Problème d'optimisation quadratique pour la synthèse de circuits VLSI et de programmes flot de données optimaux

Benoit Delahaye

20 juin 2006

Table des matières

Chapitre 1

Introduction

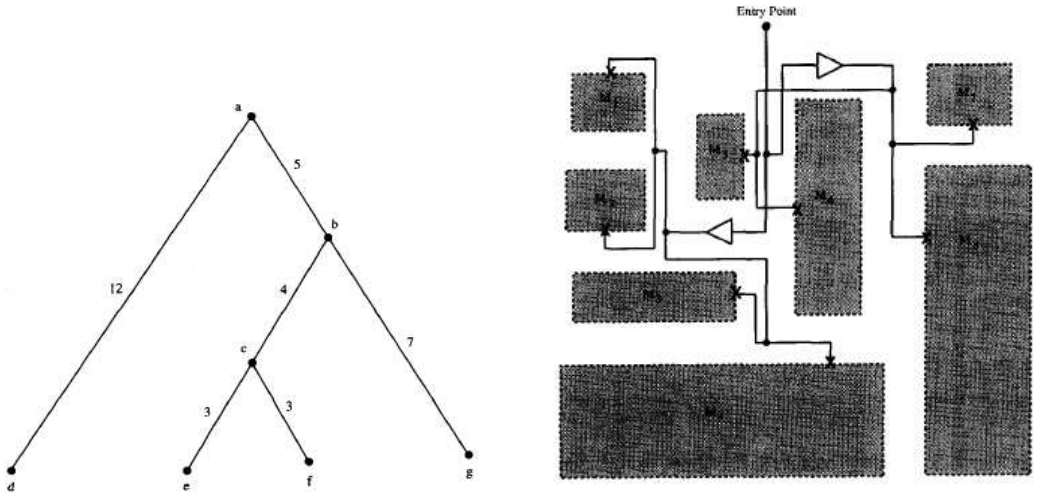
La consommation électrique dans les circuits est devenue aujourd'hui un problème majeur dans de nombreux domaines. Cette consommation provient de deux effets en particulier. En premier lieu, l'augmentation des performances des systèmes électroniques et leur miniaturisation entraine un nombre de composants croissant dans un environnement de plus en plus restreint, et donc une consommation accrue pour des batteries réduites. Le deuxième, et celui qui nous interresse particulièrement, concerne la miniaturisation des composants qui permet la fabrication de circuits de plus en plus complexes. Dans de tels circuits, une grande partie de la consommation provient de la nécéssité d'assurer une horloge synchrone en tout point.

En effet, dans les circuits synchrones, l'horloge a pour but de synchroniser les transferts de données entre les différents éléments et de définir les instants précis où le circuit peut changer d'état. Idéalement, puisque cette horloge est nécéssaire au bon fonctionnement du circuit, le signal qu'elle produit devrait être reçu simultanément par tous les éléments du circuit. En pratique, en revanche, le signal d'horloge est sujet à deux catégories de retards. La première cause de retards dans la délivrance du signal d'horloge provient du fait que ce signal est soumis à des calculs logiques et à des duplications qui ne sont pas instantanés. La solution consisterait à réduire les délais produits par ces calculs et routages, et à augmenter la tolérance des composants à des retards faibles. En revanche, la deuxième cause de retards est due à la distance entre les composants. Le signal ne se propageant pas instantanément, des retards apparaissent, fonction de la distance parcourue par le signal. Pour minimiser ces retards, la solution classique est de s'arranger pour que les distances parcourues par le signal d'horloge (et donc les délais induits) pour atteindre chaque élément soient identiques.

Par exemple, un algorithme de calcul de routage d'horloge, présenté dans [4], représente le routage par un arbre binaire de distribution et assure que le délai induit entre un noeud intermédiaire et chacune des feuilles accessibles à partir de ce noeud est le même (c.f. Fig. 1.1(a)). Des calculs sont ensuite effectués pour le placement des routeurs (nœuds intermédiaires de l'arbre) sur le circuit par rapport aux composants (feuilles de l'arbre) et le résultat est implanté physiquement (c.f Fig 1.1(b)).

L'inconvénient de cette technique est que de très nombreux routeurs d'horloge sont implantés dans un circuit et que ce sont finalement ces routeur et l'étendue de l'horloge qui sont responsables de la plus grande partie de la consommation d'énergie. En effet, ces routeurs fonctionnent en permanence pour des parties de circuits qui ne sont peut être pas utilisées.

Le but de ce stage était à l'origine d'essayer de palier à cet inconvénient en utilisant la technique du "clock-gating" (présentée en 2.1) pour éteindre l'horloge des parties inutilisées du circuit. Après une période de maturation et d'étude des techniques existantes, nous avons décidé de focaliser notre approche sur les mémoires internes (registres) des circuits pour les grouper et les éteindre lorsqu'elles ne sont pas utilisées. Cette décision nous a amené à utiliser une



(a) Arbre binaire (b) Implantation physique

FIG. 1.1 – Exemple de routage d'horloge

représentation des circuits qui nous a permi dans un premier temps de formuler le problème en un problème d'optimisation quadratique sur des variables booléennes et dans un deuxième temps d'étudier le problème de synthèse d'architectures GALS (Globally Asynchronous, Locally Synchronous) optimales en temps de calcul dont la formalisation se rapproche très fortement de notre problème initial. En effet, une architecture GALS peut être représentée dans un langage flot de données (par exemple SIGNAL) où l'on décrit les interractions entre processus sous forme de signaux qui peuvent être présents ou absents à tout instant. Dans de telles architectures, une grande partie du temps de calcul provient de l'évaluation des signaux à chaque top d'horloge. Il nous a donc semblé intéressant de regrouper entre eux les processus utilisant les mêmes signaux à certains instants pour limiter le nombre d'évaluation de ces signaux. Si ce problème ne ressemble pas vraiment, à priori, au problème de réduction de la consommation dans les circuits VLSI, il est néanmoins possible de le décrire comme un problème d'optimisation d'une fonction quadratique très semblable à celle que l'on obtient pour ce dernier.

La résolution de problèmes d'optimisation quadratiques à variables booléennes est un sujet bien connu et étudié dans la littérature actuelle (c.f. [2], [9], [5], [8], [10]). Nous nous sommes donc basés dans un premier temps sur des approches existantes (présentées dans [2] et [6]) pour résoudre ce problème de manière générale, puis nous avons développé une approche un peu plus ciblée pour notre problème à base de parcours d'arbre des solutions et d'heuristiques.

Nous présenterons donc dans un premier temps les problèmes considérés et les méthodes existantes qui nous ont amené à notre approche. Nous exposerons par la suite les notations, modèles et formalisations que nous avons utilisé pour réduire nos problèmes en problèmes d'optimisation quadratique. Cela nous amenera à étudier les méthodes utilisées pour la résolution de tels problèmes, et nous finirons par quelques tests comparatifs entre ces méthodes.

Chapitre 2

Méthodes existantes étudiées

A l'heure actuelle, de nombreuses méthodes existent pour réduire la consommation dans les circuits et pour la synthèse d'architectures GALS optimales. Nous avons choisi de ne présenter ici que les méthodes se rapprochant le plus de ce que nous voulons faire, et surtout celles qui nous ont le plus inspiré dans notre travail.

2.1 Le "clock-gating" pour la réduction de consommation dans les circuits VLSI

En ce qui concerne la réduction de consommation d'énergie dans les circuits, nous allons expliquer dans un premier temps ce qu'est le "clock-gating", technique permettant de couper l'horloge dans certaines parties du circuit, puis deux utilisations de cette technique qui nous ont inspiré pour notre travail. Nous finirons par présenter comment nous comptons utiliser le "clock-gating" pour couper l'horloge des registres internes lorsqu'ils ne sont pas utilisés.

2.1.1 Le "clock-gating"

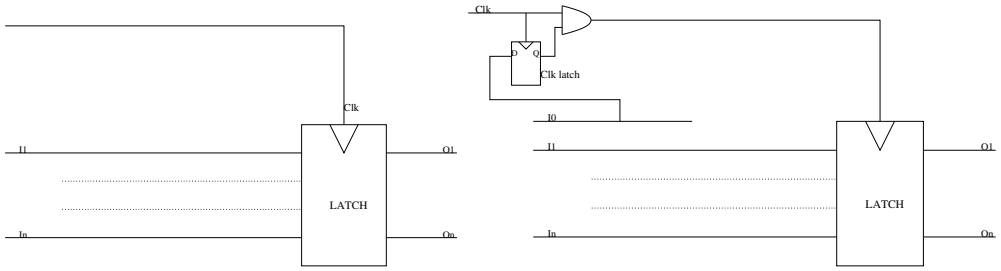
Comme nous l'avons expliqué plus haut, une grande partie de la consommation d'énergie dans les circuits provient de l'alimentation des nombreux routeurs d'horloge, dont le but est d'avoir une horloge synchrone en tout point du circuit.

Le "clock-gating" est une technique permettant de couper l'horloge (et donc les routeurs) dans certaines parties inactives du circuit. Cette technique consiste à ajouter des portes "ET" entre le signal d'horloge et un signal ajouté déterminant si la partie du circuit en question doit être active (c.f. Fig. 2.1). Lorsque la partie doit être inactive, le signal d'horloge s'arrête à la porte "ET" et n'a pas besoin d'être propagé.

Cette technique reste néanmoins très générale, elle permet de couper l'horloge de n'importe quelle partie du circuit à n'importe quel instant. Pour l'utiliser de manière efficace, il reste donc à determiner quelles parties du circuit on veut éteindre et à quels instants.

2.1.2 Détection de l'exclusion mutuelle dans un circuit

L. Benini, P. Vuillod, G. De Micheli et C. Coelho présentent dans un article de 1996 [1] une technique utilisant le flot de contrôle pour détecter des exclusions mutuelles lors du fonctionnement d'une machine et les utiliser afin de réduire la consommation d'énergie en éteignant l'horloge des sous-machines ne pouvant fonctionner.



(a) Avant clock-gating (b) Après clock-gating

FIG. 2.1 – Clock-gating sur un verrou (Latch)

Graphe de séquencement, exclusion mutuelle

A partir d'une spécification impérative de haut niveau de la machine (en VHDL par exemple), on peut extraire une représentation hierarchique associant le flot de contrôle et le flot de données. Cette représentation, le graphe de séquencement [7] (c.f. Fig. 2.2), contient deux types de sommets : les *opérations* et les *liens*. Ces derniers pointent vers des graphes de séquencement de plus bas niveaux dans la hiérarchie. Dans un tel graphe, les feuilles représentent les opérations basiques (le flot de données) alors que les *liens* représentent le flot de contrôle de plus haut niveau (lors d'une boucle ou d'une conditionnelle par exemple).

