvent observé le phénomène suivant : le premier bipartitionnement sépare un sommet, par exemple le sommet source, de tous les autres sommets. L'algorithme effectue alors une fusion du sommet source avec un sommet voisin puis recommence le bipartitionnement. Le résultat de ce deuxième bipartitionnement est la séparation du sommet puits de tous les autres sommets. L'algorithme effectue alors une fusion du sommet puits avec un sommet voisin puis recommence le bipartitionnement. Ainsi la plus petite des deux partitions croît au fur et à mesure jusqu'à ce que le bipartitionnement parvienne à déterminer la coupure qui mène à des partitions respectant les contraintes d'équilibre.

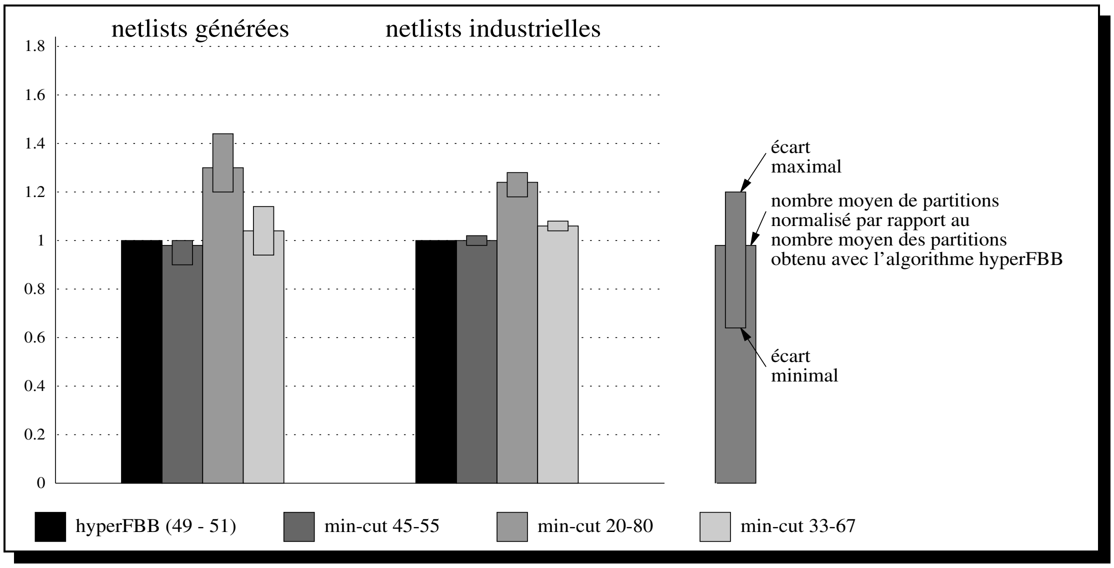


Figure 6.5: *Nombre moyen de clusters créés*

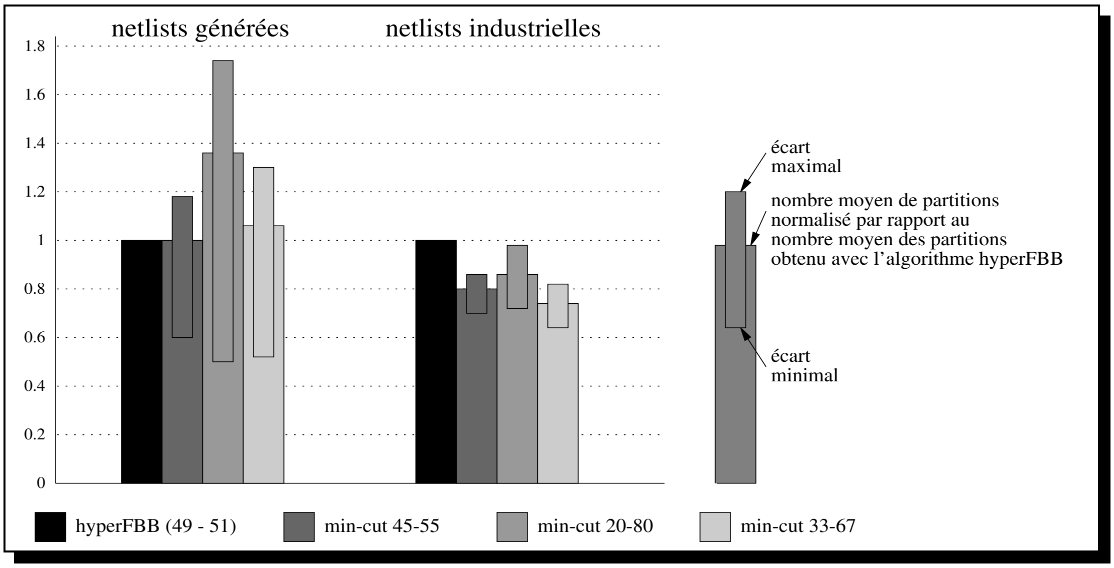


Figure 6.6: *Nombre moyen d'AVCs créés*

Ce phénomène est une des raisons pour laquelle le temps de calcul que nous observons pour ce type d'algorithmes en général est relativement élevé. C'est aussi une raison pour laquelle les versions de l'algorithme hyperFBB déterminant le sommet voisin qui minimise la coupure entre les deux partitions lors de la fusion nécessite parfois un temps de calcul plus élevé que l'algorithme de base. Ce phénomène se manifeste surtout pour le partitionnement en AVBs, où le temps CPU pour le partitionnement des *netlists* croît avec le nombre moyen de *pins* sur les *nets*.
Une autre conséquence de ce phénomène est que le temps de calcul moyen est inférieur pour la contrainte d'équilibre de 20% - 80%. En effet, chaque bipartitionnement d'une suite de bipartitionnements récursifs se termine rapidement grâce à un nombre inférieur de fusions de sommets nécessaires pour satisfaire les contraintes d'équilibre. Cette version de

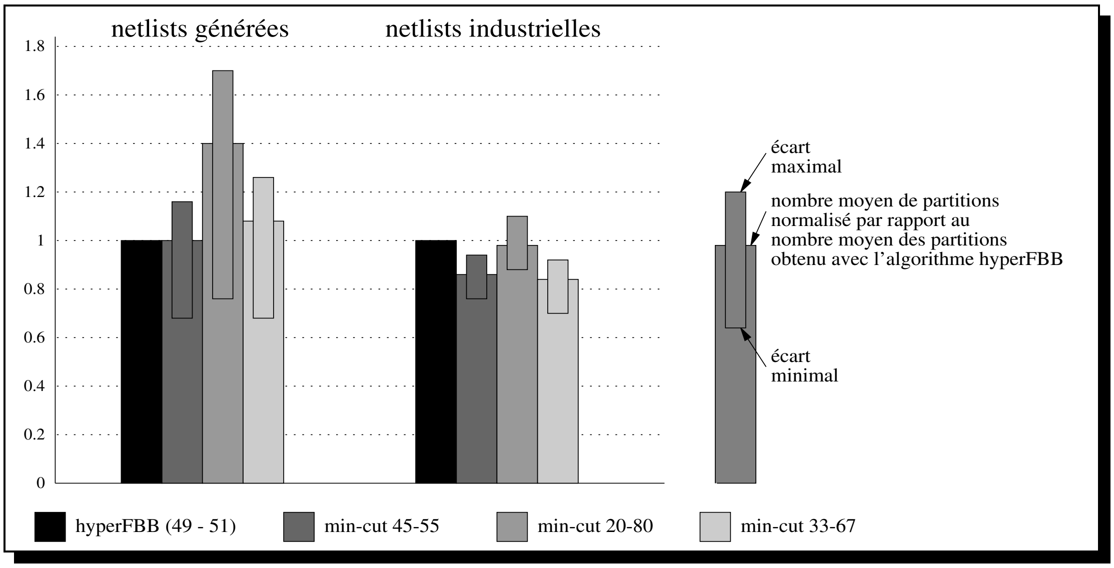


Figure 6.7: *Nombre moyen de FPGAs XC3064 créés*

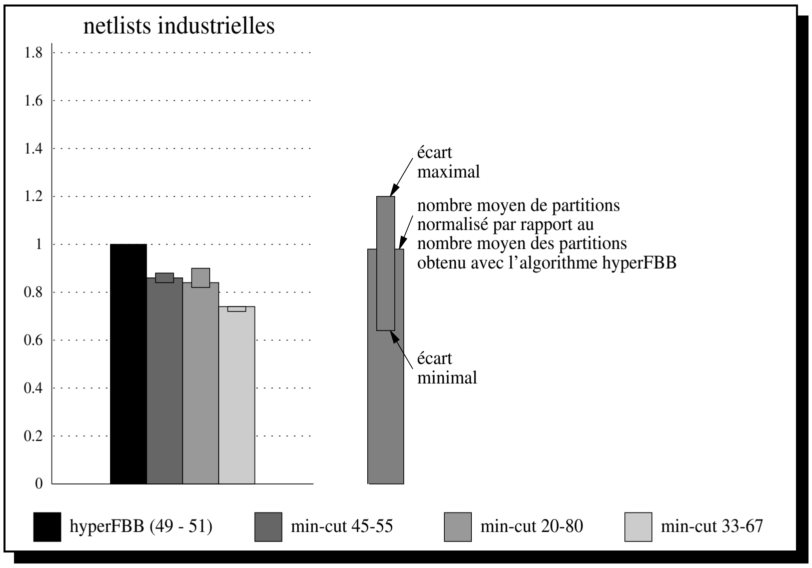


Figure 6.8: *Nombre moyen d'AVBs créés*

l'algorithme effectue par contre plus de bipartitionnement ce qui mène à la création d'un nombre de partitions plus élevé : pour une coupure à peine plus grande, nous aurions peut être trouvé un bipartitionnement beaucoup plus équilibré.

Un équilibre où les deux partitions doivent contenir chacune entre 45% - 55% des sommets de la *netlist* est trop restrictif. L'algorithme ne parvient pas à déterminer des coupures naturelles. Les bons résultats pour cette version de l'algorithme dans le cas du partitionnement en *clusters* s'expliquent du fait du nombre important de connecteurs d'entrées/sorties par rapport au nombre de CPBs (deux fois plus) du *cluster*. Le fait de couper la *netlist* presque au milieu mène à des partitions ayant approximativement la même cardinalité. Le grand nombre d'entrées/sorties du *cluster* masque le fait que ces partitions ont une coupure relativement importante par rapport aux autres versions de hyperFBB. Les partitions créées sont donc toutes valides, dès que la contrainte de cardinalité est respectée.

Le meilleur compromis menant au meilleur résultat est donc trouvé quand les deux partitions contiennent entre 33% et 67% des sommets de la *netlist* à partitionner.

Une autre observation concerne la grande dispersion des valeurs maximales et minimales par rapport à la moyenne pour le partitionnement des *netlists* générées. En effet, la valeur minimale a été observée pour les *netlists* d'opérateurs tandis que la valeur maximale est constatée pour les *netlists* de contrôleurs. Néanmoins, une certaine dépendance semble exister entre la performance d'un algorithme particulier et la topologie de la *netlist* traité. Une étude plus approfondie de cette dépendance est un sujet intéressant pour des recherches futures.

6.4 Conclusions

Dans ce chapitre, nous avons étudié FBB, un algorithme de bipartitionnement utilisant le théorème du *max-flow min-cut* [For62] pour calculer une coupure minimale séparant les deux sommets source et puits. Lors de cette étude, nous avons constaté les inconvénients suivantes :

- Les sommets source et puits sont déterminés aléatoirement. Cette méthode n'est pas envisageable pour le partitionnement en vue de l'application visée (l'émulation). En effet, puisque le résultat d'une compilation doit être reproductible, nous devons toujours déterminer les mêmes sommets source et puits.
- L'application de l'algorithme FBB sur un hypergraphe nécessite la transformation de ce hypergraphe en un graphe orienté. La transformation proposée par Yang et Wong crée des sommets et des arcs supplémentaires pour réaliser cette transformation nécessitant davantage de ressources machine en temps de calcul et en mémoire.

Pour éviter ces inconvénients, nous avons proposé dans ce chapitre les améliorations suivantes :

- Les deux sommets source et puits sont déterminé d'une manière constructive, de telles sorte qu'ils soient aux extrémités du plus long court chemin dans la *netlist*. Cette modification nous permet d'améliorer les résultats du bipartitionnement tout en garantissant leur reproductibilité.
- Développement d'un algorithme permettant le marquage direct sur l'hypergraphe lors du calcul du flot maximal. Cet algorithme permet de réduire le temps de calcul en moyenne à 53% du temps utilisé par l'algorithme de base. En même temps, l'espace mémoire occupé a pu être divisé par un facteur 10.

L'utilisation de cet algorithme dans le cadre de l'émulation est étudiée dans chapitre 8.

Chapitre 7

Multipartitionnement par découpage d'un ordonnancement de cellules : algorithme MDOC

Dans ce chapitre, nous étudions un algorithme de multipartitionnement par découpage d'un ordonnancement. Cet algorithme, proposé par Alpert et Kahng [Alp96], est constitué de deux parties : la première partie de construit un ordonnancement des sommets de l'hypergraphe modélisant la *netlist*. La deuxième partie découpe cet ordonnancement en un nombre minimal de partitions respectant les contraintes de cardinalité et de coupure maximales.
Afin de pouvoir utiliser l'algorithme dans le cadre du partitionnement d'une *netlist* pour l'implémentation sur une architecture hiérarchique, nous avons effectué les modifications et analyses suivantes présentées dans la section 7.4 :

- L'implémentation d'une heuristique permettant de déterminer le premier sommet de l'ordonnancement d'une manière constructive sur les limites de la *netlist*.
- L'analyse de l'influence de la fonction d'attraction sur la performance du partitionnement.
- L'influence de la taille de la fenêtre choisie lors de la première partie de l'algorithme sur le résultat du partitionnement.

7.1 Algorithme d'ordonnancement : "window"

7.1.1 Description de l'algorithme

L'algorithme d'ordonnancement est basé sur l'algorithme "window" présenté par Alpert et Kahng dans [Alp96]. Il s'agit d'un algorithme qui détermine un ordonnancement par une exploration locale.
Soit $H = (V, E)$ un hypergraphe contenant $|V| = n$ sommets et $|E|$ hyperarêtes modélisant une *netlist*. Soit $V = \{v_1, v_2, \ldots, v_n\}$ l'ensemble des sommets de cet hypergraphe. Un ordonnancement $v_{\pi(1)}, v_{\pi(2)}, \ldots, v_{\pi(n)}$ de ces sommets est obtenu par la bijection $\pi(1, \ldots, n) \rightarrow$

$(1,\ldots,n)$. Le sommet v_j a un rang k si $\pi(k) = j$. Ainsi, $v_{\pi(1)}$ est le premier sommet ordonné, $v_{\pi(2)}$ le deuxième, etc. L'objectif est de ranger les sommets fortement interconnectés dans des positions voisines de l'ordonnancement afin qu'un découpage en sous-ensembles de sommets consécutifs forme un partitionnement de la *netlist* avec un nombre minimal de partitions, comme par exemple

$$\{v_{\pi(1)},\ldots,v_{\pi(i-k-1)}\},\ldots,\{v_{\pi(i-k)},\ldots,v_{\pi(i-1)}\},\{v_{\pi(i)},v_{\pi(i+1)},\ldots,v_{\pi(i+j)}\},\ldots$$

Soit S l'ensemble des sommets ordonnés. L'ordonnancement est construit de manière itérative. En commençant par un ensemble vide $S = \emptyset$, un sommet quelconque v_i est sélectionné et ordonné, c'est–à–dire mis dans l'ensemble S : $\pi(1) = i$, $S = \{v_i\}$. Par la suite, le sommet v_j qui optimise une certaine fonction d'attraction est mis dans S : $\pi(1) = j$, $S = \{v_i, v_j\}$.
L'ordonnancement se termine quand tous les sommets sont ordonnés.

Les fonctions d'attraction f_{att} suivantes ont été étudiées [Fou98] :

- *DFS - depth first search* : choisir un voisin quelconque du dernier sommet ordonné, ce qui correspond à effectuer une exploration en profondeur

  $$f_{att}(j) = \max\{k | v_{\pi(k)} \in \mathrm{adj}(j) \cap S\}$$

  où $\mathrm{adj}(j)$ est l'ensemble des sommets voisins de v_j.

- *Max Adjacency* : choisir le sommet qui a le plus de *nets* en commun avec les sommets déjà rangés

  $$f_{att}(j) = |\{e \in E | (v_j \in e) \wedge (e \cap S \neq \emptyset)\}|$$

- *Absorption* : choisir le sommet qui maximise l'absorption des *nets* en cas de groupement. Un *net* participe à cette absorption avec le nombre de sommets incidents à ce *net* dans l'ordonnancement divisé par la cardinalité du même *net* moins un. L'absorption de l'ensemble S $\mathrm{abs}(S)$ est alors :

  $$\mathrm{abs}(S) = \sum_{\{e \in E | e \cap S \neq \emptyset\}} \frac{|e \cap S|}{|e| - 1}$$

  L'attraction du sommet v_i est donc l'accroissement de l'absorption en cas de regroupement :

  $$f_{att}(j) = \mathrm{abs}(S \cup v_j) - \mathrm{abs}(S) = \sum_{\{e \in E | v_j \in e \wedge e \cap S \neq \emptyset\}} \frac{1}{|e| - 1}$$

- *Scaled Cost* : choisir le sommet qui minimise le nombre de sommets ordonnés $\mathrm{card}(S)$ divisé par la coupure $\mathrm{cut}(S)$:

  $$f_{att}(j) = \frac{\mathrm{cut}(S \cup v_j)}{\mathrm{card}(S \cup v_j)}$$

  Si tous les sommets ont un poids égale à 1, le *Scaled Cost* est équivalent à l'objectif de minimiser la coupure (*Min Cut*) :

  $$f_{att}(j) = \mathrm{cut}(S \cup v_j)$$

Si l'ordonnancement est effectué en utilisant *DFS, Max Adjacency ou Absorption* comme fonction d'attraction, c'est le sommet v_j maximisant cette fonction qui est sélectionné et ordonné par la suite. Dans le cas du *Scaled Cost*, c'est celui qui minimise la fonction d'attraction f_{att}.

Exemple 7.1 Ordonnancement des sommets d'un hypergraphe selon la fonction d'attraction choisie

L'exemple d'un hypergraphe est présenté dans figure 7.1. En commençant avec le sommet

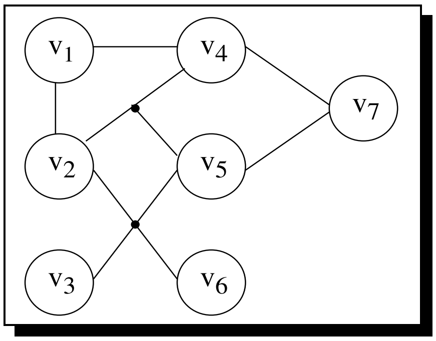


Figure 7.1: *Exemple d'un hypergraphe*

v_1 *et selon la fonction d'attraction choisie, l'ordonnancement suivant est déterminé :*

- *DFS :* $v_1, v_4, v_7, v_5, v_2, v_6, v_3$
- *Max Adjacency :* $v_1, v_4, v_2, v_5, v_7, v_6, v_3$
- *Absorption :* $v_1, v_4, v_2, v_5, v_7, v_6, v_3$
- *Scaled Cost :* $v_1, v_2, v_3, v_6, v_5, v_4, v_7$

Après avoir sélectionné et rangé un sommet v_i, la valeur de la fonction d'attraction des sommets non-ordonnés qui sont des voisins de sommets ordonnés est mise à jour.

Cette implémentation de mise à jour de la fonction d'attraction de tous les voisins des sommets ordonnés mène à une complexité de l'algorithme en $O(|V|^2)$.
Or, il apparaît clairement que si nous découpons l'ordonnancement en k partitions, les premiers sommets ordonnés ne seront pas dans la même partition que les derniers sommets non-ordonnés. Idéalement, seuls les sommets susceptibles d'appartenir à la même partition que le sommet à ordonner devraient exercer une attraction. Afin de tenir compte de cette notion, Alpert et Kahng [Alp96] ont introduit la notion de "fenêtre" de taille W. Cette fenêtre est l'ensemble des W derniers sommets ordonnés. Seuls les sommets de la fenêtre ont une attraction sur les sommets non-ordonnés.
A chaque fois qu'un sommet v_i est ordonné, la fenêtre se déplace. Le sommet $v_{\pi(k)}$ entre dans la fenêtre et le sommet $v_{\pi(k-W)}$ en sort.
L'introduction de la fenêtre permet ainsi de réduire la complexité de l'algorithme à $O(W \times |V|)$ sous condition que la cardinalité des *nets* soit limitée par une constante et que le nombre de connecteurs soit en $O(|V|)$.

7.1.2 Avantages et inconvénients

Une des difficultés de l'algorithme consiste à déterminer la taille W de la fenêtre, qui dépend de la taille des partitions à créer : à chaque fois que nous faisons varier les contraintes du partitionnement en terme de cardinalité, le paramètre W doit être modifié afin d'assurer la meilleure qualité possible du partitionnement.
Nous allons voir dans section 7.4.3 que le choix de ce paramètre peut être délicat.

Un autre choix consiste à déterminer la fonction d'attraction qui mène au meilleur partitionnement de la *netlist*, c'est–à–dire qui nécessite un nombre minimal de partitions pour couvrir toute la *netlist*.

L'avantage du choix de la fonction d'attraction est la modularité de l'algorithme. Le choix d'une nouvelle fonction d'attraction mène à un autre ordonnancement et ainsi à un autre partitionnement.

De plus, grâce à sa complexité en $O(W \times |V|)$, l'algorithme est relativement rapide par rapport à d'autres algorithmes d'ordonnancement comme par exemple ceux exploitant la méthode spectrale.

7.2 Algorithme de découpage : "DP-RP"

7.2.1 Description de l'algorithme

L'algorithme de découpage est basé sur l'algorithme DP-RP (*Dynamic Programming for Restricted Partitioning*) développé par Alpert et Kahng [Alp95c]. L'objectif de cet algorithme est le découpage d'un ordonnancement de $|V| = n$ sommets $v_{\pi(1)}, v_{\pi(2)}, \ldots, v_{\pi(n)}$ en ensembles de sommets consécutifs respectant chacun des contraintes de cardinalité et de coupure et formant ainsi un multipartitionnement de l'hypergraphe.

Soit $C_{[i,j]} = \{v_{\pi(i)}, \ldots, v_{\pi(j)}\}$ avec $i < j$ un sous-ensemble consécutif de l'ordonnancement π contenant $j - i + 1$ sommets. Nous avons n sous-ensembles qui peuvent avoir comme premier sommet le sommet $v_{\pi(1)}$, $(n-1)$ sous-ensembles avec le sommet $v_{\pi(2)}$ comme premier sommet, $(n - 3)$ sous-ensembles avec le sommet $v_{\pi(3)}$ comme premier sommet, etc. Le nombre de sous-ensembles possibles est donc $n + (n - 1) + (n - 2) + \ldots + 1 = n(n + 1)/2$. La totalité des ensembles consécutifs obtenus par le découpage d'un ordonnancement est donc définie par :

$$\begin{aligned}
\{C_{[i,j]}\} &= \{C_{[1,1]}, C_{[1,2]}, \ldots, C_{[1,n]}\} \\
&\cup \{C_{[2,2]}, C_{[2,3]}, \ldots, C_{[2,n]}\} \\
&\vdots \\
&\cup \{C_{[n-1,n-1]}, C_{[n-1,n]}\} \\
&\cup \{C_{[n,n]}\}
\end{aligned}$$

pour $1 \leq i \leq j \leq n$.
Au début, il faut calculer le coût intrinsèque $w(C_{[i,j]})$ de chaque partition possible. Deux fonctions de coût ont été implémentées pour ce faire :

- *Scaled Cost* :

$$w(C_{[i,j]}) = \frac{\mathrm{cut}(C_{[i,j]})}{\mathrm{card}(C_{[i,j]})}$$

- *Absorption* :

$$w(C_{[i,j]}) = \frac{|e \cap C_{[i,j]}| - 1}{|e| - 1}$$

Parmi l'ensemble des partitions possibles, certaines ne respectent pas les contraintes de cardinalité ou de coupure. Une telle partition est non-valide et son coût est infini[1] : $w(C_{[i,j]}) = \pm\infty$

Le calcul des différentes partitions possibles se fait itérativement : supposons que $w(C_{[i,j]})$ soit calculé. La partition $C_{[i,j+1]}$ est obtenue en ajoutant le sommet $v_{\pi(j+1)}$ à la partition $C_{[i,j]}$: $C_{[i,j+1]} = C_{[i,j]} \cup v_{\pi(j+1)}$. On peut donc induire le calcul de $w(C_{[i,j+1]})$ en fonction de $w(C_{[i,j]})$:

$$w(C_{[i,j+1]}) = w(C_{[i,j]}) + \delta(v_{\pi(j+1)})$$

où $\delta(v_{\pi(j+1)})$ est la perturbation induite par le sommet $v_{\pi(j+1)}$.
Soit U la cardinalité maximale des partitions et supposons que nous traitions des sommets de cardinalité 1. Lorsqu'une partition $C_{[k,l]}$ est limitée par les sommets $v_{\pi(k)}$ et $v_{\pi(l)}$ et $l = k + U$, nous arrêtons le calcul itératif du coût des partitions commençant avec le sommet $v_{\pi(k)}$ et continuons le calcul avec la partition $C_{[k+1,k+1]}$. Ainsi le calcul des coûts est restreint à nU partitions.

A l'origine, DP-RP a été conçu par Alpert et Kahng pour effectuer un multipartitionnement

$$P^k = \{C_1, C_2, \ldots, C_k\} = \{C_{[1,i]}, C_{[i+1,j]}, \ldots C_{[l,n]}\}$$

avec $1 \leq i \leq j \leq l \leq n$ en un nombre fixe k de sous-ensembles tel qu'une fonction de coût $f(P^k)$ soit optimisée :

$$f(P^k) = \sum_{i=1}^{k} w(C_i)$$

et $f(P^k) \neq \pm\infty$.
Grâce à la construction itérative du multipartitionnement, l'algorithme permet de trouver une solution P^k telle que k soit minimal pour un ordonnancement donné. Le fait que les fonctions de coût utilisées soient monotones en $w(C_i)$ garantit en même temps que $f(P^k)$ soit optimal pour les k partitions [Fou98].

