UNIVERSITÉ LIBRE DE BRUXELLES
Faculté des Sciences
Département d'Informatique

Prédiction de performances multicritère pour la conception des systèmes embarqués

Antonio Paolillo

Promoteur :
Prof. Frédéric Robert

Mémoire présenté en vue
de l'obtention du grade de
Master en Sciences Informatiques

Année académique 2010 – 2011

Je dédie ce travail de fin d'études à la mémoire
de mon grand-père, Antonio Paolillo.
Puisse-t-il reposer en paix.

"*I tried to picture clusters of information as they moved through the computer. What do they look like? Ships, motorcycles? With the circuits like freeways. I kept dreaming of a world I thought I'd never see.*"

Kevin Flynn, 2011

"*I discovered that hardware and software are two sides of the same coin, although they are treated completely separately in education. It's really an implementation detail whether you choose to make your logical function in software or in hardware, or some mixture of the two.*"

Andy Green, 2003

Table des matières

Remerciements

La fin de mon parcours en tant qu'étudiant est arrivée et je regrette déjà ces années passées à apprendre énormément de choses mais aussi à rencontrer et profiter de la compagnie de nombreuses personnes intéressantes.

La liste est longue quand arrive le moment de remercier ceux qui m'ont accompagné et aidé tout au long de mon parcours universitaire ainsi que pour l'écriture de mon mémoire de fin d'études.

Je vais commencer par remercier les différentes organisations que j'ai fréquentées : le Département d'Informatique, le service BEAMS et le Cercle Informatique.

Je tiens à remercier particulièrement le professeur qui a accepté d'être le directeur de ce mémoire : Frédéric Robert. Son implication et ses encouragements ont été d'une aide précieuse pour ma motivation. Je remercie également le co-directeur Gianluca Bontempi, qui m'a donné tous les outils en main pour adopter une approche formelle en termes de statistiques dans ce mémoire.

Je remercie les autres membres du jury, Raymond Devillers, Dragomir Milojevic et Gilles Geeraerts, ayant été choisis pour l'adéquation de leurs travaux au sujet mais aussi pour l'intérêt suscité chez moi par les matières qu'ils enseignent.

Je remercie également les chercheurs et étudiants de BEAMS (actuellement ou anciennement) pour leurs soutiens, aides, relectures, commentaires, encouragements à poursuivre mon travail et toutes les autres interactions : Anh Vu Doan, Axel Dero, Aliénor Richard, Nastassia Gumuchdjian et un remerciement spécial pour Anthony Leroy m'ayant suivi presque quotidiennement au cours des derniers mois de travail.

Deux professeurs d'autres services ont également été consultés, Michel Kinnaert et Christine Decaestecker, que je remercie pour m'avoir consacré du temps et donné des explications importantes.

Pour mener à bien ce cursus universitaire, l'aspect détente a été aussi important que l'aspect académique, et je tiens également à remercier ceux qui m'ont permis de m'amuser et de me consacrer à d'autres activités qui me sont chères tout au long de ces cinq années.

Je remercie ceux qui étaient avec moi sur les bancs des cours tout au long de ces années, notamment à travailler des nuits entières sur des projets, ainsi qu'à d'autres moments plus récréatifs, en particulier : Vladimir Svoboda, Olivier Sputael, Mathieu Duchêne, Aurélien Gillet, Mathieu Stennier, Filipe Miguel Gonçalves Almeida, Rukiye Akgün et Magali Thésias.

Je remercie les étudiants avec qui j'ai contribué cette année à véhiculer les valeurs du folklore estudiantin ulbiste : Colas Goeminne, Olivier Vernin, Mikaël Lenaertz, Melina Garcia, Dorian Burihabwa et Xavier Barthel.

Je remercie les personnes m'ayant parrainé au Cercle Informatique pour les nombreuses choses qu'ils m'ont apprises au niveau para-académique. Je remercie mes filleuls, les bleus 2010 ainsi que tous les autres membres du Cercle Informatique pour les bons moments passés ensemble.

Je remercie mes amis de plus longue date pour la connivence, les rires et les discussions que nous avons pu avoir, notamment : Jean-François De Backer, Bruno Gérard, Kevin-Art Deven, Nicolas Opalvens, Michael Moldawski, Alicia Ferdin, Kim Laitat, Sylvie Delloye, Neslie Soysal.

Je remercie aussi les amis que j'ai rencontré plus récemment mais avec qui j'ai passé tout autant de bons moments : Vincent Bremhorst, Wembo Vereeken, François Santy, Gaëtan Podevijn, Simon Debaets, André Madeira Cortes et Roxanne Hubesch.

Enfin, je remercie particulièrement Jonathan Moldawski et Ludivine Herchy pour leurs encouragements quotidiens et sans relâche.

Sans oublier les plus importants, les membres de ma famille, m'ayant permis d'arriver jusqu'au bout, m'ayant encouragé et supporté à tous les niveaux depuis toujours et ayant même, pour certains, eu le courage de relire ce mémoire sans formation préalable en informatique ou en électronique : Domenico Paolillo, Marie-Anne Paolillo, Manuel Paolillo, Lucia Paolillo, Luca Paolillo, Marianna Paolillo et Angélique Vanhoutte. Merci à vous tous.

Je regrette d'avance les éventuels oublis dans ces remerciements, mais je pense qu'il est impossible de lister toutes les personnes ayant de l'importance pour moi. J'espère également que ce beau monde continuera à m'entourer durant les années futures.

Préface

Lors de cet après-midi ensoleillé à Bruxelles, *monsieur X* se promène sur le campus de la Plaine. Soudainement, une onde UMTS vient frapper sa poche et le *smartphone* de ce dernier se met à vibrer et à sonner. *Monsieur X* le sort alors de sa poche, appuie sur l'écran tactile pour répondre et échange quelques propos avec son amie avant de raccrocher. Il surprend ensuite un renard et décide de le prendre en photo avec le même appareil et de partager sa découverte sur un site de réseau social. Enfin, *monsieur X* quitte la Plaine et se dirige vers l'Avenue de l'Université pour se rendre à l'autre campus de l'ULB, le Solbosch. En chemin, il s'arrête à la banque pour y retirer de l'argent à l'aide de sa carte bancaire.

Monsieur X ignore probablement le nombre incalculable d'opérations électroniques ayant eu lieu pendant ce court espace de temps.

Lors de la réception de l'appel, le circuit imprimé du téléphone est en pleine activité : les pistes métalliques voient leur potentiel énergétique varier à une vitesse phénoménale et les puces, dans leurs boîtiers, font déplacer les électrons sur le silicium grâce aux transistors qui y sont implantés. Les signaux électriques, portant l'information représentée sous la forme de nombres, sont manipulés et retransmis au travers de tout le circuit.

Lors de l'appel, l'onde entrante est numérisée, décompressée et réassemblée pour former un signal sonore numérique. Ce signal est reconverti en son à l'entrée de l'écouteur du téléphone. La voix de *monsieur X* est elle-même numérisée, traitée, compressée et codifiée sous la forme d'une onde prête à être renvoyée sur l'antenne du téléphone.

A la prise de la photo, les capteurs envoient au circuit de l'information brute sur la couleur, point par point, de l'image capturée. Cette information est elle-même compressée avant d'être sauvegardée sur la mémoire FLASH du téléphone. Une puce WiFi dédiée permettra l'envoi de l'image sur Internet.

De nombreuses autres opérations, comme la gestion de l'interface tactile, sont également exécutées en parallèle ; il serait impossible d'en réaliser une liste exhaustive au sein de cette préface.

Toutes ces opérations sont pilotées par un processeur central, le CPU, qui a chargé un système d'exploitation en mémoire centrale. Les différentes opérations sont réalisées par des puces plus ou moins spécialisées pour ces traitements.

A chaque exécution de ces opérations, les puces concernées drainent, au travers du signal d'alimentation, de l'énergie depuis la batterie du téléphone. La quantité d'énergie contenue dans la batterie est limitée et cette consommation se doit donc

d'être la plus faible possible afin de garantir une certaine autonomie à l'appareil. De plus, les opérations doivent être réalisées en respectant un certain timing restreint. Le circuit, au total, ne peut dissiper une puissance de plus de 3 watts, sinon le téléphone serait trop chaud pour être pris en main. Enfin, le prix total du téléphone ne doit pas excéder une somme acceptable pour le consommateur moyen. Le respect de toutes les contraintes énoncées précédemment ne doit donc pas être réalisé au détriment de la contrainte du coût.

Lors du retrait bancaire, des opérations cryptographiques très coûteuses sont exécutées sur un tout petit processeur enfoui dans la puce de la carte. Il existe également de nombreuses contraintes environnementales à prendre en compte pour la conception d'une telle puce : surface limitée, sécurité, fiabilité, etc.

Les cahiers de charges pour réaliser de tels systèmes électroniques sont donc très lourds en contraintes de performances.

Au sein du mémoire seront présentées les méthodes industrielles de conception de ces systèmes ainsi que deux travaux de recherche en cours visant à améliorer ces méthodes. Tout au long du texte, la nécessité de disposer de modèles de prédiction des performances sera mise en évidence. Etant donné ce contexte de conception complexe, le besoin de disposer d'outils pour réaliser de telles prédictions très tôt dans la conception du système sera illustré au niveau industriel et les deux travaux de recherche cités ci-dessus effectués dans le but de développer de tels outils seront introduits. Suite à cette observation, une idée de méthodologie de modélisation de ces systèmes et de leurs coûts en termes de performances, exploitant des techniques de *machine learning*, sera proposée.

Chapitre 1

Introduction

1.1 Contexte

Il existe traditionnellement deux approches pour aborder un problème donné : le point de vue *top-down* ou « utilisateur » et le point de vue *bottom-up* ou « concepteur, spécification et ressources ». Dans certains cas, ces approches se valent et sont même complémentaires. C'est pourquoi nous avons décidé de présenter indifféremment les deux approches pour introduire la problématique étudiée au sein de ce mémoire. D'abord, l'approche *top-down* mettra en évidence le grand nombre de systèmes embarqués disponibles aujourd'hui pour les utilisateurs. Ensuite, l'approche *bottom-up* fera le point sur le contexte économique et technologique des circuits intégrés qui sont l'une des composantes principales des systèmes embarqués.

Les systèmes embarqués

De nos jours, les sciences informatiques apparaissent comme une discipline ubiquitaire. En effet, on ne compte plus le nombre de foyers et d'entreprises qui utilisent quotidiennement un ordinateur de bureau ou un ordinateur portable. Outre cette observation, force est de constater que l'informatique ne concerne pas uniquement cette catégorie d'ordinateurs, mais également des dispositifs tels que des routeurs pour relayer de l'information au travers d'un réseau, des bornes de distribution automatique, des téléphones portables, des balladeurs MP3, des systèmes de pilotage automatique des avions ou de contrôle des réacteurs nucléaires, des téléviseurs et bien d'autres encore.

De tels systèmes contiennent en réalité de l'*informatique enfouie*, c'est-à-dire qu'ils sont constitués de circuits électroniques, implémentent des protocoles et exécutent des algorithmes de la même façon que les ordinateurs de bureau. Cependant, ces systèmes sont souvent soumis à des contraintes environnementales fortes : consommation d'énergie, dissipation de chaleur, coût de production, poids, surface et volume sont des exemples de contraintes auxquelles les dispositifs précités sont soumis. Ces dispositifs informatiques sont appelés les *systèmes embarqués*.

Pour prendre un exemple plus concret, le processeur d'un *smartphone* doit répondre à de nombreuses exigences conflictuelles en termes de performances : puissance

de calcul élevée (environ 2000 DMIPS [1]), faible consommation énergétique (environ 300 mW), temps de conception réduit (environ 6 mois), fiabilité, flexibilité, coût, etc.

Concevoir un système embarqué est autant une affaire de technologies disponibles que de choix parmi ces technologies. En effet, chacun de ces choix influe sur la satisfaction des différentes contraintes énoncées ci-dessus. Par exemple, le choix d'intégrer un puissant processeur au sein d'un téléphone portable permettra d'exécuter plus d'applications simultanément et plus rapidement, mais ce choix aura pour conséquence une plus forte consommation d'énergie et influera davantage sur le coût qu'un processeur plus modeste.

Pour répondre à ces spécifications de performances, les concepteurs ont recours à des plates-formes multiprocesseurs MPSoC (MultiProcessor System-on-Chip) intégrant sur une même puce à la fois des processeurs génériques, des processeurs spécialisés en traitement de signal (DSP), des processeurs reconfigurables ainsi que plusieurs niveaux de mémoire cache, le tout interconnecté par un réseau de communication évolué (NoC (Network-on-Chip)). Le nombre de degrés de liberté de conception offerts conjointement par l'architecture et par le logiciel est gigantesque. Pour gérer une telle complexité, les concepteurs divisent classiquement le problème en un ensemble de décisions prises à des niveaux d'abstraction décroissants. La description du système est ainsi raffinée à chaque niveau depuis les spécifications jusqu'à la puce électronique.

A chaque niveau d'abstraction, le concepteur tente d'évaluer l'impact de ses choix sur les performances du système final. Si le système ne satisfait pas aux spécifications, il est nécessaire de modifier les décisions prises au niveau précédent. Il est même parfois nécessaire de reconsidérer les tout premiers choix de conception. Or tout retour en arrière dans ce processus entraine des surcoûts importants.

Si les choix aux niveaux d'abstraction les plus bas sont le plus souvent réalisés à l'aide d'outils de conception automatiques, les premières étapes de conception sont par contre principalement basées sur l'expertise du concepteur, éventuellement aidé de quelques modèles empiriques très simples. Ces décisions sont pourtant les plus critiques car elles conditionnent les choix de toutes les étapes ultérieures et les performances finales du système.

Il est donc important de disposer à haut niveau de modèles et de méthodes d'évaluation de performances fiables qui permettent au concepteur d'évaluer l'impact de ses décisions sur les performances du système final, et ainsi de cibler les choix les plus intéressants.

Les circuits intégrés

Les *circuits intégrés* sont les composants matériels essentiels des systèmes informatiques. Les microprocesseurs centraux, les processeurs graphiques, les circuits *ASIC* [2] dédiés, les circuits *FPGA* [3] et les mémoires appartiennent, de nos jours, tous à la catégorie des circuits intégrés. Ces circuits sont des puces (en anglais *« microchips »*) qui réalisent des fonctions électroniques complexes. Le matériau principal utilisé pour

1. Le DMIPS est une unité de mesure de la vitesse d'exécution d'un processeur basée sur le programme Dhrystone [13].
2. Application Specific Integrated Circuit.
3. Field-Programmable Gate Array.

la fabrication de ces circuits est le silicium. La surface de silicium occupée par un circuit est un facteur influant fortement sur le coût de production unitaire de celui-ci. De plus, la surface disponible pour un circuit donné est limitée et dépend de la technologie de fabrication. Par conséquent, cette surface est un critère à minimiser lors de leur conception.

Etant donné que les circuits intégrés réalisent des fonctions électroniques, ceux-ci constituent *l'implémentation matérielle* du système embarqué à concevoir. Au plus bas niveau d'abstraction électronique, ces circuits sont constitués de *transistors*. Le transistor est le composant électronique essentiel et le plus basique pour la réalisation de circuits intégrés. Un assemblage cohérent de plusieurs transistors permet la réalisation de *portes logiques*, qui réalisent un traitement fonctionnel logique très simple sur des signaux électroniques binaires. L'assemblage de portes logiques permet la réalisation de modules de traitement plus complexes comme des puces spécialisées (ASIC) ou des processeurs génériques (GPP).

Au fil des années, la fabrication de ces circuits n'a cessé de s'améliorer : diminution du temps de commutation des portes logiques, miniaturisation de la taille du transistor sur une puce, augmentation de la fréquence des calculs, etc. Les limites technologiques ont été repoussées à tel point que les techniciens des matériaux font désormais face à des obstacles résultants d'effets atomiques fondamentaux.

Continuer d'augmenter sans cesse la fréquence des calculs amène des problèmes de dissipation de chaleur et de consommation d'énergie. Cette méthode n'étant plus réellement une bonne solution pour améliorer la capacité de calcul, la tendance actuelle est au *parallélisme*. A tous les niveaux hiérarchiques, le nombre d'opérations indépendantes qui s'exécutent simultanément est maximisé : circuits autonomes, pipeline, multiplication de coeurs logiques et physiques des processeurs, découpage des applications en *thread*, etc.

De plus, la très grande densité possible au sein de circuits intégrés permet dorénavant de rassembler toutes les fonctionnalités en un seul circuit final. C'est pourquoi les circuits électroniques, surtout ceux destinés à équiper les systèmes embarqués, ont de moins en moins tendance à n'être responsable que d'une seule fonctionnalité mais plutôt à intégrer tout un *système*. C'est pourquoi apparaît au niveau des circuits intégrés la notion de *Système sur Puce* ou « System-on-Chip », un circuit électronique d'un seul tenant intégrant toutes les fonctionnalités du système, comprenant les circuits de conversion analogiques-numériques, les unités de calculs génériques ou spécialisés, les mémoires, une éventuelle unité de gestion centrale, etc. Cette méthode présente de nombreux avantages : d'un point de vue coût, il ne faut gérer qu'une seule puce, la bande passante entre les éléments du circuit est très élevée car ces éléments sont situés dans le même boîtier, les problèmes de fabrication du circuit imprimé et des interconnexions entre composants discrets sont simplifiés et l'accès à la mémoire (également embarquée dans la puce) est plus rapide. En fait, toute communication avec un élément situé à l'extérieur du boîtier de la puce ralentit fortement le système et pose des problèmes de fabrication très complexes.

Le parallélisme des opérations, la conception par couches d'abstraction successives et l'intégration sur une seule puce de ces systèmes conduiront à une modélisation de ceux-ci sous la forme de *systèmes distribués hiérarchiques* dans la plupart des outils modernes de prédiction de performances.

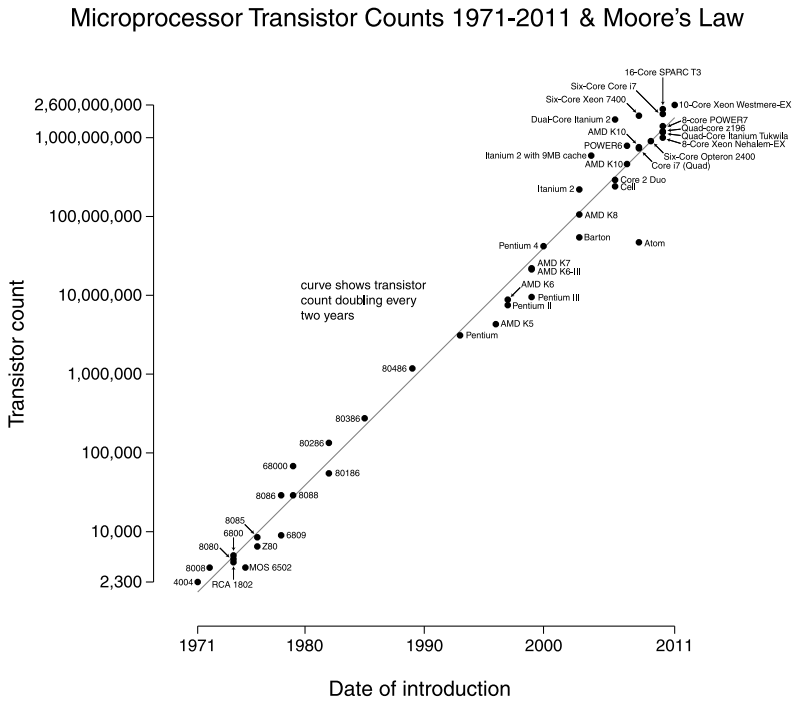


FIGURE 1.1 – Loi de Moore - Nombre de transistors sur les processeurs par rapport à l'année où ils ont été introduits [12]. L'échelle est logarithmique : la droite correspond à une croissance exponentielle.

La loi de Moore, formulée par Gordon Moore en 1965, stipule que le nombre de transistors par unité de surface double tous les 2 ans environ. Cette loi empirique s'est révélée étonnament exacte ces 30 dernières années. Cette augmentation exponentielle de la taille des circuits induit une complexité grandissante de leur conception. Les problèmes liés à la conception physique des circuits comme le placement, le routage, la génération de layout, etc sont aujourd'hui gérés automatiquement par des machines. Néanmoins, ces problèmes sont d'une part pour la plupart *NP-complets*, d'autre part travaillent sur des instances de tailles gigantesques (le nombre d'objets à traiter doublant tous les deux ans). Il est donc très difficile de résoudre, pour ces problèmes, des instances dont la taille atteint quelques milliards d'objets. Ceci nécessite énormément de ressources computationnelles et énormément de temps de calcul que les industries ne peuvent se permettre de perdre, *a fortiori* si ce temps est perdu à générer des solutions de design sous-optimales.

D'un autre côté, les innovations technologiques fleurissent et ne manquent pas de poser de nouveaux problèmes de design. Par exemple, il sera bientôt possible de réaliser des circuits par empilement de couches successives en trois dimensions. En plus de permettre un parallélisme accru, l'introduction d'une troisième dimension permet d'augmenter fortement la bande passante des canaux de communication entre les blocs situés sur des couches différentes du circuit. Par conséquent, l'espace des solutions possibles de design des circuits, déjà très grand initialement, devient énorme. De plus, de nouveaux critères de performances sont à prendre en compte, comme la puissance dissipée localisée en certains points chauds (appelés *hotspots*), qui est à

minimiser pour éviter une surchauffe en certains points précis du circuit.

L'augmentation des capacités de la technologie du silicium des précédentes décennies et ce contexte de conception et fabrication coûteuses des circuits conduisent à des designs très complexes qui deviennent fastidieux à appréhender avec les outils et méthodologies existantes. Ainsi, il serait intéressant de disposer de nouvelles méthodologies de conception. De façon similaire, la présence d'outils d'évaluation de performances *a priori* et de guides *heuristiques* pour les choix de conception de haut niveau de design serait très utile.

1.2 But du mémoire

Le but de ce mémoire est de montrer le contexte actuel de conception des systèmes embarqués et de mettre en évidence la nécessité de disposer de modèles de coûts, liant les choix de conception haut niveau aux différentes métriques pour évaluer la performance de ces systèmes. Nous présenterons le contexte industriel de la conception des systèmes embarqués, deux méthodes utilisées en recherche pour évaluer la performance et une approche originale pour créer de tels modèles.

L'objectif du mémoire est de présenter les systèmes embarqués, certaines difficultés qu'ils posent pour leur conception et, de façon essentielle, de proposer une approche pour la modélisation du lien entre les choix de conception électroniques et informatiques d'une part et les différentes métriques de performances du système résultant d'autre part. L'idée principale est de proposer l'extraction de « modèles de coûts » en termes, par exemple, de temps, de surface et d'énergie consommée sur base de larges ensembles de données issues de simulations. Ces simulations sont basées sur des cas d'études réels, comme les applications typiques du traitement du signal déployées sur architectures numériques. Afin de passer des données récoltées en simulation aux modèles prédictifs, des techniques d'apprentissage automatique (ou *machine learning*) seront exploitées. Les modèles multicritères ainsi créés permettraient l'évaluation de la pertinence de solutions de conception de systèmes en termes de performances et de guider les choix des concepteurs au travers de l'élaboration du système.

Le mémoire est structuré en trois parties :

- la partie I, couvrant les chapitres 2 à 4, réalise un panorama des méthodes de base utilisées en industrie pour concevoir des systèmes embarqués ;
- la partie II, couvrant les chapitres 5 à 8, montre deux travaux de recherche en cours dans le domaine, dédiés à l'amélioration des méthodes de conception ;
- la partie III, couvrant les chapitres 9 à 11, introduit une approche de modélisation exploitant des techniques d'analyse de données issues du *machine learning*.

Première partie

Méthodes industrielles de conception de systèmes embarqués

Chapitre 2

Conception de systèmes numériques

La présente partie I de ce mémoire, qui expose le contexte industriel de la conception de systèmes embarqués, est structurée en trois chapitres. Le chapitre 2 présente les notions générales relatives à l'électronique numérique, les systèmes de traitement d'information et leur conception. Le chapitre 3 traite de la définition et des contraintes de systèmes plus spécifiques, les systèmes embarqués. Au cours de ce chapitre, les difficultés de conception de ces systèmes sont mises en évidence, en introduisant l'explosion du nombre de choix de design et l'optimisation des métriques de performances. Le chapitre 4 concluera sur les perspectives de conception en industrie et sur le besoin de nouveaux outils et méthodologies.