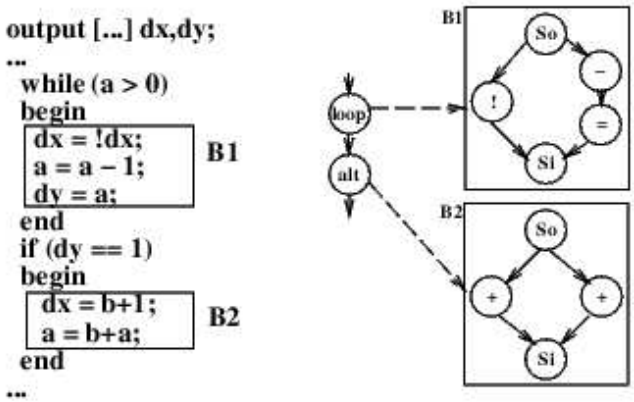


FIG. 2.2 – Graphe de séquencement à partir d'une spécification HDL

L'avantage d'une telle représentation est que les liens permettent d'identifier des sections mutuellement exclusives de l'exécution (i.e. des sections que ne peuvent pas être exécutées de manière concurrente). En effet, dès lors qu'il existe un chemin reliant deux *liens* dans le graphe de séquencement, les graphes pointés par ces *liens* représentent des parties mutuellement exclusives.

La génération d'une première machine à état finis (FSM) conforme à la spécification se base sur le graphe de séquencement. Cette première machine est appelée *implémentation monolithique du controlleur*. Elle a la même structure que le graphe séquencement et admet donc une décomposition naturelle conforme aux exclusions mutuelles détectées plus haut.

Partition de la FSM et raffinements

Une fois cette première machine construite, les auteurs proposent un partitionnement de la FSM en plusieurs étapes qui permet d'obtenir une décomposition "équilibrée" en un nombre de sous-FSM spécifié par l'utilisateur, chacune d'entre elles ayant approximativement le même nombre d'états. Le partitionnement est effectué de la manière suivante :

1. *Marking* : Marquage de certains sommets du controlleur. Les sommets marqués sont ceux associés à (i) des évaluations de conditions de boucles, (ii) des appels de procédures, (iii) des retours de procédures, (iv) des évaluations de conditionnelles, et (v) des points de jonctions de conditionnelles. Ces sommets seront appelés *racines*. Ce sont les sommets du graphe ayant plusieurs arrêtes de sortie et/ou d'entrée.
2. *Collapsing* : Transformation de la FSM marquée en graphe de transitions pondéré. Les séquences de sommets non marqués entre les racines sont fusionnées en *noeuds simples* (ce sont les sommets ayant une seule entrée/sortie), auxquels on assigne un poids égal au nombre de sommets fusionnés pour le créer. Le poids des racines est fixé à 1.
3. *Clustering* : Dernière phase de regroupement des sommets. On choisit en premier pour le regroupement les noeuds simples de poids faible. Les noeuds sélectionnés sont fusionnés avec les racines dont ils dérivent et le poids des racines est augmenté du poids des noeuds qui sont fusionnés en elles. On répète cette étape jusqu'à l'obtention du nombre désiré d'états.

La partition $\Pi(S) = \{S_1, \ldots, S_n\}$ est basée sur la hiérarchie naturelle du controlleur, et correspond donc aux exclusions mutuelles étudiées plus haut.

Implémentation et résultats

Une fois la partition $\Pi(S)$ obtenue, les auteurs proposent une méthode pour construire le controlleur comme une composition de FSMs communiquantes. Ils assurent de plus qu'une seule des sous-FSMs sera active à tout instant. La méthode proposée ajoute à toutes les sous-FSMs S_i un état dit "reset" ($s_{0,i}$) dans lequel elles seront quand elles sont inactives. Toute transition interne à une sous-FSM est conservée, et toute transition $p \to q$ entre deux sous-FSMs distinctes S_i et S_j est transformée en $p \to s_{0,i}$ et $s_{0,j} \to q$. Des signaux d'activation des sous-FSMs sont ajoutés pour permettre dans l'opération ci dessus de n'effectuer $s_{0,j} \to q$ que lorsque $p \to s_{0,i}$ est terminée (et donc lorsque S_i envoie le signal d'activation de S_j).

Dans ce système, on remarquera que les sous-FSMs sont inactives dès lors qu'elles sont dans leur état "reset". En arrêtant leur horloge dès qu'elles sont dans leur état "reset", on économise donc une quantité non négligeable d'énergie. Néanmoins, il faut faire attention à bien "réveiller" les sous-FSMs lorsque les signaux d'activation sont émis. La fonction d'activation des sous-FSMs est donc assez simple : elle vérifie que la sous-FSM est bien dans son état "reset" et qu'aucun signal d'activation lui étant destiné n'a été émis.

Après de nombreux tests, les auteurs ont obtenus en moyenne une augmentation de la taille des machines de 30% à cause des états et signaux ajoutés, mais un gain en consommation de 37%. Leur méthode permet donc un gain non négligeable en consommation, mais induit une augmentation de la taille des systèmes.

2.1.3 Gestion de la mémoire

La méthode que nous avons vu au dessus permettait de réduire la consommation des systèmes avec pour désavantage d'en augmenter la taille. On peut alors se demander s'il ne serait pas possible de réduire la taille des systèmes sans perdre le gain obtenu en consommation. La méthode

que proposent Cao Cao et Bengt Oelmann dans leur article [3] permet d'obtenir de tels résultats. En effet, ils proposent de combiner des mémoires d'état synchrones (locales) et asynchrones (globales) pour obtenir un système équivalent avec une mémoire de taille réduite et donc moins de pertes d'énergie. Le "clock-gating" peut de plus être utilisé ici pour éteindre les éléments de la mémoire globale non utilisés.

Motivations

Il existe deux manières de décomposer une FSM en sous-FSM. La première, comme on l'a vu au dessus, consiste en une partition utilisant une mémoire d'états séparée pour les sous-FSM : chaque sous-FSM a sa propre mémoire locale. Si on suppose que la partition est $S = \{S_1, \ldots, S_n\}$, le nombre total de bits nécéssaires pour différencier les états locaux sera :

$$\sum_{i=1}^{n} \lceil \mathsf{log}_2 |S_i| \rceil$$

Celà correspond à un nombre de bits énorme sachant que les sous-FSM ne fonctionneront pas en même temps.

La deuxième méthode de décomposition consiste à utiliser une mémoire d'états globale permettant de différencier laquelle des sous-FSM est active à quel instant, et une seule mémoire d'états locale partagée pour toutes les sous-FSM. Dans ce cas, le nombre de bits nécéssaires pour la mémoire globale est de $\lceil \mathsf{log}_2 n \rceil$ et le nombre de bits nécéssaires pour la mémoire locale est $\lceil \mathsf{max}(\mathsf{log}_2 |S_1|, \ldots, \mathsf{log}_2 |S_n|) \rceil$. Celà permet de réduire considérablement la taille de la mémoire nécéssaire, mais le désavantage est que la mémoire globale fonctionne en permanence et consomme donc de l'énergie inutile.

Mémoire globale asynchrone

La solution que les auteurs proposent dans cet article est d'utiliser une mémoire globale asynchrone en se basant sur le principe que les changements de sous-FSM sont peu probables, tout en conservant une mémoire locale synchrone fonctionnant en permanence.

Pour que cette méthode fonctionne, les auteurs ajoutent aux ensembles d'états de toutes les sous-FSM des états correspondant aux transitions sortantes de la sous-FSM. A chaque transition sortante correspondra un état global (*g state*) qui aura le même index que l'état de destination de la transition. Ainsi, toute transition sortante de la sous-FSM S_i vers une sous-FSM S_j sera décomposée en

- Une transition synchrone (locale) entre l'état de départ et l'état global interne à S_i ;
- Une transition asynchrone (globale) de l'état global à l'état de destination dans S_j. Elle consistera donc en une mise à jour de la mémoire globale car les deux états locaux ont le même index.

L'implémentation de cette méthode est assez simple (c.f. Fig. 2.3), et consiste en une unité synchrone de gestion de la mémoire d'états locale (LSM), une unité asynchrone de gestion de la mémoire d'états globale (GSM), et une unité synchrone de détection des états globaux permettant d'activer la gestion de mémoire globale (GDL). Elle est utile lors des changements de sous-FSM.

Le fonctionnement asynchrone de la mémoire globale permet un gain en consommation non négligeable (46 % en moyenne pour une décomposition en 2 sous-FSMs), mais l'ajout des états globaux et l'implantation de l'unité logique de détection de ces états entraine encore une fois une augmentation de la taille de la machine (les auteurs parlent de 7,5 % en moyenne).

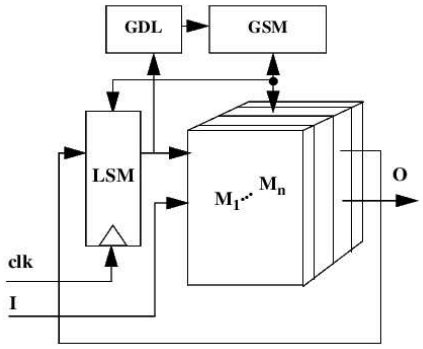


FIG. 2.3 – Modèle basé sur une gestion mixte synchrone/asynchrone de la mémoire d'états

2.1.4 Notre approche pour la gestion de la mémoire

Nous avons donc choisi pour ce stage de nous inspirer de ces deux méthodes et d'utiliser le clock-gating pour éteindre l'horloge des mémoires qui ne sont pas utilisées (sans faire de distinction entre mémoire locale et mémoire globale). En effet, si l'on considère que les états possibles d'un circuit sont entièrement décrits par les mémoires internes, on remarque que dans un comportement périodique du circuit, certaines mémoires changent peu d'état. Il pourrait donc être intéressant de regrouper entre elles les mémoires ne fonctionnant pas à certains instant pour éteindre leur horloge.

Dans l'idée, cette solution est assez proche de la solution de mémoires locales synchrones et globales asynchrones présentée au dessus, mais elle permet de regrouper les mémoires sans avoir à décomposer le circuit en sous-FSM. Dans ce contexte, la notion de mémoires locales ou globales n'a pas de sens.