L'algorithme procède de la manière suivante : il commence avec $k = 1$. Tant que les n sommets ne peuvent pas être partitionnés en k partitions k est incrémenté. A chaque itération, l'algorithme détermine le meilleur partitionnement des j premiers sommets pour $1 \leq j \leq n$ en k partitions. Pour ce faire, il additionne le coût de l'itération précédente, qui découpe l'intervalle $[1, m]$ en $k-1$ partitions, et le coût de la partition supplémentaire $w(C_{[m+1,j]})$ avec $1 \leq m \leq j$. Le coût d'un tel partitionnement est égal à $\pm\infty$ s'il n'est pas valide. Nous avons donc des coûts non-infinis uniquement pour au plus les kU premiers sommets où U est le nombre maximal de sommets dans une partition.
Soit $P^k_{[1,j]}$ le meilleur partitionnement des premiers j sommets d'un ordonnancement de

[1] L'objectif est de minimiser le *scaled cost* et de maximiser l'*absorption*. Le coût d'une partition non-valide est donc $+\infty$ pour le *scaled cost* et $-\infty$ pour l'*absorption*. Afin de simplifier l'écriture, nous noterons $\pm\infty$.

sommets $\{v_{\pi(1)}, v_{\pi(2)}, \ldots, v_{\pi(n)}\}$ en k partitions et f_b son coût. Si le *Scaled Cost* est la fonction du coût utilisée, l'algorithme DP-RP peut être décrit comme suit : *L'algorithme DP-RP*

Pour chaque j = 1,…, n :
    Calculer $w(C_{[1,j]})$
    $P^1_{[1,j]} = C_{[1,j]}$
$k = 1$
Tant que $f(P^k_{[1,n]}) = \infty$:
    $k \leftarrow k + 1$
    Pour chaque j = 1,…, n :
        $f_b = \infty$
        Pour chaque m = 2,…, j :
            Si $f_b > f(P^{k-1}_{[1,m-1]}) + w(C_{[m,j]})$:
                $f_b = f(P^{k-1}_{[1,m-1]}) + w(C_{[m,j]})$
                $P^k_{[1,j]} = P^{k-1}_{[1,m-1]} \cup C_{[m,j]}$
$P^k_{[1,n]}$ *est la solution optimale.*
Fin de l'algorithme.

La complexité de l'algorithme est en $O(kU|V|)$ où k est le nombre de partitions créées et U le nombre maximal de sommets par partition. Pour un taux de remplissage moyen $tdr = |V|/kU \leq 1$, nous avons donc une complexité en $O(|V|^2)$.

Le principe de l'algorithme est illustré à l'aide de l'exemple suivant.

Exemple 7.2 L'algorithme DP-RP
Soit $\{v_{\pi(1)}, v_{\pi(2)}, \ldots, v_{\pi(8)}\}$ *un ordonnancement de huit sommets. L'objectif est de déterminer un découpage de cet ordonnancement en un nombre minimal de partitions tel que la cardinalité maximale d'une partition soit 5.*

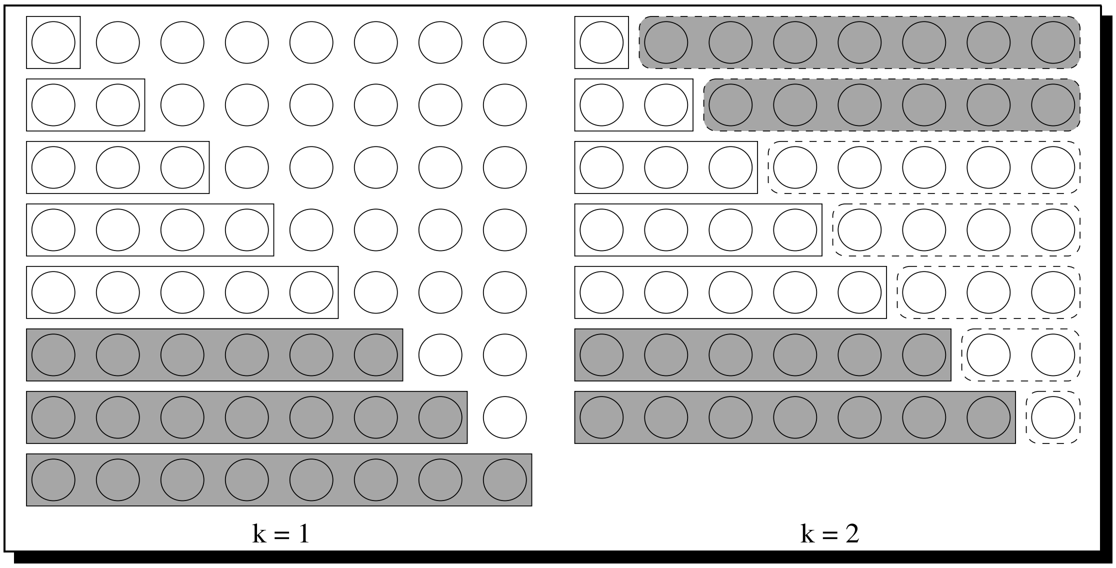


Figure 7.2: *Partitionnement d'un ordonnancement de 8 sommets*

Pour $k = 1$, *l'algorithme détermine le coût des partitions* $P^1_{[1,1]}$, $P^1_{[1,2]}$, $P^1_{[1,3]}$, $P^1_{[1,4]}$, $P^1_{[1,5]}$, $P^1_{[1,6]}$, $P^1_{[1,7]}$, $P^1_{[1,8]}$ *illustrées par les rectangles dans figure 7.2. Seulement les cinq premières*

partitions respectent la contrainte de cardinalité. Elles ont donc un coût non-infini, illustré par les rectangles blancs avec des lignes continues. Pour les autres partitions, nous avons un coût qui est infini, illustré par les rectangles grisés avec les lignes continues.
Pour $k = 2$, l'algorithme détermine au fur et à mesure le partitionnement des deux premiers sommets en deux partitions, puis le partitionnement des trois premiers sommets en deux partitions, etc. A chaque fois, il essaie toutes les configurations possibles. Le cas de $j = n$ est illustré dans figure 7.2. Les deux premiers partitionnements ne sont pas valides, car les nouvelles partitions $C_{[2,8]}$ et $C_{[3,8]}$ ne respectent pas la cardinalité maximale, illustrées par les rectangles grisés avec les lignes en pointillés. Les autres nouvelles partitions respectent la cardinalité illustrées par les rectangles blancs avec les lignes en pointillés.
Il existe donc trois solutions avec un coût non-infini. Il s'agit des partitions $P^1_{[1,3]} + C_{[4,8]}$, $P^1_{[1,4]} + C_{[5,8]}$ et $P^1_{[1,5]} + C_{[6,8]}$. Celle qui optimise la fonction du coût est retenue comme résultat du partitionnement de l'ordonnancement de huit sommets.
Notons, que la fonction de coût choisie n'a aucune influence sur le nombre de partitions créées.

7.2.2 Avantages et inconvénients

L'algorithme DP-RP permet d'effectuer un multipartitionnement d'un ordonnancement donné en un nombre minimal de partitions valides en optimisant en même temps une fonction de coût choisie.
Malgré sa complexité en $O(|V|^2)$, qui limite son domaine d'application, il a été observé lors de l'évaluation des performances des algorithmes de partitionnement (voir chapitre 8), qu'il nécessite moins de temps CPU que les algorithmes hyperFBB et multisect.

L'inconvénient majeur de l'algorithme est la dépendance de la qualité du découpage par rapport à l'ordonnancement qui lui est fourni. Un mauvais ordonnancement mène forcement à un mauvais multipartitionnement. Il est donc primordial de soigner la construction de cet ordonnancement. Néanmoins, un mauvais multipartitionnement peut être amélioré par construction d'un autre ordonnancement, éventuellement avec un autre algorithme.

Nous allons voir dans le chapitre 8 qu'un algorithme seul ne satisfait pas les besoins du partitionnement pour l'émulation. La solution consiste souvent à utiliser une certaine combinaison de plusieurs algorithmes.
Supposons l'application de l'algorithme DP-RP à un sous-ensemble d'une *netlist*. Ce sous-ensemble a été créé en utilisant un autre algorithme de partitionnement, comme par exemple bisect. L'ordonnancement construit pour les sommets de ce sous-ensemble doit être découpé en un multipartitionnement par l'algorithme DP-RP. Le découpage en k partitions valides peut alors être sous-optimal, si le découpage effectué avec bisect n'est pas optimal. En effet, il semble judicieux de permettre le rejet d'un certain nombre de sommets dans d'autres partitions afin de compenser la sous-optimalité du découpage de bisect.

7.3 Ordonnancement par calcul itératif du barycentre

Nous avons constaté dans la section précédente que le résultat du partitionnement avec l'algorithme DP-RP dépendait de l'ordonnancement. L'ordonnancement construit avec l'algorithme window proposé par Alpert et Kahng [Alp96] effectue une recherche locale

à partir d'un sommet. Il a donc une vue restreinte de la *netlist*.
Pour améliorer la qualité de l'ordonnancement et ainsi réduire le nombre de partitions déterminé avec l'algorithme DP-RP, nous avons recherché une autre méthode d'ordonnancement ayant une vue globale de la *netlist*. Cette méthode est présentée en annexe A.

Une analyse des performances de cette méthode a révélé que beaucoup de sommets ont le même ordonnancement. L'algorithme DP-RP choisit un ordre quelconque, ce qui revient à faire un ordonnancement aléatoire.
L'algorithme n'a donc pas été retenu dans la suite de ce travail.

7.4 Adaptation de l'algorithme aux architectures hiérarchiques : algorithme MDOC

7.4.1 Choix du premier sommet

Le choix du premier sommet est primordial pour le résultat de l'ordonnancement. Afin de permettre la reproductibilité du partitionnement et de garantir un bon résultat de partitionnement, nous avons choisi ce sommet sur la périphérie de la *netlist*. C'est le premier sommet rencontré ayant un connecteur externe.

7.4.2 Fonction d'attraction

Plusieurs fonctions d'attraction ont été implémentées : l'exploration en profondeur (*DFS*), *Max Adjacency*, *Absorption* et *Scaled Cost*. L'influence de la fonction d'attraction sur le résultat du partitionnement a été étudiée à l'aide des *netlists* de 15000 CPBs générées avec PartGen. En particulier, nous utilisons une *netlist* de contrôleur (*ctrl15K*), une *netlist* de combinatoire irrégulière (*glue15K*) et une *netlist* d'opérateur (*op15K*).
Ces *netlists* ont été partitionnées en respectant les contraintes de cardinalité et de coupure de l'AVC. La taille de la fenêtre a été fixée à $+\infty$ pour ce test. L'attraction de chaque sommet non-ordonné prend donc en compte la totalité des sommets ordonnés. Les résultats de ce test en terme de nombre d'AVCs nécessaires pour implémenter les circuits sont reportés dans le tableau 7.1.

	DFS	Max Adjacency	Absorption	Scaled Cost
Circuit	#AVC	#AVC	#AVC	#AVC
op15K	238	111	90	123
ctrl15K	136	93	51	70
glue15K	307	309	304	288
Moyenne	227	171	148	160

Tableau 7.1: *L'influence de la fonction d'attraction sur le résultat du partitionnement en AVCs*

Les résultats expérimentaux montrent une forte variation selon la fonction d'attraction choisie. Les fonctions d'attraction *absorption* et *scaled cost* semblent fournir les meilleurs résultats. Cette hypothèse a été confirmée par des expériences effectuées sur des *netlists* réelles.

7.4.3 Taille de la fenêtre

L'influence de la taille de la fenêtre W sur le résultat du partitionnement en AVCs a été étudiée à l'aide des mêmes *netlists* utilisées dans la section précédente. La fonction d'attraction choisie est l'*absorption*. Nous avons fait varier la taille de la fenêtre W de 1 à 800 et observé le nombre d'AVCs nécessaires pour l'implémentation de la *netlist*. Les résultats expérimentaux sont présentés dans les figure 7.3 à 7.5.

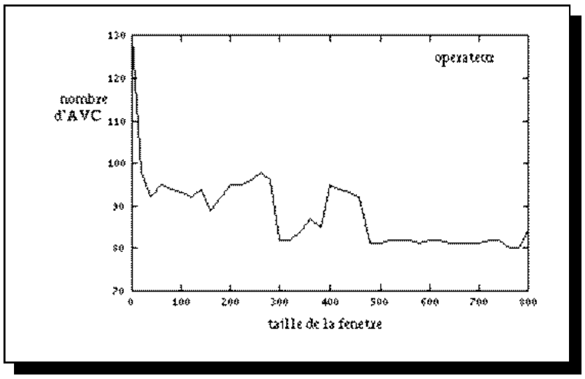


Figure 7.3: *Résultat du partitionnement d'un opérateur en AVCs en fonction de la taille de la fenêtre*

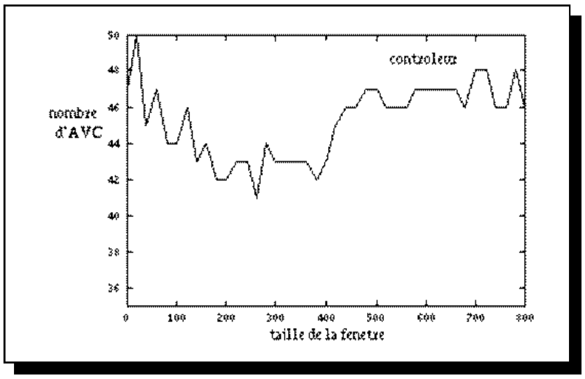


Figure 7.4: *Résultat du partitionnement d'un contrôleur en AVCs en fonction de la taille de la fenêtre*

Les résultats observés varient fortement selon le type de *netlist* partitionnée. La courbe obtenue pour la *netlist* d'opérateur décroît jusqu'à un minimum local vers 300 puis remonte pour retomber dans un minimum global entre 500 et 800.
Dans le cas de la *netlist* de combinatoire irrégulière, nous avons un minimum global vers 20, puis la courbe croît de façon monotone sous forme approximativement logarithmique.
La courbe obtenue pour le contrôleur est difficile à caractériser. Les meilleurs résultats ont été obtenus avec une fenêtre variant de 200 à 400 sommets.
Le tableau 7.2 montre pour chaque *netlist* le meilleur et le pire résultat obtenu en nombre d'AVCs en fonction de la taille de la fenêtre ainsi que l'écart entre le nombre minimal et maximal d'AVCs.

Les résultats présentés montrent qu'il est difficile d'identifier une taille de fenêtre optimale pour toutes les applications. Par défaut, cette taille a été fixée à 50 pour partCluster, à 400

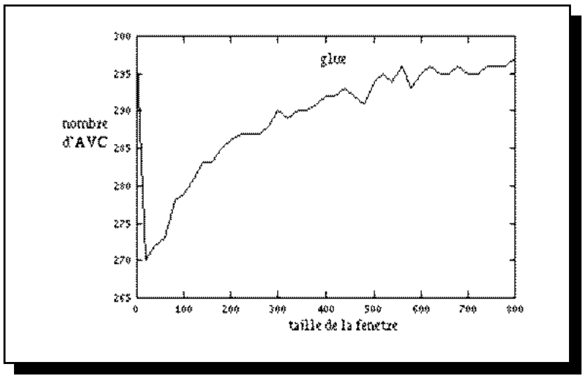


Figure 7.5: *Résultat du partitionnement de la combinatoire irrégulière en AVCs en fonction de la taille de la fenêtre*

	Meilleur résultat		Pire résultat		Ecart	
Circuit	W	#AVC	W	#AVC	absolu	pour cent
op15K	760	80	260	98	18	22%
ctrl15K	260	41	800	51	10	24%
glue15K	20	270	800	304	34	13%

Tableau 7.2: *Influence de la taille de la fenêtre sur le résultat du partitionnement en AVCs*

pour partMeta et à 12 pour partBoard.
La détermination d'une fenêtre optimale suivant la topologie de la *netlist* est un des axes de recherche pour l'optimisation de l'algorithme d'ordonnancement. D'une manière générale, une meilleure connaissance de cette dépendance permettrait d'améliorer le résultat du partitionnement et représenterait un premier pas vers un partitionnement piloté par la topologie de la *netlist* utilisant au mieux les algorithmes disponibles. En effet, il semble exister une dépendance entre la topologie d'une *netlist* et les performances des algorithmes. Ainsi, l'algorithme hyperFBB partitionne les *netlists* d'opérateur du *benchmark* présenté dans le tableau 8.1 (op5K et op15K) en 73 sous-ensembles (AVCs et FPGAs XC3064) en moyenne par rapport à 84 sous-ensembles en moyenne par bisect* (cf. les tableaux C.12 et C.13). Par contre, bisect* est plus performant pour le partitionnement des *netlist* de contrôleurs (ctrl15K, ctrl35K et ctrl50K) : il partitionne ces *netlists* en 332 sous-ensembles en moyenne, par rapport à 446 sous-ensembles en moyenne par hyperFBB.

7.5 Conclusion

Dans ce chapitre, nous avons présenté un algorithme de partitionnement, qui est composé de deux phases découplées : window et DP-RP.
Lors de la première phase, nous réalisons un ordonnancement des sommets d'un hypergraphe modélisant une *netlist*. Pendant la deuxième phase, cet ordonnancement est découpé en respectant des contraintes de cardinalité et de coupure.

Nous avons adapté cet algorithme à une utilisation dans le cadre de l'émulation menant à l'algorithme appelé MDOC. Pour cela, nous avons

- implémenté une fonction permettant de choisir le premier sommet de l'ordonnancement afin d'assurer la reproductibilité des résultats et d'améliorer la qualité de l'ordonnancement.

- analysé l'influence de la fonction d'attraction sur le résultat du partitionnement.
- analysé l'influence de la taille de la fenêtre sur le résultat du partitionnement qui dépend fortement des contraintes imposées.
- implémenté une fonction permettant d'arrêter le découpage de l'ordonnancement, même quand il reste des partitions non-valides, s'il y a un certain nombre de partitions valides qui satisfont des contraintes de qualité

L'algorithme MDOC a été retenu pour une évaluation plus approfondie. Cette évaluation des performances ainsi qu'une étude concernant son utilisation dans le cadre de l'émulation sont présentées dans chapitre 8.

Chapitre 8

Partitionnement pour une architecture hiérarchique concrète

8.1 Introduction

Au cours des chapitres précédents, nous avons analysé séparément différents algorithmes de partitionnement.
Afin de sélectionner les algorithmes les plus performants nous procédons maintenant à une comparaison expérimentale. Les algorithmes sont appliqués aux *netlists* d'un banc de test pour effectuer un partitionnement qui respecte diverses contraintes de cardinalité et de coupure.
Dans le but d'optimiser les différents outils de partitionnement, nous avons recours à des combinaisons de plusieurs algorithmes. Les résultats de l'analyse comparative des algorithmes nous permettent de déterminer les associations les plus performantes. Une étude expérimentale portant sur les interactions de ces algorithmes valide ces associations. Le résultat de cette étude permet donc de définir un ensemble d'outils de partitionnement performants.
Le processus qui permet de déterminer les outils de partitionnement intéressants, construits sur la base de différents algorithmes de partitionnement, est présentée dans la section 8.2.

L'enchaînement des différents outils de partitionnement a également une influence sur le résultat. D'autres enchaînements que celui utilisé actuellement nous semblent intéressants et méritent une analyse expérimentale approfondie. Cette analyse est le sujet de la section 8.3.

Les résultats obtenus avec notre approche sont comparés avec les résultats obtenus avec l'approche existante, en terme de niveau d'atteinte des objectifs fixés au début de ce travail. Cette analyse est présentée dans la section 8.4.

8.2 Construction des outils de partitionnement

8.2.1 Introduction

Nous disposons de trois outils de partitionnement, un pour chacun des 3 premiers niveaux hiérarchiques analysés de l'émulateur CELARO :

- partCluster : pour le partitionnement en *clusters* contenant quelques dizaines de CPBs avec autant de *pins* d'entrée et de *pins* de sortie.
- partMeta : pour le partitionnement en AVCs ayant entre 100 et 200 *pins* d'entrée et de sortie et contenant soit entre 10 et 20 *clusters*, soit entre 700 et 800 CPBs, soit la moitié des *clusters* ou la moitié des CPBs et une mémoire *xmem*.
- partBoard : pour le partitionnement en AVBs. Chaque AVB a plus de 1000 *pins* d'entrée et de sortie et peut contenir soit entre 40 et 50 AVCs, soit une combinaison de *clusters* et de mémoire *xmem*.

Ces trois outils peuvent être lancés dans un ordre arbitraire. Il est donc possible d'appliquer d'abord partCluster suivi de partMeta et partBoard ou partBoard suivi de partMeta et partCluster. Il y a donc six possibilités différentes pour l'ordre des outils. Chacun de ces outils s'exécute en trois phases :

- Phase 1 : Cette phase réalise le principe de dichotomies successives en découpant la *netlist* par identification de parties isolées pour pouvoir traiter indépendamment des ensembles plus petits. Lors de cette phase nous n'avons aucune contrainte de découpage que ce soit au niveau de la cardinalité des partitions qu'au niveau de leur coupure.
- Phase 2 : Cette phase prépare l'implémentation des cellules dans des éléments physiques. Les partitions créées lors de la phase 1 sont découpées en ensembles de cellules avec des contraintes de cardinalité et de coupure plus petites que celles de l'élément physique sur lequel ils seront implémentés.
- Phase 3 : Les ensembles créés précédemment sont regroupés en minimisant le nombre de partitions et en respectant les contraintes physiques.

Pour les deux premières phases, nous disposons de trois algorithmes[1], bisect*, cone et hyperFBB[2]. Pour la dernière phase, nous disposons de deux algorithmes, MDOC et multisect*. Nous pouvons donc réaliser chaque outil de 18 manières différentes, comme par exemple bisect* pour la phase 1, hyperFBB pour la phase 2 et MDOC pour la phase 3, ou encore hyperFBB pour la phase 1, bisect* pour la phase 2 et multisect* pour la phase 3. Les différents algorithmes sont présentés plus en détail dans la section 8.2.2.

Aux 18 possibilités pour implémenter chaque outil, s'ajoutent les 6 possibilités pour l'ordre des outils, ce qui fait $18*18*18*6 = 34992$ possibilités pour partitionner une *netlist* de CPBs en AVBs. Evidemment, il est impossible de tester toutes ces possibilités. Nous devons donc faire une présélection pour déterminer les combinaisons d'algorithmes et les enchaînements

[1] Tous les algorithmes sont présentés dans la section 8.2.2.

[2] La phase de *Clustering* peut être supprimée, ce qui donne des possibilités supplémentaires.

d'outils qui seront examinés plus en détail. Pour ce faire, nous avons d'abord analysé séparément chacun des algorithmes afin de déterminer ses avantages et inconvénients. Ensuite, nous avons utilisé ces analyses pour déterminer les algorithmes les plus appropriés pour les différentes phases de chaque outil. Cette analyse est présentée dans la section 8.2.4.

Ensuite, nous avons examiné chacune des combinaisons retenues pour un certain nombre de *netlists*. L'analyse des résultats expérimentaux nous a permis de sélectionner trois possibilités d'utilisation de partBoard et partMeta et une possibilité pour partCluster. Cette analyse est présentée dans la section 8.2.5.
Le flot des outils a été analysé par la suite. Cette analyse est présentée dans la section 8.2.6. Deux combinaisons pour enchaîner les outils de partitionnement étaient particulièrement efficaces en terme de nombre d'AVBs obtenus après le partitionnement.
Finalement, nous avons donc retenu $3 * 3 * 1 * 2 = 18$ possibilités pour partitionner une *netlist* de CPBs en AVBs. Ces combinaisons seront examinées en détail dans la section 8.3.

8.2.2 Présentation des différents algorithmes

Nous disposons à ce stade de cinq algorithmes :

- bisect* – Un algorithme de bipartitionnement ayant pour objectif de minimiser le *ratio-cut* [Wei91] entre les partitions : L'algorithme qui a été présenté dans le chapitre 4 appartient à la famille des algorithmes de voisinage. En effet, il explore l'espace des solutions en passant d'une solution à une solution voisine. Il se termine quand il a atteint un minimum local. L'algorithme gère le gain des cellules en utilisant une liste de gains comme proposé par Fiduccia et Mattheyses [Fid82].

- multisect* – Un algorithme de multipartitionnement qui minimise le nombre de partitions en respectant les contraintes de taille et de coupure a été présenté dans le chapitre 5 : L'algorithme contient une gestion complexe des partitions et cellules, qui permet entre autre de rejeter une partition si elle ne respecte pas des normes de qualité imposées par l'utilisateur. Ces cellules et partitions sont traitées ultérieurement avec d'autres cellules non-partitionnées.

- hyperFBB – Un algorithme de bipartitionnement basé sur le calcul du flot maximal dans un hypergraphe présenté dans chapitre 6 : Cet algorithme détermine la coupure minimale qui sépare deux sommets spécifiques appelés sommet source et sommet puits en respectant une certaine contrainte d'équilibre entre les deux partitions. Le flot maximal est directement déterminé sur l'hypergraphe [Pis99b]. Pour cela, il utilise implicitement une représentation en forme de graphe proposée dans [Yan94].

- MDOC – Un algorithme de multipartitionnement qui a été présenté dans le chapitre 7 coupe un ordonnancement obtenu par une exploration locale en minimisant le critère *scaled-cost* en un nombre minimal de partitions qui respectent des contraintes de taille et de coupure : En commençant avec une cellule, l'algorithme ordonne toutes les cellules une par une. A chaque fois, il détermine celle qui minimise le critère du *scaled-cost* [Zie96] par rapport aux cellules déjà ordonnées .

- cone – Un algorithme de *clustering* basé sur la construction de cônes de logique combinatoire inspiré de [Bra94] : Cet algorithme colorie par une exploration en profondeur des cônes de logique combinatoire qui se terminent par des points mémorisant de la *netlist*. Ensuite, les cônes entiers sont transformés en partitions en commençant avec le plus petit cône. Une description détaillée de l'algorithme implémenté est présenté en annexe B.

Les avantages et inconvénients de chacun de ces algorithmes ont été déterminés à l'aide du partitionnement en ensembles de cellules respectant les contraintes de taille et de coupure de FPGAs XC3064 (224 cellules, 120 I/O), de *clusters* dans l'AVC (quelques dizaines de cellules, autant d'entrées et de sorties), d'AVCs (entre 700 et 800 cellules, entre 100 et 200 I/O) et d'AVBs (entre 40 et 50 cellules, plus de 1000 I/O).
Comme *benchmark*, nous avons choisi 10 *netlists* industrielles et 10 *netlists* générées avec le générateur PartGen.

8.2.3 Description des benchmarks

Nous avons choisi 10 *netlists* générées, dont 7 *netlists* de CPBs monotype et 3 *netlists* d'AVCs multi-type (peu de sommets avec beaucoup de *pins*) [3].
Nous avons également sélectionné 10 *netlists* industrielles, dont 5 *netlists* de CPBs de type contrôleur et 3 *netlists* d'AVCs d'un microprocesseur avec un nombre de *pins* par *net* (*fanout* +1) qui varie[4] de 2.5 à 10.5.
Les caractéristiques des *benchmarks* sont présentées en détail dans le tableau 8.1.