Ce chapitre présente les bases de la conception de systèmes numériques [1] généraux. Tout d'abord, les définitions relatives à ce domaine seront fixées. Ensuite seront présentées les différentes vues possibles sur un système numérique ainsi que les différents niveaux d'abstraction existants. Enfin, un schéma classique condensant ces informations, connu sous le nom de *diagramme Y*, sera introduit.

Le contenu de ce chapitre est inspiré de deux ouvrages de Gajski [GK97, GAGS09].

2.1 Définitions

Un *système électronique* est un dispositif utilisant les propriétés de l'électricité afin de représenter, transférer et manipuler des *informations*.

L'information est portée par des signaux électriques, et l'objet de l'électronique appliquée est d'investiguer les moyens techniques pour manipuler efficacement ces signaux.

Un système électronique peut être soit *analogique* soit *numérique*, en fonction de la manière utilisée pour coder l'information dans le signal électrique.

– Les signaux analogiques sont les signaux dont l'information codée varie de façon continue en temps et en amplitude. La valeur de l'information est directement codée dans le signal électrique. Ils représentent typiquement des grandeurs

1. Ce domaine est plus connu dans sa terminologie anglaise qui est *« digital system design »*.

physiques directement issues du monde réel. Par exemple, la sortie d'un capteur de température est un signal analogique.

– Les signaux numériques, par opposition, se restreignent à coder l'information dans un espace fini de valeurs. En général, le temps est lui-même discrétisé lorsqu'on étudie un signal numérique. Le nombre de valeurs que peut prendre un signal numérique est limité et ces valeurs sont souvent codées par des nombres binaires.

Dans la suite de ce document, tous les systèmes électroniques considérés seront des systèmes numériques.

Un *processus de conception*, pour tout type de produit, explicite la démarche complète de production de ce produit, depuis la définition de ses fonctions jusqu'à la fabrication de l'objet réel.

La première partie de ce document détaillera les processus utilisés en industrie pour concevoir des systèmes numériques, en particulier les systèmes embarqués. La définition de ces derniers sera traitée au sein du chapitre 3.

2.2 Représentation de la conception

Lors du cycle de vie d'un produit, c'est-à-dire la définition, la conception et la fabrication de ce produit, il existe différentes représentations, ou *vues*, de celui-ci et donc de sa conception. Une même représentation peut également être décrite avec différents niveaux de détails dans les différentes phases du design suivant les besoins des concepteurs. Les trois représentations les plus généralement prises comme repères sont les suivantes :

– comportementale (de l'anglais, *« behavioral »*), spécifie le produit comme une boîte noire en décrivant les entrées, les sorties et le lien fonctionnel entre les deux sans spécifier la structure interne de cette fonction ;

– structurelle (de l'anglais, *« structural »*), définit l'intérieur de la boîte noire en utilisant un ensemble de composants et leurs interconnexions. Par opposition à la vue comportementale, cette représentation définit les plans de réalisation du produit sans en spécifier la fonctionnalité. Cependant, cette vue reste une vue logique sur le produit, par opposition à la vue physique ;

– physique (de l'anglais, *« physical »*), définit les caractéristiques physiques de la boîte noire, comme les dimensions et les localisations réelles de chaque composant et des connexions décrites dans la représentation structurelle. Cette représentation est utilisée pour décrire le produit après sa fabrication en explicitant son poids, sa taille, sa consommation d'énergie, la position de chaque patte d'entrée ou sortie du circuit et d'autres quantités physiques qui lui sont reliées.

Le processus de conception des systèmes numériques consiste alors en au moins trois phases centrées autour de ces trois représentations :

– Partant du cahier des charges, décrire une représentation comportementale afin de définir la fonction du système.

– En utilisant une librairie de composants donnée, convertir la représentation comportementale obtenue à l'étape précédente en une représentation structurelle.

– A l'aide de la représentation structurelle et d'outils de synthèse, produire une représentation plus détaillée encore, compatible avec une implémentation physique du système (contenant, de façon sous-jacente, la façon de fabriquer le produit). Cette nouvelle représentation s'accompagne de grandeurs physiques (comme la surface, la consommation d'énergie, etc).

Pour n'importe quel système numérique, le design passera par ces trois phases à *différents niveaux d'abstraction*.

La nécessité d'intervention humaine dans la conception varie selon les phases et les niveaux d'abstraction, ainsi que selon les époques auxquelles le système a été conçu. Historiquement, les premiers circuits numériques étaient basés sur des *layouts* de transistors dessinés à la main. Avec cette méthode, il était possible de gérer une complexité de circuit de l'ordre d'une centaine de transistors. Cependant, l'augmentation exponentielle de la densité [2] des circuits et, par conséquent, de la complexité de ceux-ci a conduit naturellement à l'introduction d'outils de CAO [3] qui permettent une automatisation des différentes phases. Chronologiquement, des outils synthétisant les phases les moins abstraites ont d'abord commencé à apparaître et de proche en proche, des outils pour réaliser la synthèse des phases de niveau de plus en plus haut furent conçus.

Abstraire la conception est non seulement un moyen de gérer plus facilement la complexité des designs, mais également (par conséquent) d'augmenter la productivité de ceux-ci. L'introduction de niveaux d'abstraction supplémentaires permet en quelque sorte de « rattraper » la loi de Moore. Il existe cependant tout de même un fossé entre la productivité humaine et la capacité des circuits, illustré par la figure 2.1. Le taux d'accroissement de la capacité des circuits de 58% est dû à l'amélioration des technologies de gravure (le support physique) tandis que le taux d'accroissement de la productivité de 21% est une conséquence de l'introduction de nouvelles méthodes de conception de niveaux de plus en plus haut.

Les concepteurs de circuits électroniques sont donc confrontés, au cours des années, à des problèmes de nature de plus en plus abstraite, étant donné le niveau élevé des outils de design disponibles. Les contraintes physiques reliées au produit, comme son temps de traitement et son énergie consommée, ne sont néanmoins pas négligées.

La section suivante présente les différents niveaux d'abstraction classiques et les confronte aux différentes représentations du design.

2.3 Couches d'abstraction

Dans le processus de conception, chacune des trois représentations décrites en 2.2 peut être utilisée à différents niveaux d'abstraction. Les niveaux d'abstraction sont définis par le type d'objets qu'ils utilisent comme composants dans la représentation structurelle. La table 2.2 présente un résumé de ces niveaux d'abstraction.

2. La densité d'un circuit numérique est définie par le nombre de transistors implantables par unité de surface.

3. Conception Assistée par Ordinateur.

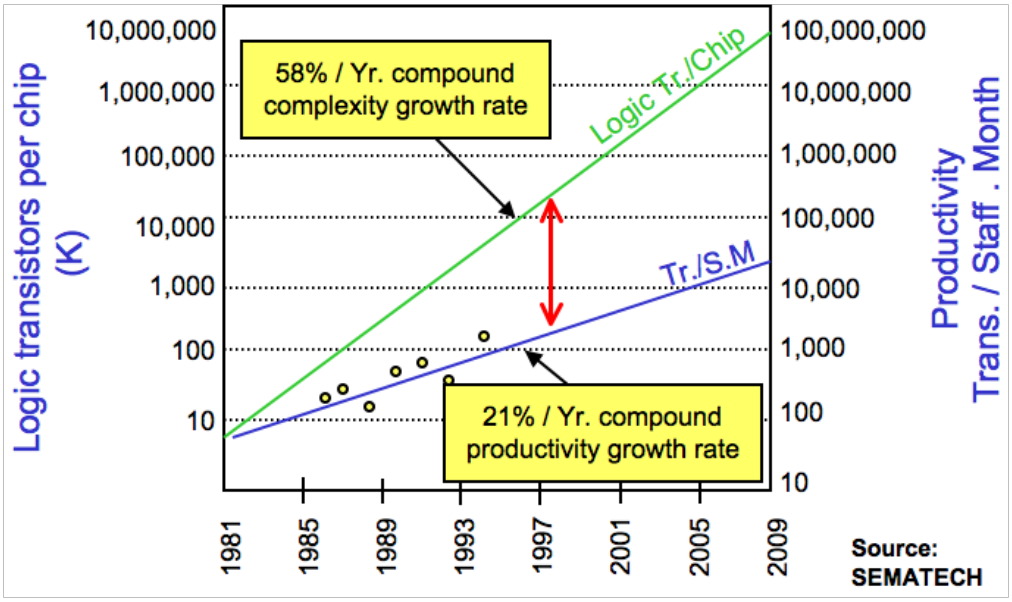


FIGURE 2.1 – Fossé de productivité entre la capacité des circuits électroniques et la capacité des humains à concevoir des systèmes [Van09]. En termes de nombre de transistors, les circuits ont un taux d'accroissement de 58%, tandis que le nombre de transistors qu'un humain implante en moyenne par mois dans un système a un taux d'accroissement de 21%.

Niveau	Outils de description du comportement	Composants structurels
Transistor	Equations différentielles, voltage en fonction du temps.	Transistors, résistances, capacités.
Porte	Equations booléennes, automates d'états finis.	Portes, bascules.
Registre	Algorithmes, diagrammes, jeux d'instructions, ...	Additionneurs, comparateurs, registres, compteurs, ...
Processeur	Spécification du système, programmes, ...	Processeurs, contrôleurs, mémoires, ...

FIGURE 2.2 – Niveaux d'abstraction et composants structurels qui y sont disponibles [GK97].

Cette section décrit les différents niveaux d'abstraction et leurs composants, en partant du niveau le plus bas, le plus proche de la physique.

Un assemblage cohérent de *transistors*, le plus petit élément actif d'électronique numérique, forme une cellule qui se comporte comme un opérateur de logique propositionnelle. Ce schéma composé forme, au niveau d'abstraction supérieur, les composants de base appelés *portes logiques*. La figure 2.3 illustre le schéma de la réalisation d'une porte NAND à l'aide de transistors CMOS. Notons que pour commuter [4], la

4. Une porte logique *commute* lorsque sa valeur de sortie change suite à une modification de

porte nécessite une alimentation en énergie, représentée par la tension entre le signal *Vdd* et la masse.

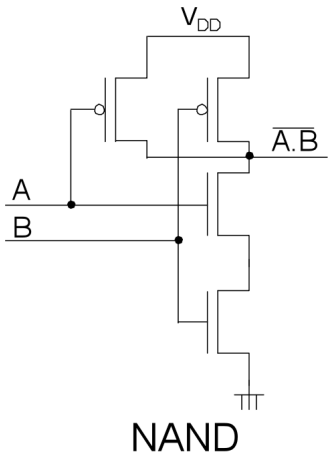


FIGURE 2.3 – Porte NAND réalisée à l'aide de 4 transistors CMOS. Notons la présence des signaux d'alimentation, *Vdd* et *Gnd*.

Une porte logique est une cellule électronique disposant d'entrées et de sorties et imposant sur ses sorties un potentiel électrique qui est une fonction logique de ses entrées. Il existe des portes AND, OR, XOR, NAND, NOR et NOT. La figure 2.4 illustre les schémas de différentes portes à ce niveau d'abstraction. Un certain assemblage de portes se comporte comme une équation de logique propositionnelle et forme, physiquement, un module. Il existe également à ce niveau d'abstraction des bascules (en anglais, *flip-flops*) qui permettent de mémoriser un *bit*. Ces composants permettent d'introduire une notion d'état, de mémoire, au sein des circuits et de représenter le comportement d'un automate d'états fini. Si tous les flip-flops sont pilotés par une même horloge centrale au système, celui-ci est appelé *système synchrone*. Ces assemblages de portes et de bascules [5] forment des modules appelés *design units*. Ces modules, qui typiquement implémentent des opérations d'arithmétique (addition, soustraction, etc.), de logique (comparaison, masques, etc.) ou de stockage (registres), sont utilisés comme composants au niveau d'abstraction suivant.

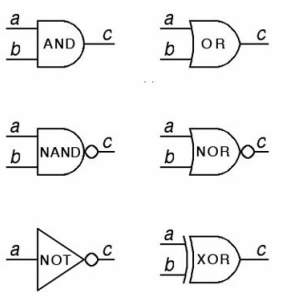


FIGURE 2.4 – Schémas des différentes portes logiques.

Le niveau d'abstraction suivant, appelé communément *Register Transfer Level*

valeur de ses signaux d'entrée. Le temps de commutation est non-négligeable (de l'ordre de quelques nanosecondes) et constitue l'une des sources des délais au sein des circuits électroniques.

5. Une bascule peut elle-même être synthétisée à l'aide de portes logiques placées en *rétroaction*.

(RTL), utilise les modules synthétisés au niveau logique pour produire des composants plus complexes, appelés *design entities*. En assemblant ces modules, on peut produire le schéma complet de circuits intégrés comme des processeurs génériques ou des puces de traitement plus spécialisées.

Le niveau suivant sort du cadre de la conception microélectronique. En effet, il s'agit d'un assemblage de composants discrets sur un circuit imprimé, pour former un circuit complet intéragissant avec le monde extérieur. Les composants de base sont les processeurs, les mémoires, les contrôleurs, les interfaces ainsi que tous les circuits dédiés qui ont pu être conçus à des niveaux d'abstraction inférieurs. L'élaboration d'un circuit imprimé complet fait intervenir de nombreuses disciplines et sort du cadre de ce mémoire.

Cependant, avec l'avènement des *System-on-Chip* (SoC), les paramètres changent considérablement. Les éléments constitutifs du système sont intégrés au sein d'un même circuit intégré et il est nécessaire de relever l'abstraction de ces circuits à un niveau supérieur. On parle alors de *system-level design*, où les composants de haut niveau cités ci-dessus, traditionnellement assemblés en discret sur un circuit imprimé, sont maintenant forgés dans un même composant et reliés entre eux soit par des bus de très grande bande passante, soit par des systèmes de communication plus évolués comme des *Network-on-Chip* (NoC) [DT01].

En combinant les différents niveaux d'abstraction et les différentes représentations d'un design, il est possible d'obtenir de multiples visions d'un produit. Pour utiliser à bien ces concepts, il faut choisir une méthodologie qui spécifiera quels sont les outils qui seront utilisés, quels niveaux d'abstraction couvriront-ils et quels sont les endroits où l'intervention d'experts humains sont nécessaires.

2.4 Diagramme Y

En 1983, Daniel Gajski introduit le « graphique en Y », pour expliquer les différents outils de CAO et leur position dans les représentations et les couches d'abstraction. La figure 2.5 montre ce graphique.

En toute logique, les concepteurs partent du niveau le plus haut du côté comportemental (la spécification du système) pour terminer au niveau le plus haut du côté physique (une représentation du système physique terminé). Le chemin parcouru à travers le graphique, quant à lui, est spécifique à chaque méthodologie de conception susceptible d'être rencontrée en industrie. La figure 2.6 illustre une trajectoire typique pour la conception d'un système dans un flot de conception haut niveau ou *system-level*. Le but de cette première partie du mémoire est de montrer quelles sont les méthodologies utilisées à ce jour, quelles sont leurs limites et quelles idées pourraient être appliquées pour les améliorer.

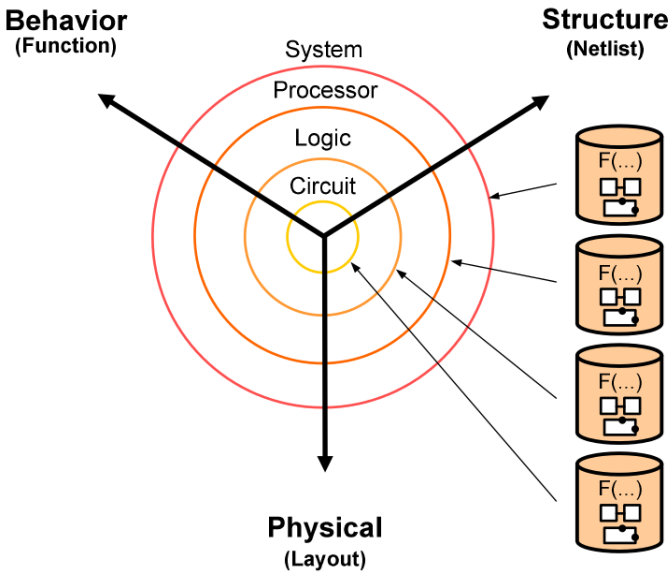


FIGURE 2.5 – Graphique en Y [GAGS09].

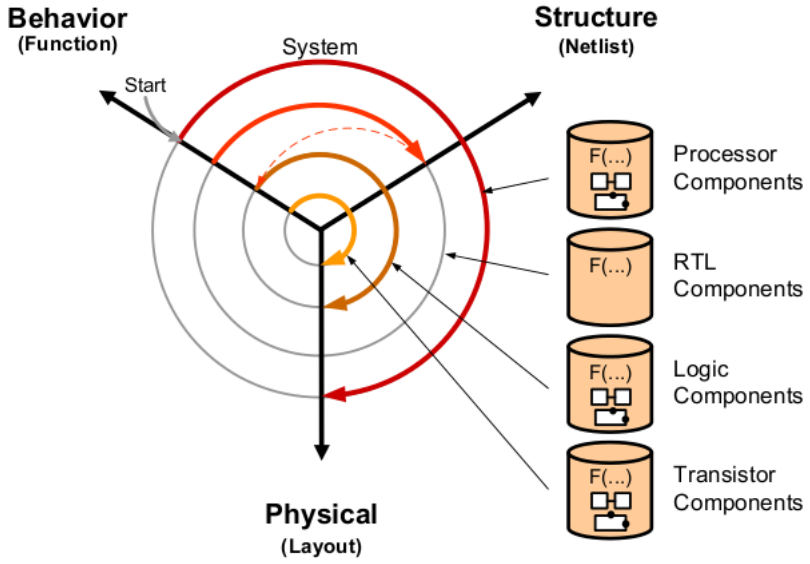


FIGURE 2.6 – Trajectoire typiquement empruntée pour concevoir un système embarqué dans un flot de conception *system-level* [GAGS09].

Chapitre 3

Les contraintes des systèmes embarqués

Dans le chapitre précédent, les systèmes numériques généraux ont été abordés. Ce chapitre va traiter d'une sous-catégorie importante de ces systèmes, les systèmes embarqués. D'abord, des exemples de ces systèmes seront fournis ainsi qu'une tentative de définition. Ensuite, le problème de l'optimisation des métriques de performances associées au système sera abordé. Enfin, un panorama des choix de conception haut niveau pour la conception de ces systèmes sera présenté.

Les notes de ce chapitre sont principalement inspirées de l'ouvrage de Vahid et Givargis [VG02].

3.1 Exemples et définitions

Définir le concept de *système embarqué* n'est pas une sinécure étant donné le grand nombre d'appareils concernés. De nombreux auteurs ont alors tendance à fournir une description de ces systèmes qui leur est personnelle. Ceci implique qu'il existe de nombreuses définitions sensiblement différentes qui parfois se contredisent.

Nous allons donc tenter de fournir une définition qui est la plus générale possible pour englober le plus possible de ces systèmes.

Historiquement, le terme « embarqué » fut utilisé car les premiers appareils dans lesquels de l'électronique et, plus tard, de l'informatique, ont été implantés furent les trains et les avions. Les systèmes ainsi conçus étaient alors « embarqués » à bord des véhicules qu'ils équipaient. Déjà à l'époque, de nombreuses contraintes, notamment de *fiabilité*, s'exercaient sur ces systèmes.

Avec le progrès exponentiel de la capacité des circuits intégrés, les appareils dotés d'électronique commencèrent à apparaître de plus en plus jusqu'à devenir indispensables dans certaines applications.

Ce type d'appareil est présent tout autour de nous au sein de notre quotidien : téléphones portables, baladeurs de musique MP3, appareils photo, caméras numériques, télévisions, consoles de jeux vidéos, photocopieuses, systèmes d'alarmes, etc. Au sein d'une automobile, un très grand nombre de processus sont gérés automatiquement

par des systèmes électroniques. Le « cruise control », l'ABS, la gestion de l'injection d'essence en sont des exemples. Faire une liste exhaustive de tels systèmes demanderait un temps considérable et représenterait un exercice difficile tellement ceux-ci sont nombreux et différents de par leur but.

Sur base de ces exemples fort différenciés, nous pourrions avoir tendance à vouloir définir un système embarqué de la façon suivante : un système embarqué est tout appareil électronique qui n'est pas un ordinateur de bureau. Cette définition toute simple atteint cependant rapidement ses limites. En effet, un ordinateur portable est une réplique à une plus petite échelle d'un ordinateur de bureau, mais il subit néanmoins quelques contraintes de l'embarqué (autonomie, poids, taille). Notons donc que la limite entre le monde de l'embarqué et celui du « desktop » est floue.

Pour aller plus loin, en partant des exemples de tels systèmes donnés ci-dessus, nous pourrions mettre en évidence certaines caractéristiques communes des systèmes embarqués :

- Le but premier d'un système embarqué est d'assurer ***une*** *fonctionnalité*.
- Un système embarqué est soumis à un certain nombre de *contraintes* en termes de performances.
- Pour concevoir un système embarqué, il existe un certain nombre de *choix de design* n'ayant pas ou peu d'impact sur la fonctionnalité mais qui influencent considérablement les performances du système final et, par conséquent, la satisfaction des contraintes liées à ses performances.

A nouveau, il est très facile de critiquer ou de contrecarrer ces caractéristiques avec de simples exemples. L'objet le plus emblématique du monde de l'embarqué en ce début de vingt-et-unième siècle est sans doute le *smartphone*. Celui-ci est assurément un système embarqué vu le nombre de contraintes auxquelles il est soumis (autonomie, puissance dissipée sous forme de chaleur, poids, volume, temps de mise sur le marché, pour n'en citer que quelques unes) mais assure un nombre de fonctions qui est largement supérieur à 1 (téléphonie, accès à Internet, photographie, baladeur MP3, etc).

De façon similaire, les circuits intégrés d'aujourd'hui ont tendance à incorporer le système complet en son sein, avec toutes ses fonctionnalités. Ces circuits se nomment *System-on-Chip*. Un exemple d'un tel circuit est schématisé sur la figure 3.1 pour la puce d'un appareil photo.

Ainsi, il est peut-être nécessaire de faire évoluer la définition de système embarqué, étant donné qu'elle ne fait pas consensus. Peut-être ne doit-on plus séparer les systèmes en deux catégories « informatique haute performance » (ordinateurs de bureau, serveurs, mainframes, etc.) et « informatique embarquée » (téléphones, appareils photos, satellites, récepteurs GPS, etc.) mais plutôt établir un spectre continu de ces systèmes et quelle est leur tendance, plutôt haute performance ou plutôt embarquée.

Nous nous contenterons cependant de cette définition en 3 points car elle est adéquate pour la plupart des systèmes embarqués.

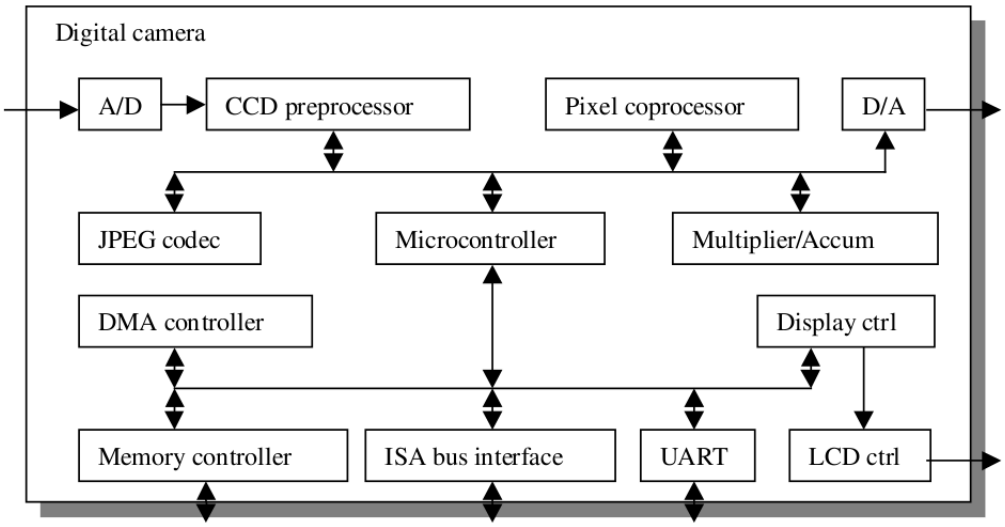


FIGURE 3.1 – System-on-Chip d'un appareil photo numérique [VG02].