La méthode que nous allons présenter par la suite (c.f.partie 3.2) a donc pour but de partitionner de manière optimale les mémoires d'un circuit en n parties (n fixé par l'utilisateur) qui ne seront actives que lorsqu'au moins un de leurs éléments est actif.

2.2 Synthèse d'architectures GALS optimales

Dans le cadre des architectures GALS et des programmes flot de données, peu de travaux existants se rapprochent de ce que nous souhaitons effectuer. En effet, les principaux travaux existant aujourd'hui ont pour but d'optimiser l'ordonnancement des tâches sur les différents processus et non, comme nous le souhaiterions, regrouper les processus entre eux pour minimiser le nombre d'échantillonnage des signaux. Les travaux qui nous ont semblé les plus intéressants sont ceux présentés dans [12], qui utilisent une méthode à base d'heuristiques et de backtracking pour construire une implémentation optimale d'un algorithme sur une machine parallèle.

Le but de cette méthode est de distribuer un programme sur différents processus et de trouver le meilleur ordonnancement pour les différentes tâches et pour les communications entre ces processus. L'algorithme est représenté par un graphe d'opération permettant de modéliser les différentes dépendances. A chaque étape, une opération prête (dont tous les prédécesseurs ont étés placés et ordonnancés) est choisie grâce à une fonction de coût, puis distribuée et ordonnancée.

Contrairement à la méthode que nous présentons en 4.2, cette méthode utilise une heuristique pour choisir le sous arbre à parcourir, et non pour couper les branches inutiles. La méthode de

backtracking permet alors de revenir en arrière lorsque le meilleur sous-arbre déterminé par la fonction de coût n'était pas unique. Dans ces travaus, le choix de la branche à parcourir et les opérations effectuées sur cette branche sont bien disjointes, ce qui ne sera pas le cas dans la méthode que nous présentons en 4.2.

Chapitre 3

Modélisation, récriture du problème

Comme nous l'avons montré dans la partie précédente, les problèmes auxquels nous sommes confrontés sont des problèmes d'optimisation, visant à trouver le minimum d'une certaine fonction de coût. Pour obtenir un problème de type classique, nous avons du jouer sur les modèles que nous avons utilisé, pour les circuits comme pour les architectures GALS, puis utiliser un certain nombre de transformations dans le but d'obtenir une fonction de coût classique. Dans les deux cas nous sommes parvenus à écrire cette fonction de coût comme une fonction quadratique, et donc à nous rapprocher de problèmes connus plus en détail dans la littérature.

Nous allons donc présenter dans un premier temps la forme classique des problèmes $MIQP$ (Mixed Integer Quadratic Programming) desquels nous souhaitons nous rapprocher, puis détailler un peu plus les modèles utilisés et les transformations faites pour arriver effectivement à de tels problèmes, dans un premier temps pour les circuits puis pour les architectures GALS.

3.1 Problème $MIQP$ classique

Un problème $MIQP$ classique est un problème d'optimisation (minimisation ou maximisation) d'une fonction quadratique à variables en partie entières sous des contraintes d'égalité et/ou d'inégalité linéaires (c.f. Def 1).

Définition 1 (Problème $MIQP$)

$$(MIQP) \quad : \quad \min\{g(x) = x^t \cdot Q \cdot x + c^t \cdot x \mid A \cdot x = b,\ A' \cdot x \leq b',\ x \in \mathbb{R}^k \times \mathbb{N}^{n-k}\}$$

avec c, b, b' des vecteurs réels de tailles réspectives n, m, p, Q une matrice symétrique réelle de taille $n \times n$, A et A' des matrices de tailles respectives $m \times n$ et $p \times n$.

Les problèmes $MIQP$ auxquels nous nous attacherons dans la suite du document auront uniquement des variables booléennes. En revanche, les matrices considérées ne sont pas nécéssairement à coefficients booléens ou même entiers. Celà est nécéssaire pour des problèmes de convexification (c.f. Part. 4.1.2).

3.2 Pour les circuits VLSI

Si le problème présenté en partie 2.1.4 correspond clairement à un problème d'optimisation, il est beaucoup plus laborieux d'obtenir une fonction de coût quadratique. Deux étapes sont nécéssaires pour cela : La première de ces étapes consiste en une formalisation du problème,

donc un choix de représentation pour les circuits et une définition formelle de la consommation considérée ; La deuxième étape quant à elle correspond à un travail de récriture du problème d'optimisation formel obtenu pour transformer la fonction de coût en fonction quadratique.

3.2.1 Formalisation de notre problème

Le but de cette première étape de formalisation est donc de choisir un modèle cohérent pour les circuits et surtout qui nous permette de définir de manière claire la consommation d'énergie que nous voulons considérer pour notre problème d'optimisation.

Modèle choisi pour les circuits

Notre but étant de regrouper entre eux les registres fonctionnant à des instants proches pour pouvoir éteindre leurs horloges, il est nécéssaire de voir apparaître de manière intuitive les instants de fonctionnement de ces registres. Les mémoires internes du circuit doivent donc avoir une importance conséquente dans la définition du circuit. Nous avons ainsi choisi de représenter un circuit par l'ensemble de ses variables (les mémoires internes) et les fonctions de transition qui leur font changer de valeur.

En effet, un circuit est une fonction booléenne qui à un certain nombre de variables d'entrées (représentant l'environnement du circuit) associe un certain nombre de variables de sorties. Les sorties physiques dépendent évidemment des entrées et de l'état courant du circuit, mais il faut aussi considerer l'état du circuit comme une variable de sortie (modifiée lors d'une transition). Une formalisation de cette définition d'un circuit est montrée en Déf. 2. Les états du circuits sont un ensemble $Q = 2^{S \cup O}$ avec S l'ensemble des variables d'états, les variables d'entrée étant I et les variables de sortie O. Le circuit est alors représenté par les fonctions de transition des variables de sortie et d'état.

Définition 2 (Circuit) *Un circuit est un n-uplet* $< I, S, O, \{T_x \mid x \in S \cup O\} >$ *où :*
- I *est l'ensemble des variables d'entrée ;*
- S *est l'ensemble des variables d'état ;*
- O *est l'ensemble des variables de sortie ;*
- $T_x : 2^I \times 2^S \to \{0, 1\}$ *représente la fonction de transition de la variable interne* $x \in S \cup O$.

On obtient alors la fonction de transition globale du circuit :

$$T : 2^S \times 2^I \to 2^S \mid T = \prod_{s \in S} T_s$$

Et la fonction globale de sortie :

$$O : 2^S \times 2^I \to 2^O \mid O = \prod_{o \in O} T_o$$

Vecteurs d'activation et consommation du circuit

Une fois le modèle du circuit choisi, il est nécéssaire de définir les instants où les mémoires internes (dont la valeur est pilotée par le comportement du circuit) sont actives. Pour pouvoir tirer parti de ces instants actifs des mémoires sans avoir à effectuer d'innombrables tests sur les circuits, nous allons nous placer dans un comportement périodique v^ω du circuit (fourni par le concepteur). Ce comportement v^ω correspond à une suite d'états du circuit $(S_1, \ldots, S_m)$ se répétant infiniment, où chaque état est le produit des valeurs des variables internes du circuit

($S_i \in 2^{S \cup O}$). On peut donc définir le vecteur d'activation d'une variable (c.f. Def. 3) comme l'ensemble des indices des états du circuit où cette variable change de valeur. Cela revient à écrire un vecteur binaire de longueur la longueur de v et où une composante vaut 1 dès lors que la variable concernée est active à cet instant.

Définition 3 (Vecteur d'activation) *Soit $x \in S \cup O$. On note $\overline{x_i^v}$ la valeur de la variable x dans l'état S_i de l'exécution périodique v^ω ($|v| = m$).*
Le vecteur $\overrightarrow{x}^v = (x_1^v, \ldots, x_{m-1}^v)$ d'activation de x relativement à v est défini tel que

$$x_i^v = 1 \iff \overline{x_i^v} \neq \overline{x_{i+1}^v}$$

Si l'on ne considère qu'une seule exécution périodique v^ω ou que cela n'introduit pas d'ambiguités, on omettra volontairement l'indice v de l'écriture de ces vecteurs.

Etant donné un circuit et une exécution périodique de ce circuit, nous voulons donc regrouper les variables internes à ce circuit (les registres) pour pouvoir leur appliquer la technique du clock-gating et ainsi réduire la consommation .

La première idée qui vient à l'esprit est de rassembler les registres ayant des domaines d'activation proches pour pouvoir les éteindre le plus longtemps possible. Supposons que le nombre de "paquets" de registres que nous voulons obtenir est fixé d'avance (n). Nous voulons donc partitionner les variables $x \in X = S \cup O$ en parties X_i tels que $X = \uplus_{1 \leq i \leq n} X_i$. Une fois une telle partition obtenue, nous voulons en évaluer la consommation. Il convient donc de définir (c.f. Def 4) le vecteur d'activation d'une partition.

Définition 4 *Soit $Y = \{\overrightarrow{x}_1, \ldots, \overrightarrow{x}_p\}$ un ensemble de vecteurs d'activations correspondant aux variables $x_1, \ldots, x_p$.*
On note $\overrightarrow{Y} = (Y_1, \ldots, Y_m)$ le vecteur d'activation de l'ensemble des variables $\{x_1, \ldots, x_p\}$ le vecteur binaire tel que $Y_k = 1 \iff \exists j \in [1,p] \;\; (x_j)_k = 1$, ie $\overrightarrow{Y} = \mathsf{Sup}_{i \in [1,p]} \; (\overrightarrow{x}_i)$.

La consommation du circuit que nous souhaitons considérer sera donc la somme des consommations de chaque partition à laquelle il faudra ajouter la consommation de la partie calculatoire du clock-gating. Nous allons ignorer cette partie en supposant qu'elle restera négligeable devant la consommation des registres (ce qui sera vrai dès que les circuits sont suffisament grands). Comme tous les éléments considérés ici sont des registres de même taille, on suppose qu'ils consomment la même quantité d'énergie, et on obtient donc que la consommation de chaque partie est proportionnelle au produit du nombre de registres de la partie par la taille "active" du vecteur d'activation.

$$C = \lambda . \sum_{i=1}^{n} |X_i| . |\overrightarrow{X}_i| \quad \text{où} \quad |\overrightarrow{X}_i| = \sum_{k=1}^{m} X_{i,k}$$

A partir de là, le problème devient purement mathématique et se résume de la manière suivante :

Problème 1 *Etant donné un ensemble $X = \{x_1, \ldots, x_p\}$ de vecteurs binaires de taille m, et un entier $n \geq 2$, trouver une partition $(X_i)_{i \in [1,n]}$ de X telle que*

$$\sum_{i=1}^{n} |X_i| . |\overrightarrow{X}_i| \;\; \textit{minimale}$$

On reconnaît bien là un problème d'optimisation, mais dont la fonction de coût est peu ordinaire (elle comporte des Sup).