Circuit	type de circuit	#CPBs	#AVCs	#XMEM	# SRAM	#nets	#pins par net
glue5K	irrégulier	5000	—	—	—	5008	4.35
op5K	régulier	5327	—	—	—	5327	3.96
ctrl15K	contrôleur	13779	—	—	—	13779	3.61
glue15K	irrégulier	15000	—	—	—	15008	4.87
op15K	régulier	15000	—	—	—	15354	3.98
ctrl35K	contrôleur	35025	—	—	—	35025	3.64
ctrl50K	contrôleur	50478	—	—	—	50478	3.63
mix50K3	mixte	—	191	—	4	10390	2.76
mix50K4	mixte	—	181	—	4	9634	2.67
mix55K5	mixte	—	199	—	5	11013	2.79
industA	mixte	6756	—	16	16	7564	4.49
industB	data path	15075	—	—	—	15333	4.14
industC	contrôleur	16893	—	—	—	17101	4.20
industD	contrôleur	20780	—	—	—	21680	4.05
industE	contrôleur	22578	—	—	—	23807	4.23
industF	contrôleur	22716	—	—	—	31049	3.02
industG	contrôleur	48900	—	64	—	49820	4.54
industK	microprocesseur	—	2123	—	—	98914	2.55
industL	microprocesseur	—	2123	—	—	98914	5.45
industM	microprocesseur	—	2123	—	—	98914	10.45

Tableau 8.1: *Caractéristiques des netlists utilisées comme benchmark*

8.2.4 Evaluation de la performance des algorithmes

L'application des différents algorithmes aux différentes *netlists* fournit les résultats présentés dans les figures 8.1 à 8.8 (voir aussi les tableaux C.11, C.12, C.13 et C.14 en annexe C) suivant les contraintes de partitionnement (*cluster*, AVC, XC3064 et AVB). Le taux de remplissage (tdr) est le nombre de cellules à partitionner divisé par le nombre de cellules disponibles dans toutes les partitions. L'application de multisect* sur ces benchmarks n'a pas été possible. En effet, au bout de 200 heures de calcul, il n'y avait aucun résultat pour le partitionnement d'une *netlist* de 15K CPBs à plat.

[3] A priori, cela n'a pas de sens de parler de type de *netlist* lorsqu'il s'agit d'une *netlist* d'AVCs.
[4] La modification du *fanout* moyen de la *netlist* entraîne également le changement de son type. D'ailleurs, le type n'est pas significatif lorsqu'il s'agit d'une *netlist* d'AVCs.

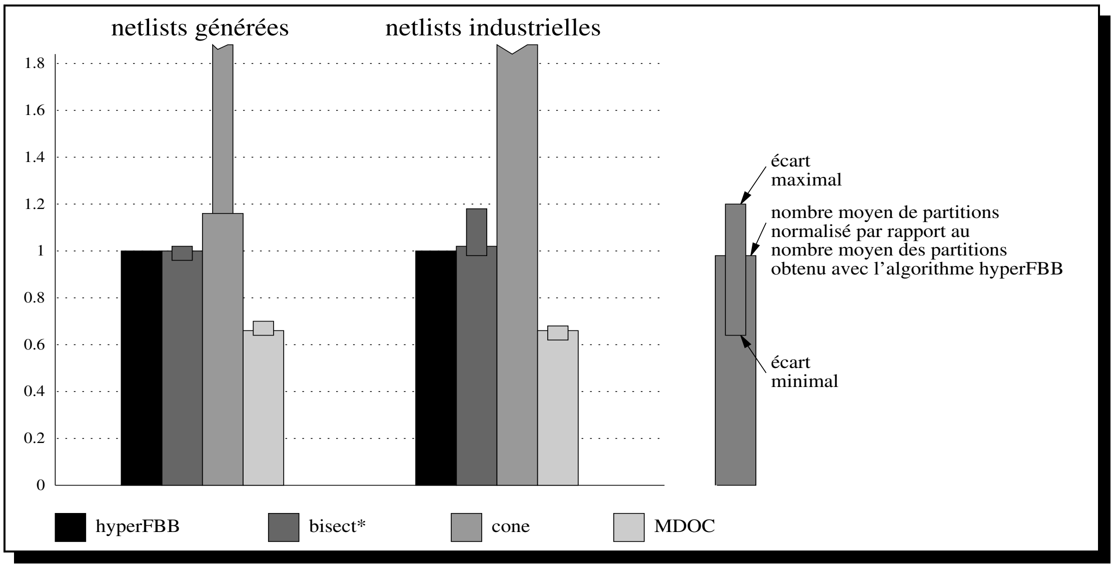


Figure 8.1: *Nombre moyen de clusters créés*

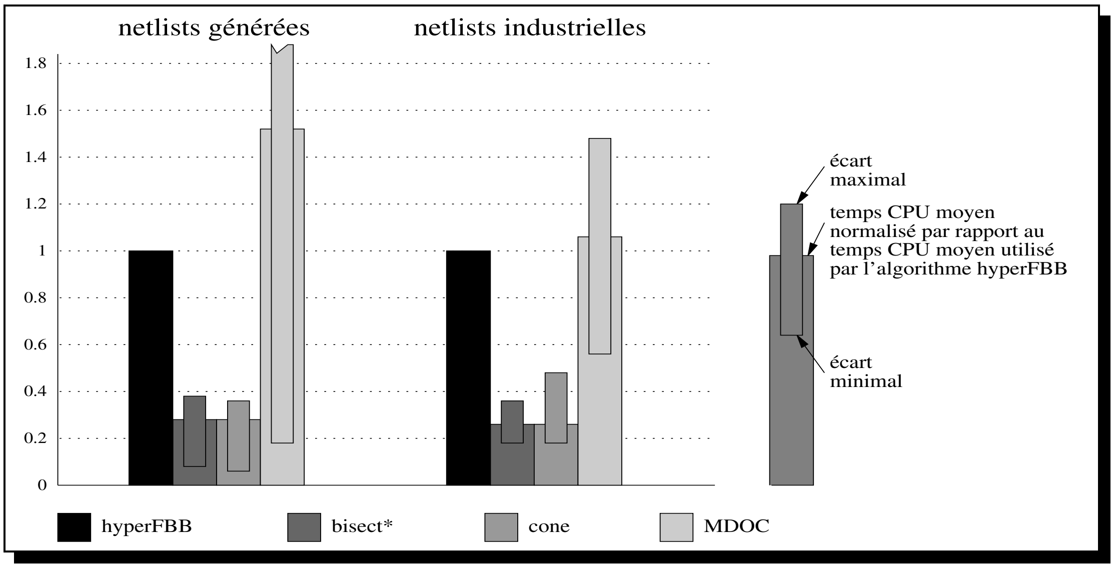


Figure 8.2: *Temps de calcul moyen pour le partitionnement en clusters*

L'algorithme bisect* fournit, indépendemment du type de la *netlist* et des contraintes de partitionnement, un taux de remplissage moyen dans un délai relativement court par rapport aux autres algorithmes. On peut en déduire qu'il est bien adapté pour traiter de très grandes netlists contenant beaucoup de CPBs, ce qui est le cas pour la phase 1 de manière générale. Grâce à de bons résultats en terme de taux de remplissage, il est également applicable dans la phase 2.

L'algorithme hyperFBB est plus lent que bisect*, mais fournit en moyenne des résultats comparables en terme de taux de remplissage. Il semble toujours applicable dans la phase 2. Il est aussi applicable dans la phase 1 mais uniquement pour des *netlists* contenant moins de 10000 cellules, ce qui est le cas pour le partitionnement de *clusters* en AVCs avec

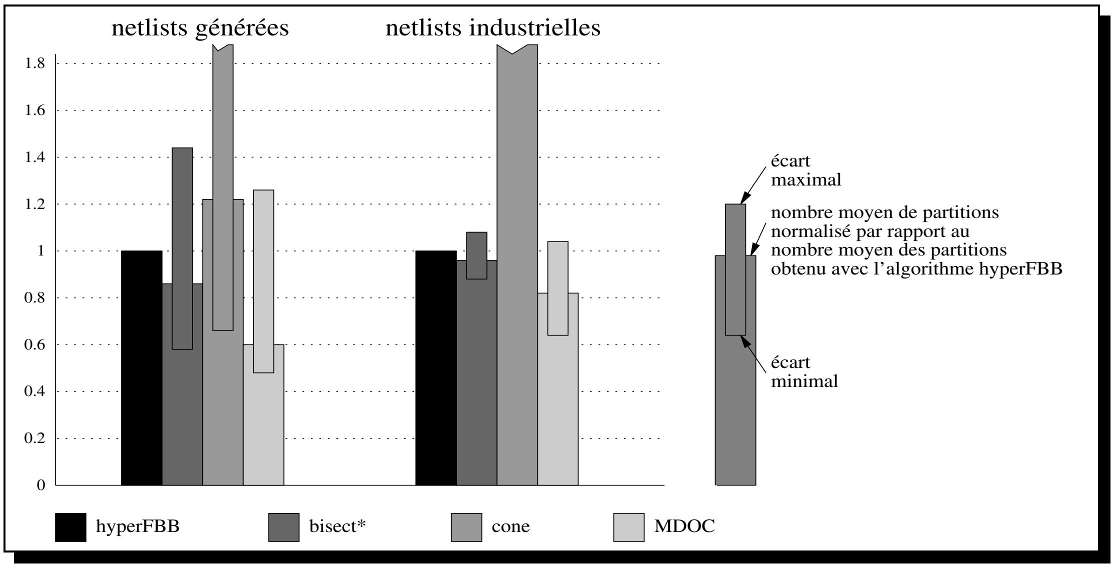


Figure 8.3: *Nombre moyen d'AVCs créés*

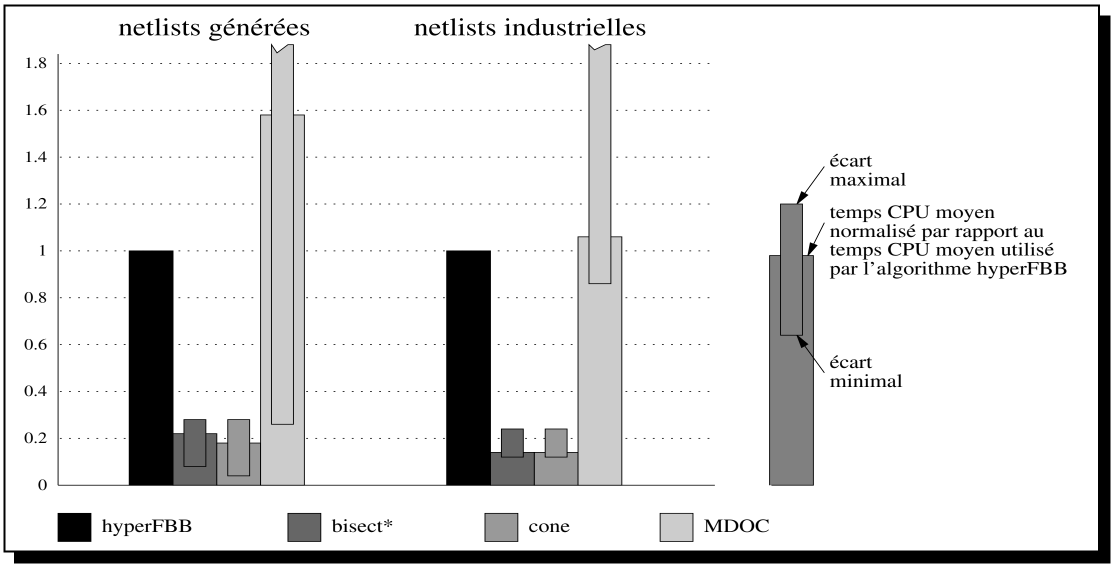


Figure 8.4: *Temps de calcul moyen pour le partitionnement en AVCs*

partMeta et pour le partitionnement d'AVCs en AVBs avec partBoard.

L'algorithme cone ne fournit pas de résultats satisfaisants. L'algorithme n'est pas adapté à la nature des *netlists* : Beaucoup de points mémorisant dans la *netlist* mènent à beaucoup de cônes de très petite taille et donc à un grand nombre de partitions. Cet effet peut être observé pour la *netlist* industD. Très peu de points mémorisant entraînent des cônes de grande cardinalité qui souvent dépassent la cardinalité d'une partition physique. Ces cônes de logique sont bipartitionnés en utilisant l'algorithme bisect* ce qui entraîne le découpage des chemins combinatoires. Dans ce cas, le nombre de partitions obtenu est approximativement le même que celui obtenu avec l'algorithme bisect*, mais l'amélioration de la fréquence n'est plus assurée. Cet effet a pu être observé pour le circuit ctrl35K. De plus,

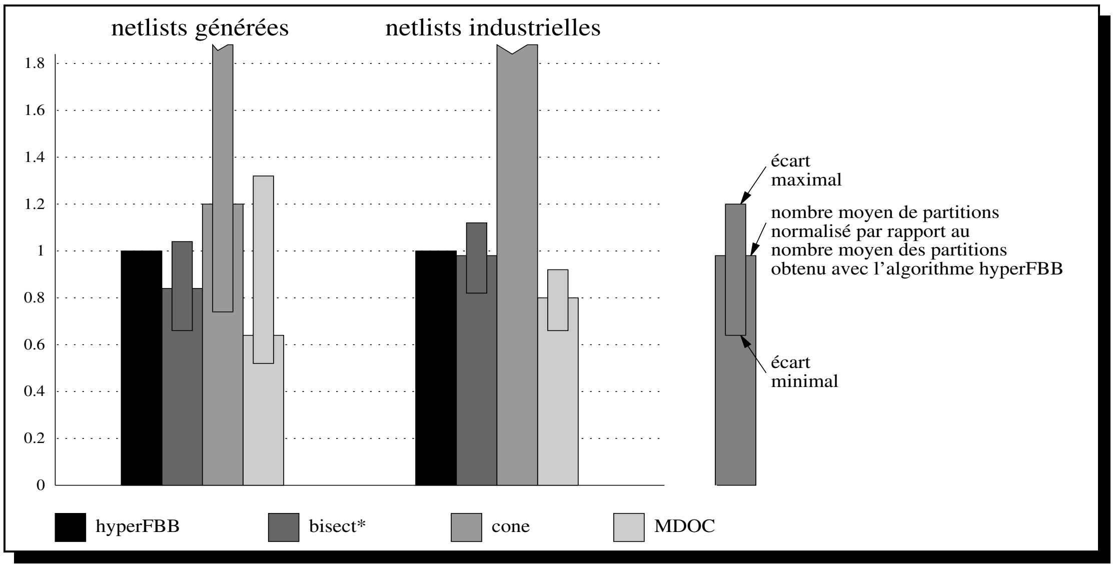


Figure 8.5: *Nombre moyen de FPGAs XC3064 créés*

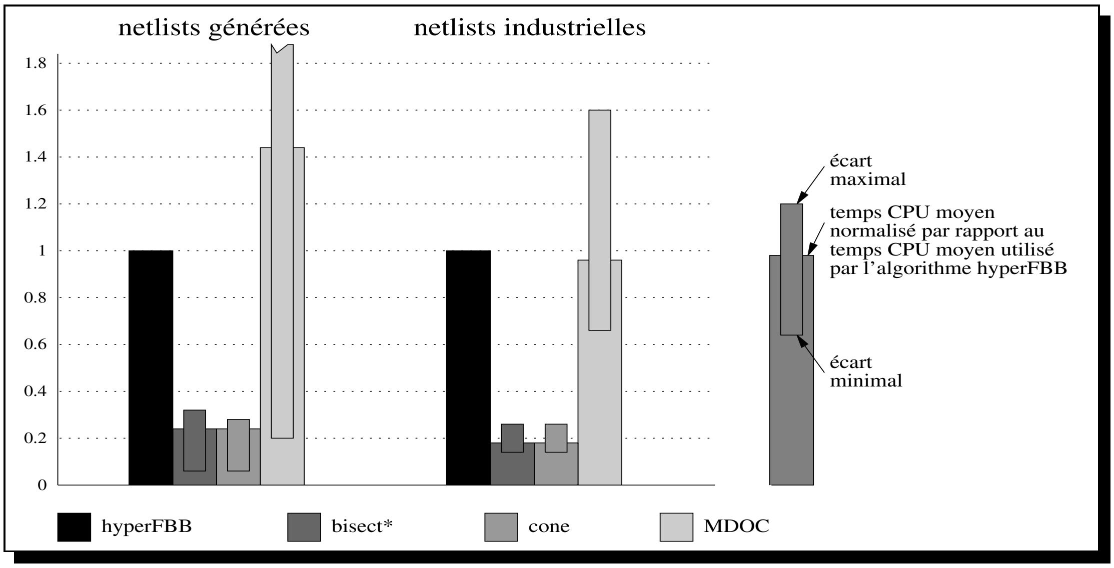


Figure 8.6: *Temps de calcul moyen pour le partitionnement en FPGAs XC3064*

chaque intersection de cônes de logique représente le découpage d'un chemin combinatoire. Le nombre de ces intersections dans la *netlist* dépend du *fanout* moyen des *nets*. Puisque nous avons observé un *fanout* moyen entre 3.5 et 4.5 dans les *netlists* industrielles, nous avons également un nombre important d'intersections de cônes. L'algorithme cone n'a donc pas été retenu par la suite.

L'algorithme MDOC fournit, pour tous les *benchmarks* et presque toutes les contraintes de partitionnement, un meilleur résultat en terme de taux de remplissage par rapport à bisect* et hyperFBB et approximativement le même résultat que multisect*.

Le partitionnement en *clusters* nécessite l'application d'algorithmes rapides. Grâce aux con-

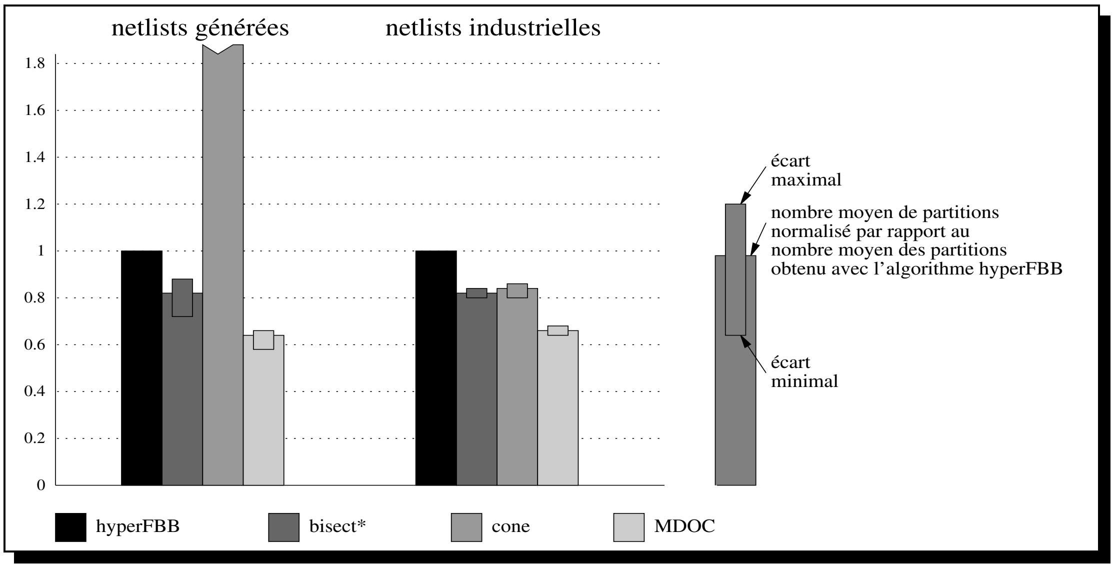


Figure 8.7: *Nombre moyen d'AVBs créés*

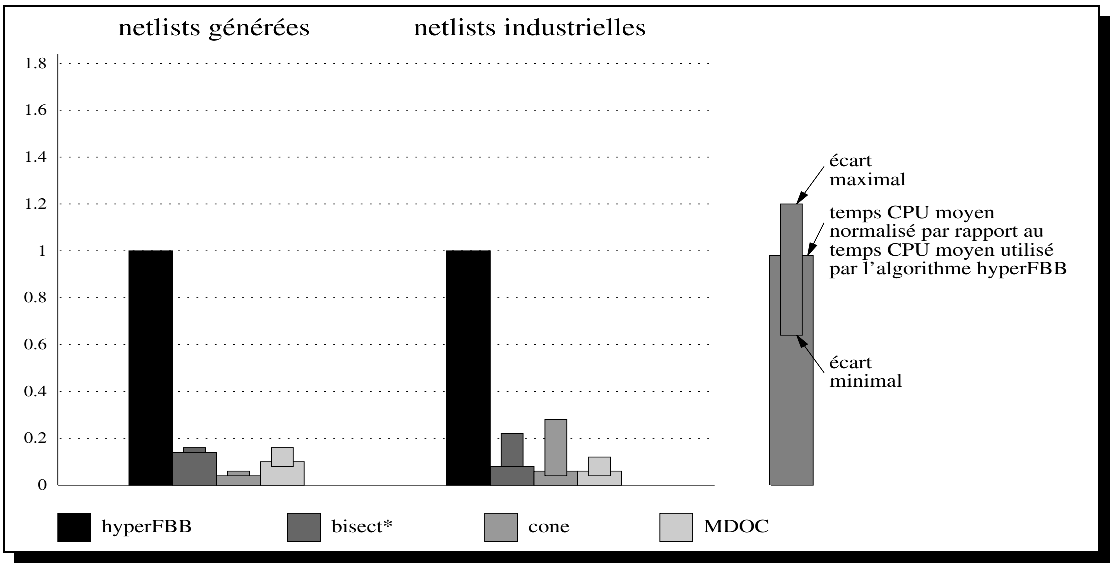


Figure 8.8: *Temps de calcul moyen pour le partitionnement en AVBs*

traintes faibles de l'interface, nous avons retenu les combinaisons bisect*-bisect*-multisect* et bisect*-bisect*-MDOC pour l'algorithme de partCluster.
Les contraintes fortes au niveau de l'interface de l'AVC exigent de tester le plus de combinaisons possibles pour partMeta. L'outil partBoard traite des *netlists* avec peu de cellules. Nous pouvons donc en relativement peu de temps tester beaucoup de combinaisons d'algorithmes. Pour cela, nous avons retenu les combinaisons bisect*-bisect*-multisect*, bisect*-bisect*-MDOC, bisect*-hyperFBB-multisect*, bisect*-hyperFBB-MDOC, hyperFBB-hyperFBB-multisect* et hyperFBB-hyperFBB-MDOC pour les outils partMeta et partBoard.

8.2.5 Evaluation de la performance des outils de partitionnement

Circuit	type de circuit	#CPBs	#clusters	#AVCs	#XMEM	# SRAM	#nets	#pins par net
glue5K	irrégulier	5000	—	—	—	—	5008	4.35
op5K	régulier	5327	—	—	—	—	5327	3.96
ctrl15K1	contrôleur	13779	—	—	—	—	13779	3.61
ctrl15K2	contrôleur	—	281	—	—	—	5820	3.01
glue15K1	irrégulier	15000	—	—	—	—	15008	4.87
glue15K2	irrégulier	—	640	—	—	—	12243	3.80
op15K1	régulier	15000	—	—	—	—	15354	3.98
op15K2	régulier	—	295	—	—	—	7555	2.49
ctrl35K1	contrôleur	35025	—	—	—	—	35025	3.64
ctrl35K2	contrôleur	—	790	—	—	—	15358	3.18
ctrl50K1	contrôleur	50478	—	—	—	—	50478	3.63
ctrl50K2	contrôleur	—	1161	—	—	—	22436	3.17
ctrl50K3	contrôleur	—	—	368	—	—	19452	3.02
mix50K3	mixte	—	—	191	—	4	10390	2.76
mix50K4	mixte	—	—	181	—	4	9634	2.67
mix55K5	mixte	—	—	199	—	5	11013	2.79
industA	mixte	6756	—	—	16	16	7564	4.49
industB	data path	15075	—	—	—	—	15333	4.14
industC1	contrôleur	16893	—	—	—	—	17101	4.20
industC2	contrôleur	—	404	—	—	—	7833	2.92
industD1	contrôleur	20780	—	—	—	—	21680	4.05
industD2	contrôleur	—	419	—	—	—	9410	2.75
industE1	contrôleur	22578	—	—	—	—	23807	4.23
industE2	contrôleur	—	560	—	—	—	12862	2.89
rct_mem1	contrôleur	14999	—	—	12	—	15793	4.03
rct_mem2	contrôleur	—	305	—	12	—	8002	2.50
industG1	contrôleur	48836	—	—	64	—	49820	4.54
industG2	contrôleur	—	1011	—	64	—	20152	2.94
industG3	contrôleur	—	—	197	—	—	9257	2.72
industH	microprocesseur	—	—	1019	—	—	41349	2.56
industI	microprocesseur	—	—	1389	—	—	46016	2.78
industK	microprocesseur	—	—	2123	—	—	98914	2.55

Tableau 8.2: *Caractéristiques des netlists utilisées comme benchmark*

Les outils ont été testés de manière indépendante : Nous avons appliqué à un ensemble de *netlists* d'AVCs les différentes combinaisons d'algorithmes dans partBoard; à un ensemble de *netlists* de *clusters* les combinaisons d'algorithmes dans partMeta et à un ensemble de *netlists* de CPBs les combinaisons d'algorithmes dans partCluster ainsi que les combinaisons d'algorithmes dans partMeta.
Les *benchmarks* utilisés pour ces expériences sont présentés dans le tableau 8.2.

Les résultats expérimentaux sont présentés dans les figures 8.9 à 8.16 (voir aussi les tableaux D.4, D.2, D.3 et D.1 en annexe D).

Indépendemment de l'algorithme utilisé dans la phase 3 (multisect* ou MDOC), l'outil partCluster occupe environ le même nombre de *clusters* en moyenne pour le partitionnement des *netlists*.
Grâce à un temps de calcul inférieur, nous choisissons l'algorithme MDOC pour la phase 3 de l'outil partCluster. PartCluster utilise donc pour les phases 1 à 3 les algorithmes bisect*, bisect* et MDOC.

Pour le partitionnement de CPBs en AVCs, les enchaînements d'algorithmes bisect*-bisect*-multisect*, bisect*-bisect*-MDOC et bisect*-hyperFBB-multisect* fournissent de bons résultats pour des *netlists* industrielles. Par contre pour les *netlists* générées, l'enchaînement bisect*-hyperFBB-multisect* nécessite un temps de calcul important, surtout pour les grandes *netlists*.