3.2 Optimisation des performances

Le plus grand défi posé par la conception de systèmes embarqués est l'optimisation de leurs performances. Bien que la satisfaction de la fonctionnalité de ces systèmes pose ses propres problèmes fondamentaux [1], ce qui pose réellement problème aux concepteurs d'aujourd'hui réside dans la création d'une implémentation qui optimise les nombreuses métriques associées à ce système. Comme énoncé ci-dessus, pour faire face à ces contraintes strictes de performances, les concepteurs ont à leur disposition un certain nombre de choix de design. Cependant, plus le système est complexe, plus le nombre de choix est grand, et les systèmes d'aujourd'hui font rapidement exploser ce nombre au delà de la perception humaine.

Voici quelques métriques [2] usuelles associées aux performances d'un système embarqué :

le coût non-récurrent d'ingéniérie (NRE) : le coût complet, payé une seule fois, pour le design du système. Une fois le système conçu, un nombre indéfini d'unités peut être produit. Ce coût inclut notamment le salaire des ingénieurs travaillant sur le design mais également, les coûts fixes entraînés par le processus de fabrication. Dans le cas d'un ASIC par exemple, il incluera le coût de fabrication des masques nécessaires pour la gravure du layout de transistors sur le silicium.

le coût unitaire : le coût de fabrication d'une copie du système, sans compter le coût non-récurrent. Dans le cas ASIC, ce coût est fortement lié à la surface de silicium occupée par le circuit. Ceci dépend de la technologie de gravure [3] et du nombre de transistors du circuit.

1. Le lecteur intéressé par ce sujet se renseignera auprès des spécialistes du domaine de la vérification formelle de systèmes informatiques.

2. Une métrique d'un design est une grandeur physique mesurable directement sur l'implémentation de ce design.

3. Essentiellement, la distance minimale nécessaire entre deux transistors. On parle de « technologie 180 nanomètres », « technologie 45 nanomètres », etc.

le délai d'exécution : le temps nécessaire que prend le système pour exécuter la fonctionnalité voulue. Celui-ci peut également être exprimé comme un débit ou une vitesse (*e.g.* nombre de tâches exécutées par seconde). Pour un système très large, il peut s'agir de mesures plus ponctuelles, comme la vitesse moyenne d'exécution de l'ensemble des tâches ou le temps de réponse à une requête spécifique. On distingue généralement le *temps de réponse* du *débit* car le parallélisme des opérations peut, dans certains cas, améliorer le débit sans changer le temps de réponse [4].

la puissance : la puissance consommée par le système. Premièrement, elle est directement reliée à l'énergie consommée et donc à l'autonomie de la batterie d'un système autonome. Deuxièmement, elle est reliée à la puissance dissipée par le circuit sous forme de chaleur : il faudra éventuellement refroidir ce circuit d'une façon ou d'une autre.

la taille : les dimensions géométriques du système. Relatif à l'objet, il peut s'agir de son volume et de son poids. Du point de vue du circuit, on fait généralement référence à sa surface.

la flexibilité : la capacité de modifier le système sans entraîner trop de surcoûts non-récurrents. Par exemple, le choix d'implémenter une fonctionnalité en *software* sur un processeur embarqué est un choix qui mène à une grande facilité, car il suffit de remplacer le programme afin de modifier la fonctionnalité.

le « time-to-prototype » : le temps requis pour construire un modèle fonctionnel du système, sans compter le temps de production du produit. Ce prototype peut être utilisé à des fins de vérification et de correction.

le « time-to-market » : le temps requis pour développer le système en un produit au point qu'il peut être délivré aux consommateurs. Ceci inclut une phase de conception, de fabrication et de tests.

la maintenabilité : la capacité de modifier le système après l'avoir initialement produit afin de pouvoir, par exemple, corriger des erreurs ou en améliorer l'efficacité. En particulier, il doit pouvoir être maintenu par des ingénieurs n'ayant pas originellement conçu le système. La capacité à mettre à jour le *firmware* d'un appareil constitue un exemple de maintenabilité.

la correction : les preuves que le système réalise effectivement la fonction requise dans le cahier des charges, sans erreur. Cette confiance peut être apportée par une série de tests logiciels (batteries de *testbenches*) ou matériels (circuiterie intermédiaire).

la fiabilité : la probabilité que le système ne tombe pas en panne et la qualité des mécanismes mis en place pour le recouvrement de telles pannes potentielles.

Les métriques sont généralement conflictuelles : à un certain point de design, améliorer l'une peut potentiellement détériorer les autres. Réduire le délai d'exécution peut augmenter le coût unitaire, et vice versa. Réduire le coût unitaire peut augmenter les coûts non-récurrents, et ainsi de suite. On peut schématiser ceci par une boule dans

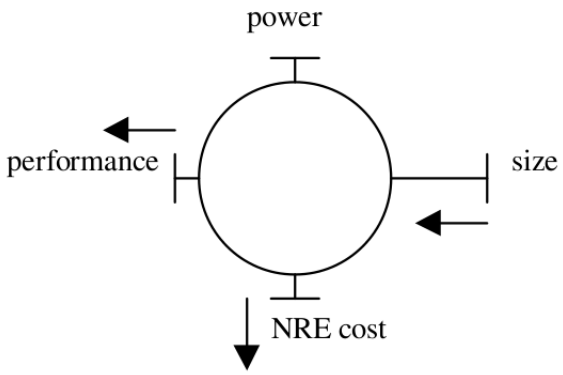


FIGURE 3.2 – Conflit des métriques [VG02] pour les délais (« performance »), la puissance (« power »), la taille (« size ») et les coûts non-récurrents (« NRE cost »).

laquelle on enfonce des aiguilles (les métriques). Lorsqu'on enfonce certaines aiguilles, d'autres sont poussées vers l'extérieur. La figure 3.2 représente cette analogie.

Suite à cette observation, le premier constat est le suivant : il n'existe pas de solution purement optimale. Les critères à optimiser étant multiples, la notion d'optimalité n'a pas de sens et il est nécessaire de *choisir* une solution de *compromis*. La partie II explicite certaines pistes entreprises dans le monde de la recherche pour faire face à ce problème. En particulier, la section 6.1 introduit l'analyse multicritère, qui répond spécialement à cette problématique de critères conflictuels à optimiser simultanément.

Afin de pouvoir réaliser les meilleurs compromis, les concepteurs doivent connaître une variété conséquente de solutions matérielles et logicielles et être capable de migrer d'une solution à l'autre afin d'être en adéquation avec les contraintes de performances. Les contraintes sont telles que les équipes de concepteurs ne peuvent plus être constituées exclusivement d'« ingénieurs hardware » ou d'« informaticiens software ». Les concepteurs doivent pouvoir communiquer entre eux, quelle que soit leur spécificité, afin de manipuler les concepts des deux mondes convergents pour réaliser les choix qui, matériel et logiciel ensemble, réaliseront le meilleur compromis d'optimisation des métriques de design.

3.3 Vue d'ensemble des choix de conception

Comme introduit ci-dessus, le nombre de degrés de liberté de conception disponible tant au niveau matériel que logiciel est très grand. Afin de gérer une telle complexité, les concepteurs divisent le problème en un ensemble de décisions prises à des niveaux d'abstraction décroissants (*cf* section 2.3). La description du système est raffinée à chaque niveau depuis la spécification jusqu'à la puce électronique.

Les choix aux niveaux d'abstraction les plus bas sont aujourd'hui réalisés à l'aide d'outils de conception automatiques, mais les premières étapes de conception sont par contre encore principalement basées sur l'expertise du concepteur. Ces décisions

4. Les architectures de type *pipeline* permettent ce genre d'optimisations.

font cependant partie des plus critiques car elles conditionnent les choix de toutes les étapes ultérieures et les performances finales du système.

Cette section présente une vue d'ensemble de ces choix haut niveau [5] et l'impact qu'ils présentent en termes de performances sur le système.

Les différentes décisions *system-level* peuvent être regroupées en différentes catégories que nous appellerons *technologies*. Nous distinguons les technologies de calcul, de support physique, de conception et d'implémentation.

3.3.1 Technologies de calcul

La technologie de calcul fait référence à l'architecture choisie pour le système qui sera la plate-forme qui exécutera la fonctionnalité. Il s'agit du moteur logique d'exécution du système. Ces technologies sont en général distinguées par rapport au degré de spécialisation de la technologie pour la fonctionnalité désirée. Ces technologies de calcul sont souvent appelées *familles de logique*. Il existe trois grandes familles de logique : les processeurs génériques, les processeurs personnalisés et, de façon intermédiaire, les processeurs orientés application. Même si dans le langage courant, le terme « processeur » fait généralement référence aux processeurs à instructions (*i.e.* les processeurs génériques), nous l'utiliserons pour qualifier tout type d'architecture logique.

Processeurs génériques

Les processeurs génériques, plus connus en anglais sous le nom *« general-purpose processor »*, sont des calculateurs capables d'exécuter n'importe quel type d'application numérique. L'application va être découpée en opérations plus élémentaires nommées *instructions*. Plutôt que de connaître à l'avance les opérations à exécuter, le processeur générique va aller chercher les instructions [6] sauvegardées dans la mémoire du système.

Cette implémentation du moteur de calcul est très généraliste et permet le support architectural d'une très vaste gamme d'application. Ainsi, le concepteur d'un tel processeur maximise le nombre de produits vendus. Néanmoins, pour une application donnée, ce processeur est en principe moins performant qu'un circuit dédié à cette application [7]. Par exemple, certaines parties de l'aire du processeur sont parfois inutilisées, étant donné que c'est le programme situé en mémoire qui décide ce qu'il exploite dans le datapath du processeur. De plus, un travail de décodage des instructions doit être effectué avant l'exécution de celles-ci, car le processeur ne sait pas *a priori* ce qu'il doit faire étant donné que l'application n'est absolument pas câblée dans le fonctionnement de celui-ci.

5. A ce niveau d'abstraction, on dit de ces choix qu'ils sont *« system-level »*.

6. Ces instructions sont celles du programme représentant l'application.

7. Cependant, le très grand nombre d'applications différentes ayant opté pour cette solution ont conduit le monde de la recherche et de l'industrie à pousser très loin les optimisations architecturales des processeurs. Ainsi, ces processeurs font partie des circuits les plus complexes et les plus performants qui existent. Ces processeurs mènent également les avancées au niveau des technologies de support physique.

Ce choix de technologie de calcul est souvent appelé *choix software*, car le problème de l'implémentation de la fonctionnalité est reporté sur la création d'un programme, suite d'instructions qui réaliseront la fonctionnalité, et le circuit qui l'exécutera est acheté ou produit indépendamment de ce programme. De façon similaire à la conception de systèmes numériques, la programmation des processeurs génériques a évolué au cours du temps par couches d'abstraction successives. Les premiers programmes de ces processeurs génériques étaient directement écrits avec les *opcodes* binaires. Les langages d'assemblage ainsi que d'autres de plus haut niveau sont rapidement apparus, accompagnés de leurs compilateurs (l'équivalent de l'outil de synthèse pour les circuits). Aujourd'hui, le langage de programmation prédominant dans le monde de l'embarqué pour la programmation de processeurs génériques est le C [8].

En termes de métriques, il existe de nombreux avantages à choisir un processeur générique. Le temps de mise sur le marché et les coûts non-récurrents sont très réduits, car le processeur étant déjà disponible, il suffit d'écrire un programme [9]. La flexibilité est également très grande car il suffit de modifier le programme (habituellement copié dans une mémoire de type ROM) pour changer la fonctionnalité. Le coût unitaire est réduit si le nombre de pièces produites du système est limité. En effet, les fabricants de processeurs génériques distribuent les coûts non-récurrents intrinsèques à la création d'un objet aussi complexe sur le grand nombre de pièces produites afin que le prix soit abordable. Les métriques plus physiques, comme les délais ou l'énergie consommée, sont réduites au vu des optimisations dont bénéficient ces processeurs. Notons que pour un processeur générique fixé, la surface du circuit et la puissance (en termes de watts) sont des constantes pour tout type d'application.

Il existe également des désavantages : pour un très grand nombre d'unités produites du système, le coût unitaire peut sembler élevé comparé à l'élaboration d'un circuit personnalisé. En effet si le nombre de pièces devant être produites est très grand, les coûts non-récurrents élevés peuvent être distribués sur ce grand nombre de pièces. Pour certaines applications (notamment en traitement intensif du signal), le débit peut être assez lent comparé à d'autres technologies. Certaines applications précises ou de petites tailles peuvent ne nécessiter qu'un nombre restreint de ressources, dans ce cas le processeur générique peut avoir une taille et une puissance encombrantes là où un circuit personnalisé aurait utilisé uniquement les ressources strictement nécessaires.

Processeurs dédiés

Par opposition aux processeurs génériques, les processeurs dédiés, connus en anglais sous le terme *« single-purpose processors »*, sont conçus pour exécuter une seule tâche spécifique. Le programme est câblé dans le fonctionnement du circuit. Souvent, ces circuits réalisent une tâche particulière pour assister un processeur générique, comme

8. Cette tendance se justifie par la proximité du C par rapport à la couche matérielle. Tout en fournissant une abstraction du type de processeur (contrairement aux différents langages d'assemblage qui dépendent des registres et instructions disponibles), le C permet un accès direct à la mémoire et une gestion personnalisée de celle-ci, ce que ne permettent pas ou peu les langages de plus haut niveau. Dans le contexte de l'embarqué, l'optimisation des accès à la mémoire est crucial en termes de métriques comme les délais ou la consommation d'énergie.

9. La partie *software* du développement est beaucoup moins coûteuse car nécessite notamment une main-d'oeuvre moins spécialisée et plus courante.

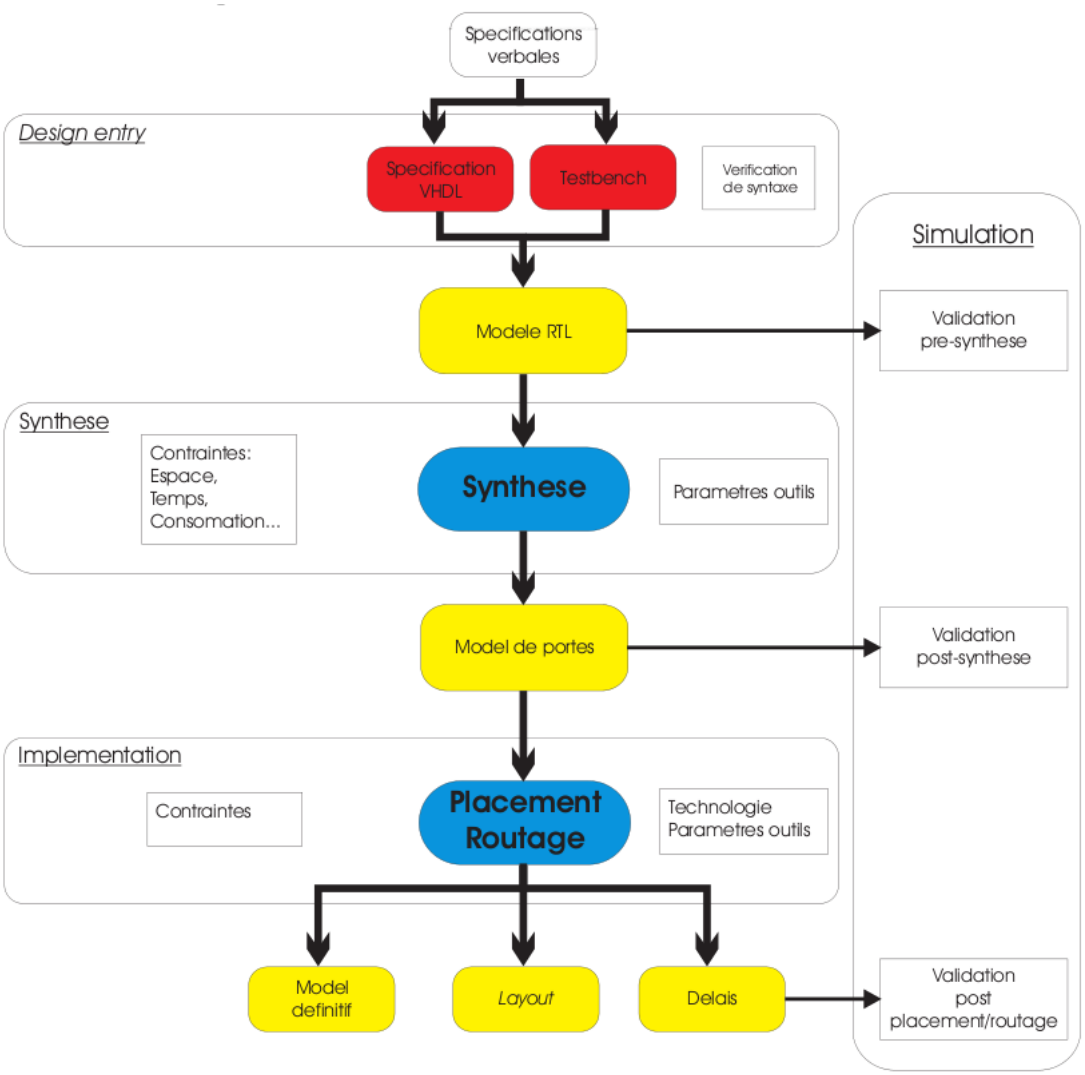


FIGURE 3.3 – Flot de conception RTL pour les processeurs dédiés [Mil11].

une tâche d'accélération graphique. Dans ce cas, le processeur dédié est alors appelé *coprocesseur*.

En pratique, les environnements de développement de tels processeurs sont sensiblement différents des environnements classiques de programmation des processeurs génériques. De nos jours, il s'agit d'écrire une spécification de l'application au niveau RTL et de l'utiliser comme entrée d'outils de synthèses dépendant de la technologie de support (ou cible) choisie. La figure 3.3 illustre un flot de conception commun de tels processeurs.

Les avantages et les inconvénients sont essentiellement les inverses du cas générique. Les délais, la taille et l'énergie consommée peuvent être réduits, tandis que le temps de prototypage et les coûts non-récurrents peuvent être très haut, la flexibilité est très réduite. Le coût unitaire résultant peut être très grand pour une petite quantité de circuits produite et les délais d'exécution peuvent ne pas atteindre la vitesse des

processeurs génériques optimisés (comme décrit plus haut).

La fonctionnalité étant câblée dans la logique du processeur, il n'est pas nécessaire de disposer d'une « mémoire de programme » comme présente dans le cas générique.

En résumé, un circuit dédié implémente exactement la fonctionnalité requise, ni plus, ni moins. Les retombées économiques ne sont cependant pas à négliger.

Processeurs orientés application

Les processeurs orientés application, en anglais *« application-specific instruction-set processor »* (ASIP), représentent la solution intermédiaire entre les processeurs totalement génériques et les processeurs totalement dédiés. Un ASIP est un processeur programmable (une mémoire de programme contient donc les instructions à exécuter) mais optimisé pour une certaine classe d'applications comme le contrôle de processus ou le traitement des signaux.

Les ASIP les plus répandus sont donc :
- les processeurs de traitement des signaux, en anglais *« Digital Signal Processor »* (DSP).
- les microcontrôleurs, utilisés pour contrôler des processus industriels.

Les concepteurs optimisent le datapath des ASIP en éliminant ce qui n'est jamais utilisé par ces classes d'application et en optimisant les manipulations les plus fréquemment utilisées. Par exemple, en traitement du signal, de nombreuses sommes de produits sont calculées. Il existe donc dans les DSP une instruction, nommée *MAC* (*« multiply-accumulate »*), permettant de réaliser une addition et une multiplication simultanément (pour réaliser l'opération $a \leftarrow a + b * c$ en un temps réduit).

Ainsi, l'intégration d'un ASIP au sein d'un système embarqué peut représenter un compromis entre les deux solutions extrêmes : la solution est flexible (il est toujours possible de modifier le programme) et le circuit voit ses délais, son énergie consommée et sa taille réduits. Cependant, l'écriture de programme pour ASIP peut être plus complexe car un compilateur depuis un langage de haut niveau peut ne pas exister, obligeant le concepteur à écrire son programme en assembleur. La conception d'un ASIP en tant que telle peut, elle, générer un coût non-récurrent très important.

La figure 3.4 représente les différences en termes d'architecture des trois technologies de calcul présentées. Les architectures du processeur générique et des ASIP sont très proches, la principale différence résidant dans la personnalisation de l'unité arithmétique et logique (ALU).

3.3.2 Technologies de support physique

Si la technologie de calcul représente l'implémentation logique du circuit, cette technologie-ci représente son implémentation physique. Chaque type de processeur doit être développé sur un certain circuit intégré. Un circuit intégré, communément appelé « puce » (ou *chip* en anglais), est un morceau de semiconducteur qui connecte ensemble différents transistors. Il existe de nombreuses façons de fabriquer les semiconducteurs, et la technologie la plus courante aujourd'hui est le CMOS [10].

10. « La technologie CMOS, pour *Complementary Metal Oxide Semiconductor*, est une technologie de fabrication de composants électroniques. Dans ces circuits, un étage de sortie est composé d'un

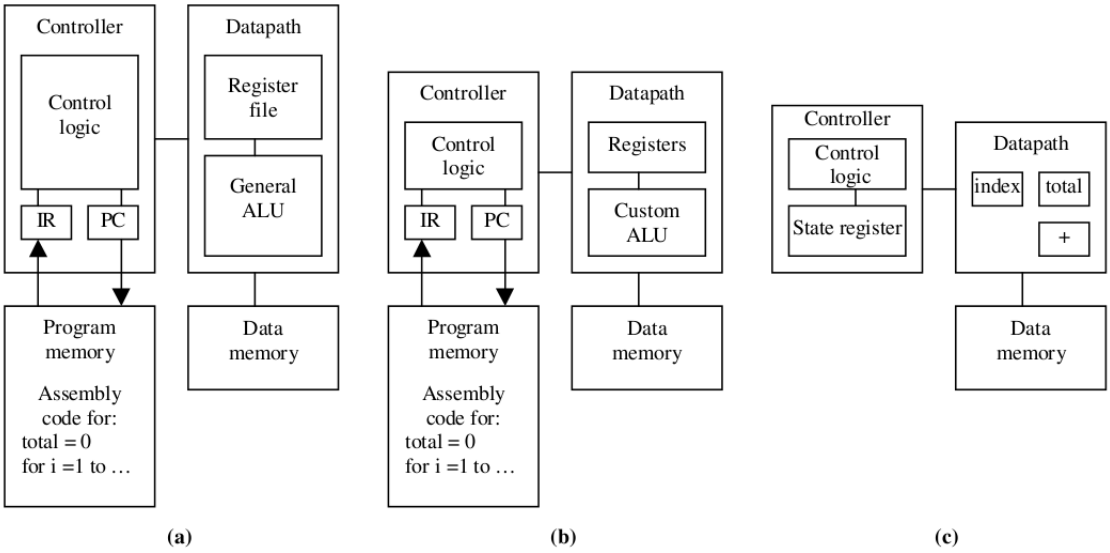


FIGURE 3.4 – Différences architecturales entre les différents types de logique [VG02]. (a) un processeur générique avec la mémoire contenant le programme et son ALU générique. (b) un ASIP avec son ALU personnalisée. (c) un processeur dédié n'utilisant que les strictes ressources nécessaires à l'application.

Les technologies de support physique se distinguent entre elles par la personnalisation du circuit intégré par rapport à son design (qui est, à ce niveau-ci, réduit à un layout de transistors ou à une netlist de portes logiques). Sans rentrer dans plus de détails, le circuit physique en tant que tel est construit en empilant une série de couches qui permettent de recréer l'effet physique correspondant aux transistors théoriques du schéma abstrait : les couches les plus basses sont les transistors, les couches du milieu permettent la formation des composants logiques (les différents blocs du design), les couches les plus hautes réalisent les connexions entre les composants à l'aide de pistes métalliques.

Pour créer chacune des couches du circuit, il faut préalablement concevoir des *masques* qui serviront de moules pour concevoir les circuits. La fabrication de ces masques représente un investissement important en termes de coût [11] et de temps.

Les technologies de support physiques, et donc les choix à ce niveau, se distinguent par la définition de qui réalise chacune de ces couches et à quel moment.