3.2.2 Récriture du problème en $MIQP$

Il convient donc d'effectuer quelques récritures sur les variables pour obtenir une fonction de coût quadratique donnant les mêmes valeurs que la fonction du problème 1.

Soit $\widehat{Z}$ le vecteur de taille $n \cdot (p+m)$ suivant :

$$\left| \begin{array}{l} \widehat{Z} = (\widehat{z}_{1,1}, \ldots, \widehat{z}_{1,p}, \widehat{z}_{2,1}, \ldots, \widehat{z}_{2,p}, \ldots, \widehat{z}_{n,p}, \widehat{y}_{1,1}, \ldots, \widehat{y}_{1,m}, \ldots, \widehat{y}_{n,m}) \\ \widehat{z}_{i,j} = 1 \quad \text{si le vecteur } x_j \text{ est présent dans la partition } i \\ \widehat{y}_{i,k} \quad \text{tq. } \forall j \in [1,p],\ (x_j)_k \cdot \widehat{z}_{i,j} \leq y_{i,k} \end{array} \right.$$

Ce vecteur regroupe toutes les informations nécessaires pour résoudre le problème, c'est à dire la présence des variables dans les éléments de la partition. Les inégalités nécessaires peuvent être formulées sous forme matricielle.

De plus, avec ce vecteur, la fonction à minimiser est la suivante :

$$q(\widehat{Z}) = \sum_{i=1}^{n} (\sum_{j=1}^{p} \widehat{z}_{i,j}) \cdot (\sum_{k=1}^{m} \widehat{y}_{i,k})$$

où $\sum_{j=1}^{p} \widehat{z}_{i,j}$ représente la taille de l'ensemble X_i, et $\widehat{y}_{i,k}$ est un majorant de la composante $(X_i)_k$ du vecteur $\overrightarrow{X_i}$ d'activation de l'ensemble (X_i). $\sum_{k=1}^{m} \widehat{y}_{i,k}$ représente donc un majorant de la taille "active" de l'ensemble X_i. Lors de la minimisation de la fonction de coût, ce majorant aura nécéssairement la valeur de $|\overrightarrow{X}_i|$.

Une autre contrainte nécessaire est que chaque variable interne doit apparaitre exactement une fois au total parmi toutes les partitions, i.e. $\forall j \in [1,p],\ \sum_{i=1}^{n} \widehat{z}_{i,j} = 1$. Elle apparaitra également sous forme matricielle.

Le problème se récrit donc de la manière suivante :

Problème 2 *Etant donnés 3 entiers n, m, p, étant donnés les matrices et vecteurs suivants :*
- $Q \in \mathcal{M}_{n\cdot(m+p)}(2)$ *symétrique ;*
- $c \in \mathbb{R}^{n\cdot(m+p)}$ *;*
- $A \in \mathbb{R}^{p,n\cdot(m+p)}$ *;*
- $A' \in \mathbb{R}^{n\cdot m\cdot p,n\cdot(m+p)}$ *;*
- $b \in \mathbb{R}^{p}$ *;*
- $b' \in \mathbb{R}^{n\cdot m\cdot p}$

Trouver $z \in 2^{n\cdot(m+p)}$ tel que la fonction $q(z) = z^t \cdot Q \cdot z + c^t \cdot z$ soit minimale sous les contraintes $A \cdot z = b$ et $A' \cdot z \leq b'$.

Les matrices considérées ici dérivent directement du problème initial et sont présentées en annexe A. On reconnaît donc bien ici la forme classique des problèmes $MIQP$.

3.3 Pour les programmes flot de données

Tout comme nous l'avons expliqué précédemment pour les circuits, la reformulation de notre problème de synthèse d'architectures GALS optimales en temps de calcul en un problème $MIQP$ nécéssite un choix de modèle cohérent pour formaliser le concept de "temps de calcul optimal", et un certain nombre de récritures pour le faire entrer dans le moule des problèmes $MIQP$.

3.3.1 Formalisation

La notion de temps de calcul optimal pour les architectures GALS n'est pas tout à fait claire. Si nous ne voulons pas remettre en question le fonctionnement interne de chacun des processus, le seul paramètre sur lequel nous pouvons jouer a pour base les communications entre processus, et donc les signaux que chacun d'entre eux va évaluer. En partant de ce principe, le temps de calcul que nous allons éssayer de réduire est uniquement lié au temps perdu par chaque processus lors de l'évaluation des signaux d'entrée.

En effet, une architecture GALS comporte un ensemble de procéssus communiquant entre eux grace à un ensemble de signaux qui peuvent être partagés par un nombre de processus supérieur à deux. Ces signaux peuvent être, à tout instant, présents ou absents, ce qui pilote le fonctionnement des processus. Ainsi, à tout instant, les processus doivent évaluer la valeur des signaux qu'ils ont en entrée et agir en fonction de la présence ou non de ces signaux. Un temps de calcul considérable est donc perdu en simple évaluation de signaux pour chacun des processus. Notre but sera donc de regrouper entre eux les processus dont les ensembles de signaux d'entrée sont proches afin de n'avoir à évaluer qu'une fois les signaux d'entrée pour chaque groupe de processus. Le coût de l'évaluation des signaux d'entrée pour un groupe de processus sera alors proportionnel, à un instant donné, au nombre de signaux d'entrée du groupe s'il y a parmi ces signaux au moins un signal présent, et nul sinon.

Modèle choisi pour les architectures GALS

Cette fois ci, il est nécéssaire de faire apparaitre de manière conséquente les signaux de chaque processus dans le modèle que nous allons utiliser pour représenter les architectures GALS. Nous allons donc représenter une architecture comme un ensemble de procéssus $(P_i)_{i\in[1,n_p]}$, un ensemble de signaux V de domaine de valeurs D. Chaque processus, quant à lui, contiendra des informations sur son ensemble de signaux d'entrée, son ensemble d'états, ses états initiaux et sa fonction de transition. Nous obtenons alors la formalisation décrite en Def. 5.

Définition 5 (Architecture GALS)
Une architecture GALS est un doublet $< (P_i)_{i\in[1,n_p]}, V >$ *où :*
- V *est un ensemble de signaux de domaine de valeurs* D
- $(P_i)_{i\in[1,n_p]}$ *est un ensemble de processus de la forme* $P_i =< V_i, S_i, \widehat{s_i}, T_i >$ *avec*
  - V_i *l'ensemble des signaux d'entrée de* P_i
  - S_i *l'ensemble de ses états*
  - $\widehat{s_i}$ *ses états initiaux*
  - $T_i \subseteq S_i \times L_{S_i}(V_i) \times S_i$ *sa fonction de transition*
  - $L_s \quad : \quad V_i \rightarrow D \cup \{\perp\}$ *la fonction d'évaluation des signaux de* V_i *dans un état* s.

Vecteurs d'activation et temps de calcul

Une fois la représentation de l'architecture choisie, il convient de définir plus formellement la notion de temps de calcul considérée. Nous avons expliqué plus haut que nous nous intéresserons au nombre de signaux évalués à chaque instant pour chaque groupe de processus. Pour que ce soit cohérent, il faut donc encore une fois se placer dans le cadre d'une exécution périodique u^ω de l'architecture (fournie par le concepteur). Cette exécution correspondra à une suite d'états internes des processus $(u_1, \ldots, u_m) = ((S_i^1)_{i\in[1,n_p]}, \ldots, (S_i^m)_{i\in[1,n_p]})$ où on peut associer à chaque état global du système u_k l'ensemble des valeurs des signaux à l'instant k.On dira qu'un signal est actif à l'instant k s'il est présent (i.e. sa valeur est différente de $\perp$). On pourra donc écrire,

pour chaque signal $v \in V$, un vecteur booléen de longueur m où la $k^{\text{ième}}$ composante dit si le signal est actif à l'instant k de l'exécution.

Définition 6 (Vecteur d'activation) *Soit $v \in V$. Le vecteur $\overrightarrow{v}^u = (v_1^u, \ldots, v_m^u)$ d'activation de v relativement à u ($|u| = m$) est défini tel que*

$$v_k^u = 1 \iff L_{u_k}(v) \neq \perp$$

Si l'on ne considère qu'une seule exécution périodique u^ω ou que cela n'introduit pas d'ambiguités, on omettra volontairement l'indice u de l'écriture de ces vecteurs.

Etant donné une architecture GALS et une exécution périodique, nous voulons donc regrouper les processus pour essayer de minimiser le nombre d'évaluation de signaux . En effet, si l'on prend un ensemble $Y = \{v_{i_1}, \ldots, v_{i_p}\}$ de signaux, on pourra dire que cet ensemble est actif à un instant k dès lors qu'un de ses signaux est actif. On obtient donc une définition de l'activation d'un ensemble de signaux (c.f. Def 7).

Définition 7 *Soit $Y = \{\overrightarrow{v}_1, \ldots, \overrightarrow{v}_p\}$ un ensemble de vecteurs d'activations correspondant aux signaux $v_1, \ldots, v_p$.*
On note $\overrightarrow{Y} = (Y_1, \ldots, Y_m)$ le vecteur d'activation de l'ensemble des signaux $\{v_1, \ldots, v_p\}$ le vecteur binaire tel que $Y_k = 1 \iff \exists j \in [1, p] \quad (v_j)_k = 1$, ie $\overrightarrow{Y} = \mathsf{Sup}_{i \in [1,p]} \ (\overrightarrow{v}_i)$.

Ainsi, La consommations que l'on considérera à chaque étape u_k de l'exécution sera la somme du nombre de signaux d'entrée pour chaque partie active à cet instant. Nous allons pour cela faire abstraction des processus dans le calcul du coût d'une solution en ne considérant que les ensembles de signaux et non les ensembles de processus.

Considérons donc le problème suivant :

Problème 3 *Etant donnés un entier n et une architecture GALS $< (P_i)_{i \in [1,n_p]}, V >$, et une exécution périodique u^ω ($|u| = m$) trouver une partition $(I_j)_{j \in [1,n]}$ de $I = [1, n_p]$ telle que*

$$\sum_{k=1}^{m} (\sum_{j=1}^{n} ((Y_j)_k \cdot |Y_j|)) \quad t.q. \quad Y_j = \bigcup_{i \in I_j} V_i \quad \textit{minimale}$$

Dans ce problème, les Y_j sont les groupes de processus, $|Y_j|$ représente le nombre de variables d'entrées pour ces groupes, et $(Y_j)_k$ représente l'activation de la partition à l'instant u_k de l'exécution.