Pour le partitionnement de *clusters* en AVCs, les enchaînements d'algorithmes bisect*-bisect*-multisect* et bisect*-hyperFBB-multisect* confirment les bons résultats obtenus pour le partitionnement de CPBs en AVCs. L'enchaînement bisect*-bisect*-MDOC utilise par contre plus d'AVCs que les autres enchaînements.
Enfin, nous retenons les enchaînements bisect*-bisect*-multisect*, bisect*-bisect*-MDOC

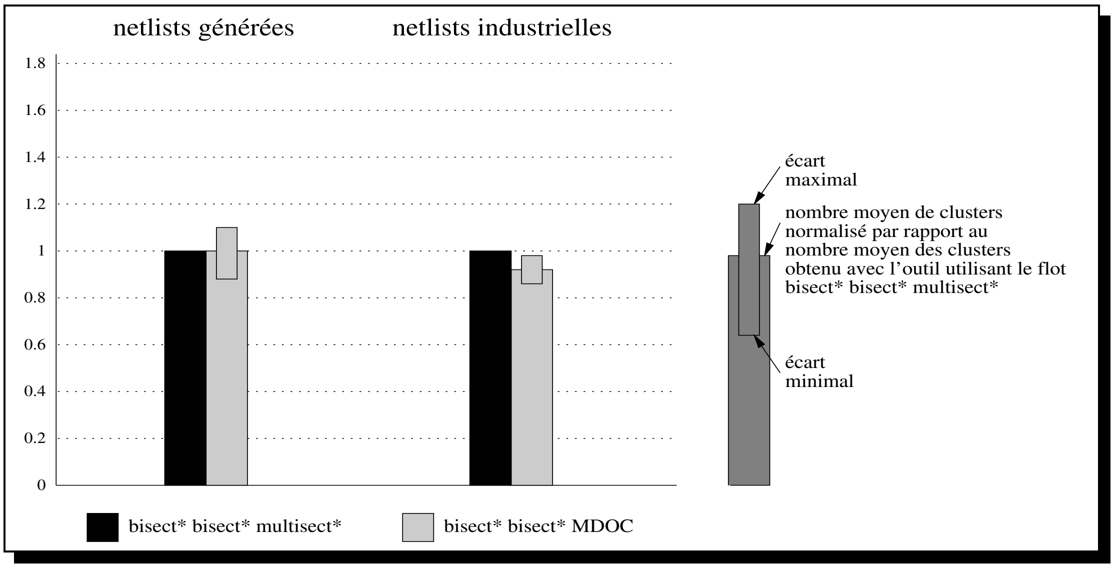


Figure 8.9: *Nombre moyen de clusters créés*

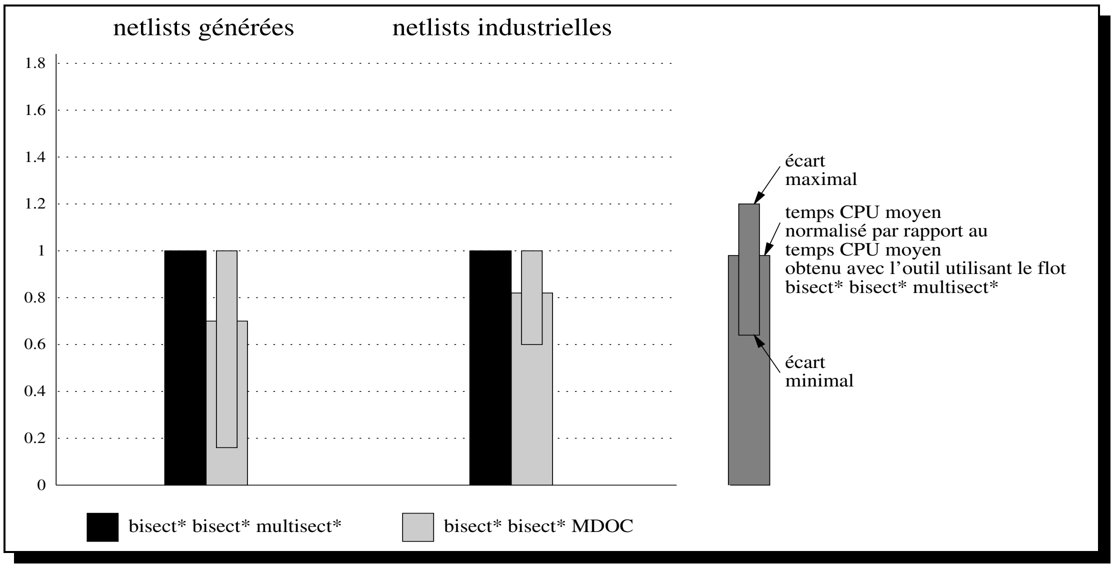


Figure 8.10: *Temps de calcul moyen pour le partitionnement en clusters*

et bisect*-hyperFBB-multisect* pour le partitionnement en AVCs.

La combinaison bisect*-bisect*-multisect* donne le meilleur résultat en terme de nombre d'AVBs utilisés. Les autres combinaisons performantes sont bisect*-bisect*-MDOC, bisect*-hyperFBB-multisect* et hyperFBB-hyperFBB-multisect* pour les phases 1 à 3 de l'outil partBoard. Afin de restreindre le nombre possible de combinaisons, nous avons retenu les trois combinaisons les plus performantes, à savoir bisect*-bisect*-multisect*, bisect*-bisect*-MDOC et hyperFBB-hyperFBB-multisect*, pour une évaluation plus approfondie.

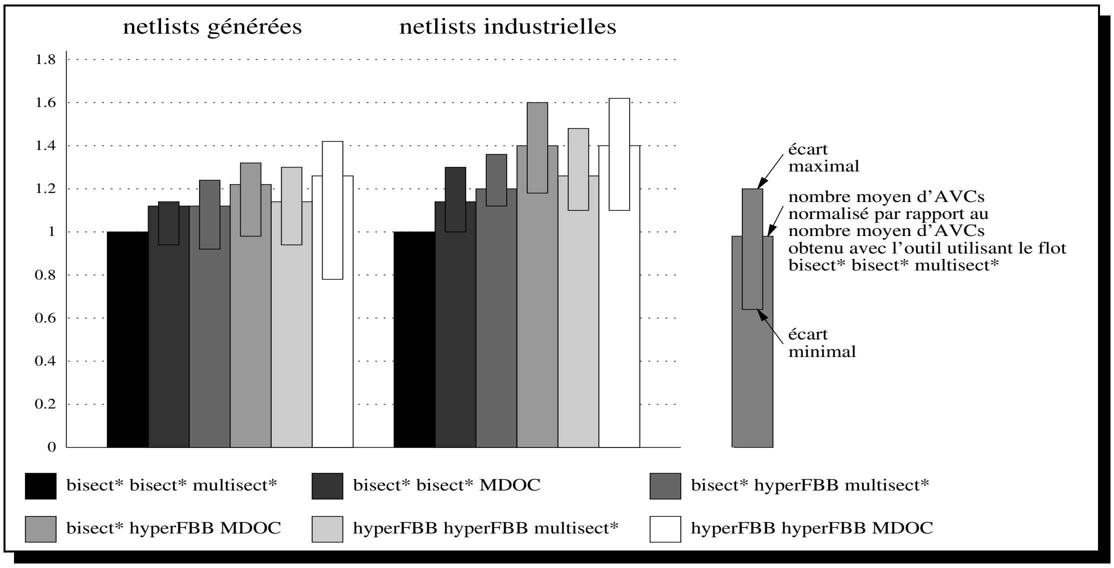


Figure 8.11: *Nombre moyen d'AVCs créés par le partitionnement de CPBs*

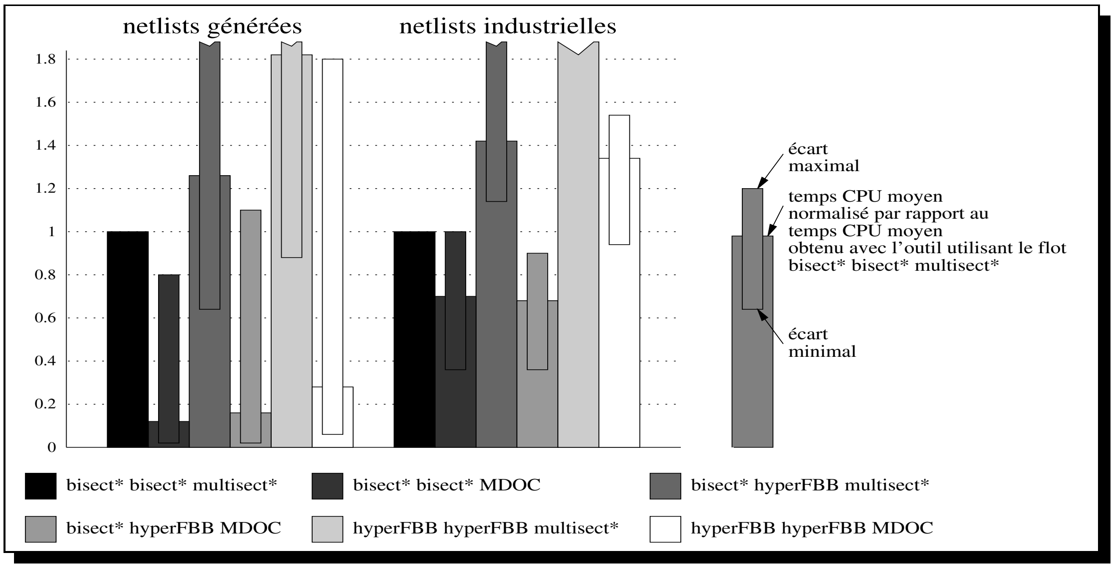


Figure 8.12: *Temps de calcul moyen pour le partitionnement de CPBs en AVCs*

8.2.6 Flots d'application des outils de partitionnement

Il existe théoriquement six possibilités pour implémenter le partitionnement hiérarchique avec les trois outils partBoard, partMeta et partCluster. Deux possibilités consistent à lancer partBoard sur la *netlist* à plat. Pour diverses raisons, cette approche n'est pas adaptée aux problèmes traités, en effet :

- L'objectif des outils de partitionnement actuels est de minimiser le nombre de sous-ensembles des partitions tout en satisfaisant la contrainte que la coupure de chacune des partitions. L'application de partBoard sur une *netlist* de CPBs doit avoir un autre objectif, par exemple : minimiser le nombre de partitions et équilibrer les cardinalités

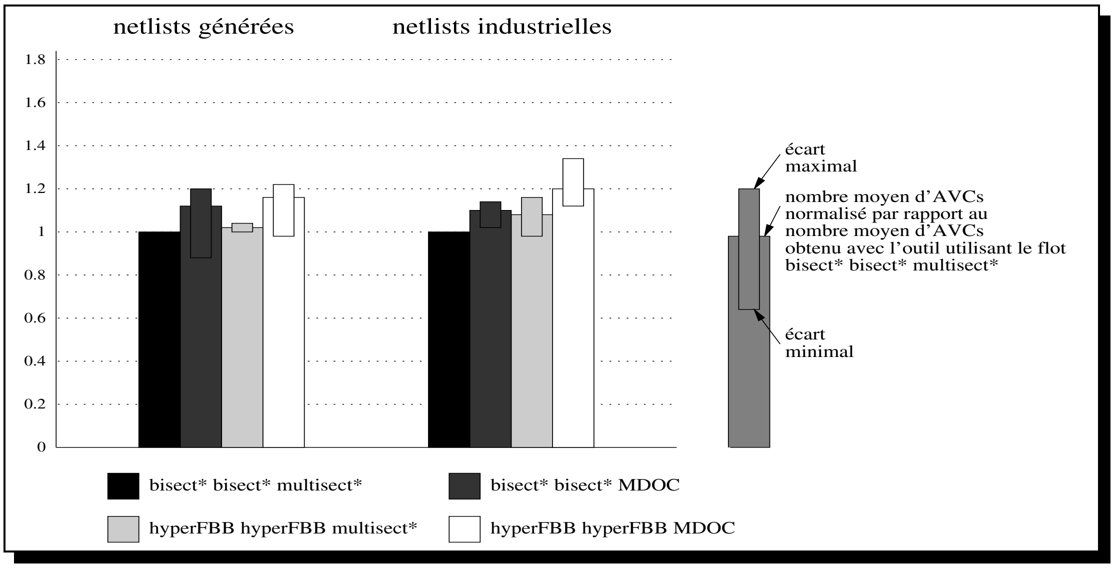


Figure 8.13: *Nombre moyen d'AVCs créés par le partitionnement de clusters en AVCs*

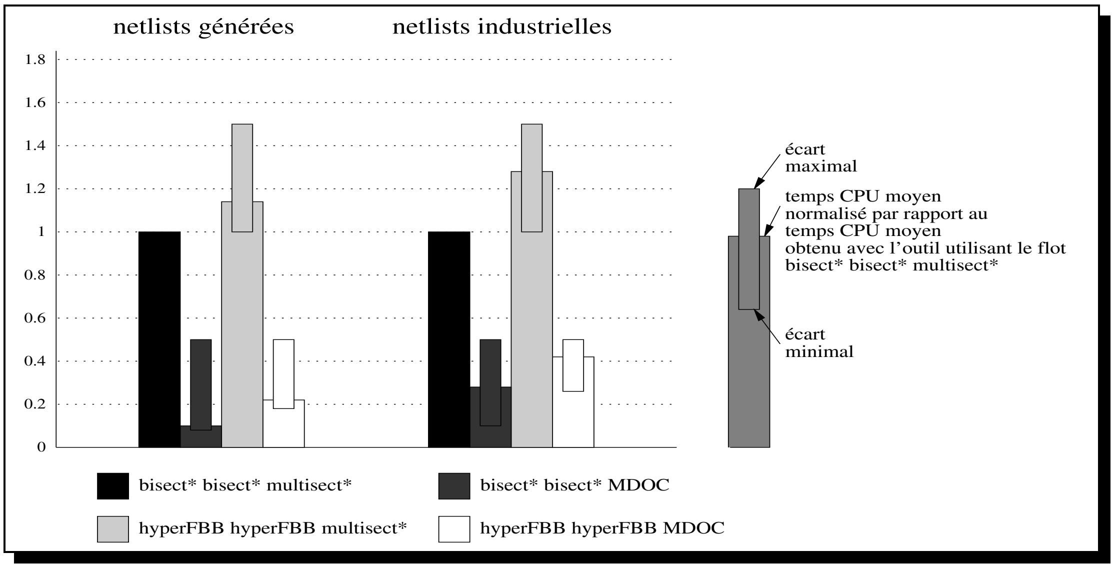


Figure 8.14: *Temps de calcul moyen pour le partitionnement de clusters en AVCs*

entre les partitions de telle sorte que la coupure de chacune des partitions soit respectée. Ainsi, on augmente la probabilité que les partitions obtenues sont valides, c'est- à-dire, que l'on pourra partitionner chacune des partitions en AVCs puis en *clusters*.

- Une partition représentant une carte AVB ne peut pas être partitionnée en AVCs ou un des AVCs ne peut pas être partitionné en *clusters*. Ces cas sont bloquants et il faut recommencer le partitionnement.

- Une partition représentant une carte AVB qui a été partitionnée en AVCs n'est pas routable ou un des *clusters* dans les AVCs n'est pas routable. Ici aussi, il s'agit de cas

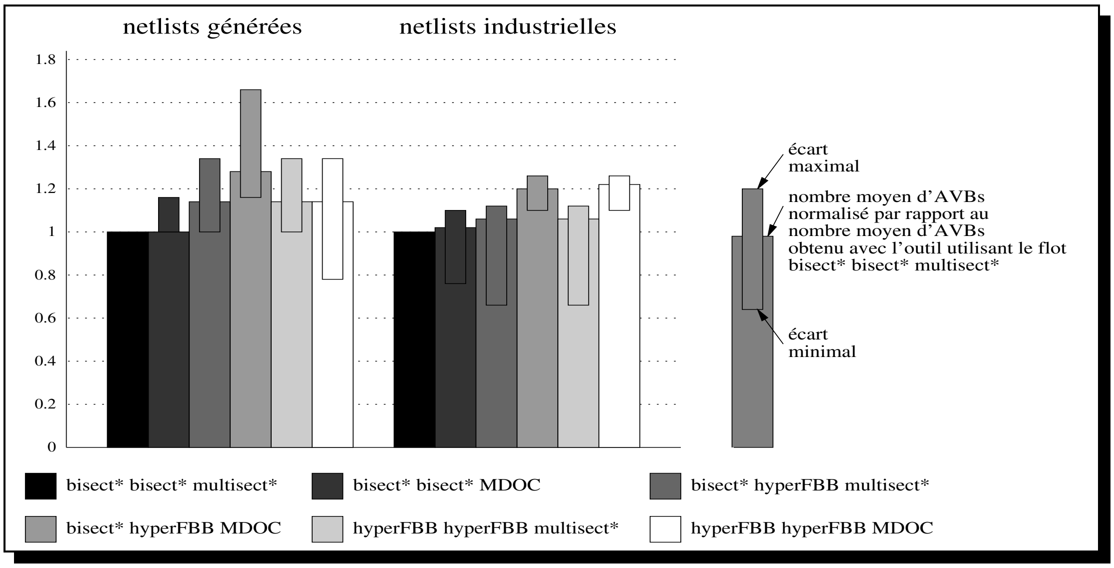


Figure 8.15: *Nombre moyen d'AVBs créés*

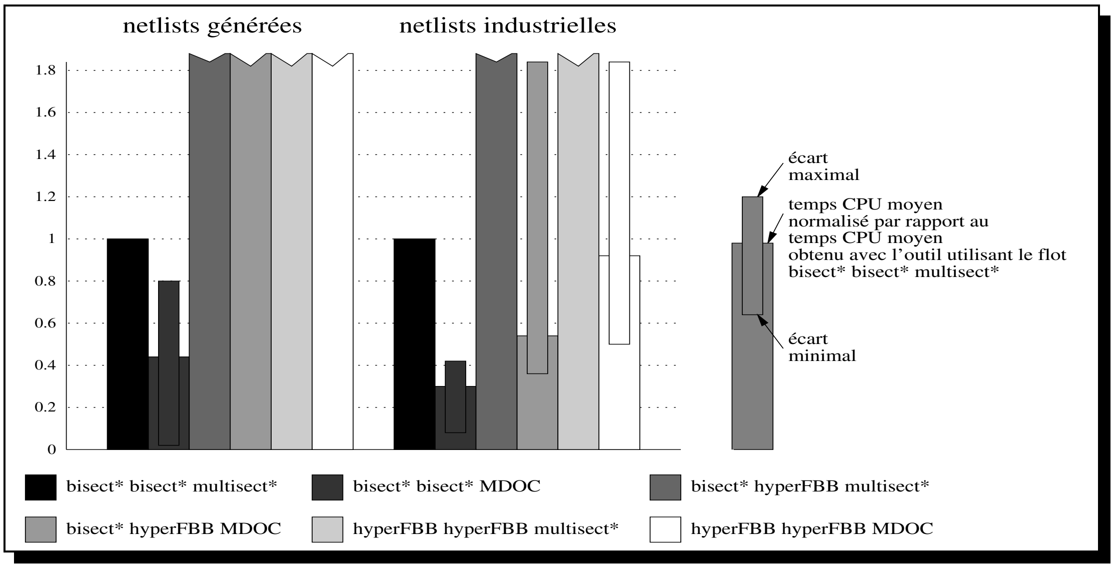


Figure 8.16: *Temps de calcul moyen pour le partitionnement en AVBs*

bloquants et il faut recommencer le partitionnement.

Puisque le routage des AVBs et des *clusters* ainsi que le partitionnement en AVCs représentent les points critiques du partitionnement/routage pour l'émulateur CELARO, l'approche consistant à commencer avec l'outil partBoard sur la *netlist* de CPBs n'a pas été retenue.

Grâce au nombre important de pins par *cluster* par rapport au nombre de pins par AVC, il est assez facile de partitionner un AVCs en *clusters*. Il est donc équivalent pour le résultat du partitionnement en terme de nombre d'AVBs si on applique d'abord partCluster ou partBoard sur la *netlist* d'AVCs obtenue avec partMeta. Nous avons donc décidé de retenir uniquement une des deux approches : partMeta suivi de partCluster suivi de partBoard.

L'approche partCluster suivi de partMeta suivi de partBoard nous semble également prometteuse. Nous avons donc retenu ces deux approches.

8.2.7 Conclusion

L'objectif de cette étude est d'identifier un enchaînement des outils de partitionnement ainsi qu'un enchaînement des algorithmes dans les outils afin d'optimiser le résultat du partitionnement pour l'émulateur CELARO en terme de nombre de cartes AVB. Le nombre de possibilités étant trop grand, nous avons procédé à une présélection en plusieurs étapes :

D'abord, nous avons analysé les performances de chacun des algorithmes pour des contraintes différentes de partitionnement. Les résultats de cette analyse nous ont permis de déterminer quel algorithme peut être employé dans quelle phase d'un outil.

Ensuite, nous avons testé les outils partCluster, partMeta et partBoard en utilisant les enchaînements d'algorithmes que nous avons retenus lors de la précédente étape. Ceci nous a permis de sélectionner un enchaînement d'algorithmes pour partCluster, trois pour partMeta et trois pour partBoard.

Enfin, une analyse des fortes contraintes du partitionnement hiérarchique de la chaîne de compilation de l'émulateur CELARO notamment le partitionnement en AVC, le routage des *clusters* et des cartes AVB, nous a permis d'identifier deux enchaînements des outils qu'il faut analyser plus en détail.

Ainsi, nous avons réduit le nombre de possibilités pour faire des AVBs à partir des CPBs de 34992 à 18. Le fait que le nombre de *clusters* dans un AVC est très petit, entraîne la suppression de la phase de *clustering* lors du partitionnement de *clusters* en AVCs. Le nombre de possibilités pour faire des AVBs à partir des CPBs se réduit ainsi de 18 à 15. Ces 15 possibilités seront examinées plus en détail.

8.3 Evaluation expérimentale du partitionnement pour l'émulateur CELARO

8.3.1 Introduction

Les études présentées dans les sections précédentes ont permis de sélectionner 15 possibilités intéressantes de partitionner une *netlist* de CPBs en AVBs. L'objectif de cette section est l'analyse expérimentale de ces 15 possibilités afin d'identifier le meilleur enchaînement possible des trois outils de partitionnement utilisant au mieux les algorithmes disponibles. Nous comparons nos résultats avec ceux obtenus pour le partitionnement de référence utilisé habituellement. Ce partitionnement enchaîne les outils partMeta, partCluster et partBoard en utilisant les algorithmes de base bisect-bisect-multisect pour les trois phases de chacun des trois outils.

8.3.2 Banc de test

L'évaluation des performances des différents enchaînements d'outils a été effectué en utilisant 10 *netlists* aléatoires produites avec PartGen et 10 *netlists* industrielles. Les *netlists* aléatoires sont réalisées dans un seul module. Pour cette raison et parce que les *netlists* mono-modules de grande taille nécessitent un temps de calcul important, nous avons limité le nombre de CPBs de ces *netlists* à environ 100K.

Les *netlists* industrielles contiennent plusieurs modules avec au moins une instance. Le partitionnement en AVCs effectué pour chacune des instances est plus rapide que celui pour une *netlist* mise à plat. Nous avons donc sélectionné des *netlists* industrielles de plus grande taille.

Les caractéristiques des *benchmarks* utilisés pour ces expériences ainsi que les résultats de partitionnement obtenus pour l'application du partitionnement de référence sont présentés dans le tableau 8.3.

Circuit	type de circuit	#CPBs	#XMEM	#SRAM	#nets	#AVBs	t_CPU
ctrl50K	contrôleur	50478	0	0	50478	18	1496'
mix50K4	mixte	31574	6	4	31896	8	153'
mix55K2	mixte	33626	15	3	34204	12	392'
mix80K	mixte	70167	20	0	70808	22	1298'
mix90K	mixte	65302	40	0	66583	22	1342'
mix100K	mixte	89339	20	0	89980	23	1358'
mix110K	mixte	100912	15	0	101393	36	2812'
mix120K	mixte	94441	33	0	95498	39	1796'
mix130K	mixte	93324	54	0	95053	37	1962'
mix140K	mixte	100723	60	0	102644	38	1471'
Moyenne						26	1408'
indust1	microprocesseur	16375	0	7	16484	5	62'
indust2	microprocesseur	67584	98	23	70330	14	209'
indust3	microprocesseur	109929	176	4	115074	21	359'
indust4	microprocesseur	86747	460	0	95114	18	32'
indust5	microprocesseur	167433	295	30	175435	27	639'
indust6	microprocesseur	173633	313	0	176936	39	727'
indust7	microprocesseur	197025	504	0	209183	70	959'
indust8	microprocesseur	163985	283	4	173860	60	1489'
indust9	microprocesseur	492284	450	163	516726	89	1461'
indust10	microprocesseur	689974	994	526	784845	236	3512'
Moyenne						58	945'

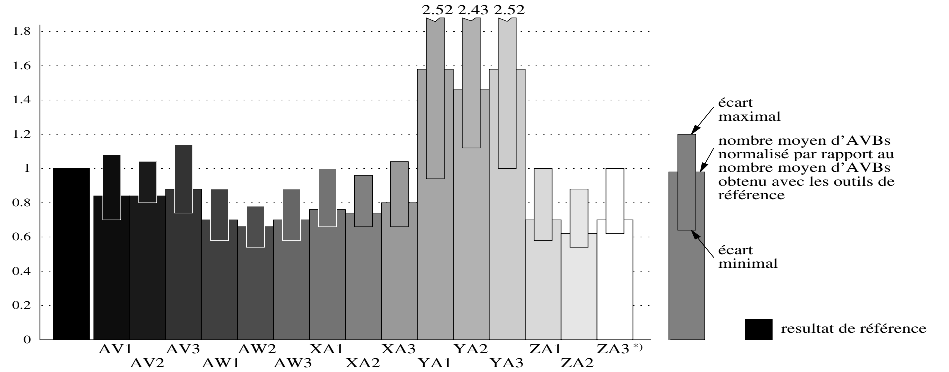
Tableau 8.3: *Caractéristiques des netlists utilisées comme benchmark*

8.3.3 Résultats expérimentaux

Nous avons testé deux enchaînements différents d'outils de partitionnement : partCluster suivi de partMeta et de partBoard ainsi que partMeta suivi de partCluster et partBoard. L'outil partCluster a été réalisé avec la combinaison d'algorithmes bisect*-bisect*-MDOC (A). L'outil partMeta utilise cinq combinaisons différentes d'algorithmes : bisect*-multisect* (V), bisect*-MDOC (W), bisect*-bisect*-multisect* (X), bisect*-hyperFBB-multisect* (Y) et bisect*-bisect*-MDOC (Z). L'outil partBoard a été réalisé avec les combinaisons d'algorithmes suivants : bisect*-bisect*-multisect* (1), bisect*-bisect*-MDOC (2) et hyperFBB-hyperFBB-multisect* (3).

Les résultats expérimentaux en terme de nombre moyen d'AVBs nécessaires à l'implémentation d'une *netlists* sont présenté dans la figure 8.17 pour les *netlists* générées et dans la figure 8.19 pour les *netlists* industrielles. Les résultats obtenus pour les différents enchaînements d'outils utilisant des algorithmes différents sont normalisés par rapport au nombre moyen d'AVBs déterminé avec le partitionnement de référence utilisant les outils de référence.