Full-custom

La technologie full-custom est celle qui personnalise et optimise le plus les couches pour le layout de transistors fourni. Ces optimisations incluent le placement des transistors de façon à minimiser leurs distances (en termes filaires), leur dimensionnement

couple de transistors MOSFET N et P placés de manière symétrique et réalisant chacun la même fonction. Du fait de leur caractéristique de fonctionnement inversée, un transistor est passant alors que l'autre est bloquant (ils sont donc complémentaires, d'où l'appellation complementary). » [16]

11. A partir d'une centaine de milliers d'euros pour des techniques classiques et jusqu'à un million d'euros pour des techniques de pointe.

pour optimiser la transmission des signaux et le routage des fils entre les transistors. Une fois les masques ainsi conçus, ils sont envoyés à une fonderie qui va les fabriquer ainsi que les circuits résultants. La fonderie nécessitera plusieurs mois pour produire ces circuits full-custom. Les clients de la fonderie (*i.e.* les concepteurs du système) devront payer un coût non-récurrent très élevé, de l'ordre de plusieurs millions d'euros. Cependant, les performances en termes de délais d'exécution, de taille et d'énergie sont excellentes.

Cette technologie est actuellement exploitée uniquement pour les produits destinés à être fabriqués dans un très grand volume (au moins 100 000 pièces) ou pour des applications où la performance est critique (*e.g.* applications spatiales ou militaires).

Semi-custom : ASIC (Gate Array, Standard Cell)

Dans la technologie ASIC (Application Specific Integrated Circuit), les éléments de bas niveau du circuit sont pré-définis.

Dans la technologie *gate array*, les masques pour les niveaux transistors et portes logiques sont déjà construites, laissant la tâche aux concepteurs de les interconnecter correctement au niveau des couches de haut niveau.

Dans la technologie *standard cell*, des cellules formant des portions de masques de niveau portes logiques sont disponibles, prêtes à être interconnectées. Ces cellules constituent des morceaux de logiques comme une porte AND ou des combinaisons du type AND-OR-INV. Reste au concepteur la tâche d'assembler ces portions de masques en masques complets pour les couches intermédiaires et de les connecter au travers des couches de haut niveau.

La technologie ASIC est la technologie de support physique la plus populaire, car elle permet d'atteindre de très bonnes performances en termes de délais, taille et consommation d'énergie tout en ayant des coûts non-récurrents bien inférieurs à la technologie full-custom. Cependant, ces coûts restent élevés et le temps de fabrication est du même ordre que pour les circuits full-custom. Cette technologie ne convient pas, par exemple, au prototypage de systèmes.

PLD

Dans la technologie des *Programmable Logic Device* (PLD), toutes les couches du circuit sont déjà conçues. En fait, le circuit est déjà fabriqué et disponible à la vente, laissant à l'utilisateur du circuit le soin de fixer sa fonctionnalité [12]. En particulier, il est possible d'acheter le circuit avant de terminer la conception du système. Cette technologie de support physique permet donc le prototypage rapide d'un système. Le circuit PLD est ainsi un circuit « programmable », mais pas de la même façon qu'un processeur générique ou un DSP ne l'est. Il ne s'agit pas ici de stocker les instructions dans la mémoire de programme du circuit, mais plutôt de définir la fonctionnalité au niveau RTL. Cette spécification RTL sera synthétisée sous la forme d'une netlist de ressources et déchargée sur le circuit PLD afin de le configurer de façon à ce qu'il se comporte fonctionnellement comme le RTL. La phase de configuration consiste à

12. De la même façon que lors de l'achat d'un processeur générique déjà produit.

créer ou détruire des liens logiques entre les cellules de base de ces systèmes. Il existe deux principaux types de PLD, les PLD simples (PLA/PAL) et les PLD complexes.

Le type de PLD complexe aujourd'hui le plus répandu est sans doute la famille appelée FPGA, pour *Field Programmable Gate Array*. Les circuits FPGA offrent un degré de connexions entre les éléments de base très élevés et peuvent être reconfigurés un nombre indéfini de fois. Les cellules de base des FPGA sont composés des éléments suivants :

- des petites mémoires RAM à quelques bits d'adresse (généralement 4) et un bit de sortie. Ces mémoires sont appelées des *LookUp Tables* (LUT) ;
- des bistables permettant la sauvegarde de bits et donc l'introduction d'une composante d'*état* au sein du système.

Ces deux types d'éléments peuvent être exploités indépendamment ou ensembles en fonction du design RTL qui est déployé sur le FPGA.

En pratique, les FPGA sont donc les PLD permettant le déploiement des systèmes les plus complexes. Bien sûr, au sein même de la famille des FPGA, il existe de nombreuses différences, comme la surface (nombres de LUT), la fréquence maximale possible (influant sur les délais et le débit d'exécution), l'énergie consommée, etc. Les entreprises produisant des FPGA sont peu nombreuses, nous pouvons citer principalement *Xilinx* et *Altera*.

Les circuits FPGA sont généralement associés à la famille des *circuits reconfigurables*.

D'un point de vue métriques de performances, les PLD ont un coût non-récurrent et un temps de prototypage très faible vis-à-vis des autres technologies de support physique (il ne faut pas avoir de contact avec une fonderie mais juste acheter les unités de circuit désirées). Cependant, un système implémenté en FPGA peut avoir une taille bien plus grande que l'équivalent ASIC, un coût unitaire plus grand, consommera plus d'énergie et peut être plus lent. Néanmoins, les performances sont assez remarquables et évoluent rapidement, ce qui en fait des outils parfaits pour réaliser des prototypes de circuits qui seront ensuite produits avec une technologie comme l'ASIC.

Il est important de prendre conscience que les choix de technologie de calcul et les choix de technologie de support physique sont totalement indépendants et toutes les combinaisons sont possibles. La figure 3.5 illustre ceci en montrant que plus les technologies employées conduisent à des solutions généralistes, plus les métriques de flexibilité, maintenabilité, coûts non-récurrents et temps de conception sont bonnes. Plus les technologies sont dédiées, plus les métriques de puissance, délais d'exécution, tailles sont bonnes. Au niveau du coût unitaire du produit, les choix pour l'amélioration dépendent du volume de production visé. Le tableau 3.6 illustre certains systèmes résultant de choix de ces deux technologies.

Un *softcore* est un processeur générique déployé sur plate-forme reconfigurable (FPGA). Les fabricants de FPGA fournissent parfois le code source RTL de processeurs génériques destinés à être synthétisés et déployés sur leurs cibles FPGA. Nous pouvons notamment citer les softcores suivant : le *Xilinx MicroBlaze* et l'*Altera Nios*.

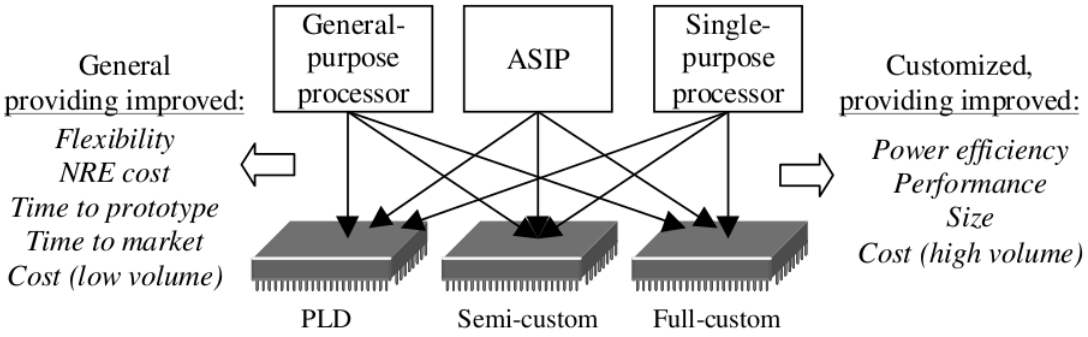


FIGURE 3.5 – L'orthogonalité des technologies de calcul et de support physique [VG02]. Toutes les combinaisons sont possibles.

Calcul \ Support	Full-custom	ASIC	PLD
Générique	Microprocesseur haut de gamme	Microprocesseur entrée de gamme	Softcore
ASIP	Applications militaires ou spatiales légères	DSP	DSP softcore
Dédié	Applications militaires ou spatiales lourdes	Puces dédiées (*e.g.* GPU, Northbridge, ...)	Application optimisée (*e.g.* JPEG2000)

FIGURE 3.6 – Tableau illustrant l'orthogonalité des technologies de calcul et de support physique et illustrant certains systèmes résultants de choix donnés.

3.3.3 Technologies de conception

Les technologies de conception font référence aux choix des outils et des niveaux d'abstraction qui seront utilisés lors de la conception du système. Il s'agit du processus qui consiste à convertir la spécification sous forme de cahier de charges en produit. Ce processus doit non seulement veiller à optimiser les métriques de performances mais doit également réaliser cette implémentation rapidement.

Differentes approches pour le design peuvent être définies. Ces approches sont détaillées précisément dans l'ouvrage de Gajski [GAGS09].

De façon plus précise, pour une technologie de calcul et de support physique données, il existe encore de nombreux choix point de vue technologies de conception : il s'agit de choisir le compilateur, l'assembleur, l'outil de synthèse haut niveau, l'outil de génération des masques, etc en fonction des besoins de tels outils. Ce besoin d'outil dépend fortement des technologies de calcul et de support physique choisies. Pour chacune de ces gammes d'outils, il existe de nombreuses variantes et même d'options parmi ces variantes (au sein des outils) qui permettent d'optimiser telle ou telle métrique.

Par exemple, il est possible d'optimiser de plusieurs façons le code cible produit

par un compilateur : en fonction du type d'optimisation choisie, le concepteur peut donner priorité à minimiser la taille du code ou à minimiser le temps d'exécution. Il s'agit à nouveau de critères pouvant être conflictuels et devant être choisis en fonction des contraintes du système à produire. Dans un exemple où la mémoire de programme est très limitée (typiquement pour un microcontrôleur), la minimisation de la taille du code produit peut constituer un avantage indéniable.

Il existe également une alternative à developper « soi-même » les différents composants du système. En effet, il est possible d'utiliser directement des composants de haut niveau pré-définis. Ces composants sont appelés IP, pour *Intellectual Properties*. Le choix entre développer le composant ou d'acquérir un IP [13] pour ce composant constitue une décision de haut niveau à part entière. Le choix parmi différents IP réalisant la même fonction mais ayant des impacts différents sur les performances fait également partie des décisions de conception haut niveau. L'avantage des IP en termes de métriques est indéniable [14], il permet de diminuer fortement le temps de développement, au détriment peut-être des coûts non-récurrents provoqués par l'achat du composant. En pratique, un IP se présente comme une description réutilisable d'un élément du système. Une analogie peut être faite avec les *librairies* du monde du *software engineering*.

3.3.4 Technologies d'implémentation

Les technologies d'implémentation constituent tous les choix ayant eu lieu pour convertir le cahier des charges en fonctionnalité effective et exécutable sur son unité logique (la technologie de calcul) et physique (la technologie de support physique).

Les choix de cette catégorie se distinguent des choix de conception : si la méthode de conception définit quels outils, quels niveaux d'abstraction et quels langages sont utilisés, les choix de technologies d'implémentation définissent la *façon* dont ceux-ci sont utilisés. Lorsque les choix de conception sont fixés, il reste des choix d'implémentation à fixer, notamment au niveau algorithmique. Citons à ce sujet une remarque de l'introduction de l'ouvrage de Cormen *et al* :

"*[...] les algorithmes, à l'instar des matériels informatiques, sont une technologie. Les performances globales du système dépendent autant des algorithmes que du matériel. Comme toutes les autres technologies informatiques, les algorithmes ne cessent de progresser.*"

En particulier, il peut s'agir du choix de l'algorithme à utiliser pour une fonctionnalité donnée. Dans le cas d'une transformée de Fourier par exemple, il existe de nombreux algorithmes aux caractéristiques de performances différentes.

D'un point de vue de plus haut niveau, le *modèle de calcul* choisi pour résoudre le problème influe également fortement sur la performance. Les algorithmes sont définis au sein d'un modèle de calcul qui fixe une série de paramètres : modèle de l'architecture (*e.g.* parallèle, séquentielle), d'accès à la mémoire (*e.g.* RAM, hiérarchique), types de calculs autorisés (*e.g.* comparaison, arithmétique), représentation des données (*e.g.*

13. Selon l'IP, il peut être acheté ou disponible gratuitement.
14. *« Il ne faut pas réinventer la roue. »*

nombres réels en virgule fixe ou flottante). Ces paramètres transcendent l'algorithme et ont une influence sur les bornes de complexité de certains problèmes [15].

Un algorithme implémenté dans un paradigme parallèle peut s'exécuter beaucoup plus rapidement (si la technologie de calcul sous-jacente supporte cette parallélisation, avec par exemple un processeur générique multi-coeur [16] ou un circuit dédié hautement parallèle). Cependant, la surface consommée par cet algorithme sera beaucoup plus grande que son équivalent séquentiel.

Le choix particulier d'utiliser un processeur générique multi-coeur est à mi-chemin entre la technologie de calcul et d'implémentation. En effet, le moteur logique doit supporter cette implémentation (plusieurs coeurs physiques copiés, une gestion de la mémoire distribuée ou commune, etc.) mais l'algorithme doit également être développé de façon à exploiter efficacement les différents coeurs du processeur.

Au sein du logiciel, l'utilisation particulière de structures de données (tableaux, listes chaînées, etc.) ou de structures de contrôle (récursivité, boucles, branchements, etc.) aura aussi son impact sur les performances (essentiellement délais et consommation d'énergie).

3.4 Conclusion

Au sein de ce chapitre, nous avons discuté des définitions possibles de systèmes embarqués et illustré le défi posé par l'optimisation des différentes métriques de performances. Ces métriques étant parfois conflictuelles, la notion de solution optimale n'a plus de sens et il est nécessaire de réaliser une solution de compromis. L'espace des solutions est défini par l'ensemble des choix de conception possible. Ce chapitre a également présenté certains des choix de conception haut niveau, qui sont critiques dans le dimensionnement du système et ont un impact sensible sur les différentes métriques de performances. Ces différents choix de conception étant relativement indépendants, l'espace des solutions possibles est gigantesque.

De ces constats, nous pouvons tirer trois conclusions intéressantes.

Premièrement, vue l'étendue des choix possibles et leurs impacts sur les performances du système, la méthode qui consiste à séparer de façon figée la partie *hardware* [17] de la partie *software* [18] d'un design n'est que peu applicable au design des systèmes embarqués. En effet, un système électronique est un objet contenant du matériel *et* du logiciel et ces deux aspects intéragissent étroitement entre eux pour réaliser la fonction du système. Les performances du système dépendent du matériel, du logiciel et de la combinaison faite entre les deux. L'approche qui consiste à séparer les deux aspects pour les concevoir est donc sous-optimale en termes d'efficacité. A l'inverse, la méthodologie du *co-design* propose d'unifier la conception des systèmes pour pouvoir considérer simultanément *hardware* et *software*. Il existe des technologies de conception de niveau système (comme *SystemC*) s'inscrivant dans cette méthodologie. La

15. Par exemple, le problème du tri dans un modèle par comparaison en exécution séquentielle est borné inférieurement par un nombre $\Omega(n \log n)$ de comparaisons.

16. L'algorithme sera alors découpé en plusieurs fils d'exécution appelés *thread*.

17. Conception des processeurs dédiés, choix des processeurs génériques, choix de la technologie de support physique, etc.

18. Ecriture des programmes pour les processeurs génériques.

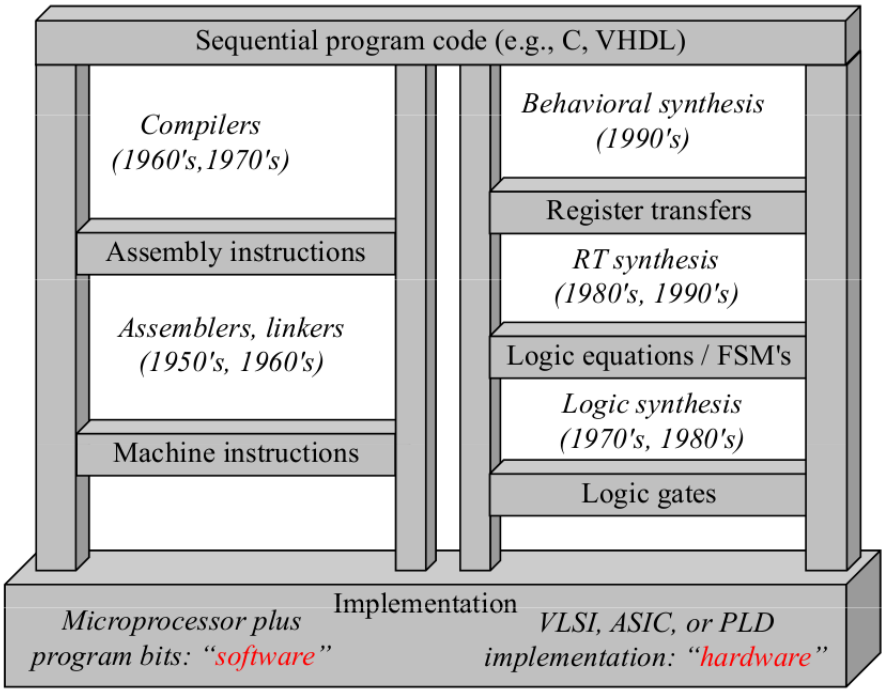


FIGURE 3.7 – L'échelle du co-design unifiant les flots de conception matériels et logiciels [VG02].

figure 3.7 illustre ces outils de haut niveau unifiant les flots de conception matériels et logiciels en les situant historiquement. Une citation de Vahid permet de résumer ce point :

" *The choice of hardware versus software for a particular function is simply a trade-off among various design metrics, like performance, power, size, NRE cost, and especially flexibility ; there is no fundamental difference between what hardware or software can implement.*"

Ensuite, vue la taille de l'espace des choix et la nature conflictuelle des critères de performances à optimiser, il est nécessaire d'établir des techniques efficaces d'exploration de ces choix en tenant compte des performances de façon à ne retenir que les solutions potentiellement intéressantes. En effet, le problème étant de nature combinatoire, le nombre de combinaisons de choix possibles explose rapidement avec l'augmentation du nombre de choix à effectuer. Une simple énumération exhaustive de toute les solutions est donc impraticable. Quand bien même nous aurions à disposition l'ensemble de toutes les solutions, il serait nécessaire de disposer d'outils pour choisir le meilleur compromis, en adéquation avec toutes les contraintes de performances du cahier des charges. C'est l'objet de la partie II de ce mémoire, en particulier au sein du chapitre 6.

Enfin, et il s'agit peut-être du message le plus essentiel de ce mémoire, il pourrait être intéressant de pouvoir évaluer à l'avance l'impact des choix sur les performances. Ceci non seulement pour aider le concepteur dans sa tâche à produire un système en adéquation avec les contraintes de performances du cahier des charges, mais aussi pour permettre l'exploration rapide des solutions de design évoquées au point précédent.

Dans l'état actuel, le concepteur se base sur sa propre expertise et évalue mentalement l'impact des choix. L'idée est de pouvoir disposer d'un modèle analytique et rationnel liant, à l'avance, l'information des choix effectués à l'information des coûts en termes de performances sur le système. De par la nature complexe du problème, ce lien est difficile à appréhender par le seul jugement humain. Ainsi, des techniques statistiques d'analyse peuvent être utilisées sur des cas d'étude afin de mettre en évidence la structure mathématique du lien entre choix et performances. L'objet de la partie III de ce mémoire est d'introduire l'une des méthodes possibles pour analyser un cas d'étude simple.

Chapitre 4

Perspectives industrielles

Ce chapitre propose une conclusion provisoire sur les méthodes industrielles de conception des systèmes embarqués et quelles sont les perspectives de celles-ci à l'avenir. En particulier, les difficultés liées à l'évaluation des performances sont mises en évidence ainsi que la nécessité de mettre en place des outils permettant la prédiction de telles performances. Ces perspectives sont enfin mises en vis-à-vis de certains travaux effectués en recherche.

4.1 Evaluation des performances

Le nombre de degrés de liberté offerts pour la conception de systèmes étant très grand, l'ensemble des choix possibles résultant en solutions différentes est gigantesque. Le concepteur, pour être en adéquation avec les contraintes de performances spécifiées dans le cahier des charges, doit tenter d'explorer intelligemment cet espace de design.

Cependant, la production d'un design de qualité prend un temps d'ingénierie certain, incluant les temps de réflexions, d'écritures, de simulations et de synthèses. En particulier, la synthèse d'un système considérable décrit en RTL peut prendre entre plusieurs heures et plusieurs jours. Or des métriques telles que le temps de mise sur le marché sont également à prendre en compte dans la conception d'un système embarqué.

L'outil de synthèse utilisé pour les systèmes RTL donne au concepteur une évaluation plus ou moins précise des différentes métriques de performances (dimensionnement de la fréquence d'horloge, surface en nombre d'équivalent portes, puissance, etc.). Cependant, l'outil de synthèse ne peut donner cette évaluation qu'une fois le design terminé et synthétisé. Pour évaluer les performances d'un système, le concepteur doit donc concevoir le système avec tous ses détails d'implémentations et payer le temps de synthèse associé. En toute logique, les choix de conception pour ce design, tels que l'architecture de calcul ou le support physique, sont donc déjà posés. Si le concepteur désire explorer un tant soit peu l'espace de design, il doit tester les combinaisons possibles et donc perdre du temps à réaliser une description RTL complète des solutions pouvant être sous-optimales : le temps passé à la conception du système sera donc égal au temps pour mettre en oeuvre une solution multiplié par le nombre de solutions explorées.

Dans son exploration de l'espace de design, le concepteur averti peut éventuellement s'aider de modèles empiriques simples [1], définis par lui-même ou ses prédécesseurs. Dans les deux cas, ces modèles sont issus de l'expérience des concepteurs. A chaque solution mise en oeuvre par les concepteurs, ceux-ci sont en mesure de compléter leurs modèles. Cependant, cette méthode est très élémentaire et il peut être nécessaire de rationaliser le processus.

L'évaluation des performances fournie par l'outil de synthèse n'est pas en parfaite adéquation avec la réalité. Comme toujours en sciences, lorsque nous effectuons des mesures sur un objet, nous nous confrontons à un certain modèle de la réalité. Le modèle utilisé par les outils de synthèse RTL a une certaine précision donnée.

Partant de ce constat, rien ne nous empêche d'élever le niveau d'abstraction de cette évaluation de performances en définissant des modèles moins précis que ceux fournis par les outils de synthèse mais permettant d'évaluer plus rapidement les solutions de design, c'est-à-dire sans passer par l'outil de synthèse et les détails d'implémentation du niveau RTL. Cette évaluation réalisée *a priori* représenterait un gain de temps considérable pour le concepteur industriel.

4.2 Prédiction des performances

La démarche du concepteur comme décrite en 4.1 est essentiellement itérative. Le concepteur est limité par le temps dont il dispose et sa propre capacité à imaginer des solutions de design toujours plus efficace. Ceci oblige ce dernier à procéder par essais et erreurs. Cette technique n'étant pas des plus efficaces, il pourrait être utile de tenter de prédire l'impact des choix de design de haut niveau sur les performances.

L'expert, dans un contexte industriel, n'a que peu de temps pour étudier le lien théorique entre choix et performances et perçoit généralement un nuage de points peu précis plutôt qu'un véritable lien fonctionnel entre ces deux types d'informations. La raison pour laquelle ce lien est difficile à appréhender est due à l'immense taille de l'espace de design, qui dépasse largement la perception humaine. Le but des recherches dans le cadre de la prédiction de performances est de clarifier ce lien.

Pour cela, il est nécessaire de déterminer une approche systématique pour quantifier et évaluer les différentes métriques de performances. La pluralité de ces métriques conduira à adopter, pour ce faire, des modèles *multicritères*.

4.3 Travaux de recherche

Pour résoudre ce problème, plusieurs approches complémentaires sont adoptées dans le monde de la recherche.

La première approche consiste à élever le niveau d'abstraction de la conception, essentiellement en introduisant des outils, langages et méthodologies de niveau système. Le *co-design*, comme présenté en 3.4, adopte cette approche. En particulier, nous pouvons citer :

1. Usuellement, des feuilles de calculs de tableur sont utilisées en industrie pour visualiser rapidement les impacts des choix sur les performances.