3.3.2 Récriture en $MIQP$, adaptation

Tout comme pour le problème d'optimisation de la consommation dans les circuits, un certain nombre de récritures sont nécéssaires pour obtenir un problème rentrant dans le "moule $MIQP$". Nous allons tout d'abord essayer de faire disparaître la notion de processus, tout en conservant les contraintes qui leur sont liés. Pour celà, considérons uniquement l'ensemble des signaux $V = \{v_j \mid j \in [1, p]\}$. Nous allons construire un recouvrement $(X_i)_{i \in [1,n]}$ de l'ensemble des signaux sous la contrainte $(P) \quad : \quad \forall i \in [1, n], v, v' \in V \ (v \in X_i) \wedge (\exists \ell \in [1, n_p] \ \{v, v'\} \subset V_\ell) \implies v' \in X_i$, qui formalise le fait que deux signaux étant en entrée d'un même processus seront nécéssairement dans le(s) même(s) groupe(s).

Soit Z le vecteur suivant :

$$
\left|
\begin{array}{l}
Z = (z_{1,1}, \ldots, z_{1,p}, z_{2,1}, \ldots, z_{2,p}, \ldots, z_{n,p}, y_{1,1}, \ldots, y_{1,m}, \ldots, y_{n,m}) \\
z_{i,j} = 1 \quad \text{si le signal } v_j \text{ est présent dans l'ensemble } X_i \\
y_{i,k} \quad \text{tq. } \forall j \in [1,p],\ (v_j)_k \cdot z_{i,j} \leq y_{i,k}
\end{array}
\right.
$$

Ce vecteur contient encore une fois une grande partie des informations nécéssaires pour la résolution du problème. Il code la présence des signaux dans les ensembles à travers les variables $z_{i,j}$ et l'activation de ces ensembles via les variables $y_{i,k}$ et les contraintes qui leur sont liées. Reste maintenant à coder la contrainte (P) avec ces variables :

$$\forall j, j' \in [1,p]\ (\exists \ell \in [1, n_p]\ \{v_j, v_{j'}\} \subseteq V_\ell \implies \forall i \in [1,n](z_{i,j} = z_{i,j'}))$$

Ces contraintes sont linéaires et pourront être transposées, pour chaque architecture considérée, sous forme matricielle. La fonction à optimiser prend alors la forme suivante :

$$q(Z) = \sum_{k=1}^{m}(\sum_{i=1}^{n}(y_{i,k} \cdot \sum_{j=1}^{p} z_{i,j})) = \sum_{i=1}^{n}(\sum_{k=1}^{m} y_{i,k}) \cdot (\sum_{j=1}^{p} z_{i,j})$$

On reconnaît ici la même fonction de coût que celle présentée dans la partie 3.2.2. La résolution de ce problème sera donc très fortement similaire à la résolution du problème 2, en ne modifiant que les matrices de contraintes. Une fois une solution identifiée, nous obtiendrons donc un recouvrement en n composantes de l'ensemble des signaux, et il restera à reconstruire à partir des ensembles X_i obtenus les groupes de processus.

Chapitre 4

Résolution du problème

La résolution de problèmes quadratiques est un thème assez discuté dans la littérature. De nombreuses méthodes de résolution existent, appliquées et distribuées dans de nombreux logiciels. Nous allons présenter ici les quelques méthodes que nous avons étudié dans le but de résoudre notre problème de manière efficace.

4.1 Résolution classique des problèmes QP

Les méthodes permettant de résoudre les problèmes quadratiques généraux diffèrent des méthodes utilisées pour résoudre les problèmes quadratiques entiers (méthodes souvent moins efficaces). Nous allons donc présenter dans un premier temps le problème quadratique continu avec une fonction de coût modifiée pour forcer les solutions à être entières, puis nous détaillerons les algorithmes visant à résoudre directement le problème entier avec et sans optimisations.

4.1.1 Le problème continu

Nous avons donc dans un premier temps supposé qu'il serait plus efficace de résoudre un problème continu, tout en s'assurant que les solutions trouvées seraient entières. Ceci peut être fait en ajoutant à la fonction de coût une fonction auxiliaire dont les minimums (nuls) sont obtenus en 0 et 1, et suffisament grande par rapport à notre fonction de coût initial pour nous assurer que les minimums de la fonction modifiée seront bien obtenus en 0 et 1.

Reconsidérons donc le problème 2. La fonction de coût de ce problème est positive et peut être aisément majorée par $n \cdot m \cdot p$. Considérons donc une constante Γ suffisament grande (par exemple $\Gamma = n \cdot m \cdot p$) et la fonction suivante :

$$q_\Gamma = q(Z) + \Gamma \cdot \sum_{i=1}^{n}((\sum_{j=1}^{p} z_{i,j} \cdot (1 - z_{i,j})) + (\sum_{k=1}^{m} y_{i,k} \cdot (1 - y_{i,k})))$$

Pour des variables booléennes, la fonction q_Γ aura clairement les mêmes solutions que la fonction q, et le choix de Γ assure qu'un minimum de la fonction q_Γ sera nécéssairement obtenu pour des valeurs booléennes des variables. Résoudre le problème modifié de la sorte devrait donc à priori donner les mêmes résultats tout en permettant d'appliquer les algorithmes de résolution de problèmes QP continus.

Le problème est qu'il n'existe d'algorithmes efficaces de résolution de problèmes QP que lorsque la fonction quadratique considérée est convexe. Hors la fonction q comme la fonction q_Γ ne le sont pas. Il existe néanmoins des méthodes permettant de convexifier une fonction sans en changer les optimums (c.f 4.1.2), mais on perd dans ce cas l'avantage de la fonction q_Γ qui tirait

partie de sa partie concave pour forcer les minimums à être obtenus pour des valeurs booléennes des variables. Nous avons donc décidé de nous arreter là dans l'exploration de cette piste et de nous contenter des algorithmes moins efficaces de résolution des $MIQP$.

4.1.2 Résolution des $MIQP$

Tout comme pour le problème continu, les méthodes les plus simples pour résoudre les problèmes $MIQP$ existent lorsque ces problèmes sont semi-définis positifs (du fait de la semi-définie positivité de la matrice Q). De tels problèmes sont appelés problèmes quadratiques convexes. Il est néanmoins clair que dans notre cas, la matrice Q n'est pas semi-définie positive. Il existe dans ce cas des méthodes permettant de convexifier la fonction et d'utiliser les méthodes classiques pour les problèmes convexes par la suite [2]. Nous allons nous placer ici dans le cadre plus général de la programmation quadratique à contraintes linéaires. Considérons donc le problème suivant :

Problème 4 ($MIQP$)

$$(MIQP) \quad : \quad \min\{g(x) = x^t \cdot Q \cdot x + c^t \cdot x \mid A \cdot x = b,\ A' \cdot x \leq b',\ x \in 2^n\}$$

avec c, b, b' des vecteurs réels de tailles réspectives n, m, p, Q une matrice symétrique réelle de taille $n \times n$, A et A' des matrices de tailles respectives $m \times n$ et $p \times n$.

Convexification du problème

La fonction g utilisée dans le problème ci-dessus n'étant pas convexe, les algorithmes pour résoudre le problème (QP) sont complexes. Nous allons donc chercher une reformulation du problème, donnant les même solutions, mais avec une fonction g' convexe. Considérons alors le problème suivant :

Problème 5 ($QP_{\alpha,u}$) *Soient α une matrice réelle de taille $m \times n$ et u un vecteur de $\mathbb{R}^n$.*

$$(QP_{\alpha,u}) \quad : \quad \min\{g_{\alpha,u}(x) = x^t \cdot Q_{\alpha,u} \cdot x + c_{\alpha,u}^t \cdot x \mid A \cdot x = b,\ A' \cdot x \leq b',\ x \in 2^n\}$$

$$o\grave{u} \begin{cases} Q_{\alpha,u} = Q + \frac{1}{2}(\alpha^t \cdot A + A^t \cdot \alpha) + \mathsf{Diag}(u) \\ c_{\alpha,u} = c - \alpha^t \cdot b - u \end{cases}$$

Il est clair que pour tous les x tels que $A \cdot x = b$, la fonction $g_{\alpha,u}(x)$ est égale à la fonction $g(x)$. Les solutions du problème $(QP_{\alpha,u})$ seront donc les mêmes que celles du problème (QP). Or il est possible de bien choisir les constantes α et u telles que la fonction $g_{\alpha,u}$ soit convexe (par exemple en prenant $\alpha = \mathbf{0}$ et $u = -\lambda \cdot \mathbf{1}$ avec λ la plus petite valeur propre de la matrice Q). Il devient donc possible de résoudre le problème en utilisant les algorithmes de résolution des problèmes convexes.

Résolution du $MIQP$ convexe

Nous avons décidé d'utiliser le logiciel CPLEX, qui permet de résoudre les problèmes $MIQP$ en utilisant un algorithme Branch-And-Bound.

Les algorithmes Branch-And-Bound permettent de résoudre les problèmes du type $\min\{g(X) \mid A \cdot X \leq B, A' \cdot X = B', X \in \mathbb{N}^n\}$ de façon itérative. A l'étape k, on suppose que l'espace

$P = \{X \mid A \cdot X \leq B, A' \cdot X = B'\}$ est découpé en un ensemble $\Pi_k = \{P_1, \ldots, P_k\}$ de polyèdres disjoints et recouvrant les vecteurs entiers de P. On détermine alors

$$\mu_j = \min\{g(X) \mid X \in P_j\}$$

Pour chaque polyèdre P_j, cette résolution s'effectue avec n'importe quelle méthode pour la résolution de problèmes continus. On choisit ensuite j^* tel que

$$\mu_{j^*} = \min\{\mu_j \mid j \in [1, k]\}$$

et $X^* = (\xi_1^*, \ldots, \xi_n^*)$ le point atteignant μ_{j^*}.