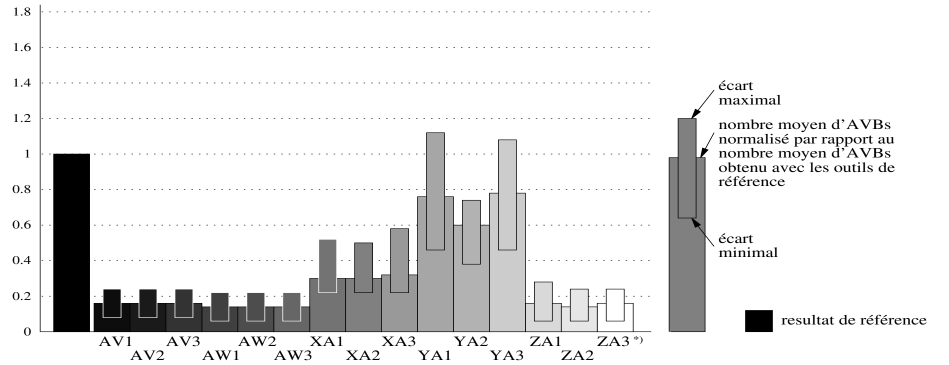
Les résultats expérimentaux en terme de temps de calcul sont illustrés dans la figure 8.18 pour les *netlists* générées et dans la figure 8.20 pour les *netlists* industrielles. Les résultats



	partCluster	partMeta					partBoard		
abréviation	A	V	W	X	Y	Z	1	2	3
phase 1	bisect*	bisect*	bisect*	bisect*	bisect*	bisect*	bisect*	bisect*	hyperFBB
phase 2	bisect*	——	——	bisect*	hyperFBB	bisect*	bisect*	bisect*	hyperFBB
phase 3	MDOC	multisect*	MDOC	multisect*	multisect*	MDOC	multisect*	MDOC	multisect*

) ZA3 : le flot du partitionnement est : Z = partMeta (bisect, bisect*, MDOC), A = partCluster (bisect*, bisect*, MDOC), 3 = partBoard (hyperFBB, hyperFBB, multisect*).

Figure 8.17: *Nombre moyen d'AVBs nécessaire à l'implémentation d'une netlist générée*



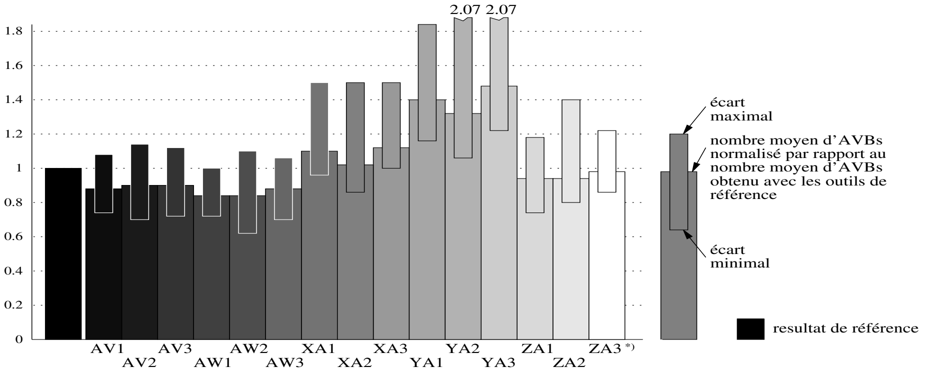
	partCluster	partMeta					partBoard		
abréviation	A	V	W	X	Y	Z	1	2	3
phase 1	bisect*	bisect*	bisect*	bisect*	bisect*	bisect*	bisect*	bisect*	hyperFBB
phase 2	bisect*	——	——	bisect*	hyperFBB	bisect*	bisect*	bisect*	hyperFBB
phase 3	MDOC	multisect*	MDOC	multisect*	multisect*	MDOC	multisect*	MDOC	multisect*

) ZA3 : le flot du partitionnement est : Z = partMeta (bisect, bisect*, MDOC), A = partCluster (bisect*, bisect*, MDOC), 3 = partBoard (hyperFBB, hyperFBB, multisect*).

Figure 8.18: *Temps CPU moyen nécessaire pour le partitionnement d'une netlist générée*

détaillés sont présentés dans les tableaux E.1, E.2, E.3, E.4 et E.5 en annexe E.

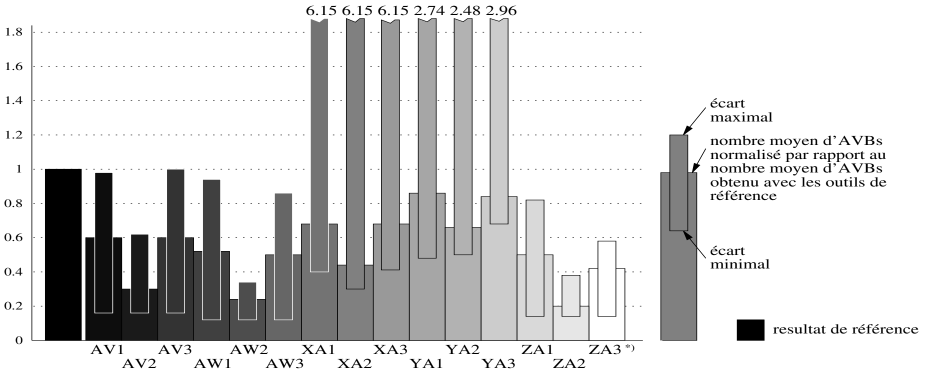
Les meilleurs résultats pour les *netlists* générées ont été obtenus par le flot partMeta suivi de partCluster et partBoard en utilisant les algorithmes bisect*-bisect*-MDOC dans chacun des outils. Le nombre moyen d'AVBs nécessaire à l'implémentation de la *netlist* passe de 26 à 16 ce qui représente une amélioration très significative de 38%. Parallèlement, le temps de compilation est divisé par 6.7. Le flot partCluster suivi de partMeta et partBoard fournit approximativement les mêmes résultats en utilisant également les algorithmes bisect*-bisect*-MDOC dans les trois outils. D'une manière générale, toutes les approches testées permettent d'améliorer le résultat obtenu avec l'approche de référence. La seule contre-performance a été constatée pour le flot utilisant l'outil partMeta avec les algorithmes



	partCluster	partMeta					partBoard		
abréviation	A	V	W	X	Y	Z	1	2	3
phase 1	bisect*	bisect*	bisect*	bisect*	bisect*	bisect*	bisect*	bisect*	hyperFBB
phase 2	bisect*	——	——	bisect*	hyperFBB	bisect*	bisect*	bisect*	hyperFBB
phase 3	MDOC	multisect*	MDOC	multisect*	multisect*	MDOC	multisect*	MDOC	multisect*

) ZA3 : le flot du partitionnement est : Z = partMeta (bisect, bisect*, MDOC), A = partCluster (bisect*, bisect*, MDOC), 3 = partBoard (hyperFBB, hyperFBB, multisect*).

Figure 8.19: *Nombre moyen d'AVBs nécessaire à l'implémentation d'une netlist industrielle*



	partCluster	partMeta					partBoard		
abréviation	A	V	W	X	Y	Z	1	2	3
phase 1	bisect*	bisect*	bisect*	bisect*	bisect*	bisect*	bisect*	bisect*	hyperFBB
phase 2	bisect*	——	——	bisect*	hyperFBB	bisect*	bisect*	bisect*	hyperFBB
phase 3	MDOC	multisect*	MDOC	multisect*	multisect*	MDOC	multisect*	MDOC	multisect*

) ZA3 : le flot du partitionnement est : Z = partMeta (bisect, bisect*, MDOC), A = partCluster (bisect*, bisect*, MDOC), 3 = partBoard (hyperFBB, hyperFBB, multisect*).

Figure 8.20: *Temps CPU moyen nécessaire pour le partitionnement d'une netlist industrielle*

bisect*-hyperFBB-multisect*. Celle-ci se traduit par un nombre élevé d'AVCs (12% de plus par rapport au partMeta qui utilise les algorithmes bisect*-bisect*-multisect*) causé par l'algorithme hyperFBB. Cet algorithme n'est pas en mesure de produire un *clustering* qui mène par la suite à un petit nombre d'AVCs avec une petite coupure moyenne.
Notons également l'influence réduite de l'outil partBoard sur le résultat final. En effet, le résultat en terme de nombre d'AVBs varie peu pour un partitionnement en AVCs donné.

Les meilleurs résultats pour les *netlists* industrielles ont été obtenus avec le flot partCluster suivi de partMeta et partBoard en utilisons les algorithmes bisect*-bisect*-MDOC dans partCluster et partMeta et soit les mêmes algorithmes dans partBoard soit les algorithmes bisect*-bisect*-multisect*. Le nombre moyen d'AVBs nécessaire à l'implémentation

d'une *netlist* passe de 58 AVBs en moyenne à 49 AVBs en moyenne, ce qui représente une amélioration de 15%.
Le temps de calcul pour ces combinaisons est en moyenne de 221 minutes (resp. 496 minutes) par rapport à 945 minutes en moyenne pour le partitionnement de référence, ce qui représente une réduction de 76% (resp. 47%).
Le meilleur temps de calcul par contre a été obtenu pour l'enchaînement des outils partMeta et partCluster suivi de partBoard en utilisant les algorithmes bisect*-bisect*-MDOC pour chacun des trois outils : il est de 181 minutes en moyenne, ce qui correspond à une division par 5.2 ou une réduction de 81%. Comparé avec le partitionnement de référence, le nombre moyen d'AVBs nécessaires pour implémenter une *netlist* industrielle est de 7% inférieur.

8.3.4 Conclusion

L'objectif de cette étude expérimentale est l'identification d'un enchaînement des outils de partitionnement utilisant au mieux les différents algorithmes disponibles pour optimiser le résultat du partitionnement pour l'émulateur CELARO.
Cette étude a été réalisée à l'aide d'un banc de test contenant 10 *netlists* aléatoires et 10 *netlists* industrielles. Ces *netlists* ont toutes les mêmes caractéristiques en taille, hiérarchie et structure que les *netlists* implémentées habituellement sur l'émulateur CELARO.
Nous avons appliqué les différents flots des outils permettant de partitionner une *netlist* de CPBs en AVBs aux *netlists* du banc de test.

Les résultats expérimentaux montrent une amélioration significative tant au niveau du nombre d'AVBs nécessaires à l'implémentation de la *netlist* qu'au niveau du temps de calcul. En effet, nous passons de 26 AVBs en moyenne à 16 AVBs et d'un temps de calcul de 1408 minutes à 210 minutes en moyenne pour les *netlists* générées, ce qui représente une amélioration de 38% du nombre d'AVBs et de 85% du temps de calcul.
En même temps, nous améliorons de 15% le nombre d'AVBs et de 76% le temps de calcul sur les *netlists* industrielles en passant de 58 AVBs en moyenne à 49 AVBs et d'un temps de calcul de 946 minutes à 221 minutes en moyenne.

8.4 Analyse des résultats

8.4.1 Amélioration du taux de remplissage

L'amélioration du taux de remplissage, ce qui se traduit par le nombre d'AVBs nécessaires à l'implémentation d'une *netlist* était l'objectif principal de ce travail. Pour parvenir à réaliser cet objectif, nous avons utilisé de nouveaux algorithmes et amélioré les algorithmes existants, fait varier les combinaisons de ces algorithmes dans les outils et testé plusieurs enchaînements de ces outils.

Ces modifications apportées conduisent finalement à une réduction significative du nombre d'AVBs comme présenté dans section 8.3.3. Le nombre d'AVBs a pu être réduit de 38% pour le partitionnement des *netlists* générées par rapport à l'approche de référence.

8.4.2 Amélioration du temps de calcul

L'amélioration du temps de calcul était également un objectif important de ce travail. L'utilisation d'algorithmes avec de faibles complexités a été l'approche choisie pour ce travail.

Les différentes modifications apportées et notamment l'utilisation de l'algorithme MDOC au lieu de l'algorithme multisect*, ont contribué à une diminution importante du temps de calcul. Celui-ci est divisé par 6.7 par rapport à l'approche de référence.

8.4.3 Amélioration de la fréquence

L'amélioration de la fréquence du *design* implémenté sur l'émulateur est un des objectifs de ce travail. La meilleure méthode au niveau du partitionnement est de diminuer le nombre de coupures des chemins combinatoires, c'est–à–dire le nombre de fois qu'un chemin combinatoire traverse la limite d'une partition. Après une étude approfondie des différents algorithmes visant à effectuer un partitionnement le long de ces chemins, l'algorithme de Brasen et Saucier [Bra94] nous semblait être le plus intéressant. Une description détaillée de cet algorithme se trouve en annexe B.

Malheureusement, l'algorithme n'est pas directement applicable dans le cas du partitionnement avec des contraintes de cardinalité et de coupure ciblant à minimiser le nombre de partitions créées. L'amélioration de la fréquence du *design* implémenté reste donc un domaine ouvert à la recherche.

8.4.4 Conclusion

Dans un premier temps, nous avons comparé les performances des différents algorithmes décrits dans les chapitres 4 à 7 pour un partitionnement avec diverses contraintes de cardinalité et de coupure. Cette comparaison nous a permis d'identifier les avantages et inconvénients de chacun d'entre eux. Sur la base de cette analyse, nous avons réalisé des combinaisons d'algorithmes constituant différentes versions des outils de partitionnement.

Les performances de ces versions d'outils ont été comparées afin d'identifier celles qui optimisent les résultats expérimentaux. Ces versions ont été utilisées par la suite lors de la comparaison expérimentale des différents enchaînements d'outils.

Cette étude nous a permis de déterminer l'enchaînement adapté et la meilleure combinaison possible des outils de partitionnement.
En comparaison avec l'approche de référence, nous avons pu diminuer le nombre moyen d'AVBs nécessaires à l'implémentation des *netlists* générés de 38% et des *netlists* industrielles de 15%. Parallèlement, le temps de calcul moyen est réduit de 85% pour les *netlists* générées et de 76% pour les *netlists* industrielles.

L'amélioration de la fréquence d'émulation est un objectif qui n'a pas pu être atteint, l'algorithme choisi n'ayant pas les caractéristiques requises pour une application dans le domaine de l'émulation.

Chapitre 9

Conclusion

Ce travail a été essentiellement motivé par la difficulté à partitionner des *netlists* de très grande dimension (de taille supérieure à 1 million de portes) pour une implémentation sur une architecture hiérarchique comme celle de l'émulateur CELARO de Mentor Graphics. En effet, le partitionnement d'une *netlist* pour son implémentation sur une architecture hiérarchique poursuit des objectifs particuliers qui se différencient significativement des objectifs classiques du partitionnement tels que la minimisation de la coupure entre les partitions. L'objectif principal de notre partitionnement spécifique est de minimiser le nombre d'éléments nécessaires sur le plus haut niveau de l'architecture afin de minimiser le besoin en ressources physiques.
De plus, le temps nécessaire pour l'implémentation d'une *netlist* dans un émulateur doit être optimisé afin d'éviter à l'utilisateur des attentes trop longues. Pour cette même raison, la fréquence de l'émulation doit être maximale.

Nous avons effectué une étude comparative d'algorithmes de partitionnement dédiés à l'implémentation d'une *netlist* sur une architecture composée de plusieurs niveaux hiérarchiques. Les résultats de cette étude ont été ensuite utilisés pour améliorer les outils de partitionnement existants.

Une étude bibliographique approfondie nous a permis d'identifier un ensemble d'algorithmes intéressants. En particulier, nous avons sélectionné un algorithme de bipartitionnement ayant pour objectif de minimiser la coupure entre les deux partitions, un algorithme de multipartitionnement qui minimise le nombre de partitions en respectant les contraintes de cardinalité maximale et de coupure maximale et un algorithme de *clustering* permettant de conserver les chemins combinatoires à l'intérieur des partitions afin d'obtenir une fréquence d'émulation plus élevée.

Lors de la construction d'un banc de test, nous avons constaté une carence en *netlists* de test ayant des caractéristiques semblables à celles des *netlists* traitées habituellement dans le domaine de l'émulation. Cette carence nous a amené à développer un générateur de *netlists* paramétrable permettant de produire artificiellement des *netlists* de tailles et de structures très variées tout en reproduisant les caractéristiques statistiques et structurelles présentes dans des *netlists* réelles. Le travail effectué dans ce domaine a fait l'objet d'une publication (cf. [Pis99a]).

Des *netlists* créées avec ce générateur ainsi que des *netlists* industrielles ont été utilisées par la suite pour évaluer les performances des différents algorithmes. Les résultats de ces expériences nous ont permis d'apporter plusieurs améliorations significatives aux algorithmes étudiés. En particulier, nous avons :

- développé un algorithme de *clustering* utilisé en prétraitement à l'algorithme bisect,
- étudié l'influence du bipartitionnement initial sur le résultat de partitionnement de bisect en concluant que le bipartitionnement initial aléatoire fournit le meilleur résultat,
- introduit une nouvelle fonction de coût pour l'algorithme bisect permettant un bipartitionnement plus équilibré que la fonction du *ratio-cut* tout en conservant approximativement la même coupure,
- analysé des fonctions de coût différentes pour l'algorithme multisect permettant de réduire le nombre de sous-ensembles dans une partition tout en respectant des contraintes de cardinalité maximale et de coupure pour chacun d'entre eux,
- développé un algorithme permettant le marquage directement sur l'hypergraphe lors du calcul du flot maximal dans l'algorithme hyperFBB. Cet algorithme a été soumis à publication (cf. [Pis99b]),
- développé un algorithme permettant de déterminer un ordonnancement linéaire des sommets d'un graphe,
- adapté les algorithmes window et DP-RP aux besoins du partitionnement pour des architectures hiérarchiques permettant de minimiser le nombre de sous-ensembles d'une partition tout en respectant les contraintes de cardinalité maximale et de coupure.

Nous avons comparé les performances de ces algorithmes lors d'un partitionnement avec différentes contraintes de cardinalité et de coupure, l'objectif étant de déterminer les combinaisons des algorithmes à utiliser pour chaque outil de partitionnement.

Une étude comparative préliminaire de ces différentes versions d'outils nous a permis de mettre en évidence les plus performantes. Ces versions ont été enchaînées de diverses façons afin d'effectuer le partitionnement sur tous les niveaux de l'architecture hiérarchique. Ainsi, nous avons testé de multiples possibilités de partitionner une *netlist* de CPBs en AVBs.

Les résultats expérimentaux pour ces différents enchaînements d'outils de partitionnement montrent que le nombre de cartes AVBs nécessaires à l'implémentation d'une *netlist* a pu être réduit de 15 à 38% en moyenne pour les *netlists* du banc de test. Parallèlement, le temps de calcul a été diminué jusqu'à 85% pour ces mêmes *netlists*. Ces résultats confirment la pertinence et l'efficacité de la méthodologie et de l'ensemble des outils développés au cours de cette thèse.

Après l'amélioration significative de la densité et du temps de calcul pour le partitionnement en vue d'une implémentation sur une architecture hiérarchique, un des axes de recherche future concernera l'amélioration de la fréquence de fonctionnement. Nous avons vu dans chapitre 8 section 8.2.4 qu'une première approche envisagée, basée sur la construction de tous les cônes de logique à la fois, s'est révélée insuffisante. Néanmoins, d'autres idées devront être étudiées, par exemple la construction successive de cônes de logique en combinaison avec une duplication des cellules sur les intersections de cônes.

A plus long terme, il serait intéressant d'envisager une approche par partitionnement adaptatif fondé sur l'analyse des caractéristiques (statistiques, topologiques) de la *netlist*. Certains des résultats obtenus dans notre travail (cf. entre autres chapitre 7 section 7.4.3 et annexe C.4) peuvent constituer un point de départ intéressant pour une telle étude.

Glossaire

– A –

accelerated verification	Méthode permettant la vérification fonctionnelle d'un circuit intégré en utilisant l'implémentation matérielle de son équivalent logique.
ANF	Format de *netlist* utilisé par CELARO.
arc	*directed edge.* Couple ordonné de sommets dans un graphe. Si $e = (v_i, v_j)$ est un arc dans le graphe, alors v_i est l'extrémité initiale de e et v_j l'extrémité terminale de e.
arête	*edge.* Couple non-ordonné de sommets dans un graphe.
AutoLogic II	Ensemble d'outils de Mentor Graphics pour la synthèse logique.
AVB	*Accelerated Verification Board.* Carte utilisateur contenant entre 40 et 50 AVCs.
AVC	*Accelerated Verification Chip.* Circuit dédié à la vérification accélérée.

– B –

bipartitionnement	Division d'un ensemble en deux sous-ensembles tels qu'une fonction de coût soit optimisée.
bissection	Division d'un ensemble en deux sous-ensembles de même taille tels qu'une fonction de coût soit optimisée.
BLIF	*Berkeley Logic Interchange Format.* Langage permettant de décrire un circuit au niveau logique.
BLP	Bloc Logique Programmable. Elément de base de l'architecture Meta I pouvant réaliser toute fonction logique à quatre entrées et une sortie. La sortie peut être temporisée dans une *flip-flop.*

– C –

Cabinet	Ensemble de unités. Un cabinet peut contenir jusqu'à quelques dizaines de cartes utilisateur.

chaîne augmentante . Une chaîne augmentante d'un sommet v_i vers un sommet v_j dans un graphe est une chaîne élémentaire d'extrémité v_i et v_j telle que le flot sur les arcs dans le sens du parcours est inférieur aux capacités de ces arcs et que le flot sur les arcs dans le sens opposé du parcours est positif.

chemin augmentant *augmenting path.* Chaîne augmentante entre deux sommets v_i et v_j dont tous les arcs sont orientés dans le même sens de v_i vers v_j.

CLB *Configurable Logic Block.* Groupement d'éléments logiques dans une architecture FPGA.

Cluster Ensemble de quelques dizaines de BLPs avec autant d'entrées et de sorties. Chaque AVC contient entre 10 et 20 de ces clusters.

Clustering Technique permettant d'identifier des éléments fortement connexes avec peu de connexions vers l'extérieur.

CPB *Configurable Physical Block.* Elément de base de l'architecture Meta II pouvant réaliser toute fonction logique à plusieurs entrées et une sortie. La sortie peut être temporisée dans une *flip-flop*.

– F –

fan Nombre de connecteurs d'un *net*.

fanout Nombre de connecteurs pilotés par un *net*.

fichier.anf Fichier contenant la description d'une *netlist* en format ANF.

fichier.blp.anf Fichier contenant la description d'une *netlist* de BLPs. C'est le format d'entrée du partitionnement.

FPGA *Field Programmable Gate Array.* Circuit contenant une matrice de CLBs, des blocs d'entrée/sortie configurables et des interconnexions entre blocs configurables.

– G –

graphe *directed graph.* Un graphe $G = (V, E)$ est déterminé par la donnée d'un ensemble V dont les éléments sont appelés des sommets et d'un ensemble E dont les éléments sont appelés des arcs.

graphe non-orienté *undirected graph.* Un graphe non-orienté $G = (V, E)$ est déterminé par la donnée d'un ensemble V dont les éléments sont appelés des sommets et d'un ensemble E dont les éléments sont appelés des arêtes.

– H –

heuristique Méthode réalisant le choix dans un algorithme non-déterministe.

hyperarc *directed hyperedge, net.* Etant donné un hypergraphe $H(V, E)$. Un hyperarc $e(\{v_i\}, \{v_j, v_k\}) \in E$ est déterminé par la donnée d'un couple ordonné formé d'un ensemble de sommets $V_S = \{v_i\} \subseteq V$ appelé l'extrémité initiale de e et d'un ensemble de sommets $V_T = \{v_j, v_k\} \subseteq V$ appelé l'extrémité terminale de e.

hyperarête *hyperedge.* Etant donné un hypergraphe $H(V, E)$. Une hyperarête $e(v_i, v_j, v_k) \in E$ est déterminée par la donnée d'un ensemble de sommets $V_S \subseteq V$.

hypergraphe *hypergraph.* Un hypergraphe $H = (V, E)$ est déterminé par la donnée d'un ensemble V dont les éléments sont appelés des sommets et d'un ensemble E dont les éléments sont appelés des hyperarêtes.

hypergraphe orienté *directed hypergraph.* Un hypergraphe orienté $H = (V, E)$ est déterminé par la donnée d'un ensemble V dont les éléments sont appelés des sommets et d'un ensemble E dont les éléments sont des couples ordonnés de sous-ensembles de sommets appelés hyperarcs.

– I –

IOB *Input/Output Block.* Bloc logique programmable qui peut être configuré en entrée, en sortie, ou bidirectionnel.

– L –

LUT *Look-up Table.* Une LUT de n-entrées est une mémoire qui peut implémenter n'importe quelle fonction booléenne de n variables.

– M –

MCM *Multichip module.* Implémentation compact de plusieurs *chips* de petite taille sur un même substrat.

mémoire intégrée Mémoire réalisée en meta_xmem.

mémoire isolée Mémoire réalisée en bancs de meta_sram_bank.

meta.env Fichier de configuration pour la compilation d'une *netlist.*

meta_xmem Mémoire multi-ports intégrée aux AVCs.

meta_sram_bank Mémoire mono-port sur des cartes mémoires.

multipartitionnement Division d'un ensemble en plusieurs sous-ensembles tels qu'une fonction de coût soit optimisée.

multisection Division d'un ensemble en plusieurs sous-ensembles de même taille tels qu'une fonction de coût soit optimisée.

– P –

PCB	*Printed circuit board.* Carte imprimé utilisée comme support pour des composants électronique et permettant leurs interconnexions.
PI	*Primary input.* Entrée primaire d'une *netlist.*
PO	*Primary output.* Sortie primaire d'une *netlist.*

Bibliographie

[Alp94] C.J. Alpert et A.B. Kahng
"Multi-Way Partitioning Via Spacefilling Curves and Dynamic Programming"
In *Proc. 31st ACM/IEEE Design Automation Conference*, pp. 652-657, 1994.

[Alp95a] C.J. Alpert et A.B. Kahng
"Recent Directions in Netlist Partitioning: A Survey"
In *Integration: The VLSI Journal 19(1-2)*, pp. 1-81, 1995.

[Alp95b] C.J. Alpert et S.Z. Yao
"Spectral Partitioning: The More Eigenvectors, The Better"
In *Proc. 32nd ACM/IEEE Design Automation Conference*, pp. 195-200, 1995.

[Alp95c] C.J. Alpert et A.B. Kahng
"Multi-Way Partitioning Via Geometric Embeddings, Orderings, and Dynamic Programming"
In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 14(11)*, pp. 1342-1358, 1995.

[Alp96] C.J. Alpert et A.B. Kahng
"A General Framework for Vertex Orderings with Applications to Circuit Clustering"
In *IEEE Transactions on VLSI Systems 4(2)*, pp. 240-246, 1996.