- le profil MARTE et l'outil GASPARD2, qui proposent un flot de conception *system-level* ;
- SystemC ;
- les flots émergeants « C vers RTL », qui consistent à traduire des programmes écrits en langage C en circuits au niveau RTL directement synthétisables à l'aide d'outils classiques.

La seconde approche consiste à créer des outils auxiliaires aux outils de conception présentés ci-dessus, ceux-ci se focalisant sur la prédiction de performances des mêmes systèmes. Nous pouvons citer l'outil NESSIE.

La partie II de ce mémoire présentera essentiellement deux travaux effectués en recherche allant dans le sens de cette conclusion provisoire.

Deuxième partie

Travaux de recherche pour la conception de systèmes embarqués

Chapitre 5

Conception de haut niveau

La partie I a introduit les problèmes auxquels l'industrie est confrontée en matière de conception de systèmes embarqués, en particulier vis-à-vis du respect du cahier des charges en termes de performances. La présente partie II vise à introduire certains travaux de recherche effectués afin de trouver des solutions élégantes et efficaces à ces problèmes. Cette partie est structurée de la façon suivante : le chapitre 5 introduit l'idée de l'utilisation d'outils de prédiction de performances au sein d'un flot de conception de haut niveau. Le chapitre 6 présente l'outil NESSIE développé dans l'équipe *BEAMS digital* à l'Université Libre de Bruxelles. Cet outil permet une exploration des choix de conception, tant au niveau matériel que logiciel, en se basant sur des modèles multicritères des performances des composants du système. Le chapitre 7 présente l'outil GASPARD2 développé au sein de l'équipe *DaRT* à l'*INRIA Lille Nord Europe*. Ce dernier outil permet la synthèse de systèmes spécifiés par un modèle de haut niveau, conforme au profil MARTE de spécification des systèmes embarqués et temps réel. Enfin, le chapitre 8 conclue sur ces différents travaux de recherche et met en évidence les lacunes de ces derniers au niveau de la modélisation des composants de base des systèmes embarqués.

Ce chapitre présente les bases de la conception de systèmes embarqués à haut niveau (*system level*) et des outils imaginés dans le monde de la recherche pour l'assister. En premier lieu, le lien entre les choix de design et les performances du système seront rappelés et la méthodologie de conception en industrie sera résumée. Une automatisation partielle de la démarche est ensuite proposée pour aboutir à un modèle de la prédiction des performances. Enfin, la démarche est résumée en mettant en évidence les différents outils nécessaires à sa mise en place.

5.1 Degrés de liberté et performances résultantes

Comme présenté au chapitre 3, la *fonctionnalité* assurée par le système embarqué est fixée par le cahier des charges, mais il existe un ensemble de choix possibles, indépendants de celle-ci, pour la conception de ce système. Cet ensemble constitue les *degrés de liberté* du design. Chaque choix effectué fixe un de ces degrés de liberté et influe sur les performances du système produit. Ces choix ne sont pas tout à fait « libres » dans la mesure où le cahier des charges spécifie également l'ensemble de

contraintes *non-fonctionnelles* sur les performances. Ces contraintes peuvent être, par exemple :

– le circuit ne doit pas dissiper une puissance de plus de 3 watts,

– le circuit doit calculer 50 transformées de Fourier discrètes par seconde.

Par conséquent, les concepteurs du système vont fixer les degrés de liberté du design en réalisant des choix de façon à tenter de respecter les contraintes de performances. Une fois les choix réalisés, les concepteurs peuvent effectuer une implémentation effective du système et vérifier l'adéquation des performances du système produit avec les contraintes non-fonctionnelles du cahier des charges [1].

Ainsi énoncé, le problème de la conception d'un système embarqué revient à trouver, à un niveau d'abstraction donné, comment *fixer les choix* de façon à obtenir des résultats de performances *« optimaux »*. Il est cependant difficile de prévoir la performance d'un tel système avant de l'avoir entièrement conçu. Les concepteurs vont donc procéder itérativement : partant d'une idée de départ, ils fixent les choix [2], ensuite procèdent au design et finalement évaluent les performances du système produit. Si les performances sont satisfaisantes par rapport aux contraintes du cahier des charges, le travail est terminé et le concepteur peut passer au niveau d'abstraction inférieur [3]. Sinon, il faut recommencer l'itération en changeant les choix de départ et en recommençant la conception, ceci jusqu'à satisfaction des contraintes de performances. La figure 5.1 illustre ce principe de développement.

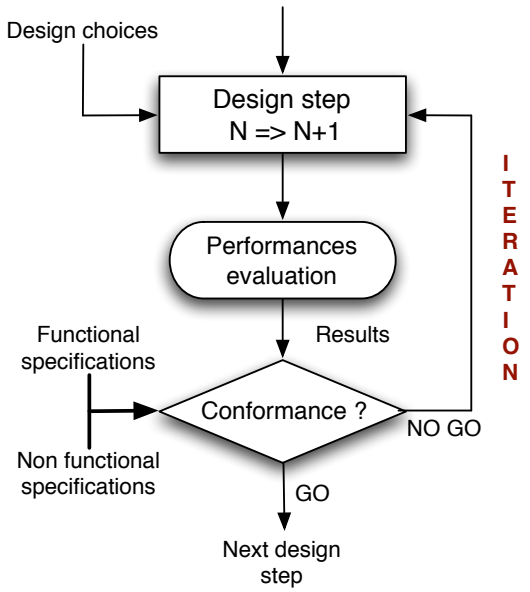


FIGURE 5.1 – Représentation de la conception itérative de système [Van09].

1. La vérification des contraintes fonctionnelles (*i.e.* vérifier que le système réalise bien la fonction pour laquelle il a été conçu, sans aucune considération pour les performances) sort du cadre de ce mémoire.
2. En se basant sur leur expérience de ce qui peut s'avérer efficace dans tel ou tel cas.
3. Essentiellement au travers d'un processus de synthèse.

5.2 La prédiction de performances

Avec l'augmentation de la complexité des systèmes à concevoir, cette façon de développer, un peu « en aveugle » par rapport au lien entre choix de design et performances, peut souffrir d'un très grand nombre d'itérations. Le temps pour chacune de ces itérations étant non négligeable, le temps de prototypage du système est donc très long et la conception devient un processus pouvant être très fastidieux et répétitif. Les concepteurs de tels systèmes, notamment dans le monde de la recherche appliquée, ont imaginé comment l'outil informatique pourrait les aider à réaliser ces étapes plus rapidement. Le but étant de minimiser le nombre d'itérations *réelles* du développement du système, les concepteurs vont construire un outil d'évaluation automatique qui donnera une estimation *a priori* des performances du système pour un ensemble de choix donnés, sans devoir réaliser concrètement l'implémentation de ces choix. Ceci permettra donc aux concepteurs du système de disposer d'un « avis » sur les choix à réaliser sur le design. La figure 5.2 illustre un tel outil, schématisé comme une routine qui prend en entrée les choix de design fixant les degrés de liberté et qui, en fonction de ceux-ci, génère en sortie la performance du système. La performance générée est représentée sous la forme d'un rapport faisant intervenir de nombreux critères, comme les délais d'exécution, la surface occupée et l'énergie consommée.

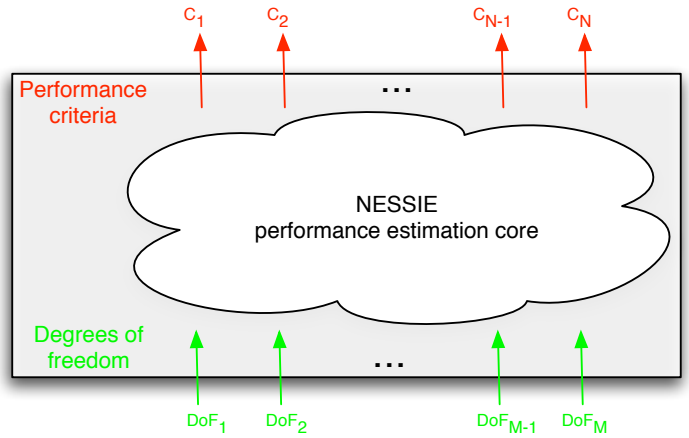


FIGURE 5.2 – Outil d'estimation des performances d'un système [Van09].

Il est important de remarquer qu'étant donné que le système n'est pas conçu réellement mais est émulé par un outil automatique, les résultats de performances obtenus seront seulement *approchés* par rapport au système tel qu'il le serait s'il avait été effectivement réalisé. Ceci est expliqué par le fait que l'outil se base sur un *modèle* du système plus ou moins précis. Le but de cet outil serait donc d'accélérer la conception du système en supprimant des itérations inutiles résultantes de mauvais choix des concepteurs humains.

Cet outil d'évaluation des performances nécessitera donc une importante étape de modélisation des systèmes. Plus cette modélisation sera fine et proche de la réalité, plus les résultats de l'estimateur seront proches des performances du système réel. Les problèmes liés au choix des modèles seront introduits plus loin dans ce mémoire.

D'un autre côté, le concepteur humain a besoin d'assistance pour être guidé dans l'assignation des choix relatifs aux degrés de liberté du design. Ces choix ne peuvent

pas être totalement arbitraires ou aléatoires, et la seule expertise des concepteurs dans ce domaine peut s'avérer insuffisante pour aboutir aux performances optimales du système. De plus, l'espace des choix possibles est très grand et nécessite l'aide des ordinateurs pour l'explorer efficacement. L'outil requis ici est *l'aide à la décision multicritère*, une branche importante de la recherche opérationnelle. Le but est d'obtenir une procédure qui, sur base des résultats $\langle Choix, Performances \rangle$ des différentes « itérations virtuelles » déjà calculées par l'estimateur, fournit les choix à effectuer pour l'itération suivante, et ce afin de tendre vers une solution dite « optimale ». Cet outil utiliserait l'estimateur comme une boîte noire qui en entrée reçoit les choix et en sortie renvoie l'estimation de performances. La figure 5.3 schématise une telle procédure et son lien avec l'estimateur de performances.

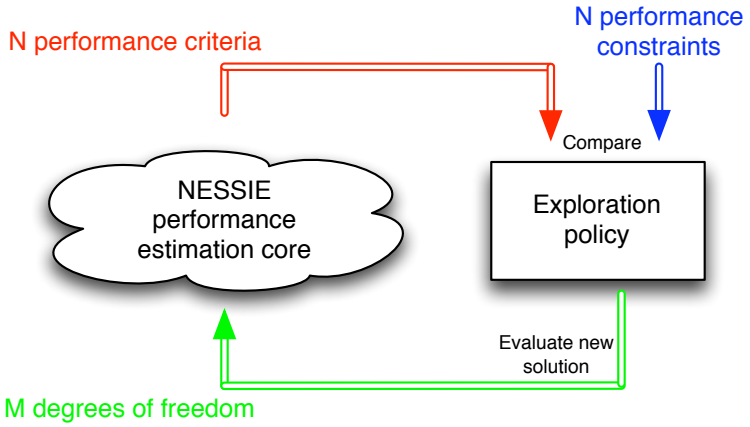


FIGURE 5.3 – Représentation de l'estimateur de performances et de l'outil d'exploration des choix possibles pour la conception du système [Van09].

5.3 Schéma final et éléments du système de prédiction

Ce schéma complet, représenté par la figure 5.3, permet donc de guider efficacement les choix des concepteurs au travers du design de systèmes embarqués. En résumé, cette approche nécessite trois éléments fondamentaux :

- Une méthodologie de modélisation réaliste des performances des systèmes, appliquée dans le but de fournir une bibliothèque de modèles des composants d'un système (*cf.* le diagramme en Y de Gajski introduit à la section 2.4).
- Une méthode pour évaluer rapidement les performances d'un système complet en fonction des choix des composants de ce système, exploitant les modèles de ces composants présents dans la bibliothèque. Cette méthode est schématisée par la boîte noire de la figure 5.2.
- Une méthode d'exploration de l'espace de solutions de choix de design, schématisée par la boîte rectangulaire de la figure 5.3.

Trouver ces méthodes et concevoir les outils qui les implémentent est précisément l'objet de nombreuses équipes de recherche. Dans cette deuxième partie, nous introduirons les travaux des équipes de recherche *Inria Lille DarT* et *ULB Beams Digital*. On peut notamment citer :

- Le profil MARTE, un standard de modélisation haut niveau de systèmes embarqués temps réel, basé sur le langage UML.
- L'outil GASPARD2, respectant le profil MARTE, permet de concevoir un système à haut niveau d'abstraction et la synthèse de ce système vers des cibles et langages de niveaux d'abstraction inférieurs.
- L'outil NESSIE, permettant la spécification d'un système à haut niveau d'abstraction, l'exploration de solutions et implémente une routine d'évaluation de l'impact des choix sur les performances.

Les chapitres suivants décriront ces différents outils, leurs buts, leur fonctionnement et leurs éventuelles failles.

Chapitre 6

Analyse multicritère et l'outil Nessie

Ce chapitre présente le fonctionnement de l'outil Nessie, ses buts, son fonctionnement et ses limitations.

Il est cependant nécessaire d'introduire d'abord l'analyse multicritère.

6.1 Aide à la décision multicritère

Cette section décrit plus en détail l'analyse multicritère et l'aide qu'elle peut apporter à la conception des systèmes embarqués.

6.1.1 Nature du problème

Le problème considéré, à savoir fixer les choix de design en vue d'optimiser les performances, est de nature combinatoire. En effet, les choix que sont en mesure d'effectuer les concepteurs sont très grands, mais pas infinis. Par conséquent, l'espace des solutions possibles est représentable par une combinaison de choix dans des ensembles finis.

En outre, le problème demande d'optimiser *plusieurs* critères qui sont généralement *conflictuels*. En effet, lorsque l'optimisation pousse les critères à leur limite, il n'est plus possible d'améliorer un critère sans en détériorer un autre.

Prenons un exemple simple en restreignant notre analyse à deux critères : le temps d'exécution moyen du système et le coût par pièce de celui-ci. Il est évident qu'*à partir d'une certaine limite*, diminuer le coût du système aura un impact négatif sur sa performance et en particulier sur son temps d'exécution. De façon symétrique, la réalisation de certains choix pour diminuer le temps d'exécution conduira à une augmentation du coût unitaire du système.

Etant donnée la nature conflictuelle de ces critères, il est impossible de trouver une solution qui optimise tous les critères simultanément. En fait, la notion de *solution optimale* du monde unicritère n'a pas de sens en analyse multicritère. Il est nécessaire de trouver des solutions de compromis.

6.1.2 Formalisation du problème

Une modélisation plus formelle du problème peut se faire de la façon suivante : le concepteur peut influer sur une série de degré de libertés modélisés par une variable x. Ces choix de design influeront sur les métriques des performances du système représentés par une variable y.

La variable x, dite *d'entrée*, prend ses valeurs dans un espace fini qui est l'ensemble des solutions de design possible. Cet ensemble de solutions est le résultat d'une série de choix. Chaque choix conditionne un degré de liberté x_i de l'ensemble $x_1, x_2, ..., x_n$. Un degré de liberté x_i prend sa valeur dans un espace de choix scalaire et fini. La variable $x = \langle x_1, x_2, ..., x_n \rangle$ est donc une variable vectorielle.

La variable y, dite *de sortie*, voit sa valeur conditionnée par la valeur de la variable x et est également vectorielle. Chaque métrique de design $y_1, y_2, ..., y_m$ prend sa valeur dans un espace unidimensionnel continu. La variable $y = \langle y_1, y_2, ..., y_m \rangle$ est donc également une variable vectorielle.

Le lien entre x et y (entre les choix de design et les métriques de performances) est modélisé par une fonction $f(.)$ telle que :

$$y = f(x)$$

Ou, de façon plus explicite :

$$\begin{cases} y_1 &= f_1(x_1, x_2, ..., x_n) \\ y_2 &= f_2(x_1, x_2, ..., x_n) \\ ... & \\ y_m &= f_m(x_1, x_2, ..., x_n) \end{cases}$$

L'étude et le calcul de ce lien $y = f(x)$ au niveau « système » font l'objet d'outils comme NESSIE, au sein de sa procédure *« NESSIE performance estimation core »* représenté par la boîte en forme de nuage sur la figure 5.3. L'étude de ce lien au niveau des composants de base du système est l'objet de la partie III de ce mémoire.

Le but du concepteur est d'*optimiser* les différentes métriques de performances, c'est-à-dire maximiser ou minimiser (selon le cas) les variables $y_1, y_2, ..., y_m$. Le problème de l'analyse multicritère consiste à trouver la ou les valeurs de x pour lesquelles le y relié est « le plus optimal », ou plutôt représente la « meilleure » solution de compromis pour l'optimisation des différentes composantes du vecteur y. L'aide à la décision multicritère consiste à donner des algorithmes d'exploration efficace de l'espace des valeurs de x afin de se rapprocher de ce meilleur compromis. C'est le but de la routine représentée par un rectangle sur la figure 5.3.

En résumé, comme schématisé sur la figure 5.3, l'optimisation de systèmes à multiples critères nécessite deux routines relativement indépendantes :

- une routine d'exploration efficace de solutions, qui choisit des valeurs de x et détermine l'intérêt, du point de vue optimisation multicritère, de la valeur de y reliée ;
- une routine de prédiction de performances, qui puisse associer rapidement les choix x à leurs coûts y, en modélisant le lien $f(.)$.

6.1.3 Solutions dominées et frontière Paréto-optimale

Malgré l'inexistence de solution optimale, il existe des « bonnes » et des « moins bonnes » solutions. En effet, il convient de distinguer les *solutions dominantes* et les *solutions dominées*.

- Les solutions dominées sont les solutions telles qu'il est encore possible d'améliorer au moins un critère sans détériorer les autres.
- Les solutions dominantes sont les solutions telles qu'il n'est plus possible d'améliorer un critère sans en détériorer un autre.

Pour trouver les meilleures solutions de compromis, il est nécessaire d'éliminer les alternatives dominées de l'espace de solutions. Ce procédé permet de déterminer la frontière Paréto-optimale [14], qui est l'ensemble des solutions dominantes. La figure 6.1 illustre un exemple de cette frontière.

Pour déterminer cette frontière, il existe deux types de méthodes :

- les méthodes exactes, qui sont coûteuses en temps de calcul,
- les méthodes approchées, qui sont rapides mais fournissent une valeur approchée de la frontière.

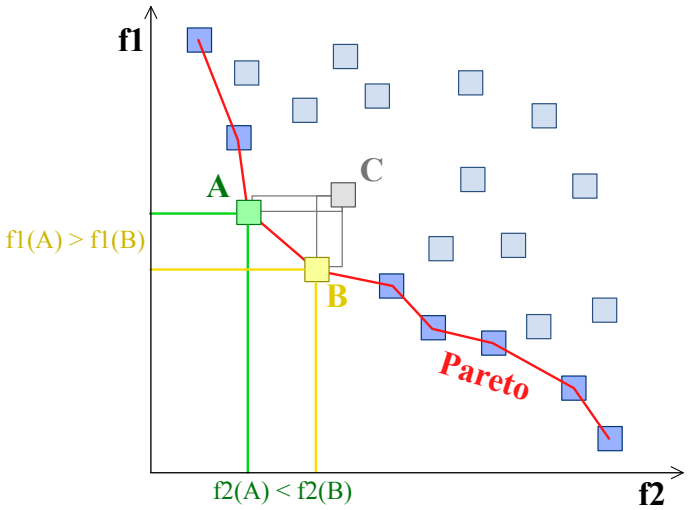


FIGURE 6.1 – Frontière Paréto-optimale dans un espace de solutions à deux critères [14], $f1$ et $f2$. La solution C est dominée par les solutions A et B qui sont sur la frontière.

6.1.4 Méta-heuristiques d'exploration

Ce problème combinatoire possède un espace de solutions très grand. En effet, chaque choix de design possible conduit à une solution située dans l'espace. Pour connaître la situation d'une solution par rapport aux différents critères, il faut lancer la routine d'évaluation de performances (qui sera par ailleurs décrite dans la section 6.2). Les solutions dominantes étant inconnues a priori, il est nécessaire d'explorer l'espace des solutions pour les identifier. Etant donnée la taille de l'espace de design, une exploration exhaustive n'est pas viable : il faut donc implémenter des méta-heuristiques

pour explorer le plus rapidement possible l'espace des solutions en vue d'approcher la frontière Paréto-optimale.

Une fois la frontière de Paréto approchée, il faut « choisir son point » [1] sur cette frontière. Il existe de nombreuses méthodes d'élection de solutions dans la littérature du multicritère. Citons à titre indicatif les méthodes suivantes [FGE05] :

- AHP, étudiée par Saaty en 1971 ;
- PROMETHEE-GAIA, étudiée par Brans et Vincke en 1985 ;
- MAUT, étudiée par Fishburn en 1970 ainsi que par Keeney et Raiffa en 1976 ;
- ELECTRE, étudiée par Roy en 1968.

6.2 L'outil NESSIE

6.2.1 Introduction

Les chapitres précédents ont montré que la conception d'un système embarqué est caractérisée par la définition conjointe du matériel et du logiciel qui sera exécuté sur ce matériel. La conception de logiciel se fait par *programmation* et consiste à écrire dans un langage de programmation, comme le C, les programmes qui seront exécutés sur les différents processeurs du système. La conception de matériel consiste à écrire dans un langage de *description de hardware*, comme le VHDL, la spécification du matériel. Ces processus se situent en amont d'une chaîne d'outils automatiques destinés à générer le système final : compilation de programmes et synthèse de matériel en l'occurrence. La figure 6.2 illustre ces processus.

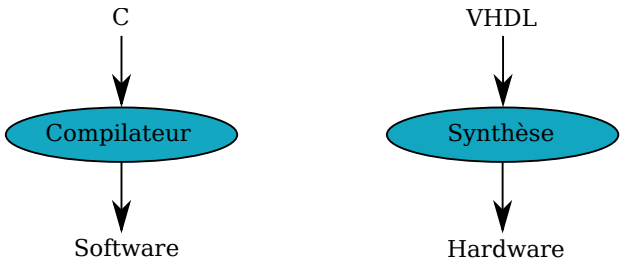


FIGURE 6.2 – Conception d'un système embarqué : aspects matériels et logiciels.

Comme cela a été illustré, cette façon de procéder pour concevoir des systèmes embarqués correspond à l'état de l'art actuel. Cependant, cette méthode souffre de certains désavantages, comme décrits au sein du chapitre 3 : les choix matériels et les choix logiciels, *mais également certaines de leurs combinaisons*, influent sur les performances du système final. Ainsi, la séparation entre les équipes « matériel » et les équipe « logiciel » en milieu industriel pose des problèmes essentiels de communication et influent négativement sur les performances du système ainsi produit. De plus, les systèmes actuels deviennent très complexes et l'écriture de code C ou VHDL, aussi abstraite soit-elle par rapport au système final, est loin d'être une sinécure.

1. Ceci correspond au choix final d'implémentation pour le système.

En constatant ces problèmes, l'idée de réaliser une couche d'abstraction supplémentaire pour le design des systèmes embarqués est apparue. Cette couche, appelée *« system-level design »*, oblige la conception conjointe de logiciel et de matériel (co-design).

L'outil NESSIE [3], développé dans l'équipe « digital » du service *BEAMS*[2] de l'Université Libre de Bruxelles, est un estimateur de performances[3] situé sur cette couche d'abstraction de très haut niveau.

L'idée de base de NESSIE est très simple : il prend en entrée la modélisation de plusieurs variantes du système à concevoir[4], autant au niveau logiciel que matériel, et cherche à trouver quelle est la *meilleure combinaison* en termes de performances du système final. Cette approche permet de guider tous les concepteurs du système, qu'ils soient responsables du logiciel ou du matériel, dans une même direction en adoptant des solutions estimées comme de bons compromis en termes d'optimisation.

6.2.2 Domaine d'application

> *"D'après la loi de Amdahl, il faut optimiser ce qui est exécuté le plus fréquemment."*
>
> **Pierre Mathys, 2010**

Bien que l'outil NESSIE puisse être utilisé pour résoudre et optimiser de nombreux problèmes de natures différentes, il a été développé dans le contexte de la conception d'un certain type particulier d'applications embarquées. Dans cette sous-section (et uniquement dans celle-ci), nous ferons abstraction de la différence entre matériel et logiciel et utiliserons plutôt la notion d'« activité », afin d'expliquer quel est le domaine où s'applique principalement NESSIE.