1. Si $X^* \in \mathbb{N}^n$, alors le minimum de g est atteint en X^*. L'algorithme termine.
2. Sinon, on choisit une composante de X^* non entière, par exemple ξ_i^* et on définit les polyèdres suivants :

$$\begin{cases} Q_1 = \{X = (\xi_1, \ldots, \xi_n) \in P_{j^*} \mid \xi_i \leq \lfloor \xi_i^* \rfloor\} \\ Q_2 = \{X = (\xi_1, \ldots, \xi_n) \in P_{j^*} \mid \xi_i \geq \lceil \xi_i^* \rceil\} \end{cases}$$

   On définit alors $\Pi_{k+1} = \{P_1, \ldots, P_{j^*-1}, Q_1, Q_2, P_{j^*+1}, \ldots, P_k\}$, qui recouvre toujours les vecteurs entiers de P et dont les composantes sont toujours disjointes. On passe à l'étape $k+1$.
3. Si tous les P_j sont vides, alors le problème est insoluble.

Il a été prouvé ([11]) qu'un tel algorithme termine dès lors que le polyèdre de départ P est borné (ce qui est le cas dans notre problème puisque nous travaillons sur des booléens).

Il nous suffit donc d'appliquer cet algorithme sur notre problème convexifié pour obtenir une solution. Néanmoins, lorsque la fonction à minimiser n'est pas suffisament régulière, il peut s'avérer utile de commencer l'algorithme à l'étape 2 par exemple, en précisant le premier point autour duquel les polyèdres vont être placés. En effet, si ce premier point (la racine de l'arbre de recherche) est choisi de manière efficace, le nombre d'itérations à faire peut s'en retrouver diminué.

Choix de la meilleure racine

Il est ainsi proposé, dans [2], une méthode permettant de choisir les constantes α et u pour que l'algorithme Branch-And-Bound soit plus efficace.

Le but de cette méthode est donc de trouver la meilleure racine possible pour l'arbre de recherche de l'algorithme Branch-And-Bound. Comme nous l'avons expliqué plus haut, cette racine est égale à la valeur optimale de la relaxation continue du problème $(QP_{\alpha,u})$. Ils proposent donc de déterminer les valeurs de $\alpha \in \mathbb{R}^{m,n}$ et $u \in \mathbb{R}^n$ telles que $g_{\alpha,u}$ soit convexe et telles que la relaxation continue de $(QP_{\alpha,u})$ soit maximale (et donc plus proche de la meilleure solution entière). Ils proposent donc de résoudre le problème suivant :

$$(C(QP)) \quad : \quad \max_{\substack{\alpha,u \\ Q_{\alpha,u} \succeq 0}} \min_{X \in P} g_{\alpha,u}(X)$$

Ils montrent alors que ce problème est équivalent à une relaxation semidéfinie du problème $MIQP$ initial, et que les meilleures valeurs de α et u peuvent être obtenues en solvant le problème $SDQP$ suivant :

Problème 6 ($SDQP$)

$$(SDQP)\begin{cases} & \min\ c^t \cdot x + \sum_{i=1}^{n}\sum_{j=1}^{n} q_{i,j} \cdot X_{i,j} & & \\ t.q. & X_{i,i} = x_i & i = 1,\dots,n & (1) \\ & -b_k \cdot x_i + \sum_{j=1}^{n} a_{k,j} \cdot X_{i,j} = 0 & i = 1,\dots,n; j = 1,\dots,m & (2) \\ & A \cdot x = b & & \\ & A' \cdot x \leq b' & & \\ & \begin{pmatrix} 1 & x^t \\ x & X \end{pmatrix} \succeq 0 & & \\ & x \in \mathbb{R}^n, X \in \mathcal{M}_n(\mathbb{R}) & & \end{cases}$$

Une fois ce problème résolu (ce qui peut être fait en utilisant le software SB pour la résolution de programmes semi-définis, c.f. [6]), les valeurs optimales u^* et α^* sont les valeurs optimales des variables duales associées aux contraintes (1) et (2).

Il ne reste donc plus qu'à résoudre le problème quadratique convexe QP_{α^*,u^*} en utilisant les méthodes classiques de Branch-And-Bound, telles que celle implémentée dans CPLEX. Nous n'avons malheureusement pas eu accès aux logiciels permettant d'effectuer cette optimisation et avons du nous contenter des valeurs non optimales de α et u dans la résolution du problème (c.f. 5.1).

4.2 Utilisation d'heuristiques et parcours de l'arbre

La deuxième piste que nous avons choisi d'étudier pour résoudre nos problèmes quadratiques booléens est un simple parcours de l'arbre des solutions en utilisant des heuristiques pour couper les branches dont on sait qu'elles ne permettront pas d'aboutir à une solution optimale. En effet, si l'on s'en tient uniquement à la définition formelle des problèmes, ils consistent juste à partitionner un ensemble de vecteurs binaires (pour la consommation des circuits) ou d'entiers (pour la synthèse d'architectures GALS).

Nous n'avons testé cette méthode que pour le problème de la consommation dans les circuits mais il sera facile de l'adapter pour résoudre le problème de la synthèse d'architectures GALS. De plus, nous allons présenter cette méthode en supposant que le nombre de groupes de vecteurs souhaités est 2 (pour plus de clarté), mais cela s'adapte encore une fois sans aucun mal à des partitions plus étendues.

Le principe est le suivant :

1. On ordonne les vecteurs pour les insérer un à un dans les partitions ;
2. Construire l'arbre de toutes les partitions possibles en plaçant les vecteurs un à un ;
3. Pour toutes les feuilles de l'arbre, calculer la consommation obtenue ;
4. Conserver la meilleure des solutions trouvée.

Cette solution est bien entendue naïve puisqu'elle implique de construire tout l'arbre des solutions alors qu'on pourraît trouver la meilleure solution dès la première feuille. Il pourrait donc être intéressant de construire cet arbre en profondeur en conservant en mémoire la meilleure solution trouvée pour le moment et en ne construisant pas les branches dont on peut déduire qu'elles ne donneront pas une meilleure solution

Parcours de l'arbre en profondeur et heuristiques

Pour ce faire, nous allons essayer de calculer, à chaque noeud de l'arbre des solutions (et donc avec des solutions incomplètes) un minorant de la consommation que l'on pourra trouver dans

le sous arbre partant de ce noeud. Si ce minorant est plus grand que la meilleure consommation trouvée précédemment, il n'y aura aucun intérêt à continuer de parcourir cette branche.

Considérons donc un ensemble de vecteurs binaires $X = \{x_1, \ldots, x_m\}$ que nous voulons partitionner en deux parties X_1 et X_2 telles que $c(X_1, X_2) = |X_1| \cdot |\overrightarrow{X_1}| + |X_2| \cdot |\overrightarrow{X_2}|$ soit minimale.

Un noeud interne de l'arbre des solutions est donc un triplet $N = (X_1, X_2, Y)$ où X_1 et X_2 sont l'état actuel des partitions et Y l'ensemble des vecteurs restant à placer dans ses partitions. Nous voulons donc trouver une fonction $d(N)$ qui permette d'assurer :

Propriété 1

$$\forall N = (X_1, X_2, Y)\ \forall X_1', X_2'\ (X = X_1' \uplus X_2') \wedge (X_1 \subseteq X_1') \wedge (X_2 \subseteq X_2') \implies d(N) \leq c(X_1, X_2)$$

Une fois une telle fonction établie, on peut construire l'arbre des solutions en profondeur en conservant la meilleure solution trouvée pour l'instant c, en calculant à chaque noeud ajouté la valeur de d. Si $d(N) \geq c$ alors on peut être sûr qu'on ne trouvera pas de meilleure solution et on ne génère donc pas la branche associée.

Heuristiques proposées

Nous avons donc cherché des fonctions vérifiant la propriété 1 pour pouvoir couper efficacement l'arbre des solutions :

1. $d_1(N) = c(X_1, X_2) + |Y| \cdot \min(|\overrightarrow{X_1}|, |\overrightarrow{X_2}|)$
2. $d_2(N) = c(X_1, X_2) + |Y| \cdot \min\{|\overrightarrow{y}| \mid y \in Y\}$
3. $d_3(N) = c(X_1, X_2) + \sum_{y \in Y} |y|$

La première heuristique d_1 se base sur le fait que la meilleure solution possible à partir d'une solution partielle est de rajouter tous les vecteurs restants dans la partie la moins active sans en augmenter l'activité. C'est un minorant plutôt grossier, vu que généralement les vecteurs restants n'on pas nécessairement un vecteur d'activation plus petit que celui des parties déjà générées. La deuxième heuristique d_2 se base sur le fait que tous les vecteurs restants ont une activité supérieure à celle de leur minorant, et qu'ils seront au mieux tous placé dans une partition les contenant et ayant une activité égale à celle du minorant. La troisième et dernière heuristique proposée ici d_3 est une version moins grossière de d_2 et utilise le fait que les vecteurs seront au mieux tous placés dans une partition les contenant exactement. Chacune de ces heuristiques a des avantages et peut être plus efficace dans certains cas. d_1 n'est pas comparable à d_2 et d_3 au niveau de la qualité de minoration, mais elle est moins coûteuse à calculer puisqu'elle ne nécéssite pas de parcourir tous les vecteurs restants de Y. En revanche, il est clair que d_2 est plus grossière que d_3 mais elle peut être plus facile à calculer si la structure des données est bien choisie.

On pourrait aussi utiliser toutes ces heuristiques en même temps en prenant $d = \max(d_1, d_2, d_3)$ ou quelconque combinaison de ces heuristiques. L'avantage de cette méthode comparée à la résolution du $MIQP$ présentée précédemment est qu'elle évite la phase de récriture où un nombre important de variables sont ajoutées au problème pour se débarasser du Sup et rentrer dans le moule des $MIQP$.

Chapitre 5

Tests et Résultats

Pour des raisons de manque de temps et de ressources, nous n'avons pu tester l'efficacité de nos solutions sur les circuits ou les architectures GALS de taille réelle. Nous nous sommes uniquement concentré sur la résolution des problèmes $MIQP$ par les méthodes présentées dans la partie précédente, et sur leur efficacité en temps de calcul. Nous allons présenter ici les deux principales méthodes que nous avons implémenté, la première utilisant Cplex pour résoudre le problème 2, et la seconde implémentant le parcours en profondeur de l'arbre des solutions avec heuristiques. Nous présenterons finalement un exemple sur un circuit basique.

5.1 Résolution du problème $MIQP$ avec Cplex

Comme nous l'avons expliqué en 4.1.2, les outils principaux pour résoudre les problèmes $MIQP$ nécéssitent que la fonction de coût étudiée soit convexe. Nous avons donc convexifié le problème 2 en utilisant la méthode présentée en 4.1.2. Etant donné que Cplex est une application payante et que nous n'y avons eu accès que très tard durant ce stage, les premiers essais que nous avons effectué avec des algorithmes Branch-And-Bound ont eu lieu sous Matlab, peu efficace, mais permettant de convexifier le problème sans utiliser d'outils auxiliaires. Nous avons donc choisi de continuer à utiliser Matlab par la suite pour la génération des données et la convexification de problèmes.