[Alp97] C.J. Alpert, J.H. Huang et A.B. Kahng
"Multilevel Circuit Partitioning"
In *Proc. 34th ACM/IEEE Design Automation Conference*, pp. 530-533, 1997.

[Alp98] C.J. Alpert
"The ISPD98 Circuit Benchmark Suite"
In *Proc. ACM International Symposium on Physical Design*, pp. 85-90, 1998.

[Bea93a] D. Beasley, D.R. Bull et R.R. Martin
"An Overview of Genetic Algorithms: Part 1, Fundamentals"
In *University Computing 15(2)*, pp. 58-69, 1993.

[Bea93b] D. Beasley, D.R. Bull et R.R. Martin
"An Overview of Genetic Algorithms: Part 2, Research Topics"
In *University Computing 15(4)*, pp. 170-181, 1993.

[Beh96a] D. Behrens, K. Harbich et E. Barke
"Hierarchical Partitioning"
In *Proc. IEEE International Conference on Computer-Aided Design*, pp. 470-477, 1996.

[Beh96b] D. Behrens, K. Harbich et E. Barke
"Circuit Partitioning using High Level Design Information"
In *Proc. IDPT'96: 2nd World Conference on Integrated Design and Process Technology*, pp. 259-266, 1996.

[Ber83] C. Berge
Graphes, 3e édition
Bordas, 1983.

[Bra94] D. Brasen et G. Saucier
"FPGA Partitioning For Critical Paths"
In *Proc. IEEE European Design and Test Conference*, pp. 99-103, 1994.

[Bra95] D. Brasen
"Prototypage Automatique d'ASIC Sur FPGAs"
Thèse de l'Institut National Polytechnique de Grenoble, 1995.

[Bra85] R. Brayton, G. Hachtel, C. McMullen et A. Sangiovanni-Vincentelli
Logic Minimization Algorithms for VLSI Synthesis
Kluwer Academic, 1985.

[Bui94] T.N. Bui et B.M. Moon
"A Fast and Stable Hybrid Genetic Algorithm for the Ratio-Cut Partitioning Problem on Hypergraphs"
In *Proc. 31st ACM/IEEE Design Automation Conference*, pp. 664-669, 1994.

[Bui98] T.N. Bui et B.M. Moon
"GRCA: A Hybrid Genetic Algorithm for Circuit Ratio-Cut Partitioning"
In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 17(3)*, pp. 193-204, 1998.

[But91] M.R. Butts et J.A. Batcheller
Method of Using Electronically Reconfigurable Logic Circuits
United States Patent N^o 5036473, 1991.

[Cal99a] A.E. Caldwell, A.B. Kahng, A.A. Kennings et I.L. Markov
"Hypergraph Partitioning for VLSI CAD: Methodology for Heuristic Development, Experimenation and Reporting"
In *Proc. 36th ACM/IEEE Design Automation Conference*, pp. 349-354, 1999.

[Cal99b] A.E. Caldwell, A.B. Kahng et I.L. Markov
"Hypergraph Partitioning with Fixed Vertices"
In *Proc. 36th ACM/IEEE Design Automation Conference*, pp. 355-359, 1999.

[Cal99c] A.E. Caldwell
Personal communication.
Avril 1999.

[Cam87] R. Camposano et R.K. Brayton
"Partitioning Before Logic Synthesis"
In *Proc. IEEE International Conference on Computer-Aided Design*, pp. 324-326, 1987.

[Cho94] N.-C. Chou, L.-T. Liu, C.-K. Cheng, W.-J. Dai et R. Lindelof
"Circuit Partitioning for Huge Logic Emulation Systems"
In *Proc. 31st ACM/IEEE Design Automation Conference*, pp. 244-249, 1994.

[Che97] C.S. Chekuri, A.V. Goldberg, D.R. Karger, M.S. Levine et C. Stein
"Experimental Study of Minimum Cut Algorithms"
In *Proc. 8th ACM/SIAM Symposium on Discrete Algorithms*, pp. 324-333, 1997.

[Che91] C.K. Cheng et Y.C. Wei
"An Improved Two-Way Partitioning Algorithm with Stable Performance"
In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 10(12)*, pp. 1502-1511, 1991.

[Con93] J. Cong et M.L. Smith
"A Parallel Bottom-Up Clustering Algorithm With Applications to Circuit Partitioning in VLSI Design"
In *Proc. 30th ACM/IEEE Design Automation Conference*, pp. 755-760, 1993.

[Con94a] J. Cong, Z. Li et R. Bagrodia
"Acyclic Multi-Way Partitioning of Boolean Networks"
In *Proc. 31st ACM/IEEE Design Automation Conference*, pp. 670-675, 1994.

[Con94b] J. Cong, W.J. Labio et N. Shivakumar
"Multiway VLSI Circuit Partitioning Based on Dual Net Representation"
In *Proc. IEEE International Conference on Computer-Aided Design*, pp. 56-62, 1994.

[Con96] J. Cong, W.J. Labio et N. Shivakumar
"Multiway VLSI Circuit Partitioning Based on Dual Net Representation"
In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 15(4)*, pp. 396-409, 1996.

[Con97] J. Cong, H.P. Li, S.K. Lim, T. Shibuya et D. Xu
"Large Scale Circuit Partitioning with Loose/Stable Net Removal and Signal Flow Based Clustering"
In *Proc. IEEE International Conference on Computer-Aided Design*, pp. 441-446, 1997.

[Con98] J. Cong et S.K. Lim
"Multiway Partitioning with Pairwise Movement"
In *Proc. IEEE International Conference on Computer-Aided Design*, pp. 512-516, 1998.

[Dar96] J. Darnauer et W.W.-M. Dai
"A Method for Generating Random Circuits and Its Application to Routability Measurement"
In *Proc. 4th ACM/SIGDA International Symposium on FPGAs*, pp. 66-72, 1996.

[Din70] E.A. Dinic
"Algorithms for Solution of a Problem of Maximum Flow in Networks with Power Estimation"
In *Soviet Math. Doklady 11*, pp. 1277-1280, 1970.

[Dut96a] S. Dutt et W. Deng
"A Probability-Based Approach to VLSI Circuit Partitioning"
In *Proc. 33rd ACM/IEEE Design Automation Conference*, pp. 100-105, 1996.

[Dut96b] S. Dutt et W. Deng
"VLSI Circuit Partitioning by Cluster-Removal Using Iterative Improvement Techniques"
In *Proc. IEEE International Conference on Computer-Aided Design*, pp. 194-200, 1996.

[Edm72] J. Edmonds et R.M. Karp
"Theoretical improvements in algorithmic efficiency for network flow problems"
In *Journal of the Association for Computing Machinery 19(2)*, pp. 248-264, 1972.

[Fan97] W.-J. Fang et A.C.-H. Wu
"Multi-Way FPGA Partitioning by Fully Exploiting Design Hierarchy"
In *Proc. 34th ACM/IEEE Design Automation Conference*, pp. 518-521, 1997.

[Fan98] W.-J. Fang et A.C.-H. Wu
"Performance-Driven Multi-FPGA Partitioning Using Functional Clustering and Replication"
In *Proc. 35th ACM/IEEE Design Automation Conference*, pp. 283-286, 1998.

[Fid82] C.M. Fiduccia et R.M. Mattheyses
"A Linear-Time Heuristic for Improving Network Partitions"
In *Proc. 19th ACM/IEEE Design Automation Conference*, pp. 175-181, 1982.

[For62] L.R. Ford, Jr. et D.R. Fulkerson
Flows in Networks
Princeton University Press, 1962.

[Fou98] C. Fouassier
"Implémentation et Intégration d'un Nouvel Algorithme de Partitionnement"
Rapport de DESS CIMI du Laboratoire d'Informatique de Paris VI (LIP6), 1998.

[Gho98] D. Ghosh, N. Kapur, J. Harlow et F. Brglez
"Synthesis of Wiring Signature-Invariant Equivalence Class Circuit Mutants and Applications to Benchmarking"
In *Proc. Design, Automation and Test in Europe Conference*, pp. 656-663, 1998.

[Glo89] F. Glover
"Tabu Search - Part I"
In *ORSA Journal on Computing 1(3)*, pp. 190-206, 1989.

[Gol88] A.V. Goldberg et R.E. Tarjan
"A New Approach to the Maximum-Flow Problem"
In *Journal of the Association for Computing Machinery 35(4)*, pp. 921-940, 1988.

[Gol83] M.K. Goldberg et M. Burstein
"Heuristic Improvement Technique for Bisection of VLSI Networks"
In *Proc. International Conference Computer Design*, pp. 122-125, 1983.

[Gom61] R.E. Gomory et T.C. Hu
"Multi-Terminal Network Flows"
In *Journal of the SIAM 9*, pp. 551-570, 1961.

[Gon95] M. Gondran et M. Minoux
Graphes et Algorithmes, 3e édition
Collection de la Direction des Etudes et Recherches d'Electricité de France: Editions Eyrolles, 1995.

[Gue98] A. Guettaf et P. Bazargan-Sabet
"Reducing Communication Overhead in Distributed Logic Simulation of VLSI Circuits"
In *Proc. Summer Computer Simulation Conference*, pp. 278-283, 1998.

[Had92] S.W. Hadley, B.L. Mark et A. Vannelli
"An Efficient Eigenvector Approach for Finding Netlist Partitions"
In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 11(7)*, pp. 885-892, 1992.

[Hag91] L. Hagen et A.B. Kahng
"Fast Spectral Methods for Ratio Cut Partitioning and Clustering"
In *Proc. IEEE International Conference on Computer-Aided Design*, pp. 10-13, 1991.

[Hag92a] L. Hagen et A.B. Kahng
"New Spectral Methods for Ratio Cut Partitioning and Clustering"
In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 11(9)*, pp. 1074-1085, 1992.

[Hag92b] L. Hagen et A.B. Kahng
"A New Approach to Effective Circuit Clustering"
In *Proc. IEEE International Conference on Computer-Aided Design*, pp. 422-427, 1992.

[Hag95a] L. Hagen, D.J.H. Huang et A.B. Kahng
"Quantified Suboptimality of VLSI Layout Heuristics"
In *Proc. 32nd ACM/IEEE Design Automation Conference*, pp. 216-221, 1995.

[Hag95b] L. Hagen, D.J.H. Huang et A.B. Kahng
"On Implementation Choices for Iterative Improvement Partitioning Algorithms"
In *Proc. ACM/IEEE European Design Automation Conference*, pp. 144-149, 1995.

[Hal70] K.M. Hall
"An r-dimensional quadratic placement algorithm"
In *Management Science Vol.17(3)*, pp. 219-229, 1970.

[Hao92] J. Hao et J.B. Orlin
"A Faster Algorithm for Finding the Minimum Cut in a Directed Graph"
In *Proc. 3rd ACM/SIAM Symposium on Discrete Algorithms*, pp. 165-174, 1992.

[Hau97] S. Hauck et G. Borriello
"An Evaluation of Bipartitioning Techniques"
In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 16(8)*, pp. 849-866, 1997.

[Hol75] J. Holland
Adaptation in Natural and Artificial Systems
University of Michigan Press, 1975.

[Hu85] T.C. Hu et K. Moerder
"Multiterminal Flows in a Hypergraph"
In *VLSI Circuit Layout: Theory and Design*
Hu and Kuh Eds., IEEE Press, pp. 87-93, 1985.

[Hua95] D.J.H. Huang et A.B. Kahng
"When Clusters Meet Partitions: New Density-Based Methods"
In *Proc. IEEE European Design and Test Conference*, pp. 60-64, 1995.

[Hut96] M.D. Hutton, J.P. Grossman, J. Rose et D. Corneil
"Characterization and Parameterized Random Generation of Digital Circuits"
In *Proc. 33rd ACM/IEEE Design Automation Conference*, pp. 94-99, 1996.

[Hut97] M. Hutton, J. Rose et D. Corneil
"Generation of Synthetic Sequential Benchmark Circuits"
In *Proc. 5th ACM/SIGDA International Symposium on FPGAs*, pp. 149-155, 1997.

[Hwa95] L.J. Hwang et A. El Gamal
"Min-Cut Replication in Partitioned Networks"
In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 14(1)*, pp. 96-106, 1995.

[Ihl93] E. Ihler, D. Wagner et F. Wagner
"Modeling Hypergraphs by Graphs With the Same Mincut Properties"
In *Information Processing Letters 45(4)*, pp. 171-175, 1993.

[Joh96] F.M. Johannes
"Partitioning of VLSI Circuits and Systems"
In *Proc. 33rd ACM/IEEE Design Automation Conference*, pp. 83-87, 1996.

[Kar93] D.R. Karger et C. Stein
"An $O(n^2)$ Algorithm for Minimum Cuts"
In *Proc. 25th ACM Symposium on Theory of Computing*, pp. 757-765, 1993.

[Kar96] D.R. Karger
"Minimum Cuts in Near-Linear Time"
In *Proc. 28th ACM Symposium on Theory of Computing*, pp. 56-63, 1996.

[KaA96] G. Karypis, R. Aggarwal, V. Kumar et S. Shekhar
"Hypergraph Partitioning: Application in VLSI Domain"
Technical Report No. TR-96-060, CS Dept.. University of Minnesota, Minneapolis, 1996.

[KaA97] G. Karypis, R. Aggarwal, V. Kumar et S. Shekhar
"Multilevel Hypergraph Partitioning: Application in VLSI Domain"
In *Proc. 34th ACM/IEEE Design Automation Conference*, pp. 526-529, 1997.

[KaK98] G. Karypis et V. Kumar
"Multilevel Algorithms for Multi-Constraint Graph Partitioning"
Technical Report No. TR-98-019, CS Dept.. University of Minnesota, Minneapolis, 1998.

[KaK99] G. Karypis et V. Kumar
"Multilevel k-way Hypergraph Partitioning"
In *Proc. 36th ACM/IEEE Design Automation Conference*, pp. 343-348 1999.

[Ker70] B.W. Kernighan et S. Lin
"An Efficient Heuristic Procedure for Partitioning Graphs"
In *Bell System Technical Journal 49(2)*, pp. 291-307, 1970.

[Kir83] S. Kirkpatrick, C.D. Gelatt, Jr. et M.P. Vecchi
"Optimization by Simulated Annealing"
In *Science (220)*, pp. 671-680, 1983.

[Kle91] J.M. Kleinhans, G. Sigl, F.M. Johannes et K.H. Antreich
"GORDIAN: VLSI placement by quadratic programming and slicing optimization"
In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 10(3)*, pp. 356-365, 1991.

[Koz91] K. Koźmiński
"Benchmarks of layout synthesis - evolution and current status "
In *Proc. 28th ACM/IEEE Design Automation Conference*, pp. 265-270, 1991.

[Kri91] C. Kring et A.R. Newton
"A Cell-Replicating Approach to Mincut-Based Circuit Partitioning"
In *Proc. IEEE International Conference on Computer-Aided Design*, pp. 2-5, 1991.

[Kri84] B. Krishnamurthy
"An Improved Min-Cut Algorithm for Partitioning VLSI Networks"
In *IEEE Transactions on Computers 33(5)*, pp. 438-446, 1984.

[Kru97] H. Krupnova, A. Abbara et G. Saucier
"A Hierarchy-Driven FPGA Partitioning Method"
In *Proc. 34th ACM/IEEE Design Automation Conference*, pp. 522-525, 1997.

[Kuo97] M.-T. Kuo et C.-K. Cheng
"A Network Flow Approach for Hierarchical Tree Partitioning"
In *Proc. 34th ACM/IEEE Design Automation Conference*, pp. 512-517, 1997.

[Kuz93] R. Kužnar, F. Brglez et K. Koźmiński
"Cost Minimization of Partitions into Multiple Devices"
In *Proc. 30th ACM/IEEE Design Automation Conference*, pp. 315-320, 1993.

[Kuz94a] R. Kužnar, F. Brglez et B. Zajc
"Multi-Way Netlist Partitioning into Heterogeneous FPGAs and Minimization of Total Device Cost and Interconnect"
In *Proc. 31st ACM/IEEE Design Automation Conference*, pp. 238-243, 1994.

[Kuz94b] R. Kužnar, F. Brglez et B. Zajc
"A Unified Cost Model for Min-Cut Partitioning with Replication Applied to Optimization of Large Heterogeneous FPGA Partitions"
In *Proc. ACM/IEEE European Design Automation Conference*, pp. 271-276, 1994.

[Lan94] S. Lan, A. Ziv et A. El Gamal
"Placement and Routing for a Field Programmable Multi-Chip Module"
In *Proc. 31st ACM/IEEE Design Automation Conference*, pp. 295-300, 1994.

[Lan71] B.S. Landman et R.L. Russo
"On a Pin Versus Block Relationship for Partitions of Logic Graphs"
In *IEEE Transactions on Computers 20(3)*, pp. 1469-1479, 1971.

[Lei91] C.E. Leiserson et J.B. Saxe
"Retiming Synchronous Circuitry"
In *Algorithmica 6(1)*, pp. 5-35, 1991.

[Len90] T. Lengauer
Combinatorial Algorithms for Integrated Circuit Layout
Wiley-Teubner, 1990.

[Li95] J. Li, J. Lillis et C.K. Cheng
"Linear Decomposition Algorithm for VLSI Design Applications"
In *Proc. IEEE International Conference on Computer-Aided Design*, pp. 223-228, 1995.

[Li96] J. Li, J. Lillis, L.-T. Liu et C.K. Cheng
"New Spectral Linear Placement and Clustering Approach"
In *Proc. 33rd ACM/IEEE Design Automation Conference*, pp. 88-93, 1996.

[Liu94] L.-T. Liu, M. Shih et C.K. Cheng
"Data Flow Partitioning for Clock Period and Latency Minimization"
In *Proc. 31st ACM/IEEE Design Automation Conference*, pp. 658-663, 1994.

[Liu95a] L.-T. Liu, M.-T. Kuo, C.K. Cheng et T.C. Hu
"Performance-Driven Partitioning Using a Replication Graph Approach"
In *Proc. 32nd ACM/IEEE Design Automation Conference*, pp. 206-210, 1995.

[Liu95b] L.-T. Liu, M.-T. Kuo, S.-C. Huang et C.K. Cheng
"A Gradient Method on the Initial Partition of Fiduccia–Mattheyses Algorithm"
In *Proc. IEEE International Conference on Computer-Aided Design*, pp. 229-234, 1995.

[Liu97] H. Liu et D.F. Wong
"Network Flow Based Multi-Way Partitioning with Area and Pin Constraints"
In *Proc. ACM International Symposium on Physical Design*, pp. 12-17, 1997.

[Liu98a] H. Liu et D.F. Wong
"Network Flow Based Multi-Way Partitioning with Area and Pin Constraints"
In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 17(1)*, pp. 50-59, 1998.

[Liu98b] H. Liu, K. Zhu et D.F. Wong
"Circuit Partitioning with Complex Resource Constraints in FPGAs"
In *Proc. 6th ACM/SIGDA International Symposium on FPGAs*, pp. 77-84, 1998.

[Liu98c] H. Liu et D.F. Wong
"Network Flow Based Circuit Partitioning for Time-multiplexed FPGAs"
In *Proc. IEEE International Conference on Computer-Aided Design*, pp. 497-504, 1998.

[Liu99] H. Liu et D.F. Wong
"Circuit Partitioning for Dynamically Reconfigurable FPGAs"
In *Proc. 7th ACM/SIGDA International Symposium on FPGAs*, pp. 187-194, 1999.

[Law76] E.L. Lawler
Combinatorial Optimization: Networks and Matroids
Holt, Rinehart, and Winston, 1976.

[Mak96] W.-K. Mak et D.F. Wong
"Minimum Replication Min-Cut Partitioning"
In *Proc. IEEE International Conference on Computer-Aided Design*, pp. 205-210, 1996.

[Men99a] Mentor Graphics Corp., Meta Systems Div. "CELARO User's Manual" Mentor Graphics, 1999.

[Men99b] Mentor Graphics Corp., Meta Systems Div. "CELARO Reference Manual" Mentor Graphics, 1999.

[Met53] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller et E. Teller
"Equation of State Calculation by Fast Computing Machines"
In *Journal of Chem. Phys. 21*, pp. 1087-1091, 1953.

[Min89] M. Minoux et E. Pinson
"Lower bounds to the graph partitioning problem through generalized linear programming and network flows"
In *R.A.I.R.O. Recherche opérationelle/Operations Research 21(4)*, pp. 349-363, 1989.

[Min93] M. Minoux
"LP relaxation of the hypergraph partitioning problem and network flows"
In *Proc. Netflow 93*, pp. 170-174, 1993.

[Min95] M. Minoux
"On some large-scale LP relaxations for the graph partitioning problem and their optimal solutions"
In *Annals of Operations Research 58(1995)*, pp. 143-154, 1995.

[Nag92] H. Nagamochi et T. Ibaraki
"Computing Edge-Connectivity in Multigraphs and Capacitated Graphs"
In *SIAM Journal of Discrete Mathematics 5(1)*, pp. 54-66, 1992.

[Nag94] H. Nagamochi, T. Ono et T. Ibaraki
"Implementing an Efficient Minimum Capacity Cut Algorithm"
In *Mathematical Programming 67(3)*, pp. 297-324, 1994.

[Obe95] U. Ober et M. Glesner
"Multiway Netlist Partitioning onto FPGA-based Board Architectures"
In *Proc. ACM/IEEE European Design Automation Conference*, pp. 150-155, 1995.

[Pad90] M. Padberg et G. Rinaldi
"An Efficient Algorithm for the Minimum Capacity Cut Problem"
In *Mathematical Programming 47*, pp. 19-36, 1990.

[Pal83] C.A. Palesko et L.A. Akers
"Logic Partitioning for Minimizing Gate Arrays"
In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 2(2)*, pp. 117-121, 1983.

[Pis99a] J. Pistorius, E. Legai et M. Minoux
"Generation of Very Large Circuits to Benchmark the Partitioning of FPGAs"
In *Proc. ACM International Symposium on Physical Design*, pp. 67-73, 1999.

[Pis99b] J. Pistorius et M. Minoux
"An Improved Direct Labeling Method for the Maximum Flow Problem on Hypergraphs"
soumis à publication.

[Raj93] R. Rajaraman et D.F. Wong
"Optimal Clustering for Delay Minimization"
In *Proc. 30th ACM/IEEE Design Automation Conference*, pp. 309-314, 1993.

[Ram96] S. Raman et L.M. Patnaik
"Performance-Driven MCM Partitioning Through an Adaptive Genetic Algorithm"
In *IEEE Transactions on VLSI Systems 4(4)*, pp. 434-443, 1996.

[Rie94] B.M. Riess, L. Doll et F.M. Johannes
"Partitioning Very Large Circuits Using Analytical Placement Techniques"
In *Proc. 31st ACM/IEEE Design Automation Conference*, pp. 646-651, 1994.

[Rie95] B.M. Riess et A.A. Schoene
"Architecture Driven K-Way Partitioning for Multichip Modules"
In *Proc. IEEE European Design and Test Conference*, pp. 71-76, 1995.

[Roy93] K. Roy et K. Sechen
"A Timing Driven N-Way Chip and Multi-Chip Partitioner"
In *Proc. IEEE International Conference on Computer-Aided Design*, pp. 240-247, 1993.

[Saa90] Y.G. Saab et V.B. Rao
"Fast Effective Heuristics for the Graph Bisectioning Problem"
In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 9(1)*, pp. 91-98, 1990.

[San89] L.A. Sanchis
"Multiple-Way Network Partitioning"
In *IEEE Transactions on Computers 38(1)*, pp. 62-81, 1989.

[Saw95] P. Sawkar et D. Thomas
"Multi-Way Partitioning for Minimum Delay for Look–Up Table Based FPGAs"
In *Proc. 32nd ACM/IEEE Design Automation Conference*, pp. 201-205, 1995.

[Sch72] D.M. Schuler et E.G. Ulrich
"Clustering and Linear Placement"
In *Proc. 9th ACM/IEEE Design Automation Conference*, pp. 50-56, 1972.

[Sen92] E.M. Sentovich et al.
"SIS: A System for Sequential Circuit Synthesis"
Technical Report No. UCB/ERL M92/41. University of California, Berkeley, 1992.

[She93] N.A. Sherwani
Algorithms for VLSI Physical Design Automation
Kluwer Academic Publishers, 1993.

[Sig91] G. Sigl, K. Doll et F.M. Johannes
"Analytical placement : A linear or a quadratic objective function?"
In *Proc. 28th ACM/IEEE Design Automation Conference*, pp. 427-432, 1991.

[Son98] J. Song, Z. Shen, et W. Zhuang
"An Effective General Connectivity Concept for Clustering"
In *Proc. Design, Automation and Test in Europe Conference*, pp. 398-405, 1998.

[Van90] A. Vannelli et S.W. Hadley
"A Gomory-Hu Cut Tree Representation of a Netlist Partitioning Problem"
In *IEEE Transactions on Circuits and Systems 37(9)*, pp. 1133-1139, 1990.

[Vij90] G. Vijayan
"Partitioning Logic on Graph Structures to Minimize Routing Cost"
In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 9(12)*, pp. 1326-1334, 1990.

[Wei90] Y.C. Wei et C.K. Cheng
"A Two-Level Two-Way Partitioning Algorithm"
In *Proc. IEEE International Conference on Computer-Aided Design*, pp. 516-519, 1990.

[Wei91] Y.C. Wei et C.K. Cheng
"Ratio Cut Partitioning for Hierarchical Designs"
In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 10(7)*, pp. 911-921, 1991.

[Xil92] Xilinx, Inc. "The programmable Gate Array Data Book" Xilinx, 1992.

[Yan94] H.H. Yang et D.F. Wong
"Efficient Network Flow Based Min-Cut Balanced Partitioning"
In *Proc. IEEE International Conference on Computer-Aided Design*, pp. 50-55, 1994.

[Yan95a] H.H. Yang et D.F. Wong
"Circuit Clustering for Delay Minimization under Area and Pin Constraints"
In *Proc. IEEE European Design and Test Conference*, pp. 65-70, 1995.

[Yan95b] H.H. Yang et D.F. Wong
"New Algorithms for Min-Cut Replication in Partitioned Circuits"
In *Proc. IEEE International Conference on Computer-Aided Design*, pp. 216-222, 1995.

[Yan96] H.H. Yang et D.F. Wong
"Efficient Network Flow Based Min-Cut Balanced Partitioning"
In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 15(12)*, pp. 1533-1540, 1996.

[Zie96] J.Y. Zien, M.D.F. Schlag et P.K. Chan
"Multi-level Spectral Hypergraph Partitioning with Arbitrary Vertex Sizes"
In *Proc. IEEE International Conference on Computer-Aided Design*, pp. 201-204, 1996.