Dans de nombreux systèmes embarqués, l'exécution est en général partitionnée en deux phases :

- La phase de contrôle, qui représente en général 90% de la taille de l'activité mais dans laquelle le système reste en moyenne 10% du temps. Il s'agit de nombreux branchements conditionnels pour déterminer le comportement du système.
- La phase de traitement intensif des données, qui représente environ 10% de la taille de l'activité mais dans laquelle le système reste pendant 90% du temps. Il s'agit de traitements répétitifs comprenant de nombreuses itérations.

La partie qu'il faut effectivement optimiser est celle dans laquelle le système reste le plus longtemps, c'est-à-dire la phase de traitement de données. Ces types d'applications (manipulations numériques, traitement du signal, compressions, chiffrements, etc.) sont les systèmes à évaluer et optimiser dans NESSIE.

2. Bio-, Electro- And Mechanical Systems [4].
3. Comme présenté en 5.1.
4. Ces variantes constituent les degrés de liberté du design, comme introduit en 5.1.

6.2.3 Modélisation du système

Au sein de NESSIE, la modélisation d'un système est séparée en deux composantes essentielles :

- la fonctionnalité,
- la plate-forme.

Il est important de remarquer que ces deux notions sont abstraites par rapport à la distinction matériel-logiciel. La fonctionnalité est tout à fait indépendante de la plate-forme. Elle décrit l'architecture distribuée de l'application à développer, c'est-à-dire à quel point celle-ci est *parallélisable*. La plate-forme explicite les ressources disponibles pour implémenter la fonctionnalité ainsi que le lien entre les ressources (*i.e.* leur capacité de communication).

Comme présenté en 3.3, une même fonctionnalité peut s'exécuter sur des plates-formes différentes :

- sur un processeur, choix « logiciel »,
- sur un circuit dédié, choix « matériel »,
- etc.

De la même façon, une même plate-forme peut exécuter des fonctionnalités différentes. De plus, une fonctionnalité peut être décrite de plusieurs façons différentes, indépendamment de la plate-forme :

- par un algorithme purement séquentiel,
- par un algorithme distribué,
- par un algorithme avec différentes composantes, certaines parallèles et d'autres séquentielles,
- etc.

Une description plus complète des alternatives disponibles est proposée à la section 3.3.

Modélisation de la plate-forme

Une plate-forme est définie récursivement de la façon suivante :

- Cas de base : la plate-forme est décrite par un *listing* des *fonctionnalités élémentaires* qu'elle est capable d'implémenter. Ce listing est fourni avec, pour chaque *fonctionnalité élementaire* implémentable, le coût engendré en termes de performances.
- Cas inductif : la plate-forme est décrite par un *graphe des ressources*.

Un graphe des ressources est un formalisme défini par le triplet $\langle B, P, L \rangle$ où :

- B : un ensemble fini de *blocs*.
- $P : B \mapsto \mathbb{N}$: une fonction qui indique, pour chaque bloc, le nombre de *ports* de communication dont il dispose.
- $L \subseteq B \times B$: un ensemble fini de *liens logiques* entre les blocs, représentés par des paires de blocs.

Notons que le nombre de liens logiques par bloc doit être inférieur ou égal à son nombre de ports.

Chaque bloc du graphe est ensuite défini récursivement par une autre plate-forme. Ce système permet d'obtenir une représentation *hiérarchique* de la plate-forme. De plus, cette modélisation sous forme de graphe permet de lier facilement des blocs matériels concrets aux blocs abstraits. La figure 6.3 illustre un exemple de graphe des ressources.

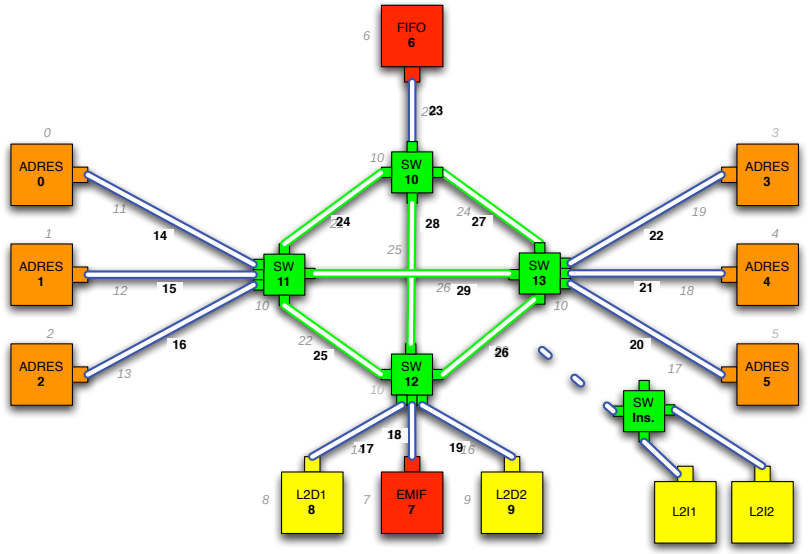


FIGURE 6.3 – Modélisation d'une plate-forme multiprocesseur à l'aide d'un graphe des ressources [Van09].

Modélisation de la fonctionnalité

Une fonctionnalité est également définie récursivement :

- Cas de base : la fonctionnalité est décrite par le nom d'une *fonctionnalité élémentaire*, en lien avec le cas de base des plates-formes.
- Cas inductif : la fonctionnalité est décrite par un Réseau de Petri.

Un réseau de Petri [15] est un formalisme défini par le 5-uplet $\langle P, T, A, M_0, W \rangle$ où :

- P : un ensemble fini de *places*.

- T : un ensemble fini de *transitions*
- $A \subseteq (P \times T) \cup (T \times P)$: un ensemble fini d'*arcs*, reliant les places aux transitions et les transitions aux places.
- $M_0 : P \mapsto \mathbb{N}$: le *marquage initial*, une fonction qui indique le nombre de jetons par place avant l'exécution du Réseau de Petri.
- $W : A \mapsto \mathbb{N}^+$: le nombre de *jetons consommés* ou *produits* au travers de chacun des arcs.

Une place représente une certaine tâche à accomplir. Comme dans le cas de la plate-forme, chaque place définit récursivement une autre fonctionnalité, c'est-à-dire soit une fonctionnalité élémentaire, soit un autre Réseau de Petri.

L'exécution d'un réseau de Petri se déroule par *distribution de jetons* entre places et transitions. Un *jeton* modélise l'utilisation d'une ressource pour l'exécution d'une place. Les transitions permettent de synchroniser l'exécution des différentes places.

La représentation de la fonctionnalité sous la forme d'un Réseau de Petri hiérarchique permet de représenter les systèmes distribués, leurs différents degrés de parallélisme ainsi que la dépendance entre les tâches de ces systèmes. La figure 6.4 illustre un exemple de réseau de Petri.

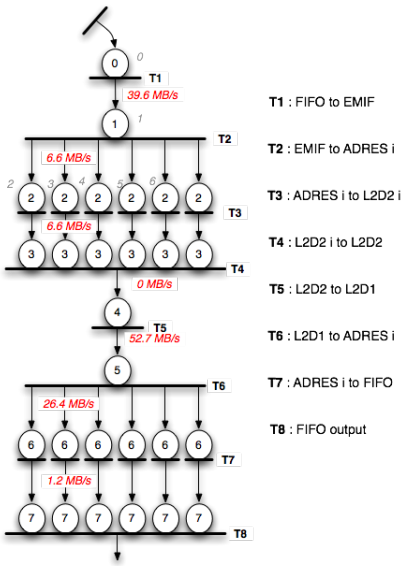


FIGURE 6.4 – Modélisation d'un décodeur vidéo temps-réel à l'aide d'un réseau de Petri [Van09].

6.2.4 Le mapping

A partir de la représentation du système comme présenté en 6.2.3, NESSIE va tenter de relier la fonctionnalité sur la plate-forme. Essentiellement, il va exécuter un algorithme de *mapping* dont le but est d'assigner les places du Réseau de Petri aux blocs du graphe des ressources. Cet algorithme est schématisé par la figure 6.5.

Le résultat de ce mapping, s'il a réussi, permet d'évaluer facilement les performances du système sur base du modèle des caractéristiques des primitives données en entrée.

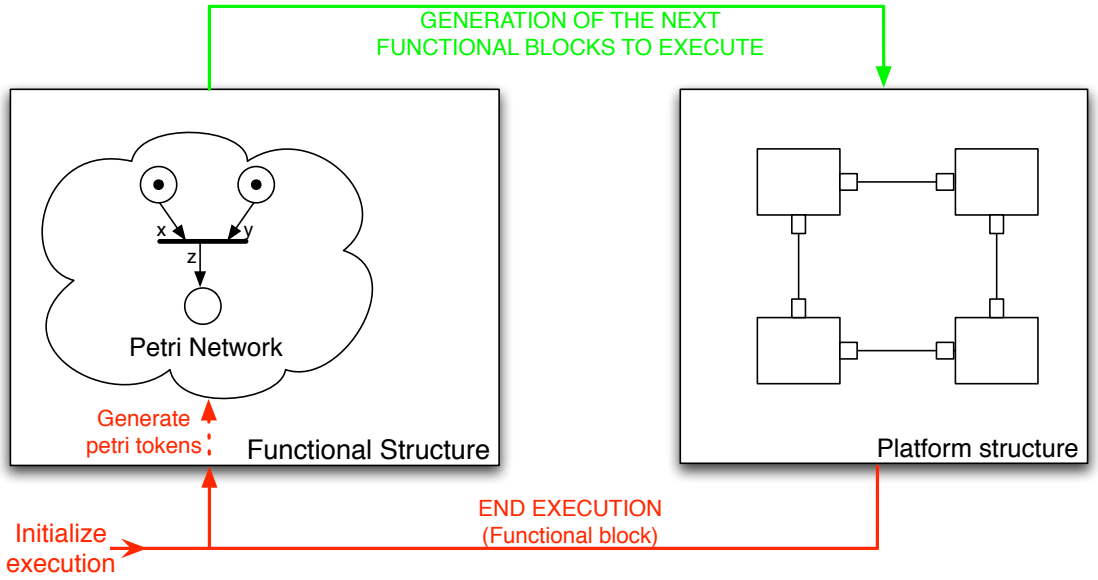


FIGURE 6.5 – Mapping entre la fonctionnalité et la plate-forme [Van09].

6.2.5 Routine d'exploration des solutions

Sur base des résultats du mapping, le système peut soit produire un *output* pour l'utilisateur, soit tenter d'autres combinaisons. Il incombe à la routine d'exploration des choix de choisir quelle combinaison est à tester pour se rapprocher de l'optimalité. Une fois les choix effectués, NESSIE relance son moteur de mapping pour à nouveau évaluer les performances du système. Cette démarche est appliquée itérativement jusqu'à satisfaction des contraintes.

6.2.6 Format des entrées/sorties

NESSIE communique avec le monde extérieur au travers de fichiers XML. En effet, le système modélisé comme en 6.2.3, qui forme l'entrée de l'outil, doit être décrit avec le formalisme d'XML. Les résultats produits sont également dans ce format.

6.2.7 Etat actuel de l'outil et idées de développements futurs

L'outil NESSIE a été conçu par Alexis Vander Biest dans le cadre de sa thèse de doctorat [Van09]. Aliénor Richard, au sein de sa propre thèse, a contribué à la validation de NESSIE sur deux cas industriels réels [Ric10].

Dans l'état actuel des choses, NESSIE souffre néanmoins de certains défauts notables :

- La routine d'exploration des solutions est implémentée pour le moment de façon à tester *toutes* les possibilités de design. Cette solution est la moins optimale qui existe et pose des problèmes de latence quand la taille des entrées augmente. Or la taille des systèmes réels, tels que veulent en produire les industries, est beaucoup plus grande que les cas d'études réalisés en recherche. Il serait donc nécessaire d'implémenter des algorithmes ou des heuristiques d'exploration efficace de l'ensemble des solutions possibles de design. L'analyse multicritère peut s'avérer utile dans ce cas, et un doctorant de *BEAMS*, Anh Vu Doan, travaille activement à l'intégration de méta-heuristiques multicritères au sein de la procédure d'exploration de NESSIE [Doa09].
- Le moteur de mapping de NESSIE connaît certains bugs et souffre de certains problèmes théoriques. Notamment au niveau de l'ordonnancement des tâches, certains problèmes de *deadlocks* apparaissent. Il serait nécessaire de résoudre ces problèmes et de réécrire ce moteur pour qu'il fonctionne correctement.
- Modéliser un système sous forme de Réseaux de Petri et de graphes des ressources, et ensuite écrire cette spécification en XML est une tâche longue et fastidieuse. Or le but de NESSIE est de faciliter la tâche des concepteurs, pas de la complexifier avec de nouveaux concepts et syntaxes à apprendre et à maîtriser. Il serait dès lors profitable que NESSIE s'intègre, au niveau des entrées et sorties, aux standards de *system-level design* des systèmes embarqués. La possibilité d'intégration de NESSIE avec le profil MARTE est abordée au chapitre suivant.
- Pour évaluer la performance d'un système avec NESSIE, des modèles de coût doivent être spécifiés pour les tâches et blocs élémentaires. Pour cela, un outil pour encoder des formules a été prévu avec NESSIE. Cet outil s'appelle YETI. Des modèles de coût ont été fournis pour des cas d'étude bien particuliers, notamment au travers de la thèse d'Aliénor Richard [Ric10]. Cependant, la communauté scientifique souffre d'un manque de modèles de coût exploitables dans le cas général et appliquables aux simulations proposées par NESSIE.

Chapitre 7

Le profil Marte et l'outil Gaspard

Ce chapitre présente l'outil de conception *system-level* Gaspard2 ainsi que le profil Marte. Une brève introduction au *Model-Driven Engineering* est également fournie afin de fixer le contexte de la création de cet outil.

7.1 Introduction

L'équipe *DaRT* [5] est spécialisée dans la conception efficace d'applications de traitement intensif de signal déployées sur des systèmes embarqués. La plupart du temps, ces systèmes sont des applications avec contraintes de *temps réel* et sont exécutées sur des plates-formes *parallèles*. En parallèle au développement de Nessie à l'ULB, les chercheurs de l'équipe ont conçu un environnement de développement, appelé Gaspard2, dont le but est de permettre une modélisation haut niveau (system-level design), sous forme de diagrammes UML, de tels systèmes. Cette modélisation, conforme au profil Marte, permet la vérification de propriétés sur le système et la génération de code pour plusieurs *cibles* d'implémentation différentes. Cette démarche permet de réduire le temps de conception de ces systèmes, de conserver un lien fort entre la modélisation et l'implémentation et de permettre le choix de la cible finale indépendamment de la modélisation du système. Contrairement à Nessie donc, le but de Gaspard2 est d'obtenir en fin de compte un système fonctionnel (ou du moins une partie de ce système).

Pour concevoir cet outil, l'équipe a utilisé le paradigme de développement *Model-Driven Engineering*, ou MDE.

Cette équipe a également grandement contribué au développement du standard de modélisation des *System-on-Chip* Marte[1], aujourd'hui reconnu et validé par l'*Object Management Group*[2]. Gaspard2 exploite grandement ce standard, surtout en matière de modélisation des systèmes.

Pour mener à bien ces deux objectifs, les chercheurs de *DaRT* ont exploité une méthodologie connue depuis quelques années au niveau du *software engineering* mais émergente dans le monde de l'informatique embarquée : le *Model-Driven Engineering*.

1. Modeling and Analysis of Real Time and Embedded systems [2].
2. L'Object Management Group, ou OMG, est un organisme dont le but est d'établir des standards de modélisation dans de nombreux domaines d'applications (notamment l'électronique embarquée), et ce en appliquant une approche *Orientée Objets*.

7.2 L'approche Model-Driven Engineering

Le *Model-Driven Engineering*, ou en Français « ingénierie dirigée par les modèles », se veut être un nouveau paradigme de conception d'applications, surpassant, en termes d'abstraction, la méthodologie *Orientée Objets*.

L'idée essentielle derrière le *MDE* est la définition et l'utilisation de modèles pour créer l'applicatif. Tout comme un programme en méthodologie Orientée Objets manipulera uniquement des objets, une application MDE exploitera des modèles.

La définition d'un modèle est fortement dépendante du domaine de l'application qui exploite ce modèle. Le domaine d'application qui nous intérresse ici est celui des systèmes embarqués, les applications relatives utiliseront donc des modèles de ce domaine. Un modèle appartenant à un domaine d'application se veut *parlant* pour les utilisateurs familiers à ce domaine. Ainsi, les concepteurs de *System-on-Chip* ne sont pas dépaysés par rapport à leurs habitudes : l'approche par modèles les aide juste à être plus abstraits dans la description de leur système. L'idée est bien de *modéliser* le système, c'est-à-dire en définir le fonctionnement général et abstrait, en masquant les détails d'implémentation : utiliser un modèle pour réaliser ceci est par conséquent une démarche naturelle.

Un concept essentiel propre à la méthodologie MDE est la notion de *méta-modèle*. Un méta-modèle permet de définir « toutes les valeurs » que peut prendre un modèle d'un certain domaine d'application. De la même façon que tout objet est issu d'une certaine classe en méthodologie Orientée Objets, tout modèle est issu d'un méta-modèle en MDE, et c'est le méta-modèle qui est fortement relié au domaine d'application. En outre, il est également possible de définir plusieurs méta-modèles par domaine d'application avec plusieurs niveaux de raffinement. Ainsi, au sein d'un même domaine d'application, il peut exister plusieurs couches d'abstraction de modèles. Ceci convient particulièrement bien dans le milieu des systèmes embarqués qui est également structuré en plusieurs couches d'abstraction, comme présenté en 2.3.

Enfin, la notion la plus importante en Model-Driven Engineering est sans doûte celle de transformation de modèles. Il est possible de définir des « méthodes de traduction » pour transformer un modèle issu d'un certain méta-modèle en un nouveau modèle équivalent mais défini par un autre méta-modèle. L'avantage de cette méthode est de proposer la définition de la traduction *au niveau des méta-modèles*, automatisant donc une fois pour toute la traduction de modèles. Ainsi, au lieu de devoir traduire « manuellement » un modèle d'un domaine d'application à l'autre (ou d'une couche d'abstraction à l'autre), la méthode de traduction sera *générique* pour tous les modèles issus du méta-modèle traduit. La figure 7.1 illustre ce principe.

En conclusion, l'approche par le MDE permet d'assurer une *compatibilité des concepts* entre applications, et ce, de façon *transversale* (entre plusieurs domaines d'application) et *verticale* (entre plusieurs niveaux d'abstraction d'un même domaine d'application).

7.3 Le standard MARTE

Le standard MARTE constitue un méta-modèle de haut-niveau pour les *System-on-Chip*. L'intérêt de la définition d'un tel standard est sa validation par un organisme

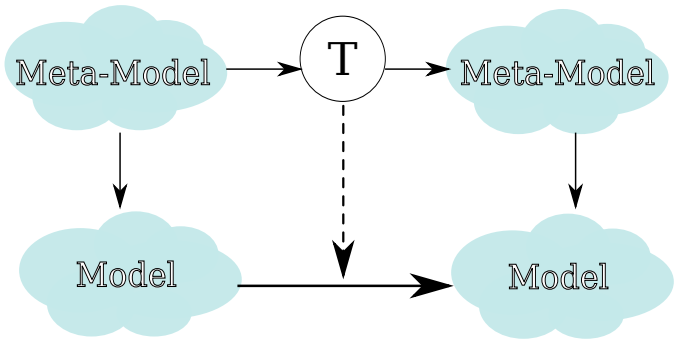


FIGURE 7.1 – Notion de modèle, méta-modèle et transformation de méta-modèles.

officiel. En effet, les industries devraient très vite commencer à utiliser ce standard dans leur propre intérêt : facilité de communication, rapidité et efficacité de développement, disponibilité de personnel formé à ce standard, etc.

De plus, le fait que MARTE s'inscrive dans la méthodologie MDE permet de le rendre facilement compatible avec toute application imaginable : il suffit aux concepteurs d'implémenter une traduction de MARTE vers le méta-modèle de leur application, pour peu que ce méta-modèle soit défini.

Enfin, cette facilité de traduction permet de créer aisément des outils réalisant de la génération de code. Un code source écrit dans les langages comme le C ou le VHDL est un exemple de solution générable depuis un modèle défini du méta-modèle MARTE.

7.4 L'outil GASPARD2

GASPARD2 se présente comme un outil de modélisation et de développement de systèmes embarqués. L'outil en tant que tel est implémenté par une « chaîne de transformations MDE ».

Une vue simplifiée de cet outil peut être la suivante : l'utilisateur conçoit son système en définissant un modèle de ce système, dans le formalisme de MARTE ; ensuite GASPARD2 réalise la simulation, la validation et la transformation du système vers, par exemple, des langages comme le C ou le VHDL, par une suite de traductions du modèle défini par l'utilisateur en modèles de moins en moins abstraits. La figure 7.2 schématise le fonctionnement de GASPARD2.

7.4.1 Méthodologie de conception

Présentation

Cette section présente un sous-ensemble du profil MARTE, dont le but est la modélisation à haut niveau d'abstraction de systèmes embarqués temps réel.

En particulier, cette section décrit la méthodologie proposée par GASPARD, utilisant MARTE, pour la conception de *System-on-Chip*.

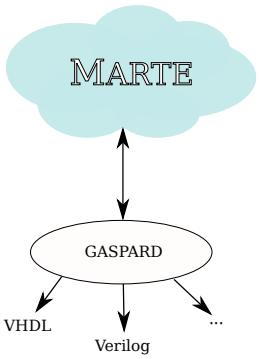


FIGURE 7.2 – Fonctionnement de GASPARD2 et interactions avec MARTE et les langages cibles.

Pour concevoir un modèle conforme au méta-modèle MARTE d'un tel système, il faut notamment établir trois schémas UML :

un modèle d'application décrivant la fonctionnalité du système,

un modèle d'architecture décrivant la plate-forme d'exécution du système,

un modèle d'association décrivant la façon dont la fonctionnalité s'exécutera sur la plate-forme.

Pour chacun de ces modèles, le profil MARTE propose un méta-modèle standard. Dans l'état actuel de l'outil GASPARD, l'utilisateur doit saisir chacun de ces modèles pour concevoir un système.

Méta-modèle d'application

Le méta-modèle d'application permet la définition de la fonctionnalité du système. L'utilisateur doit établir la structure de son application en la divisant en tâches périodiques et inter-dépendantes. Afin de structurer l'application, l'utilisateur peut grouper un ensemble de tâches élémentaires dans une tâche inductive. Ainsi, pour modéliser l'application, l'utilisateur possède à sa disposition :

des tâches élémentaires, qui sont les plus petits éléments applicatifs modélisés,

des tâches répétées, ou « shaped », qui ont les mêmes caractéristiques qu'une tâche classique mais itérées un certain nombre de fois[3],

des tâches inductives, qui sont définies par un *flot* de tâches élémentaires, répétées et inductives. Une tâche inductive permet d'identifier les dépendances entre les tâches qui la composent.

Il est possible d'associer à chaque tâche modélisée des caractéristiques comme une échéance dans le cas de contraintes *temps réel* ou la consommation d'énergie relative à une certaine gamme de processeurs[4].

3. Typiquement, les tâches répétitives sont très présentes au sein d'une application de traitement du signal.

4. Dans son état actuel, GASPARD2 ne permet pas la saisie de ces caractéristiques.

Cette structure hiérarchique de l'application en tâches périodiques et inter-dépendantes permet de modéliser fidèlement la partie applicative d'un système embarqué temps réel.

Méta-modèle d'architecture

Le méta-modèle d'architecture permet la définition de la plate-forme du système. L'utilisateur établit la structure de l'architecture modélisée en la divisant en blocs, de la même façon que le matériel réel est divisé. Pour modéliser l'architecture, l'utilisateur dipose :

de blocs élémentaires, qui peuvent être de différents types : unités de traitements (*e.g.* processeurs), mémoires (*e.g.* caches), etc.

de blocs répétés, ou « shaped », qui sont constitués d'une matrice de blocs identiques. La figure 7.3 illustre ce type de blocs.

de blocs inductifs, qui sont définis par un assemblage de blocs élémentaires, répétés et inductifs. Un bloc inductif identifie également les points de communication entre les blocs qui le composent.