Génération des données

La génération des données, sous Matlab, se décompose en 4 étapes :

1. Choix de la taille du problème (nombre de vecteurs p, nombre de partitions n, taille des vecteurs m) ;
2. Génération des vecteurs booléens aléatoirement ;
3. Construction de la matrice Q et calcul de sa plus petite valeur propre λ
4. Convexification du problème (construction de la matrice $Q_{\alpha,u}$, de la partie linéaire $c_{\alpha,u}$ et des matrices de contraintes).

Utilisation de Cplex et résultats

Il existe deux utilisations possibles du logiciel Cplex, au travers d'une bibliothèque C ou alors au travers d'une interface interactive. Nous n'avons malheureusement pas eu accès à la première solution durant ce stage, et avons donc du nous contenter de l'interface intéractive bien qu'elle soit certainement moins efficace et plus contraignante. Comme nous le voyons dans

Nombre de vecteurs	Noeuds visités	Temps de calcul(s)
10	355	0.34
15	3469	3.76
20	47569	58.12
25	2478782	4522.53

FIG. 5.1 – Performances avec Cplex ($n = 2, m = 15$)

le tableau 5.1, l'utilisation que nous faisons de cplex ne nous permet pas de traîter de gros problèmes, dès que nous dépassons les 25 vecteurs, le temps de calcul devient trop important. Nous n'avons malheureusement pas eu accès aux outils permettant de choisir une racine optimale pour l'algorithme de Branch-And-Bound comme présenté dans 6, et n'avons donc pas pu tester l'efficacité de Cplex avec une meilleure racine.

5.2 Parcours heuristique de l'arbre des solutions

Cette deuxième méthode est plus simple d'accès que la première, puisqu'elle n'a pas demandé l'utilisation de logiciels ou programmes soumis à des licenses particulières. En effet, l'implémentation du parcours en profondeur de l'arbre des solutions a été écrite en C, et peut tourner sur n'importe quelle machine. La génération aléatoire de vecteurs, dans un premier temps effectuée en C elle aussi, a été remplacée par la suite. En effet, dans un soucis de comparaison entre cette méthode et celle implémentée dans Cplex, nous avons voulu effectuer des tests sur les mêmes jeux de données. Nous avons donc utilisé, dans une deuxième phase de tests, les mêmes données générées avec Matlab, pour les deux méthodes.

Principe de cette méthode

Comme cela a été expliqué dans la partie précédente, cette méthode génère en profondeur l'arbre des solutions du problème en coupant grâce à des heuristiques les branches ne présentant pas d'intérêt. Chaque noeud de l'arbre est un triplet (X_1, X_2, Y) où X_1,X_2 et Y sont des listes de vecteurs. L'algorithme de résolution est donc le suivant :

Algorithme 1 (solver(X_1, X_2, Y))

si Y *non vide*
  si heur$(X_1, X_2, Y) <$ *coûtmin*
    $x := \mathsf{first}(Y)$
    solver$(x :: X_1, X_2, \mathsf{tail}(Y))$
    solver$(X_1, x :: X_2, \mathsf{tail}(Y))$
  sinon ne rien faire
sinon si coût$(X_1, X_2) <$ *coûtmin*
  coûtmin = *coût*(X_1, X_2, Y)
  solution = (X_1, X_2)
  retour

Les résultats avec cet algorithme (montrés en Fig. 5.2 et Fig. 5.3) sont étonnament bien meilleurs que ceux obtenus avec Cplex. Cela pourrait s'expliquer par le fait que la transformation en problème $MIQP$ classique engendre une explosion du nombre de variables (on passe de $n \cdot p$ à $n \cdot m + p$ variables), mais ce n'est pas suffisant au vu de la différence de performances. La cause principale, à notre avis, est que les algorithmes de Branch-And-Bound implémentés dans Cplex

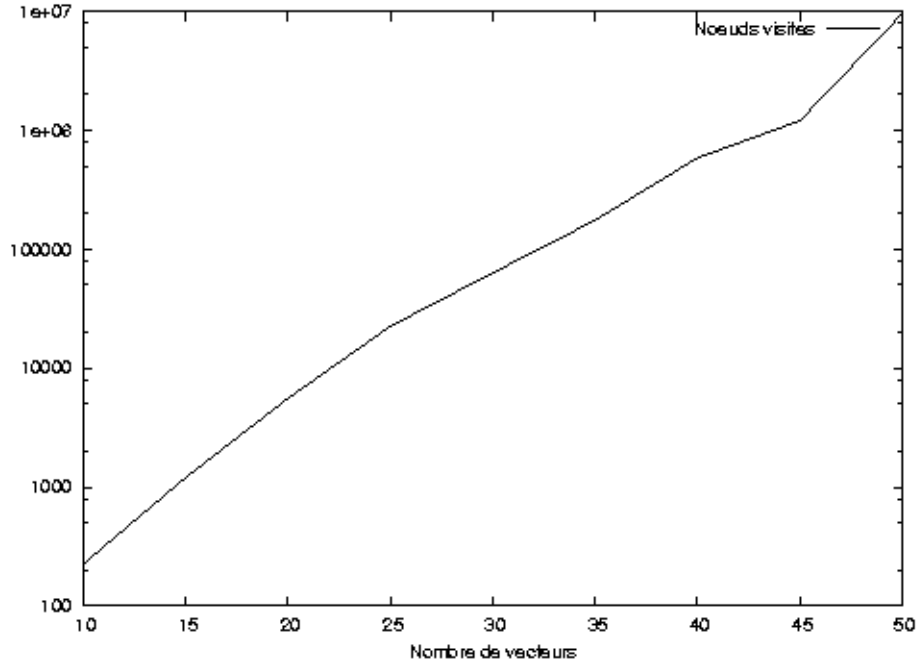


FIG. 5.2 – Quantité de noeuds visités en fonction du nombre de vecteurs (log) pour $n = 2, m = 15$

calculent à chaque noeud les minimums locaux de la fonction sur des polygones, alors que notre algorithme de recherche par heuristiques se contente de calculer la valeur de la fonction de coût et/ou de l'heuristique à chaque solution partielle. Non seulement le problème traité par Cplex possède un nombre plus grand de variables, et donc un arbre de recherche plus profond, mais il doit en plus effectuer à chaque noeud des opérations plus coûteuses.

Le programme C que nous avons écrit pour résoudre ce problème avec la méthode de recherche par heuristiques n'est néanmoins pas parfait, il mériterait plus de réflexion sur les structures de données utilisées (pour le moment des listes de vecteurs, à remplacer certainement par des tableaux ou des matrices) et surtout plus d'adaptation et de flexibilité aux différents problèmes. En effet, la taille maximale des problèmes qu'il nous permet de traiter actuellement sur un Pentium 4 muni d'1 Go de RAM est limitée à une cinquantaine de vecteurs de taille 15. De plus, il n'a été implémenté que pour une partition en 2 parties ($n = 2$), mais l'adaptation à des partitions plus grande ne devrait pas entrainer de difficultés majeures.

5.3 Exemple d'un circuit

Nous allons maintenant appliquer notre algorithme à un circuit (c.f. Fig. 5.4) qui reçoit en entrée des suites de valeurs X_k et Y_k de manière asynchrone ainsi que des signaux P_x et P_y permettant de savoir lorsque les valeurs reçues sont prêtes, et qui rend en sortie la concaténation des signaux reçus en entrée (X_k, Y_k). Le circuit présenté n'est pas parfait et suppose que les temps entre réception des signaux X_k et Y_k restent faibles et que deux Y successifs ne peuvent pas arriver sans qu'un X ait été reçu (et réciproquement). Si c'est le cas, le premier Y reçu sera ignoré.

Ce circuit est constitué de 7 mémoires, dont nous allons calculer les vecteurs d'activation. Plaçons nous tout d'abord dans le cas d'une exécution périodique, et supposons que la suite des signaux reçus est présentée dans la figure 5.5. On obtient dans ce cas les vecteurs d'activation

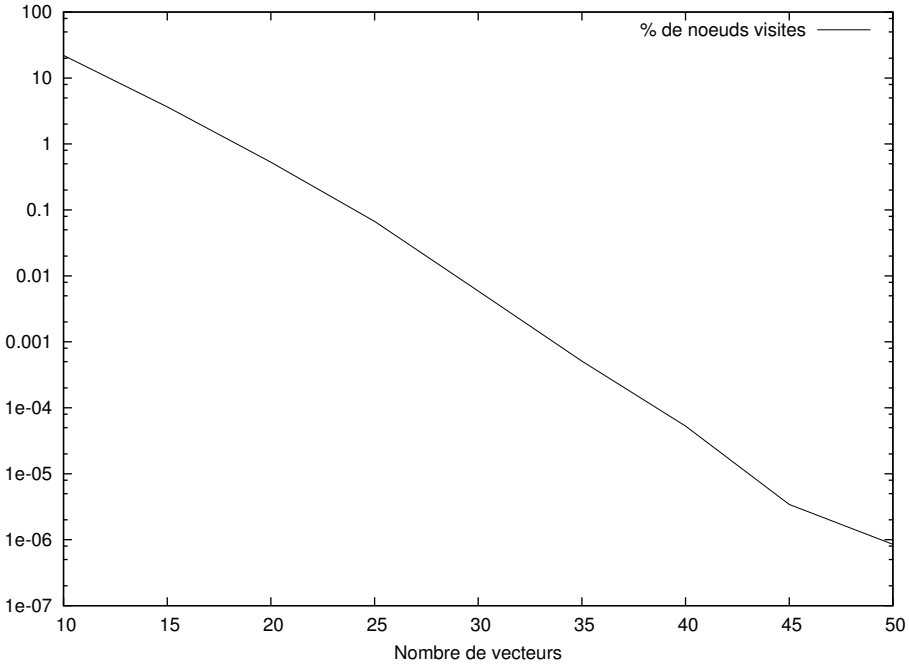


FIG. 5.3 – Pourcentage de noeuds visités en fonction du nombre de vecteurs (log) pour $n = 2, m = 15$

des mémoires présentés dans la figure 5.6.

Nous avons alors calculé avec notre programme de recherche de la meilleure solution avec heuristiques la partition de l'ensemble des mémoires à 2 éléments permettant d'obtenir une consommation minimale (2 éléments uniquement pour des raisons de simplicité), et nous avons obtenu la partition suivante : $X_1 = \{L_4, L_5, L_6, L_7\}$ et $X_2 = \{L_1, L_2, L_3\}$.