Annexe

Annexe A

Ordonnancement par calcul itératif du barycentre

A.1 Principe de l'algorithme

Les différents algorithmes basés sur la méthode spectrale présentée dans section 2.3.3 nécessitent un temps de calcul trop important.
Une autre approche consiste à calculer itérativement le barycentre de tous les sommets par rapport à deux sommets de référence. Cette approche est présentée par la suite. Elle est basée comme l'algorithme de Hall [Hal70] sur la minimisation de la distance carrée entre tous les sommets (voir aussi section 2.3.3). Soit c_{ij} le nombre d'arrêtes entre les sommets v_i et v_j. Soit x_i la coordonnée X du sommet v_i et x_j la coordonnée X du sommet v_j. Soient v_s et v_t deux sommets sélectionnés. v_s est placé à la coordonnée $x_s = 0$ et v_t à la coordonnée $x_t = 1$. L'objectif est donc de placer tous les sommets dans l'intervalle $[0;1]$ tel que la longueur carré des arrêtes soit minimale :

$$\begin{aligned} \min\left(z \right. &= \left.\frac{1}{2}\sum_{ij}(x_i - x_j)^2 c_{ij}\right) \\ \text{tel que} \quad & x_s = 0 \\ & x_t = 1 \end{aligned}$$

La solution du problème est le suivant :

$$\begin{aligned} \frac{\partial z}{\partial x_i} &= \sum_{\{j|v_j \in adj(v_i)\}} c_{ij}(x_i - x_j) \\ &= m_i\left(x_i - \frac{1}{m_i}\sum_{\{j|v_j \in adj(v_i)\}} c_{ij}x_j\right) \qquad m_i = \sum_{\{j|v_j \in adj(v_i)\}} c_{ij}x_j \\ &= m_i(x_i - \hat{x}_i) \qquad \hat{x}_i = \frac{1}{m_i}\sum_{\{j|v_j \in adj(v_i)\}} c_{ij}x_j \end{aligned}$$

avec $\hat{x}_i$ le barycentre du sommet v_i. La solution du problème est donc optimal pour $x_i = \hat{x}_i$, c'est–à–dire quand le sommet v_i est ordonné dans son barycentre. Dans ce cas nous avons

$\frac{\partial z}{\partial x_i} = 0$.

A.2 Description de l'algorithme

Nous en déduisons l'algorithme suivant :

1. Choix de deux sommets initiaux appelés source s et puits t en utilisons l'algorithme décrit dans section 6.2.1, puis placement des ces sommets : $x_s = 0$ et $x_t = 1$.
2. Détermination d'un ordonnancement initial arbitraire, où tous les sommets sont placés dans l'intervalle $[0;1]$:
$$\forall_{v_i \in V\{s,t\}} x_s, \ldots x_i, \ldots x_t$$
3. Choix d'une tolérance $eps > 0$.
4. Tant qu'il y a des changements dans la solution actuelle :
Effectuer l'étape 5).
5. Calculer pour tous les sommets v_i, $v_i \in V$
$\{s,t\}$:
m_i, x_i et $\delta = m_i|x_i - \hat{x}_i|$.
Si $eps < \delta$: $\hat{x}_i = x_i$.

Cet algorithme a été implémenté et testé à l'aide de la *netlist industC* du *benchmark* présenté dans le tableau 4.1 de la section 4.3.1.

A.3 Analyse des performances de l'algorithme

La *netlist* a été découpée en partitions respectant les contraintes des FPGAs XC3064 (224 sommets et 120 connecteurs d'entrées/sorties) et en variant la tolérance *eps*. Les résultats sont présentés dans le tableau A.1.

Circuit	eps	#itérations	#XC3064	t_CPU
industC	10^{-4}	28	517	14min
industC	10^{-6}	43	446	13min
industC	10^{-9}	54	416	12min
industC	10^{-15}	83	416	13min

Tableau A.1: *Ordonnancement par calcul itératif du barycentre*

En même temps, le partitionnement de la même *netlist* avec l'algorithme window utilisant la fonction d'attraction de *scaled cost* nécessite un temps CPU de 11 minutes pour partitionner en 126 FPGAs XC3064. Un ordonnancement arbitraire selon la suite de la lecture de la *netlist* mène à 311 partitions en un temps CPU de 9 minutes.

Le déséquilibre de la distribution des sommets sur l'intervalle $[0;1]$ explique le mauvais résultat du partitionnement avec ordonnancement par calcul itératif du barycentre. La distance maximale entre les sommets source et puits a été déterminée en prenant compte uniquement des *nets* avec une cardinalité inférieure à 20 (voir aussi section initFBB). Dans

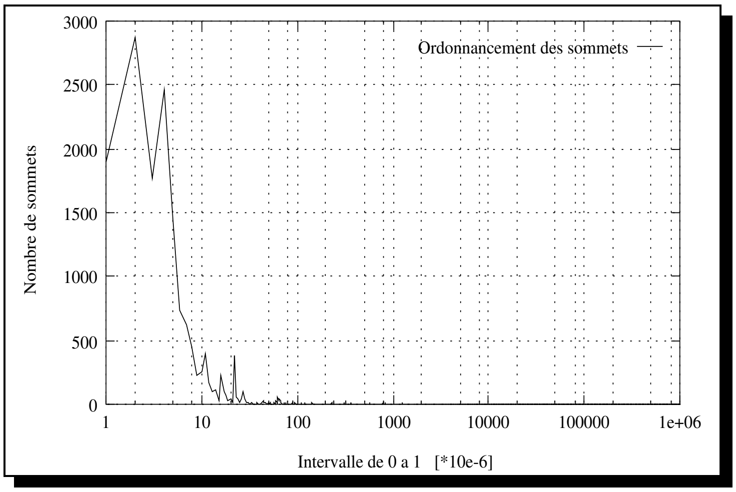


Figure A.1: *Distribution des sommets sur l'intervalle de 0 à 1*

ce cas, la distance déterminée est de 32. Statistiquement, nous avons donc en moyenne 16735/32 sommets voisins du sommets source. En effet, en analysant la distribution des sommets sur l'intervalle $[0;1]$ illustré dans figure A.1, nous constatons qu'environ 90% des sommets sont ordonnés dans l'intervalle $[0;10^-5]$. Beaucoup de sommets ont donc le même ordonnancement. L'algorithme DP-RP choisit un ordre quelconque, ce qui revient à faire un ordonnancement aléatoire.
L'algorithme n'a donc pas été retenu dans la suite de ce travail.

Annexe B

Algorithme de clustering par détermination de cônes de logique combinatoire

Le principe de l'algorithme consiste à déterminer des cônes de logique combinatoire et à créer une partition pour chacun de ces cônes. Le résultat est un *clustering* de la *netlist* d'origine où chaque *cluster* contient un maximum de chemins de logique combinatoire. L'algorithme présenté dans ce chapitre est inspiré de [Bra94].

B.1 Description de l'algorithme

Dans un premier temps, l'algorithme détermine des cellules de la *netlist* qui sont à l'origine des cônes de logique combinatoire. Il s'agit de cellules séquentielles ou de cellules pilotant des *nets* sur des sorties primaires de la *netlist*. A chacune de ces cellules est associée une couleur. L'algorithme marque récursivement toutes les cellules à l'entrée de la cellule courante avec la couleur de la cellule qui est à l'origine du cône de logique combinatoire en commençant avec celle-ci et jusqu'à atteindre une entrée primaire ou une autre cellule séquentielle. Les cellules se trouvant sur des intersections de cônes de logique combinatoire sont marquées avec plusieurs couleurs.
Après avoir effectué le marquage avec toutes les couleurs, l'algorithme trie les cônes de logique combinatoire selon leur cardinalité. En commençant avec le plus petit cône, des cellules ayant la même couleur que l'origine de ce cône sont regroupées dans une partition si elles ne font pas déjà parties d'autres partitions. Le principe de l'algorithme est illustré dans un exemple :

Exemple B.1 *Partitionnement selon l'algorithme basé sur la construction de cônes de logique combinatoire*

Etant donnée une netlist contenant 27 cellules combinatoires et ayant 4 sorties et 8 entrées primaires comme présentée dans figure B.1. Puisqu'il n'y a pas de cellules séquentielles dans la netlist, seulement les 4 cellules pilotant les nets sur les sorties primaires sont des origines de cônes de logique combinatoire. Une couleur différente est attribuée à chacune de ces 4 cellules. Ensuite, toutes les cellules se trouvant sur les entrées des origines de

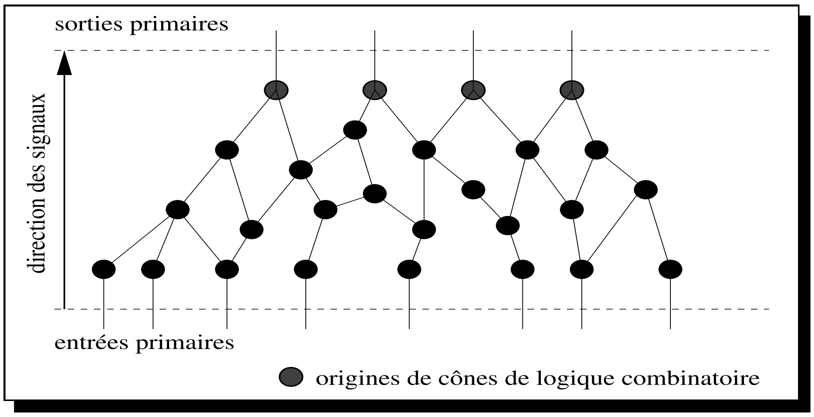


Figure B.1: *Netlist combinatoire avec des entrées/sorties primaires*

cônes, et récursivement toutes les cellules sur les entrées de ces cellules jusqu'à atteindre des entrées primaires, sont marquées avec la couleur de l'origine du cône respectif. Nous obtenons alors la configuration illustrée dans figure B.2.

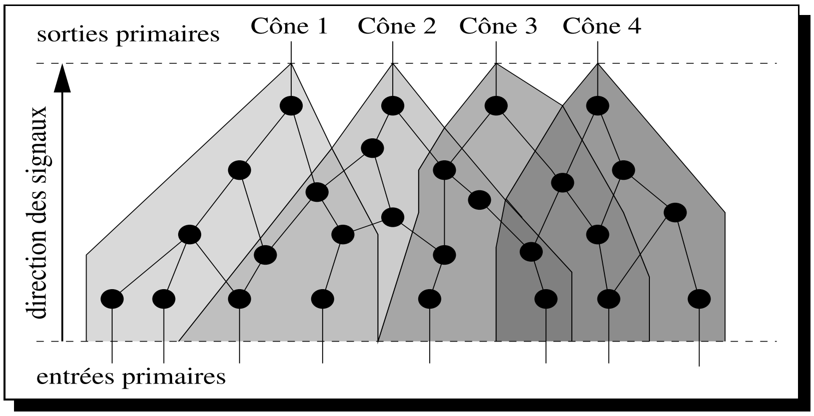


Figure B.2: *Marquage de cellules*

Cône 1 contient alors 10 cellules, cône 2 a 14 cellules, cône 3 est composé de 10 cellules et cône 4 a 9 cellules. L'ordre de regroupement est alors cône 4 suivi de cône 1, cône 3 et cône 2. Nous avons donc le découpage présenté dans figure B.3.

Les partitions qui dépassent la cardinalité ou la coupure maximale d'une partition physique sont découpées en sous-ensembles jusqu'à ce que chaque sous-ensemble respecte ces contraintes. L'algorithme utilisé à cette occasion est l'algorithme bisect*.

B.2 Avantages et inconvénients

Les avantages de l'algorithme sont sa rapidité grâce à une complexité linéaire en $O(p|V|)$ où p est le nombre moyen de connecteurs par cellule et $|V|$ le nombre de cellules, ainsi qu'une augmentation potentielle de la fréquence du *design* implémenté sur l'émulateur.
L'inconvénient majeur est la taille des différentes partitions ou *clusters* qui dépend uniquement de la nature de la *netlist* et qui n'est pas contrôlable par cet algorithme: de nombreux

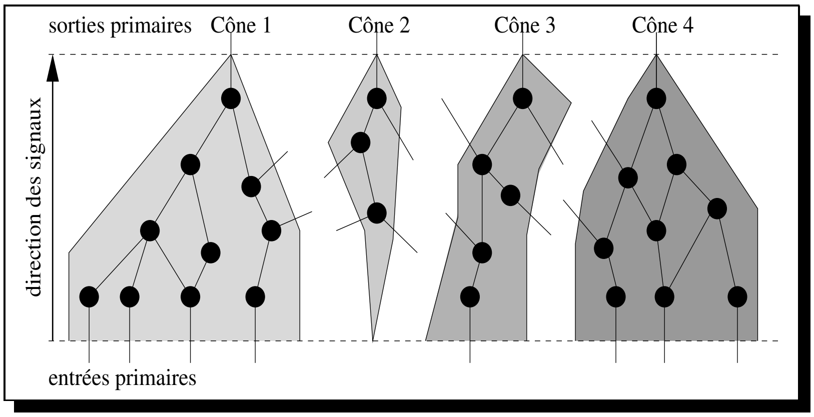


Figure B.3: *Formation de partitions à partir des cônes de logique combinatoire*

points mémorisant dans la *netlist* conduisent à un grand nombre de cônes de très petite taille et donc à un grand nombre de partitions. Très peu de points mémorisant entraînent la construction de cônes de logique combinatoire ayant des cardinalités ou des coupures qui dépassent souvent la cardinalité ou la coupure maximale d'une partition physique. Ces cônes de logique sont bipartitionnés en utilisant l'algorithme bisect*, ce qui entraîne le découpage des chemins de logique combinatoire que l'on veut préserver.
Un autre inconvénient se présente pour les *nets* pilotant plusieurs cellules: chaque *net* de multiple sortance (*fanout*) piloté par une cellule non-séquentielle mène à des intersections de cônes et ainsi à une coupure de chemins combinatoires. Nous avons observé qu'environ 30% des *nets* dans des *netlists* de CPBs traitées habituellement ont un *fanout* supérieur à un et que plus de 95% des cellules ont plus de trois *nets* en entrée, ce qui augmente significativement la probabilité de rencontrer de nombreuses intersections de cônes.
Un problème semblable se produit pour des cellules pilotant plusieurs *nets*, comme par exemple dans le cas d'une *netlist* d'AVCs. Cette cellule appartient naturellement à plusieurs partitions.
Les résultats obtenus lors de l'évaluation expérimentale de cet algorithme dans section 8.2.4 confirment ces hypothèses.

B.3 Conclusion

Nous avons observé de nombreux inconvénients de l'algorithme. basé sur la construction de cônes de logique combinatoire présenté dans [Bra94]. Il n'est donc pas adapté au partitionnement avec des contraintes de cardinalité et de coupure ciblant à minimiser le nombre de partitions tout en optimisant la fréquence du *design* implémenté sur l'émulateur. Néanmoins, l'approche de construction de cônes de logique combinatoire nous semble intéressante, en particulier associée à un algorithme de duplication de logique permettant la suppression d'un certain nombre d'intersections de cônes. De plus, la cardinalité et la coupure maximale des partitions physiques doivent être prise en compte lors de la construction des cônes, ce qui favorise une construction des cônes de logique en commençant avec les entrées primaires.
La réalisation et la vérification de ces hypothèses est un domaine ouvert à la recherche.

Annexe C

Evaluation des performances des algorithmes de partitionnement

C.1 Bipartitionnement par une approche de voisinage : algorithme bisect*

Les résultats expérimentaux obtenus avec les différentes modifications apportées à l'algorithme *bisect** sont détaillés dans cette section. En particulier, nous présentons les résultats obtenus pour :

- la fonction de coût permettant un bipartitionnement équilibré comme décrit dans section 4.2.3 (*balanced*),
- le bipartitionnement initial déterminé aléatoirement décrit dans section 4.2.2 en combinaison avec le bipartitionnement aléatoire (*initAlea*),
- le bipartitionnement initial déterminé aléatoirement deux fois avec sélection de la meilleure solution comme décrit dans section 4.2.2 en combinaison avec le bipartitionnement équilibré (*2InitAlea*),
- le regroupement initial de cellules décrit dans section 4.2.1 avec le bipartitionnement équilibré (*initClustering*),
- le regroupement initial de cellules suivi du bipartitionnement initial aléatoire en combinaison avec le bipartitionnement équilibré (*initAleaClust*).

Les résultats expérimentaux obtenus pour ces modifications ont été comparés avec ceux obtenus pour l'algorithme de base (*bisect*), décrit dans section 4.1.1.
Les résultats expérimentaux sont triés en fonction des contraintes de partitionnement (*cluster*, AVC, XC3064 et AVB), et présentés dans les tableaux C.1, C.2, C.3 et C.4.

	bisect*		balanced		initAlea		2InitAlea		initClustering		initAleaClust	
Circuit	#cl	t_CPU	#cl	t_CPU	#cl	t_CPU	#cl	t_CPU	#cl	t_CPU	#cl	t_CPU
glue5K	212	58s	131	33s	130	33s	132	113s	133	30s	131	25s
op5K	171	40s	121	25s	124	23s	131	69s	132	23s	128	19s
ctrl15K	562	257s	319	160s	302	115s	307	24min	320	143s	327	122s
glue15K	837	444s	511	264s	478	235s	489	23min	492	245s	497	220s
op15K	500	243s	353	150s	339	128s	359	9min	358	148s	358	134s
ctrl35K	1334	28min	853	20min	822	11min	825	150min	827	15min	825	12min
ctrl50K	1771	61min	1148	41min	1090	22min	1084	205min	1095	29min	1099	22min
Moyenne	769	912s	491	613s	469	359s	475	59min	480	461s	481	351s
industA	260	64s	192	54s	178	38s	195	239s	177	40s	177	35s
industB	853	7min	347	3min	375	3min	380	33min	449	3min	339	2min
industC	601	5min	407	4min	401	3min	415	14min	409	3min	397	3min
industD	1636	17min	585	6min	493	4min	495	18min	479	5min	473	4min
industE	713	7min	550	7min	543	5min	536	20min	525	6min	523	5min
industF	678	10min	546	7min	523	7min	552	23min	552	7min	532	6min
industG	1347	38min	1150	29min	1156	22min	1167	83min	1116	24min	1142	21min
Moyenne	870	12min	540	8min	524	6min	534	28min	530	7min	512	6min

Tableau C.1: *Performances des différentes version de l'algorithme bisect* pour le partitionnement en clusters*

	bisect*		balanced		initAlea		2InitAlea		initClustering		initAleaClust	
Circuit	#AVC	t_CPU	#AVC	t_CPU	#AVC	t_CPU	#AVC	t_CPU	#AVC	t_CPU	#AVC	t_CPU
glue5K	137	48s	81	27s	80	27s	75	104s	85	24s	90	21s
op5K	62	26s	33	15s	40	14s	27	36s	11	7s	12	7s
ctrl15K	129	111s	53	74s	44	39s	49	21min	59	56s	72	47s
glue15K	650	367s	425	231s	409	213s	406	22min	424	222s	433	201s
op15K	156	134s	112	77s	123	69s	83	7min	41	38s	44	37s
ctrl35K	731	20min	271	13min	256	5min	269	151min	282	7min	300	6min
ctrl50K	1106	49min	464	28min	427	12min	447	173min	463	17min	481	12min
Moyenne	424	689s	205	326s	197	197s	194	54min	195	255s	205	199s
industA	40	26s	53	29s	31	16s	35	152s	31	16s	30	14s
industB	350	250s	129	101s	124	82s	151	34min	126	80s	120	75s
industC	104	142s	94	95s	89	78s	118	11min	87	71s	93	65s
industD	251	367s	81	133s	83	88s	90	7min	96	97s	67	71s
industE	121	139s	151	182s	157	138s	121	12min	171	150s	151	121s
industF	101	225s	90	151s	90	123s	112	14min	66	109s	70	89s
industG	211	18min	252	13min	206	7min	293	47min	203	7min	215	6min
Moyenne	168	318s	121	210s	111	135s	131	18min	111	135s	107	114s

Tableau C.2: *Performances des différentes versions de l'algorithme bisect* pour le partitionnement en AVCs*

C.2 Multipartitionnement par une approche de voisinage : algorithme multisect*

Les résultats expérimentaux pour le partitionnement en ensembles respectant les contraintes de cardinalité et de coupure des FPGAs XC3064 obtenus pour les différentes combinaisons de fonctions de coût et de fonction de gain, décrit dans section 5.2.1, ont été comparés avec ceux obtenus pour l'algorithme de base (*multisect*), décrit dans section 5.1.1.
Ces résultats sont présentés dans tableau C.5.

C.3 Bipartitionnement par calcul du flot maximal dans un hypergraphe

C.3.1 Résultats expérimentaux pour le marquage direct sur l'hypergraphe

La comparaison des performances en terme de temps de calcul et de mémoire occupée a été effectuée à l'aide d'hypergraphes H1 à H13 modélisant des *netlists* industrielles et des *netlists* générées avec PartGen. Les résultats expérimentaux détaillés sont présentés dans le tableau C.6.

	bisect*		balanced		initAlea		2InitAlea		initClustering		initAleaClust	
Circuit	#3064	t_CPU	#3064	t_CPU	#3064	t_CPU	#3064	t_CPU	#3064	t_CPU	#3064	t_CPU
glue5K	193	54s	124	32s	123	32s	124	115s	123	28s	124	24s
op5K	76	30s	49	17s	62	17s	40	40s	34	11s	35	10s
ctrl15K	333	190s	117	100s	112	65s	124	23min	134	84s	138	68s
glue15K	857	423s	505	263s	498	241s	506	24min	496	247s	506	222s
op15K	210	150s	140	86s	148	76s	123	8min	118	67s	104	57s
ctrl35K	982	24min	430	15min	395	7min	425	140min	422	9min	450	8min
ctrl50K	1382	54min	662	32min	619	14min	646	190min	664	20min	681	15min
Moyenne	576	790s	290	474s	280	242s	284	55min	284	311s	291	252s
industA	55	29s	66	29s	54	21s	60	194s	61	22s	60	18s
industB	484	303s	204	125s	186	95s	226	32min	186	100s	180	87s
industC	244	198s	171	127s	158	109s	181	11min	153	101s	148	87s
industD	525	511s	170	185s	162	125s	153	9min	159	133s	140	104s
industE	229	202s	239	231s	219	180s	218	14min	275	209s	258	185s
industF	249	355s	189	217s	174	175s	211	16min	176	190s	172	157s
industG	504	23min	471	17min	433	11min	487	51min	424	11min	419	9min
Moyenne	327	425s	216	276s	198	195s	219	19min	205	202s	197	168s

Tableau C.3: *Performances des différentes versions de l'algorithme bisect* pour le partitionnement en FPGAs XC3064*

	bisect*		balanced		initAlea		2InitAlea	
Circuit	#AVB	t_CPU	#AVB	t_CPU	#AVB	t_CPU	#AVB	t_CPU
mix50K3	10	9s	8	8s	8	8s	15	14s
mix50K4	15	10s	10	7s	11	6s	15	14s
mix55K5	9	9s	8	8s	8	7s	16	14s
Moyenne	11	9s	9	8s	9	7s	15	14s
industK	140	8min	88	4min	92	4min	161	18min
industL	126	12min	108	10min	108	5min	202	32min
industM	169	29min	184	31min	195	10min	300	47min
Moyenne	145	16min	127	15min	132	6min	221	32min

Tableau C.4: *Performances des différentes versions de l'algorithme bisect* pour le partitionnement en AVBs*

C.3.2 Performances de l'algorithme hyperFBB

L'évaluation des performances des différentes améliorations apportées à l'algorithme hyperFBB sont présentées dans cette section. L'influence de la fusion du sommet voisin minimisant la coupure entre les deux partitions avec les contraintes d'équilibres 20% - 80% (*min-cut 20-80*), 33% - 67% (*min-cut 33-67*) et 45% - 55% (*min-cut 45-55*) par rapport à l'algorithme hyperFBB de base (hyperFBB) est présentée séparément selon les contraintes de partitionnement dans les tableau C.7 à C.10.