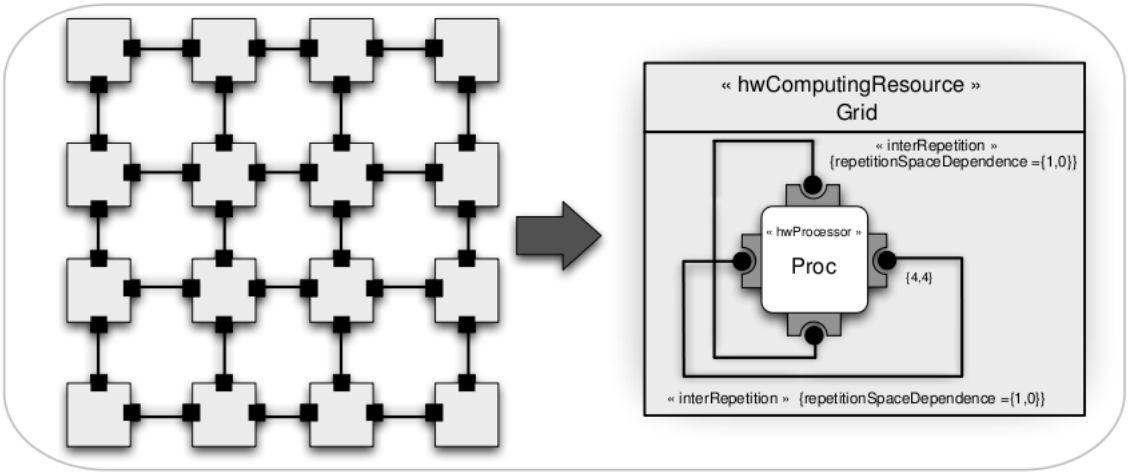


FIGURE 7.3 – Illustration d'une architecture « shaped » [Red10].

Méta-modèle d'association

Le méta-modèle d'association définit les relations possibles entre le schéma applicatif et architectural. En pratique, il exprime le placement des tâches sur leur plate-forme d'exécution.

L'association application-architecture permet l'évaluation des performances du système (temps d'exécution, énergie consommée, etc.).

La figure 7.4 illustre le design flow d'un système conçu avec la méthodologie proposée par GASPARD.

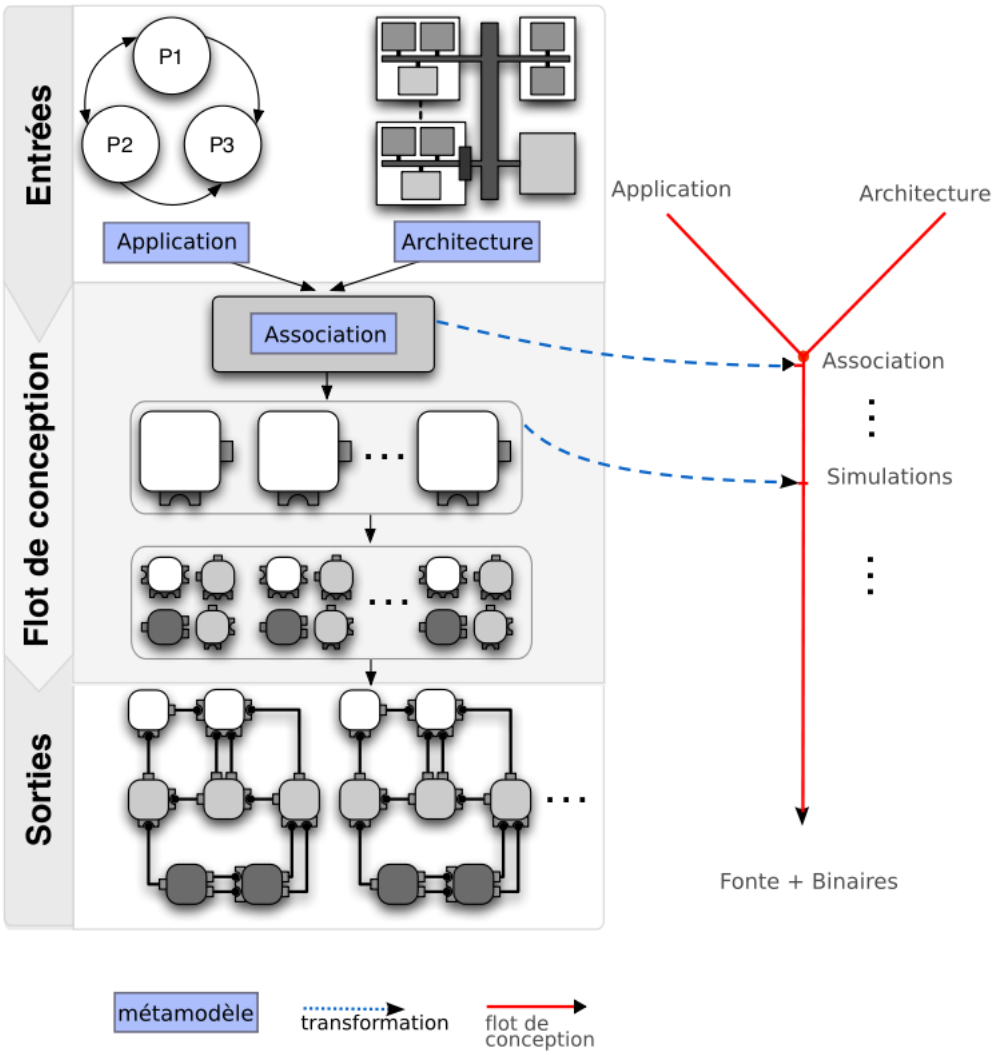


FIGURE 7.4 – Illustration du design flow d'un système conforme au profil MARTE [Red10].

Autres méta-modèles

Le profil MARTE est un standard très riche qui possède de nombreuses ressources de modélisation. Cependant, dans le cadre de ce mémoire, nous nous limiterons à ce qui a été présenté dans cette section. Pour plus d'informations, veuillez vous référer au Wiki de l'OMG [6].

7.4.2 L'environnement de conception Gaspard

Présentation

Comme présenté ci-dessus, GASPARD2 est un environnement de conception de systèmes embarqués. Le but de ce dernier est de permettre une conception indépendante

des concepts d'implémentation [5], c'est-à-dire permettre la conception du système à partir d'un *modèle* indépendant de la *cible* finale d'implémentation [6].

Pour modéliser un système embarqué, l'utilisateur de GASPARD2 doit saisir quelques diagrammes UML décrivant la fonctionnalité à réaliser et les ressources matérielles à exploiter. Ce modèle UML est conforme au profil MARTE de spécification de systèmes embarqués décrit dans la section 7.4.1. L'utilisateur choisit ensuite une des cibles implémentées au sein de GASPARD2 en vue de générer le code correspondant au modèle qu'il a défini. Un moteur transforme alors le modèle initial en code pour la cible, en faisant passer le modèle par une suite de transformations automatiques, visant à l'enrichir, le complexifier et le rendre de plus en plus dépendant de la cible choisie, jusqu'à la dernière étape qui consiste à transformer le modèle en code source (sous forme de fichier texte) correspondant à cette cible.

La saisie de diagrammes UML en deux dimensions permet d'exprimer un parallélisme *in situ*, par opposition à un code source (typiquement en C) qui est intrinsèquement séquentiel. En effet, lors de l'établissement d'un graphe de tâches et de ressources, les potentiels calculs parallèles apparaissent directement. Cette modélisation bidimensionnelle est donc par essence adaptée aux systèmes électroniques actuels où le parallélisme est omniprésent.

Dans la suite de ce chapitre, nous dénoterons par « utilisateur GASPARD » tout concepteur de système embarqué qui utilise ou utilisera GASPARD2 pour réaliser son système. Par opposition, le « développeur GASPARD » dénotera toute personne impliquée dans la conception du logiciel GASPARD2 lui-même.

Model-Driven Engineering appliqué à l'électronique

Afin de réaliser ces transformations, la méthodologie utilisée pour la conception de GASPARD2 est le *Model-Driven Engineering*, souvent traduite par *l'Ingénierie Dirigée par les Modèles*. A l'aide d'outils et de langages spécifiques, il est possible d'établir des méta-modèles, véritables cadres pour les modèles, ainsi que des transformations entre méta-modèles. Ceci permet d'automatiser la traduction d'un modèle abstrait (défini en UML) vers un code dépendant d'un langage spécifique (par exemple, le VHDL). Cette démarche, qui était auparavant effectuée *manuellement*, permet au concepteur de se concentrer sur les caractéristiques de son système (représenté sous forme de modèle abstrait) sans se soucier des détails d'implémentation propres à la cible choisie. L'automatisation de cette traduction réduit considérablement le risque d'erreur et permet de disposer facilement du système implémenté selon plusieurs cibles différentes.

Afin de passer du modèle conforme au profil MARTE au code généré dépendant de la cible, plusieurs transformations intermédiaires du modèle sont souvent nécessaires. Pour chaque cible, il existe une chaîne constituée de transformations « Modèle vers Modèle », qui part du modèle en UML/MARTE et qui se termine par une transformation « Modèle vers Texte » qui génère effectivement le code pour la cible sélectionnée. Ces transformations intermédiaires peuvent contenir des algorithmes complexes visant

5. Comme les langages, les librairies et tous les détails techniques propres au *déploiement* du système.

6. Cette cible est généralement un langage telle que le VHDL, SystemC, etc.

à créer de l'information (typiquement : fournir un ordonnancement des tâches à exécuter) ou à traduire des informations (typiquement : fournir le nombre de *thread* sur une cible en fonction du nombre de tâches abstraites du modèle) nécessaires à la génération de code. La figure 7.5 illustre une transformation classique.

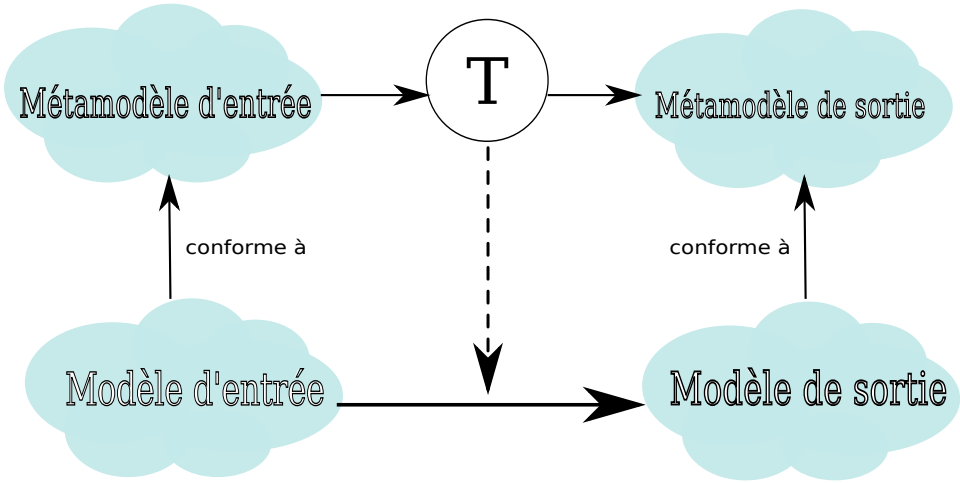


FIGURE 7.5 – Transformation de modèle.

Il existe plusieurs méthodes pour réaliser les transformations « Modèle vers Modèle » :

- Une transformation qui prend un méta-modèle d'entrée et le traduit complètement en un méta-modèle de sortie.
- Une transformation qui prend un méta-modèle d'entrée et l'enrichit avec des nouvelles informations pour former un nouveau méta-modèle de sortie.

La deuxième méthode est plus souple et modulaire car elle ne nécéssite pas de traduire toutes les informations du méta-modèle d'entrée qui ne varient pas au cours de la transformation. Cette méthode permet de définir uniquement les éléments qui varient ou qui sont nouveaux par rapport au méta-modèle d'entrée. C'est donc cette méthode qui a été utilisée pour GASPARD2.

Afin de réaliser ces chaînes de transformations, plusieurs outils sont utilisés :

Eclipse Modeling Framework (EMF) est un environnement, présenté comme une extension d'*Eclipse*, permettant la modélisation des systèmes. C'est avec cet outil que sont définis les méta-modèles et les variations entre méta-modèles au cours des transformations (aussi appelés *delta*)[7].

Query/View/Transformation (QVT) est un langage standardisé pour exprimer les transformations « Modèle vers Modèle ».

Acceleo est un langage *Open Source* pour exprimer les transformations « Modèle vers Texte », c'est-à-dire les transformations finales de chaque chaîne permettant la génération de code.

7. Comme chaque modèle est conforme à un méta-modèle donné, chaque méta-modèle est conforme au « méta-méta-modèle » standard, nommé *e-core*. Notons qu'*e-core* est défini par lui-même, stoppant ainsi la récursivité des définitions.

Papyrus est un logiciel, également intégré à *Eclipse*, permettant la saisie de diagrammes UML.

7.4.3 Etat actuel

Eclipse étant par défaut multiplateforme, EMF et tout ce qui en provient, incluant GASPARD2, le sont également. GASPARD2 est donc un logiciel pouvant s'exécuter sur Linux, Windows, MacOS et toutes les plates-formes supportées par *Eclipse*.

GASPARD2, dans son état actuel, implémente les cibles suivantes :

SequentialC : cette cible est la plus simple qui existe. Elle vise à générer un code C conforme à l'application visant à s'exécuter sur une plate-forme monoprocesseur. Le code C peut être exécuté et compilé sur un simple processeur d'un ordinateur « general purpose ». Un algorithme d'ordonnancement est exécuté lors des transformations : celui-ci aplatit l'ensemble des tâches sur une ligne du temps monoprocesseur.

PThread : cette cible, comme son nom l'indique, génère du code pour le framework C *pthread*. Ce framework permet de découper l'exécution d'une application en sous-processus appelés « thread », exécutant chacun une série de tâches en parallèle. Sur une plate-forme multiprocesseur, chaque coeur peut prendre la responsabilité d'un ou plusieurs threads. La chaîne de transformations partitionne, place et ordonnance les tâches de l'application en différents threads. En fin de chaîne, le code source des tâches et un *Makefile* sont générés. Il est possible de tester le résultat sur un processeur multi-coeur.

OpenCL : cette cible permet la génération de code pour le framework C *OpenCL* qui permet l'exécution d'applications sur des plates-formes hétérogènes comme des CPU, des GPU, des DSP, etc. C'est également un framework qui permet un grand parallélisme.

OpenMP : cette cible permet la génération de code pour l'API C *OpenMP*, qui permet l'exécution d'applications parallèles à mémoire partagée.

SystemC : cette cible génère du code pour *SystemC*, un environnement de simulation de systèmes électroniques basé sur le langage C++.

Lustre : cette cible permet la génération des équations synchrones correspondant au système. Ces équations, formalisées dans le langage Lustre, pourront ensuite être confrontées à des propriétés à satisfaire à l'aide d'outils de vérification formelle.

VHDL : cette cible permet la génération de code en VHDL, le langage de description du *hardware*, afin de permettre la réalisation du système sur un support FPGA ou une implémentation ASIC.

Dans l'état courant du logiciel GASPARD2, les deux dernières cibles citées sont temporairement non disponibles.

La modélisation de systèmes embarqués, comme présentée dans la section 7.4.1, définit un ensemble de tâches et de plates-formes élémentaires. Cependant, le comportement interne de ces éléments n'est lui-même pas décrit dans GASPARD ou dans

le profil MARTE. En effet, ils sont laissés à l'appréciation de l'utilisateur. C'est à ce dernier de définir les « Intellectual Properties », ou IP, qui les définissent.

Ces IP doivent être décrites dans le langage de la cible d'implémentation choisie par l'utilisateur. Chaque tâche élémentaire et chaque bloc élémentaire doit être liée à une IP dans le diagramme de déploiement. Si l'utilisateur désire plusieurs implémentations de son système, il doit disposer d'un diagramme de déploiement pour chaque cible d'implémentation désirée. Par conséquent, chaque tâche élémentaire doit être décrite par une IP dans le langage de la cible choisie.

Le but du générateur de code, en fin de chaîne de transformations, est donc de mettre ensemble les IP [8], à l'aide d'algorithmes plus ou moins complexes. Les IP sont copiés et injectés au sein des fichiers qui formeront le système final après compilation (ou synthèse).

Il existe également un module spécial, appelé la GASPARDLIB, contenant une définition d'IP pour des fonctions classiques. Il s'agit de l'équivalent d'une *bibliothèque* pour GASPARD.

7.4.4 Perspectives

L'outil GASPARD est un projet en constante évolution. A terme, il veut fournir une solution viable de développement haut niveau de systèmes embarqués temps réel. Pour ce faire, la pure traduction de modèles ne suffit pas : GASPARD doit intégrer les concepts actuels et futurs des designs microélectroniques, présentés ci-dessous.

Reconfiguration dynamique

La reconfiguration dynamique est la capacité, pour un circuit électronique, de changer sa fonction après sa fabrication. Cette reconfiguration peut avoir lieu à différentes échelles spatiales ou temporelles. Il existe donc de nombreuses gammes de circuits dynamiquement reconfigurables parmi lesquelles nous pouvons citer :

- les circuits programmés, comme les processeurs génériques, les DSPs, etc.
- les circuits programmables, comme les FPGA.

Les circuits programmables peuvent changer leur architecture, la logique de leur circuit, entre deux utilisations (*e.g.*, pour la programmation d'un *prototype*) ou pendant l'utilisation (*e.g.*, le passage en mode *basse consommation d'énergie* lorsqu'un seuil déterminé d'utilisation de la batterie est atteint).

Les développements futurs de GASPARD viseront à intégrer la modélisation des architectures dynamiquement reconfigurables en proposant, par exemple, des modèles pour des architectures FPGA présentes sur le marché. Des travaux sont en cours pour identifier ce qui permet, dans le profil MARTE, de modéliser les propriétés de ces circuits.

8. Qui correspondent formellement à des fichiers sources.

Design Space Exploration

Comme décrit dans le chapitre précédent, l'utilisateur de GASPARD doit saisir un diagramme d'application, d'architecture et d'association. Ces diagrammes modélisent, à haut niveau, le système à concevoir. Ainsi, les découpes de l'application en tâches et de l'architecture en blocs ainsi que le placement sont laissés à l'appréciation de l'utilisateur. Il s'agit de degrés de liberté laissés au concepteur du système. Cependant, cette découpe résulte d'un ensemble de choix non arbitraires qui influence considérablement les performances du système final.

En effet, il peut exister différentes variantes d'applications et d'architectures, réalisant la même fonction mais ayant des performances différentes : temps d'exécution, énergie consommée, surface utilisée, coût, etc. sont autant de critères que le concepteur voudra optimiser en fonction de ses besoins et de son cahier des charges.

Afin de réaliser un compromis en termes de performances parmi une série de propositions de choix applicatifs et architecturaux, l'outil informatique peut aider le concepteur à faire les bons choix. Si l'utilisateur modélise également les critères de performances qui l'intéressent et la façon dont les composants sont reliés à ces critères, un algorithme d'exploration de solutions peut rechercher, parmi les propositions de modèles d'applications et d'architectures, quels sont les couples qui constituent les meilleurs compromis en matière de performances [9]. NESSIE, l'outil présenté au chapitre 6, réalise précisément cette tâche : il explore des variantes d'architectures et d'applications dans le but d'établir le meilleur compromis en termes de performances.

Afin de comparer différents couples application/architecture optimaux, il est nécessaire de pouvoir évaluer le coût d'un tel couple. En fait, il existe plusieurs manières d'associer l'architecture et l'application, et celles-ci influent différemment sur le coût en termes de performances. Pour une seule variante d'application et d'architecture, il peut exister de nombreuses solutions de *mapping*, avec différents compromis de performances. A nouveau pour accélérer le travail du concepteur humain, il est nécessaire qu'un algorithme explore ces solutions dans le but de fournir un diagramme d'association de façon automatique [10]. C'est l'objet de l'outil AAS, également développé à Lille. Notons que NESSIE fournit également un *mapping* entre l'application (la fonctionnalité) et l'architecture (la plate-forme). Ce *mapping* est réalisé par le coeur du logiciel, nommé « Nessie performance estimation core ».

En définitive, NESSIE représente une solution de choix pour optimiser un design construit à l'aide de GASPARD2. Il présente l'avantage de fournir une optimisation sur deux niveaux différents :

- l'un au niveau de l'association directe entre un modèle d'application (la fonctionnalité) et un modèle d'architecture (la plate-forme),
- l'autre au niveau de l'exploration de solutions alternatives de modèles d'application et de plate-forme.

Il serait donc idéal de pouvoir lier ces deux outils. Les concepteurs disposeraient ainsi d'un environnement de conception (GASPARD2) de modèles de haut niveau et

9. Typiquement, les solutions à rechercher sont celles situées sur la frontière Paréto-optimale.

10. Ou du moins proposer à l'utilisateur quels sont les meilleurs compromis d'associations fonctions/plates-formes.

conforme à un standard (MARTE), fournissant une assistance au design (optimisations proposées par NESSIE) et réalisant la synthèse du système (les différentes cibles de GASPARD2).

Les développements futurs de GASPARD viseront l'intégration des outils d'analyse multicritère pour l'exploration de variantes de design (ou *Design Space Exploration Tools*) au travers de transformations de modèles. En effet, il suffit de générer, à l'aide des langages présentés ci-dessus, les fichiers d'entrées des outils à intégrer (il s'agit très souvent de fichiers formattés en XML) et de récupérer les fichiers de sorties en les traduisant en modèles. Cette méthode permet l'intégration d'outils dans GASPARD2 de façon légère, efficace et intuitive.

La section suivante présente l'idée de l'implémentation de cette liaison entre GASPARD2 et NESSIE et les démarches entreprises dans ce sens.

7.5 Lien avec NESSIE

L'avantage d'un environnement comme GASPARD2 est indéniable : il réduit considérablement le temps de développement. Cependant, il serait utile de pouvoir réaliser une analyse de performances multicritères des systèmes modélisés à l'aide de cet outil. Or, comme expliqué au chapitre précédent, il s'agit justement de l'objectif de NESSIE. Un juste questionnement survient alors : comment serait-il possible de combiner les facilités de modélisation et d'implémentation fournies par MARTE et GASPARD2 avec l'analyse des performances du système prodiguée par NESSIE ?

Une idée naturelle serait de définir un méta-modèle des entrées et sorties de NESSIE et d'implémenter ensuite la traduction de ce méta-modèle vers MARTE, et vice-versa. Ainsi, la démarche de conception d'un système embarqué serait, à première vue, la suivante :

- Création de modèles (conformes à MARTE) haut niveau, à l'aide de GASPARD2, de différentes variantes d'implémentation du système à partir de son cahier des charges.
- Traduction de l'ensemble de ces modèles vers NESSIE, avec injection des contraintes de performances (issues également du cahier des charges) dans le processus.
- Evaluation, à l'intérieur de NESSIE, de la meilleure solution de design.
- Récupération des résultats de l'analyse de NESSIE dans un modèle conforme à MARTE.
- Injection du modèle du système ainsi obtenu dans GASPARD2 pour simulation du système et génération de code.

Les phases de traduction étant automatisées, cette démarche réduirait considérablement la complexité du flot de conception des *System-on-Chip*. La figure 7.6 représente le schéma de cette idée de design flow.

Dans le cadre de son mémoire de fin d'études, Nastassia Gumuchdjian a réalisé une étude de faisabilité d'une telle compatibilité entre les différents outils [Gum10]. Elle a travaillé à la méta-modélisation des entrées et sorties de NESSIE avec les outils du

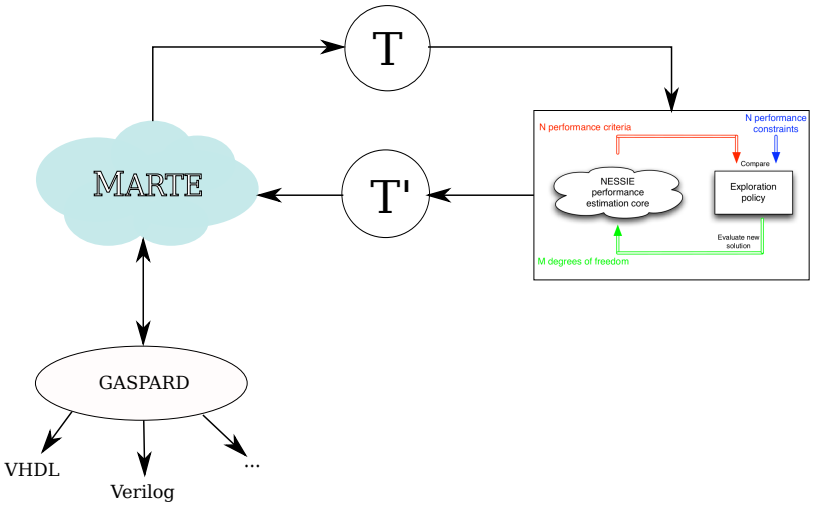


FIGURE 7.6 – Schéma général des interactions possibles entre NESSIE, MARTE et GASPARD2.