Ce résultat est tout à fait logique puisqu'il correspond à mettre toutes les mémoires liées à X dans une partie et toutes les mémoires liées à Y dans l'autre.

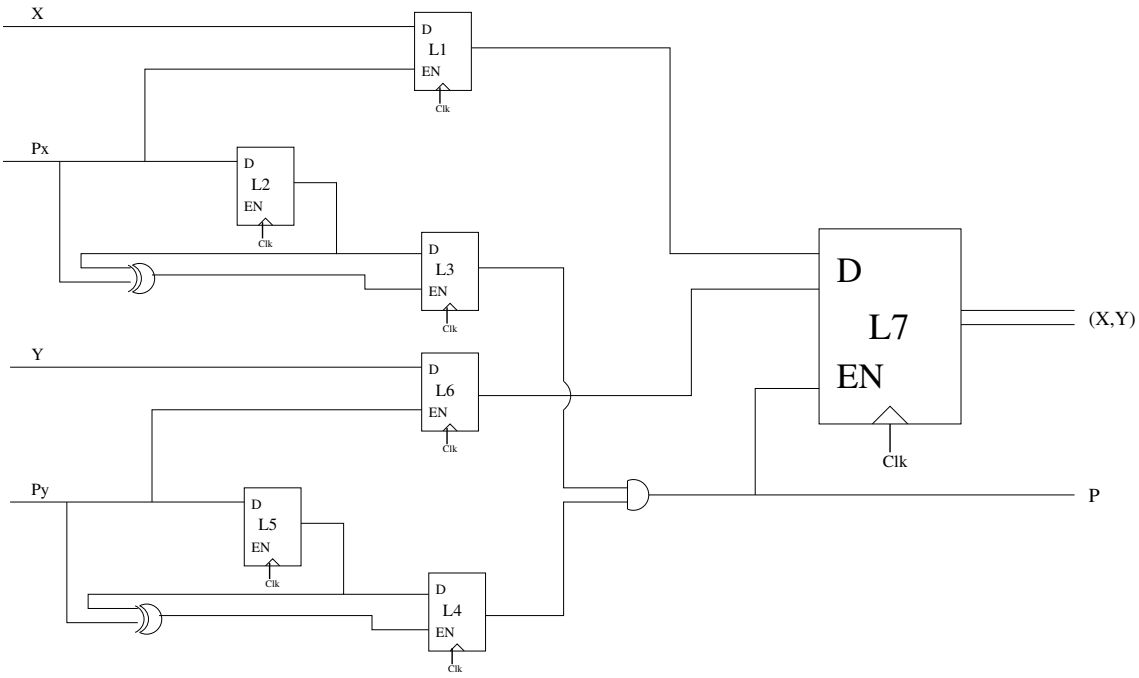


FIG. 5.4 – Circuit étudié

X	:	1	1	0	1	1	0	0	1	0	0	1
P_x	:	1	0	0	1	1	0	0	0	1	1	0
Y	:	0	0	0	1	1	1	1	0	0	0	0
P_y	:	1	0	0	0	1	1	0	0	0	1	1

FIG. 5.5 – Signaux périodiques reçus

L_1	:	1	1	0	1	0	1	0	0	1	0	1
L_2	:	1	0	0	0	0	0	0	0	1	0	0
L_3	:	1	1	0	1	1	0	0	0	1	0	1
L_4	:	0	1	0	0	1	0	1	0	0	1	0
L_5	:	0	0	0	0	1	0	0	0	0	1	0
L_6	:	0	1	0	0	1	0	1	0	0	1	0
L_7	:	0	0	1	0	0	0	1	0	0	0	0

FIG. 5.6 – Vecteurs d'activation des mémoires du circuit

Chapitre 6

Conclusion

Ce stage avait pour but d'étudier et de mettre en oeuvre des méthodes de réduction de consommation dans les circuits. Bien que nous n'ayons pas poussé jusqu'au bout les tests par manque de temps et de sources d'information principalement, nous avons proposé et étudié une méthode permettant de regrouper les mémoires internes d'un circuit pour pouvoir utiliser la technique du "Clock-Gating" afin d'éteindre leurs horloges durant leurs périodes d'inactivité. Cette étude nous a amené à transformer le problème de regroupement des mémoires en un problème d'optimisation quadratique, plus connu dans la littérature dans d'autres cadres, et nous a entrainé à rapprocher ce problème au problème de la synthèse d'architectures GALS optimales en temps de calcul. Nous avons donc étudié les méthodes de résolution existantes pour ce genre de problèmes, et proposé un algorithme de parcours d'arbre avec heuristiques. Les tests que nous avons effectué ont montré que notre algorithme était bien plus efficace et plus adapté à notre problème que les méthodes de résolution existantes.

De nombreux points restent néanmoins à étudier et améliorer, en commençant par l'utilisation faite du logiciel Cplex. Ce logiciel est très complet et peut être utilisé pour la résolution de nombreux problèmes de programmation linéaire et quadratique, et n'ayant obtenu une licence pour l'utiliser que très tard dans ce stage, nous n'avons certainement pas tiré parti de la totalité de ses capacités. Le programme que nous avons écrit pour le parcours de l'arbre avec heuristiques n'est, quant à lui, pas optimal pour la gestion des données et mériterait à être récrit. On remarque néanmoins que la taille des problèmes que l'on peut traiter reste assez faible, et qu'il faudra passer par d'autres optimisations pour pouvoir l'augmenter de façon non négligeable. Une idée pour celà serait de permettre à l'utilisateur de préciser avant la recherche des meilleures solutions, des groupes de signaux qui, selon lui, mériteraient à être groupés.

Bibliographie

[1] L. Benini, P. Vuillod, C.J.N. Coelho Jr., and G. De Micheli. Synthesis of low-power selectively-clocked systems from high-level specification. In *ISSS*, page 57, 1996.

[2] A. Billionnet, S. Elloumi, and M.C. Plateau. Convex quadratic programming for exact solution of 0-1 quadratic programs. 2005.

[3] C. Cao and B. Oelmann. Mixed synchronous/asynchronous state memory for low power FSM design. In *DSD*, pages 363–370, 2004.

[4] A.J. Dupont, K.G. Shin, and P. Ramanathan. Clock distribution in general VLSI circuits. *IEEETCS : IEEE Transactions on Circuits and Systems*, 41, 1994.

[5] Nicholas I. M. Gould. Numerical methods for large-scale non-convex quadratic programming. Technical report, 2001.

[6] C. HELMBERG and F. RENDL. A spectral bundle method for semidefinite programming. *siopt*, 10(3) :673–696, 2000.

[7] D. C. Ku and G. De Micheli. *High Level Synthesis of ASICs Under Timing and Synchronization Constraints*. Kluwer Academic, 1992.

[8] F. Oustry and C. Lemarechal. Semidefinite relaxations and lagrangian duality with applications to combinatorial optimization. Technical Report RR–3170, INRIA, Rhone-Alpes, June 1999.

[9] S. Poljak, F. Rendl, and H. Wolkowicz. A recipe for semidefinite relaxation for (0,1)-quadratic programming. *Journal of Global Optimization*, 7(1) :51–73, 1995.

[10] Svatopluk Poljak and Henry Wolkowicz. Convex relaxations of 0-1 quadratic programming. Technical Report 93-18, Center for Discrete Mathematics and Theoretical Computer Science (DIMACS) at Rutgers University, New Jersey, 1993.

[11] Schrijver, A. *Theory of Linear and Integer Programming*. John Wiley & Sons, 1986.

[12] A. Vicard and Y. Sorel. Formalization and static optimization of parallel implementations. In *Proc. Austrian-Hungarian Workshop on Distributed and Parallel Systems (DAPSYS'98)*, SZTAKI, H-1132 Budapest, Victor Hugo str., September 1998.

Annexe A

Définition des matrices du problème 2

$$Q = \begin{pmatrix} D_1 & R \\ R^t & D_2 \end{pmatrix} \text{ avec } \begin{cases} D_1 \in \mathcal{M}_{n\cdot p}(\mathbb{R}) = \mathbf{0} \\ D_2 \in \mathcal{M}_{n\cdot m}(\mathbb{R}) = \mathbf{0} \\ R \in 2^{n\cdot p, n\cdot m} \text{ tq } r_{i,j} = 1 \iff \lfloor i/p \rfloor = \lfloor j/m \rfloor \end{cases}$$

$$c = \mathbf{0}$$

$$A \in 2^{p, n\cdot(m+p)} \text{ tq } \begin{cases} a_{i,j} = 0 & \forall j > n \cdot p \\ a_{i,j} = 1 & \forall j \leq n \cdot p \text{ tq } j \equiv i[p] \\ a_{i,j} = 0 & \forall j \leq n \cdot p \text{ tq } j \not\equiv i[p] \end{cases}$$

$$b \in 2^p \text{ tq } \forall i,\ b_i = 1$$

$$b' \in 2^{m\cdot n\cdot p} = \mathbf{0}$$

$$A' = \begin{pmatrix} A'_1 & A'_2 \end{pmatrix},\ A'_1 \in 2^{n\cdot m\cdot p, n\cdot p},\ A'_2 \in 2^{n\cdot m\cdot p, n\cdot m} \text{ avec}$$

$$A'_1 = (M_{i,j})_{i,j\in[1,n]}, M_{i,j} \in 2^{m\cdot p, p} \text{ et telles que } M_{i,j} = \begin{cases} \mathbf{0} & \text{si } i \neq j \\ M_0 & \text{si } i = j \end{cases}$$

$$A'_2 = (D_{i,j})_{i,j\in[1,n]}, D_{i,j} \in \mathbb{R}^{m\cdot p, m} \text{ et telles que } M_{i,j} = \begin{cases} \mathbf{0} & \text{si } i \neq j \\ D_0 & \text{si } i = j \end{cases}$$

$$M_0 = (c_{i,j})_{i,j\in[1,p]},\ c_{i,j} \in 2^m \text{ tq } c_{i,j} = \begin{cases} \mathbf{0} & \text{si } i \neq j \\ \overrightarrow{x}_i & \text{si } i = j \end{cases}$$

$$D_0 \in \mathbb{R}^{m\cdot p, m}\ D_0 = \begin{pmatrix} \mathsf{Diag}_m(-1) \\ \cdots \\ \mathsf{Diag}_m(-1) \end{pmatrix}$$