C.4 Comparaison des performances des différents algorithmes de partitionnement

L'application des différents algorithmes aux différentes *netlists* du tableau 8.1 fournit les résultats présentés dans les tableaux C.11, C.12, C.13 et C.14. suivant les contraintes de partitionnement (*cluster*, AVC, XC3064 et AVB). Le taux de remplissage (tdr) est le nombre de cellules à partitionner divisé par le nombre de cellules disponibles dans toutes les partitions. L'application de multisect* sur ces benchmarks n'a pas été possible. En effet, au bout de 200 heures de calcul, il n'y avait aucun résultat pour le partitionnement d'une *netlist* de 15K CPBs à plat.

	multisect*			cost1			cost2			cost3		
Circuit	#3064	#pins	t_CPU	#3064	#pins	t_CPU	#3064	#pins	t_CPU	#3064	#pins	t_CPU
glue500	3	104	5s	3	112	11s	3	111	3s	3	90	18s
op500	4	97	18s	3	119	13s	3	114	9s	3	81	20s
s5378	4	114	67s	4	119	24s	4	95	7s	5	81	65s
glue750	6	119	360s	7	112	231s	7	119	253s	10	77	571s
op750	6	108	124s	5	119	72s	6	109	41s	5	88	195s
s9234	6	100	172s	5	118	80s	5	97	8s	5	76	72s
op1000	7	103	480s	7	119	575s	7	98	58s	6	97	505s
op1250	10	118	2888s	12	116	2160s	9	117	628s	10	101	1748s
s13207	9	108	957s	9	119	562s	9	96	206s	9	80	1173s
op1500	13	113	4111s	14	119	5245s	11	116	1230s	13	84	2825s
Moyenne	6.8	108	918s	6.9	117	897s	6.4	107	244s	6.9	86	719s

Tableau C.5: *Evaluation des performances de l'algorithme multisect* pour le partitionnement en FPGAs XC3064*

Hypergraphe				FBB		hyperFBB		Rapport	
Nom	|V|	|E|	p	t (Mcycles)	m (Ko)	t (Mcycles)	m (Ko)	t(hyperFBB)/t(FBB)	m(hyperFBB)/m(FBB)
H1	20	531	2.35	9.1	264	3.1	32	0.34	0.12
H2	51	3531	2.05	94.5	1632	30.0	176	0.32	0.11
H3	64	3750	2.61	124.4	1992	48.6	200	0.39	0.10
H4	192	4988	2.47	66.1	2584	32.1	264	0.49	0.10
H5	245	11417	2.55	65.7	6000	35.7	592	0.54	0.10
H6	265	14256	2.38	138.0	7160	68.5	720	0.50	0.10
H7	407	8302	2.87	66.7	4584	60.2	464	0.90	0.10
H8	419	9410	2.75	158.1	5000	99.1	512	0.63	0.10
H9	678	14913	2.84	53.1	8224	32.8	824	0.62	0.10
H10	1389	46016	2.78	1412.8	25504	274.2	2448	0.66	0.10
H11	2123	98915	2.55	1266.8	51808	887.3	5080	0.70	0.10
H12	2123	98915	5.45	486.9	77864	199.8	5888	0.41	0.08
H13	2123	98915	10.58	2694.3	126566	1089.0	7408	0.40	0.06
Rapport moyen entre hyperFBB et FBB								0.53	0.1

Tableau C.6: *Comparaison des performances du marquage avec et sans transformation en graphe*

	hyperFBB		min-cut 45-55		min-cut 20-80		min-cut 33-67	
Circuit	#clust	t_CPU	#clust	t_CPU	#clust	t_CPU	#clust	t_CPU
glue5K	127	4min	122	5min	176	6min	132	5min
op5K	125	1min	122	2min	151	2min	126	2min
ctrl15K	302	5min	303	11min	436	12min	345	9min
glue15K	554	54min	494	61min	643	61min	521	58min
op15K	347	12min	341	11min	425	12min	360	11min
ctrl35K	816	44min	810	64min	1061	59min	860	54min
ctrl50K	1141	96min	1126	142min	1528	118min	1223	105min
Moyenne	487	31min	474	42min	631	39min	510	35min
industB	335	17min	335	17min	416	16min	363	15min
industC	393	15min	393	17min	464	16min	402	15min
industD	477	18min	486	17min	581	17min	509	17min
industE	514	25min	516	26min	628	26min	536	25min
industF	522	18min	515	21min	649	17min	541	16min
industG	1112	125min	1134	126min	1429	151min	1192	120min
Moyenne	559	36min	563	37min	695	41min	590	35min

Tableau C.7: *Performances des différentes version de l'algorithme hyperFBB pour le partitionnement en clusters*

	hyperFBB		min-cut 45-55		min-cut 20-80		min-cut 33-67	
Circuit	#AVC	t_CPU	#AVC	t_CPU	#AVC	t_CPU	#AVC	t_CPU
glue5K	104	4min	84	5min	116	6min	95	5min
op5K	42	1min	25	1min	32	2min	22	1min
ctrl15K	93	4min	97	10min	129	10min	100	8min
glue15K	438	52min	384	58min	498	58min	395	56min
op15K	153	11min	105	9min	78	10min	85	9min
ctrl35K	313	38min	368	57min	528	53min	405	48min
ctrl50K	486	86min	561	136min	841	113min	628	100min
Moyenne	233	28min	232	38min	317	36min	247	32min
industB	167	16min	125	15min	160	15min	135	14min
industC	155	14min	116	15min	129	14min	102	12min
industD	117	15min	85	14min	84	13min	80	14min
industE	247	22min	173	23min	178	23min	156	22min
industF	106	14min	90	17min	97	13min	89	12min
industG	348	108min	297	113min	340	133min	283	110min
Moyenne	190	32min	148	33min	165	35min	141	31min

Tableau C.8: *Performances des différentes versions de l'algorithme hyperFBB pour le partitionnement en AVCs*

	hyperFBB		min-cut 45-55		min-cut 20-80		min-cut 33-67	
Circuit	#XC3064	t_CPU	#XC3064	t_CPU	#XC3064	t_CPU	#XC3064	t_CPU
glue5K	135	4min	121	5min	177	6min	131	5min
op5K	67	1min	45	1min	56	2min	50	2min
ctrl15K	152	4min	160	11min	223	11min	183	9min
glue15K	596	57min	536	61min	691	59min	555	60min
op15K	220	11min	166	10min	168	11min	148	10min
ctrl35K	470	42min	526	64min	796	57min	588	52min
ctrl50K	701	88min	807	141min	1170	113min	869	99min
Moyenne	334	30min	337	42min	469	37min	361	34min
industB	246	16min	206	16min	241	15min	208	15min
industC	233	15min	188	16min	214	15min	164	14min
industD	184	16min	154	15min	178	15min	157	15min
industE	373	23min	284	24min	325	25min	271	23min
industF	210	15min	187	18min	213	14min	189	13min
industG	536	113min	500	116min	592	133min	495	111min
Moyenne	297	33min	253	34min	294	36min	247	32min

Tableau C.9: *Performances des différentes versions de l'algorithme hyperFBB pour le partitionnement en FPGAs XC3064*

	hyperFBB		min-cut 45-55		min-cut 20-80		min-cut 33-67	
Circuit	#AVB	t_CPU	#AVB	t_CPU	#AVB	t_CPU	#AVB	t_CPU
industK	138	20min	116	20min	113	19min	101	18min
industL	171	35min	151	53min	153	79min	128	104min
industM	304	239min	263	366min	250	348min	227	415min
Moyenne	204	98min	177	146min	172	149min	152	179min

Tableau C.10: *Performances des différentes versions de l'algorithme hyperFBB pour le partitionnement en AVBs*

	hyperFBB			bisect*			cone			MDOC		
Circuit	#clusters	tdr	t_CPU	#clusters	tdr	t_CPU	#clusters	tdr	t_CPU	#clusters	tdr	t_CPU
glue5K	129	60.6%	303s	131	59.7%	33s	199	39.3%	56s	90	86.9%	56s
op5K	124	67.1%	94s	121	68.8%	25s	235	35.4%	18s	84	99.8%	58s
ctrl15K	333	64.7%	586s	319	67.5%	160s	326	66.4%	166s	216	99.7%	498s
glue15K	511	45.9%	57min	511	45.9%	4min	652	36.0%	3min	358	65.5%	11min
op15K	360	66.6%	10min	353	68.0%	3min	613	39.1%	3min	246	97.5%	9min
ctrl35K	830	65.9%	54min	853	64.2%	20min	860	63.6%	21min	548	99.9%	68min
ctrl50K	1197	65.9%	106min	1148	68.7%	41min	1163	67.8%	38min	789	99.9%	274min
Moyenne	498	62.4%	35min	491	63.3%	10min	578	49.6%	10min	333	92.7%	53min
industA	188	56.2%	188s	192	55.0%	54s	363	29.8%	40s	116	90.7%	111s
industB	350	67.3%	16min	347	67.9%	3min	1110	21.2%	4min	236	99.8%	9min
industC	400	66.0%	16min	407	64.9%	4min	1161	22.7%	3min	265	99.6%	12min
industD	492	66.0%	17min	585	55.5%	6min	6469	5.1%	4min	326	99.6%	20min
industE	533	66.2%	25min	550	64.1%	7min	966	36.5%	5min	354	99.7%	24min
industF	533	66.7%	17min	546	65.0%	7min	785	45.2%	8min	362	98.4%	25min
industG	1180	64.8%	123min	1150	66.4%	29min	1452	52.6%	29min	776	98.4%	141min
Moyenne	525	64.7%	31min	540	62.7%	8min	1758	30.4%	8min	348	98.0%	33min

Tableau C.11: *Performances des différents algorithmes pour le partitionnement en clusters*

	hyperFBB			bisect*			cone			MDOC		
Circuit	#AVCs	tdr	t_CPU	#AVCs	tdr	t_CPU	#AVCs	tdr	t_CPU	#AVCs	tdr	t_CPU
glue5K	89	7.3%	295s	81	8.4%	27s	175	3.7%	8s	65	10.3%	141s
op5K	23	30.2%	78s	33	21.1%	15s	153	4.5%	9s	14	49.5%	141s
ctrl15K	96	18.6%	8min	53	33.9%	1min	63	28.5%	1min	55	32.6%	11min
glue15K	398	4.9%	57min	425	4.6%	4min	445	4.4%	2min	276	7.7%	15min
op15K	80	25.0%	9min	112	17.8%	1min	443	4.5%	2min	103	19.4%	14min
ctrl35K	397	11.5%	48min	271	16.8%	13min	275	16.6%	13min	193	23.6%	66min
ctrl50K	600	11.0%	102min	464	14.2%	28min	496	13.3%	25min	290	22.7%	252min
Moyenne	240	15.5%	33min	205	16.7%	7min	293	10.8%	6min	142	23.7%	52min
industA	49	18.0%	158s	53	16.6%	29s	251	3.5%	18s	29	30.2%	230s
industB	134	14.6%	15min	129	15.2%	2min	832	2.4%	2min	118	16.6%	13min
industC	107	20.6%	14min	94	23.4%	2min	877	2.5%	1min	69	31.9%	17min
industD	83	32.6%	14min	81	33.4%	2min	6294	0.4%	3min	81	33.4%	23min
industE	147	20.0%	21min	151	19.5%	3min	725	4.5%	3min	154	19.8%	28min
industF	91	32.5%	13min	90	32.9%	3min	378	7.8%	3min	73	40.5%	29min
industG	278	22.9%	107min	252	25.3%	13min	598	10.6%	13min	201	31.7%	112min
Moyenne	127	23.0%	29min	121	23.8%	4min	1422	4.5%	4min	104	29.2%	31min

Tableau C.12: *Performances des différents algorithmes pour le partitionnement en AVCs*

	hyperFBB			bisect*			cone			MDOC		
Circuit	#XC3064	tdr	t_CPU	#XC3064	tdr	t_CPU	#XC3064	tdr	t_CPU	#XC3064	tdr	t_CPU
glue5K	127	17.6%	303s	124	18.2%	32s	194	11.5%	11s	95	23.5%	79s
op5K	47	50.6%	84s	49	48.5%	17s	207	11.5%	15s	32	74.3%	77s
ctrl15K	177	34.8%	9min	117	52.6%	2min	130	47.3%	2min	122	50.4%	515s
glue15K	539	12.4%	58min	505	13.3%	4min	635	10.6%	3min	385	17.4%	12min
op15K	143	47.9%	9min	140	49.0%	1min	618	11.9%	3min	190	36.7%	11min
ctrl35K	568	27.5%	51min	430	36.4%	15min	447	35.0%	16min	293	53.4%	62min
ctrl50K	839	26.9%	102min	662	34.4%	32min	687	32.8%	29min	428	52.7%	246min
Moyenne	349	31.1%	34min	290	36.1%	8min	417	22.9%	8min	221	44.1%	49min
industA	81	37.2%	173s	66	45.7%	29s	152	19.8%	25s	54	55.6%	2min
industB	205	32.8%	15min	204	33.0%	2min	916	7.3%	3min	176	38.2%	10min
industC	167	45.2%	15min	171	44.1%	2min	943	8.0%	2min	121	62.3%	12min
industD	153	60.6%	14min	170	54.6%	3min	6371	1.5%	3min	137	67.7%	19min
industE	265	38.3%	24min	239	42.2%	4min	784	12.9%	3min	246	41.0%	24min
industF	184	55.1%	15min	189	53.7%	4min	458	22.1%	4min	147	69.0%	24min
industG	494	44.2%	116min	471	46.3%	17min	787	27.7%	16min	357	61.1%	108min
Moyenne	221	44.8%	29min	216	45.7%	5min	1487	14.2%	5min	177	56.4%	28min

Tableau C.13: *Performances des différents algorithmes pour le partitionnement en FPGAs XC3064*

	hyperFBB			bisect*			cone			MDOC		
Circuit	#AVBs	tdr	t_CPU	#AVBs	tdr	t_CPU	#AVBs	tdr	t_CPU	#AVBs	tdr	t_CPU
mix50K3	9	49.2%	53s	8	55.4%	8s	195	2.3%	2s	6	72.3%	8s
mix50K4	14	30.3%	52s	10	42.4%	7s	185	2.3%	2s	8	51.4%	4s
mix55K5	10	46.4%	59s	8	58.0%	8s	204	2.3%	3s	6	75.4%	5s
Moyenne	11	42.0%	55s	9	51.9%	8s	195	2.3%	2s	7	66.4%	6s
industK	109	44.3%	18min	88	54.8%	4min	89	54.2%	5min	72	67.1%	2min
industL	131	36.8%	100min	108	44.7%	10min	106	45.5%	11min	86	56.1%	6min
industM	221	21.8%	421min	184	26.2%	31min	188	25.7%	21min	152	31.7%	21min
Moyenne	154	34.3%	180min	127	41.9%	15min	128	41.8%	12min	103	51.6%	10min

Tableau C.14: *Performances des différents algorithmes pour le partitionnement en AVBs*

Annexe D

Evaluation des performances des outils de partitionnement

Les outils de partitionnement ont été testés de manière indépendante : Nous avons appliqué à un ensemble de *netlists* d'AVCs les différentes combinaisons d'algorithmes dans partBoard; à un ensemble de *netlists* de *clusters* les combinaisons d'algorithmes dans partMeta et à un ensemble de *netlists* de CPBs les combinaisons d'algorithmes dans partCluster ainsi que les combinaisons d'algorithmes dans partMeta.
Les *benchmarks* utilisés pour ces expériences sont présentés dans le tableau 8.2.

Les résultats expérimentaux détaillés sont présentés dans les tableaux D.4, D.2, D.3 et D.1.

phase 1 phase 2 phase 3	bisect* bisect* multisect*		bisect* bisect* MDOC	
Circuit	#clusters	t_CPU	#clusters	t_CPU
glue5K	188	6min	167	1min
op5K	105	1min	107	1min
ctrl15K1	272	8min	274	8min
glue15K1	721	72min	796	11min
op15K1	290	9min	302	9min
ctrl35K1	733	57min	714	53min
ctrl50K1	1072	127min	1027	115min
Moyenne	483	40min	484	28min
industA	160	3min	146	2min
industB	353	13min	310	9min
industC1	395	16min	340	12min
industD1	425	22min	414	21min
industE1	552	35min	472	21min
industF1	307	10min	297	10min
industG1	1034	105min	990	93min
Moyenne	461	29min	424	24min

Tableau D.1: *Performances des différents algorithmes pour le partitionnement en clusters*

phase 1	bisect*		bisect*		bisect*		bisect*		hyperFBB		hyperFBB	
phase 2	bisect*		bisect*		hyperFBB		hyperFBB		hyperFBB		hyperFBB	
phase 3	multisect*		MDOC		multisect*		MDOC		multisect*		MDOC	
Circuit	#AVCs	t_CPU	#AVCs	t_CPU	#AVCs	t_CPU	#AVCs	t_CPU	#AVCs	t_CPU	#AVCs	t_CPU
glue5K	73	48min	82	12min	71	94min	81	16min	71	124min	84	25min
op5K	36	2min	34	1min	36	4min	35	2min	34	6min	28	3min
ctrl15K1	85	10min	80	8min	106	56min	101	11min	105	72min	104	18min
glue15K1	307	1373min	354	12min	288	870min	358	20min	287	1208min	353	81min
op15K1	94	13min	98	10min	94	13min	101	11min	94	33min	100	21min
ctrl35K1	249	89min	270	53min	310	381min	330	64min	325	600min	343	109min
ctrl50K1	374	210min	434	112min	450	763min	491	134min	472	1128min	529	240min
Moyenne	174	249min	193	30min	194	312min	214	37min	198	453min	220	71min
industA	43	4min	43	4min	53	10min	51	3min	51	11min	54	5min
industB	99	30min	107	11min	110	47min	129	11min	125	92min	127	28min
industC1	68	24min	84	15min	87	36min	110	15min	100	128min	103	39min
industD1	76	29min	91	30min	88	33min	98	26min	84	49min	84	35min
industE1	103	47min	133	27min	139	99min	165	27min	149	25min	167	52min
industF1	64	16min	78	11min	74	21min	83	11min	77	31min	95	16min
industG1	227	148min	257	109min	269	192min	312	109min	271	332min	321	229min
Moyenne	97	43min	113	30min	117	61min	135	29min	122	95min	136	58min

Tableau D.2: *Performances des différents algorithmes pour le partitionnement de CPBs en AVCs*

phase 1	bisect*		bisect*		hyperFBB		hyperFBB	
phase 2	bisect*		bisect*		hyperFBB		hyperFBB	
phase 3	multisect*		MDOC		multisect*		MDOC	
Circuit	#AVCs	t_CPU	#AVCs	t_CPU	#AVCs	t_CPU	#AVCs	t_CPU
ctrl15K2	97	2min	102	1min	100	3min	108	1min
glue15K2	308	27min	333	2min	310	32min	332	7min
op15K2	100	2min	87	1min	100	2min	97	1min
ctrl35K2	288	22min	334	2min	298	25min	346	4min
ctrl50K2	431	45min	513	5min	450	49min	525	8min
Moyenne	244	19min	274	2min	251	22min	282	4min
industC2	97	4min	110	1min	113	5min	130	1min
industD2	89	2min	93	1min	98	3min	103	1min
industE2	158	10min	177	1min	180	14min	206	3min
industF2	89	3min	91	1min	88	3min	100	1min
industG2	287	16min	314	4min	298	21min	331	8min
Moyenne	144	7min	157	2min	155	9min	174	3min

Tableau D.3: *Performances des différents algorithmes pour le partitionnement de Clusters en AVCs*

phase 1	bisect*		bisect*		bisect*		bisect*		hyperFBB		hyperFBB	
phase 2	bisect*		bisect*		hyperFBB		hyperFBB		hyperFBB		hyperFBB	
phase 3	multisect*		MDOC		multisect*		MDOC		multisect*		MDOC	
Circuit	#AVBs	t_CPU	#AVBs	t_CPU	#AVBs	t_CPU	#AVBs	t_CPU	#AVBs	t_CPU	#AVBs	t_CPU
ctrl50K2	3	29s	3	30s	3	240s	4	150s	3	274s	4	151s
ctrl50K3	13	189s	13	41s	14	457s	16	138s	13	368s	15	294s
mix50K3	6	23s	6	13s	8	126s	8	64s	8	131s	8	66s
mix50K4	6	40s	7	11s	7	133s	10	53s	7	136s	10	54s
mix55K5	6	13s	7	13s	7	106s	7	66s	7	108s	7	66s
Moyenne	7	49s	7	22s	8	212s	9	94s	8	203s	8	631s
industG2	12	2min	9	27s	8	7min	15	4min	8	7min	15	4min
industG3	10	2min	11	10s	10	6min	11	66s	10	6min	11	1min
industH	41	17min	45	4min	45	42min	49	6min	45	51min	52	12min
industI	43	14min	47	4min	48	27min	54	6min	46	35min	54	17min
industK	67	31min	70	13min	75	78min	81	17min	75	74min	84	27min
Moyenne	35	13min	36	4min	37	32min	42	7min	37	35min	43	12min

Tableau D.4: *Performances des différents algorithmes pour le partitionnement en AVBs*

Annexe E

Evaluation des performances du partitionnement pour l'architecture hiérarchique de l'émulateur CELARO

Nous avons testé deux enchaînements différents d'outils de partitionnement : partCluster suivi de partMeta et de partBoard (cf. tableaux E.1 et E.2) ainsi que partMeta suivi de partCluster et partBoard (cf. tableaux E.3, E.4, E.5). L'outil partCluster a été réalisé avec la combinaison d'algorithmes bisect*-bisect*-MDOC (BiBiOr).
L'outil de partitionnement en cartes, partBoard, utilise trois enchaînements d'algorithmes : bisect*-bisect*-multisect* (BiBiMu), bisect*-bisect*-MDOC (BiBiOr) et hyperFBB-hyperFBB-multisect* (FlFlMu).
L'outil partMeta a été réalisé avec cinq combinaisons différentes d'algorithmes. Chacun des tableaux correspond à une version de l'outil partMeta suivi des différentes version de l'outil partBoard. Ainsi, nous trouvons

- partMeta avec l'enchaînement des algorithmes bisect*-multisect* (BiMu) suivi des différentes versions de l'outil partBoard dans le tableau E.1.
- partMeta avec l'enchaînement des algorithmes bisect*-MDOC* (BiOr) suivi des trois versions de l'outil partBoard dans le tableau E.2.
- partMeta avec l'enchaînement des algorithmes bisect*-bisect*-multisect* (BiBiMu) suivi des différentes versions de l'outil partBoard dans le tableau E.3.
- partMeta avec l'enchaînement des algorithmes bisect*-hyperFBB-multisect* (BiFlMu) suivi des différentes versions de l'outil partBoard dans le tableau E.4.
- partMeta avec l'enchaînement des algorithmes bisect*-bisect*-MDOC (BiBiOr) suivi des différentes versions de l'outil partBoard dans le tableau E.5.

Nous distinguons entre les résultats expérimentaux obtenus sur des *netlists* générées et ceux obtenus sur des *netlists* industrielles.

partCluster BiBiOr , partMeta BiMu						
partBoard	BiBiMu		BiBiOr		FlFlMu	
Circuit	#AVB	t_CPU	#AVB	t_CPU	#AVB	t_CPU
ctrl50K	17	114'	16	108'	16	113'
mix50K4	8	37'	7	37'	8	37'
mix55K2	13	45'	11	45'	13	45'
mix80K	20	194'	22	181'	25	202'
mix90K	20	166'	19	165'	19	175'
mix100K	25	306'	24	301'	23	328'
mix110K	32	393'	29	377'	36	394'
mix120K	27	367'	32	321'	29	367'
mix130K	33	338'	31	320'	30	347'
mix140K	29	378'	31	356'	31	383'
Moyenne	22	234'	22	221'	23	239'
indust1	5	12'	4	12'	5	13'
indust2	14	56'	16	54'	15	56'
indust3	21	166'	20	95'	21	98'
indust4	17	12'	17	10'	16	10'
indust5	25	167'	27	155'	27	164'
indust6	42	614'	38	453'	44	633'
indust7	52	160'	49	160'	51	161'
indust8	47	371'	52	229'	49	389'
indust9	84	667'	86	401'	85	678'
indust10	204	3421'	208	1216'	204	3543'
Moyenne	51	565'	52	279'	52	575'

Tableau E.1: *Enchaînement partCluster, partMeta, partBoard*

partCluster BiBiOr , partMeta BiOr						
partBoard	BiBiMu		BiBiOr		FlFlMu	
Circuit	#AVB	t_CPU	#AVB	t_CPU	#AVB	t_CPU
ctrl50K	14	90'	14	90'	14	94'
mix50K4	7	35'	6	35'	7	35'
mix55K2	10	41'	9	41'	10	41'
mix80K	17	170'	17	168'	16	170'
mix90K	20	154'	15	154'	16	156'
mix100K	18	295'	16	292'	19	296'
mix110K	26	347'	23	346'	25	351'
mix120K	24	306'	21	303'	24	308'
mix130K	23	301'	25	300'	23	304'
mix140K	22	343'	21	341'	22	346'
Moyenne	18	208'	17	207'	18	210'
indust1	5	10'	5	10'	5	11'
indust2	14	48'	12	48'	14	48'
indust3	19	86'	19	86'	19	93'
indust4	17	10'	17	8'	16	8'
indust5	24	170'	27	150'	26	173'
indust6	38	264'	43	241'	41	268'
indust7	50	120'	44	120'	49	123'
indust8	45	242'	50	185'	49	245'
indust9	81	739'	82	424'	81	766'
indust10	199	3274'	192	936'	211	2989'
Moyenne	49	496'	49	221'	51	472'

Tableau E.2: *Enchaînement partCluster, partMeta, partBoard*

partMeta BiBiMu , partCluster BiBiOr						
partBoard	BiBiMu		BiBiOr		FlFlMu	
Circuit	#AVB	t_CPU	#AVB	t_CPU	#AVB	t_CPU
ctrl50K	18	171'	12	169'	14	171'
mix50K4	7	51'	6	51'	7	51'
mix55K2	8	79'	8	79'	8	79'
mix80K	16	304'	16	300'	17	306'
mix90K	16	270'	16	268'	17	270'
mix100K	23	525'	22	515'	24	540'
mix110K	30	913'	28	891'	36	926'
mix120K	29	590'	27	585'	27	604'
mix130K	28	665'	27	645'	31	718'
mix140K	28	776'	26	741'	29	842'
Moyenne	20	434'	19	424'	21	451'
indust1	5	381'	5	382'	5	381'
indust2	21	341'	21	318'	21	335'
indust3	26	185'	27	181'	28	195'
indust4	18	15'	16	12'	18	17'
indust5	39	287'	40	213'	38	276'
indust6	54	638'	46	377'	51	626'
indust7	78	392'	69	392'	73	395'
indust8	57	871'	58	453'	63	903'
indust9	104	1443'	106	909'	110	1556'
indust10	236	2438'	202	1337'	245	2257'
Moyenne	64	649'	59	408'	65	644'

Tableau E.3: *Enchaînement partMeta, partCluster, partBoard*

partMeta BiFlMu , partCluster BiBiOr						
partBoard	BiBiMu		BiBiOr		FlFlMu	
Circuit	#AVB	t_CPU	#AVB	t_CPU	#AVB	t_CPU
ctrl50K	32	686'	28	559'	34	689'
mix50K4	10	121'	9	121'	8	122'
mix55K2	11	192'	14	192'	12	192'
mix80K	28	671'	25	628'	26	751'
mix90K	32	778'	24	691'	26	820'
mix100K	58	1508'	56	1012'	58	1455'
mix110K	76	2347'	67	1768'	77	2427'
mix120K	53	1348'	51	1081'	56	1478'
mix130K	52	1579'	52	1234'	61	1645'
mix140K	53	1389'	49	1020'	54	1349'
Moyenne	41	1080'	38	847'	41	1112'
indust1	7	170'	6	154'	7	184'
indust2	28	353'	29	274'	29	337'
indust3	35	523'	36	505'	34	549'
indust4	22	31'	20	21'	22	29'
indust5	49	509'	48	320'	53	489'
indust6	52	751'	48	415'	54	738'
indust7	92	650'	84	650'	90	651'
indust8	69	1057'	63	551'	73	1112'
indust9	144	1962'	140	1178'	152	1889'
indust10	310	3925'	293	2150'	317	3837'
Moyenne	81	993'	77	622'	83	982'

Tableau E.4: *Enchaînement partMeta, partCluster, partBoard*

partMeta BiBiOr , partCluster BiBiOr						
partBoard	BiBiMu		BiBiOr		FlFlMu	
Circuit	#AVB	t_CPU	#AVB	t_CPU	#AVB	t_CPU
ctrl50K	13	91'	13	89'	13	90'
mix50K4	8	44'	7	37'	8	37'
mix55K2	8	41'	7	41'	8	41'
mix80K	16	174'	14	174'	18	178'
mix90K	14	148'	12	148'	15	149'
mix100K	20	274'	18	271'	20	278'
mix110K	27	360'	25	350'	27	360'
mix120K	23	325'	21	321'	24	328'
mix130K	24	324'	22	318'	25	328'
mix140K	23	354'	22	353'	24	368'
Moyenne	18	213'	16	210'	18	216'
indust1	5	13'	5	13'	5	14'
indust2	15	61'	15	56'	15	67'
indust3	24	107'	23	105'	25	107'
indust4	18	13'	17	12'	18	13'
indust5	32	187'	38	173'	33	195'
indust6	44	249'	43	194'	41	261'
indust7	63	137'	60	137'	67	140'
indust8	58	410'	54	203'	62	435'
indust9	93	669'	90	390'	102	650'
indust10	198	2860'	191	522'	201	2045'
Moyenne	55	471'	54	181'	57	393'

Tableau E.5: *Enchaînement partMeta, partCluster, partBoard*