MDE (QVT, etc.). Il suffirait de réaliser une transformation entre les méta-modèles GASPARD et les méta-modèles NESSIE pour lier les deux outils.

Cependant, nous avons constaté, après une étude plus approfondie de l'outil GASPARD2, qu'il reste à ce jour certaines lacunes au sein de cet outil rendant pour le moment impraticable une tentative de liaison :

- Il n'est pas possible de modéliser les performances des éléments du système : temps d'exécution des tâches, consommation d'énergie, surface de silicium occupée par le circuit, etc sont des éléments cruciaux pour réaliser une optimisation multiobjectif. Dans ce domaine, une piste à explorer est le stéréotype MARTE *« ressource usage »* décrit dans le papier de Kamoun [KB05]. L'idée est d'implémenter ce composant MARTE au sein des modèles GASPARD2.
- Il est nécessaire de pouvoir spécifier un « diagramme d'associations potentielles », c'est-à-dire l'impact en termes de performances du choix de placer une telle tâche sur un tel processeur. Ce point rejoint également le point précédent.

- Des menus pour l'optimisation doivent être intégrés à l'interface de GASPARD. Ces menus doivent permettre de choisir les critères de performances à tenir compte, etc. De la même façon il est nécessaire de pouvoir traiter le résultat au retour dans GASPARD : il faut pouvoir effectuer un choix d'une solution parmi celles générées par l'outil. Ceci nécessite également des ajouts dans l'interface graphique.

Une autre solution, qui a été pensée par Gumuchdjian, est de réaliser une transformation intermédiaire, et de rajouter les modèles de coût manuellement à l'aide d'une interface graphique. Cette solution a été implémentée par Gumuchdjian et Sébastien Leduc dans le cadre du stage de ce dernier [Led10].

Cependant, même si une possibilité de spécifier les coûts des systèmes existait, il n'empêche que les informations des coûts pour les composants fondamentaux des systèmes embarqués restent globalement indisponibles. La nécessité de disposer de tels modèles se fait fortement ressentir au sein de la communauté scientifique.

Chapitre 8

Conclusion provisoire

Ce chapitre conclue sur l'état de l'art des deux travaux de recherche présentés au sein de la partie II.

8.1 Fonctionnalité abstraite et monde physique

L'élément primordial caractérisant un système électronique est sa *fonctionnalité*. Il peut être tentant pour le concepteur du système de considérer cette fonction comme abstraite, existant à part entière dans un monde mathématique [1].

Cependant, l'implémentation du système consiste à projeter cette fonction mathématique sur une plate-forme physique, supportant l'exécution de la fonction dans le monde réel. Cette projection s'accompagne d'une série de caractéristiques, ou métriques, mesurables physiquement sur le système comme les délais d'exécution, les dimensions géométriques, l'énergie consommée, les coûts, etc.

Ces différentes métriques de performances sont conditionnées par les choix de conception effectués lors de la projection de la fonction sur la plate-forme. Ces choix concernent aussi bien les aspects matériels que logiciels de la conception.

Néanmoins, ce lien entre choix de conception et métriques de performances est difficile à évaluer et l'exploration de l'ensemble des choix afin d'obtenir les meilleurs performances possibles nécessite de perdre du temps à concevoir itérativement différentes solutions. Ce processus étant fastidieux et coûteux en termes de temps, les concepteurs industriels sont souvent amener à retenir des solutions sous-optimales. C'est pourquoi de nombreux travaux de recherche ont pour but de tenter de prédire les performances sur base de choix haut niveau, en amont des flots complets de conception de systèmes numériques.

8.2 Framework de la recherche

Les travaux effectués à l'INRIA Lille Nord et à l'Université Libre de Bruxelles ont permis de définir un framework, une méthodologie pour la conception visant l'optimisation multicritère de différentes métriques de performances.

1. Ceci correspond à description comportementale du diagramme Y (*cf.* chapitre 2).

Un système numérique est défini par un modèle conforme à un standard de modélisation *system-level* comme MARTE.

Avant de passer à une étape de synthèse de haut niveau (avec un outil comme GASPARD2), une phase d'optimisation est réalisée en analysant le modèle du système avec un outil (comme NESSIE) de prédiction de performances et d'exploration des choix de conception possible.

Au sein de cet outil, le système est perçu comme un assemblage de composants[2]. Nous pouvons imaginer que chacun de ces composants dispose d'un certain nombre de *leviers* qui représentent les degrés de liberté de la conception. La position des leviers représentent les choix de conception, c'est-à-dire la façon dont a été fixé le degré de liberté correspondant. La position de l'ensemble des leviers conditionne les performances du système :

- localement, à l'aide de modèles de coûts (en termes de performances) de composants élémentaires du système ;
- globalement, en agrégeant les modèles de coûts locaux des différents composants constitutifs du système à une information de performances caractérisant les métriques du système complet.

Le but de NESSIE est de composer les différents modèles des blocs élémentaires pour produire une prédiction des performances du système complet. Cependant, les méthodologies d'identification des modèles des composants élémentaires du système, faisant intervenir de multiples critères, sont encore peu explorées aujourd'hui.

La recherche de la « position idéale » de l'ensemble des leviers est réalisée par la procédure d'aide à la décision multicritère.

8.3 Utilité des modèles de coûts

Les chapitres 6 et 7, ainsi que la section précédente, concluent sur le besoin de disposer de modèles de coût et, de façon plus générale, la nécessité d'une méthodologie d'identification rapide de tels modèles.

Une fois ces modèles obtenus, nous pourrions les intégrer dans les outils et flots de conception présentés jusqu'ici, comme MARTE, GASPARD2 ou YETI (qui est utilisé pour encoder les modèles exploités par NESSIE).

Ces modèles pourraient également être utilisés tels quels par les concepteurs industriels, qui disposeraient ainsi d'une information quantitative pertinente sur l'impact des choix de conception sur les performances.

La partie III du mémoire a pour objet de proposer une approche pour l'identification de tels modèles.

2. Une transformée de Fourier ou un filtre constituent des exemples typiques de composants d'un système numérique.

Troisième partie

Création de modèles de performances

Chapitre 9

Description de l'approche

Les parties I et II ont mis en évidence la nécessité de disposer de modèles de performances pour les composants essentiels des systèmes embarqués, tant au point de vue matériel que logiciel. Cette dernière partie du mémoire présentera une approche, initiée au sein de l'équipe *BEAMS digital*, pour la création de tels modèles.

Le chapitre 9 présentera l'approche généralisée pour la création d'une méthodologie de création de modèles pour les blocs fondamentaux des applications de traitement des signaux sur plates-formes reconfigurables. Le chapitre 10 décrira en détails l'un de ces blocs de base, la transformée de Fourier, en proposant différents algorithmes pour la calculer. Dans le chapitre 11, nous réaliserons un premier exercice de création de modèles simples pour la transformée de Fourier déployée sur FPGA. Cet exercice sera réalisé dans le but de valider l'idée de l'approche de cette partie du mémoire. Les résultats expérimentaux seront discutés et une conclusion sera proposée.

9.1 Introduction

Comme conclu aux chapitres 4 et 8, il est important de disposer à haut niveau de modèles et de méthodes d'évaluation de performances fiables qui permettent au concepteur d'évaluer l'impact de ses décisions sur les performances du système final, et ainsi de cibler les choix les plus intéressants.

Pour résoudre le problème de conception haut niveau posé dans la partie I, d'importants efforts de recherche sont en cours pour définir des flots de conception *system-level* comme ceux décrits au sein de la partie II.

Néanmoins, les modèles de prédiction de performances exploitables par de tels flots et indispensables pour les rendre pleinement opérationnels ont été peu investigués dans le cas général. Plus spécifiquement, de nombreuses recherches ont été menées dans le domaine mais celles-ci portent principalement sur l'analyse de plates-formes à microprocesseur générique ou ASIP (*e.g.* DSP), en se focalisant sur un seul critère de performance : le temps d'exécution ou la consommation énergétique [MP02, EPTP07, LDA+06, HNG09, BK02].

Nous voudrions établir une approche où nous considérons la création de modèles des blocs critiques des applications de traitement des signaux (*i.e.* les algorithmes classiques de la littérature) déployés sur des architectures reconfigurables (type FPGA) et

d'en étudier simultanément de multiples critères de performances (temps d'exécution, surface, consommation, coût, etc).

La création de modèles de performances repose sur des techniques statistiques d'analyse des données obtenues par simulations. Ces techniques sont regroupées sous le nom de *machine learning*. Parmi les techniques existantes, nous distinguons [HTF01] :

- les techniques d'apprentissage non-supervisé, qui permettent l'étude et la réduction de la dimensionnalité d'un problème (*e.g.* l'analyse en composantes principales ou le clustering).
- les techniques d'apprentissage supervisé, qui permettent d'étudier les liens entre les variables d'entrées et de sorties d'un problème afin d'extraire un modèle. En fonction de la nature du problème, on applique des méthodes de classification ou de régression, linéaires ou non-linéaires. Il existe également des techniques de sélection de variables dans le cas supervisé.

Les techniques exploitées afin d'analyser le résultat des simulations appartiennent à la famille des techniques de régression. A titre d'exemple, nous utiliserons dans ce mémoire une régression linéaire aux moindres carrés.

Concrètement, la création des modèles prédictifs de performances s'appuiera sur de larges campagnes de simulations (ou plus exactement de synthèses [1]) couvrant différentes variantes d'implémentation de différents blocs algorithmiques de base sélectionnés. Les fonctions typiques qui pourraient être l'objet de ces simulations sont la transformée de Fourier rapide (FFT), la transformée en ondelette discrète (DWT) et la transformée en cosinus discrète (DCT). Ces blocs constituent l'épine dorsale de la plupart des codecs audio/vidéo existants (JPEG, JPEG 2000, MPEG...) ainsi que dans de très nombreuses applications de traitement du signal. Leur caractérisation permettra donc d'offrir un outil précieux au concepteur pour un large spectre d'applications.

Pour chaque implémentation, les paramètres fondamentaux de conception du bloc de base seront identifiés. L'espace des choix de ces paramètres sera exploré afin de récolter un maximum de données caractérisant la performance des implémentations.

Les techniques de *machine learning* telles qu'évoquées ci-dessus (régressions linéaires et non-linéaires) permettront de condenser la masse d'informations générées par les simulations et d'en extraire des modèles prédictifs de performances.

Au final, nous vérifierons que la prédiction ainsi obtenue permet effectivement d'accélérer la conception et/ou d'obtenir de meilleures performances sur le système final.

Le but à long terme consiste à formaliser cette approche en développant une méthodologie précise de production de ces modèles de performances, intégrable dans les flots de conception les plus récents.

Nous pouvons résumer l'originalité de cette approche dans les éléments suivants :

- il s'agit d'une approche systématique de l'application de techniques statistiques et algorithmiques à la conception micro-électronique avec de nombreuses combinaisons de résultats possibles pour différentes variantes d'algorithmes, différents types de cible, d'outils et de techniques d'analyse ;

1. Ces synthèses seront réalisées à l'aide d'outils classiquement utilisés dans les flots de conception.

- le choix, dans un premier temps, des architectures FPGA comme cibles matérielles ;
- l'approche multicritère des modèles [2] ;
- le souci de veiller à la possibilité d'exploiter les résultats obtenus (modèles et méthodologie de production de modèles) dans le cadre de flots de conception actuels et émergents.

Les détails de la démarche sont explicités au sein de la section 9.3.

Nous nous intéresserons aussi en particulier au problème de la composabilité des modèles de performances, c'est-à-dire la possibilité de prédire les performances globales d'un système sur base des modèles de performances des éléments de ces systèmes.

Dans le cadre du mémoire, nous étudierons un premier cas d'étude : la transformée de Fourier rapide déployée sur plate-forme FPGA.

9.2 Formalisation du problème

En reprenant la modélisation du problème de l'optimisation multicritère comme défini en 6.1.2, nous avons :

- $x = \langle x_1, x_2, ..., x_n \rangle$, l'ensemble des choix de design,
- $y = \langle y_1, y_2, ..., y_m \rangle$, l'ensemble des mesures en termes de coûts de performances du système.

Nous modélisons dans ce cas le lien fonctionnel entre x et y de la façon suivante :

$$\begin{cases} y_1 &= f_1(x_1, x_2, ..., x_n) \\ y_2 &= f_2(x_1, x_2, ..., x_n) \\ ... & \\ y_m &= f_m(x_1, x_2, ..., x_n) \end{cases}$$

La nature de ce lien $f(.)$ est *a priori* inconnue. Les techniques de *machine learning* proposent, sur base d'un ensemble de réalisations des variables $\langle x, y \rangle$, des modèles pour $f(.)$, permettant de prédire les valeurs de y pour certains x donnés.

Au sein de ce mémoire, le but des analyses se limitera à illustrer l'existence de ce lien fonctionnel $f(.)$ entre les choix de design x et les performances du système y, sans considération profonde pour la qualité de prédiction des modèles de $f(.)$. Cette analyse préliminaire est réalisée au sein du chapitre 11.

La définition claire et cohérente de la nature des paramètres x et y ainsi que l'identification de méthodes de modélisation appliquables à ce problème, la méthodologie d'extraction rapide de modèles et l'intégration au sein des flots de conception existants rentrent dans le cadre d'une éventuelle future thèse de doctorat.

2. Les modèles seront multicritères dans le sens où nous considérerons pour les différents cas d'études de multiples métriques (délais, surface, coût, etc.) associées aux performances du système.

9.3 Méthodologie d'acquisition de modèles

Cette section détaille les étapes précises prévues pour acquérir les modèles de coûts reliés aux fonctions étudiées.

Pour l'ensemble des cas d'études décrits section 9.1, on élaborera des modèles de performances de la manière suivante :

- Sélection d'un bloc algorithmique comme cas d'étude.
- Identification des variantes d'implémentations de cet algorithme.
- Acquisition ou écriture d'une implémentation de ces variantes.
- Validation fonctionnelle des différentes implémentations.
- Sélection d'une plate-forme matérielle de déploiement et d'outils de synthèse.
- Réalisation d'une campagne de simulations/synthèses et génération de résultats (temps d'exécution, surface occupée, énergie consommée, etc).
- Construction de modèles à l'aide des techniques de régression.
- Validation de la démarche et des modèles obtenus, en particulier par rapport à un flot de référence n'exploitant pas les modèles prédictifs.

Ces étapes seront réitérées en faisant varier les algorithmes, les variantes d'implémentation, les cibles matérielles, les outils de synthèse et les techniques d'analyse de données afin d'affiner la démarche d'extraction de modèles.

9.4 Cas d'étude

Pour des raisons évidentes de délai, nous nous restreindrons, dans le cadre de ce mémoire, à une analyse préliminaire constituant seulement une partie limitée de l'approche présentée au sein de ce chapitre. La poursuite de ce travail, c'est-à-dire l'exploration de l'approche complète, fera l'objet d'une potentielle thèse de doctorat.

Nous étudierons la transformée rapide de Fourier (FFT) déployée sur plate-forme reconfigurable. Pour ce faire, nous étudierons les performances de deux implémentations RTL existantes de cette fonction, synthétisées sur différentes cibles FPGA.

Les métriques de performances étudiées seront, en particulier, les délais d'exécution de la fonction, la surface occupée par le circuit et la puissance consommée.

La technique principale utilisée pour créer les modèles sera la régression linéaire aux moindres carrés.

Le chapitre 10 décrira en détails la définition et les algorithmes de la transformée de Fourier rapide. Ensuite, le chapitre 11 décrira le déploiement de la campagne de simulations et présentera les résultats de l'analyse préliminaire.

Chapitre 10

La Transformée de Fourier Rapide

> "*Cooley and Tukey apparently developed their algorithm in 1965 to help detect Soviet nuclear tests without actually visiting Soviet nuclear facilities, by interpolating off-shore seismic readings. Without their rediscovery of the FFT algorithm, the nuclear test ban treaty would never have been ratified, and we'd all be speaking Russian, or more likely, whatever language radioactive glass speaks.*"
>
> **Jeff Erickson, 2003**

10.1 Introduction

La transformée de Fourier rapide, en abrégé FFT (de l'anglais « *Fast Fourier Transform* ») est un algorithme permettant de calculer efficacement la transformée discrète de Fourier.

La transformée discrète de Fourier est un composant de base essentiel pour l'analyse et le traitement des signaux. Elle est notamment utilisée pour le filtrage et la compression de données.

La transformée de Fourier transforme un signal donné en entrée du domaine temporel vers le domaine fréquentiel. Le signal donné en entrée est représenté dans le domaine du temps, c'est-à-dire qu'il s'agit d'une fonction qui associe à chaque instant une amplitude. Après l'application de la transformée de Fourier, le signal est exprimé sous la forme d'une somme pondérée de sinusoïdes à décalage de phase et à fréquence variable. Ces poids et ces phases associés aux fréquences représentent le signal dans le domaine des fréquences.

Ce chapitre introduit la transformée discrète de Fourier et l'algorithme le plus communément utilisé pour la calculer, à savoir la transformée de Fourier rapide. Les explications de ce chapitre sont tirées de l'ouvrage de Cormen *et al* [CLRS01].

10.2 Représentation de signaux sous la forme de polynômes

Nos outils de calculs ayant une mémoire de taille finie, la représentation des signaux ne peut être faite de façon continue [1]. Ainsi, les signaux sont représentés de façon discrète, en échantillonnant à intervalles de temps réguliers l'amplitude des véritables signaux analogiques du monde réel.

Une représentation possible d'un signal discret dans le domaine temporel est la représentation sous forme d'un *polynôme*. En effet, si n points ont été échantillonnés, il existe toujours un polynôme de degré au plus $n-1$ passant par chacun de ces points.

Un polynôme est une fonction d'une variable construite uniquement avec des additions, soustractions et multiplications de cette variable. Plus formellement, un polynôme A en x sur le corps complexe $\mathbb{C}$ est défini de la façon suivante :

$$A(x) = \sum_{j=0}^{n-1} a_j x^j \tag{10.1}$$

Où les $a_0, a_1, ..., a_{n-1} \in \mathbb{C}$ et sont les *coefficients* du polynôme. La représentation par coefficients d'un polynôme $A(x)$ de degré borné par $n-1$ est le vecteur de coefficients $a = \langle a_0, a_1, ..., a_{n-1} \rangle$.

Trois opérations de base sont réalisables sur les polynômes :

– l'addition de deux polynômes $A(x)$ et $B(x)$:

$$A(x) = \sum_{j=0}^{n-1} a_j x^j \quad \wedge \quad B(x) = \sum_{j=0}^{n-1} b_j x^j$$
$$\Rightarrow$$
$$A(x) + B(x) = \sum_{j=0}^{n-1} (a_j + b_j) x^j$$

– la multiplication de deux polynômes $A(x)$ et $B(x)$:

$$A(x) = \sum_{j=0}^{n-1} a_j x^j \quad \wedge \quad B(x) = \sum_{j=0}^{n-1} b_j x^j$$
$$\Rightarrow$$
$$A(x)B(x) = C(x) = \sum_{j=0}^{n-1} c_j x^j$$

où

$$c_j = \sum_{k=0}^{j} a_k b_{j-k}$$

1. Ceci nécessiterait une mémoire de taille infinie continue.

– l'évaluation d'un polynôme $A(x)$ en un nombre $x_0 \in \mathbb{C}$:

$$y_0 = \sum_{j=0}^{n-1} a_j x_0^j$$

10.3 Une réprésentation alternative pour les signaux

Le vecteur des coefficients n'est pas la seule façon de représenter un polynôme. Par exemple, la *représentation par paires point-valeur* est une alternative. Celle-ci consiste à conserver un ensemble de n points du plan représentés par leurs coordonnées cartésiennes

$$\{(x_0, y_0), (x_1, y_1), ..., (x_{n-1}, y_{n-1})\}$$

telles que les x_k sont tous différents et que

$$y_k = A(x_k) \quad \forall\, 0 \leq k \leq n-1$$

c'est-à-dire que la coordonnée y d'un point est égale à l'évaluation du polynôme en sa coordonnée x.

Pour passer d'une représentation par paires point-valeur à une représentation par coefficients, il faut interpoler les n points de l'ensemble en un polynôme. Pour n points distincts, il existe un et un seul polynôme de degré $n-1$ qui passe par ces n points.

Pour passer d'une représentation par coefficients à une représentation par paires point-valeur, il suffit de choisir un ensemble de n valeurs distinctes sur l'axe des x et d'évaluer le polynôme $A(x)$ en ces points.

Le coût de pareilles opérations dépend des algorithmes choisis pour les réaliser. Un algorithme naïf pour cette dernière opération consisterait à évaluer, un par un, les n points choisis. Cet algorithme aurait un coût en temps s'exprimant en $O(n^2)$. En effet, une évaluation d'un polynôme de degré n coûte n additions et n multiplications, et il faut itérer l'évaluation sur les n points choisis.

Cependant, il est possible d'améliorer le coût de cette opération en utilisant un meilleur algorithme. En fait, le choix des n points n'est pas anodin et peut influer sur la complexité de l'opération. Si nous choisissons un ensemble de valeurs x_k avec la bonne forme récursive, il est possible de réaliser la conversion plus rapidement. C'est précisément l'objet de la transformée de Fourier discrète.

10.4 Les racines complexes de l'unité

Les *racines nièmes complexes* de l'unité sont les nombres complexes z satisfaisant l'équation

$$z^n - 1 = 0$$

Usuellement, une telle racine est notée ω, tel que $\omega^n = 1$.

Pour tout n entier strictement positif, il existe exactement n racines $n$ièmes complexes de l'unité :

$$\omega_n^k = e^{\frac{-2\pi ki}{n}} = \cos(\frac{-2\pi k}{n}) + i\sin(\frac{-2\pi k}{n}) \qquad k = 0, 1, ..., n-1$$

avec

$$i^2 = -1$$
$$e^{i\theta} = \cos(\theta) + i\sin(\theta) \quad \forall\, \theta \in \mathbb{R}$$

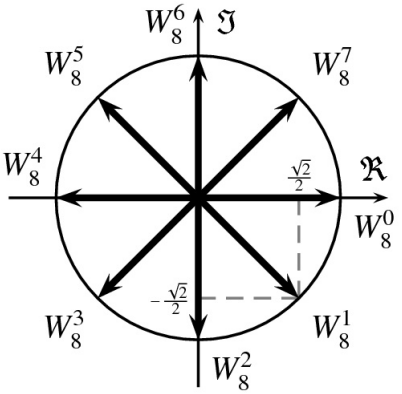


FIGURE 10.1 – Les 8 racines 8èmes de l'unité : $\omega_8^0, \omega_8^1, \omega_8^2, ..., \omega_8^7$.

La figure 10.1 illustre les 8 racines 8-èmes de l'unité. En fait, les racines $n$ièmes sont espacées de façon uniforme sur le cercle de rayon unitaire centré à l'origine dans le plan complexe.

La racine $n$ième principale de l'unité est $\omega_n = e^{\frac{-2\pi i}{n}}$, les autres racines étant des puissances de ω_n. Nous avons effectivement :

$$(\omega_n)^k = (e^{\frac{-2\pi i}{n}})^k = e^{\frac{-2\pi ki}{n}} = \omega_n^k$$

Ainsi $\omega_n^0, \omega_n^1, ..., \omega_n^{n-1}$ forment les n racines $n$ièmes de l'unité, parcourant le cercle unité du plan complexe dans le sens horloger.

Ces nombres complexes jouissent de propriétés intéressantes, pour tous entiers positifs n et k :

- Il y a exactement n racines $n$ièmes de l'unité car $\omega_n^k = \omega_n^{k \bmod n}$,
- Si n est pair, alors $\omega_n^{k+\frac{n}{2}} = -\omega_n^k$. En particulier, $\omega_n^{\frac{n}{2}} = -\omega_n^0 = -1$
- $\forall\, d > 0$: $\omega_{dn}^{dk} = \omega_n^k$. En particulier, $\omega_n = \omega_{kn}^k$. Donc toute racine $n$ième de l'unité est aussi une racine (nk)ième de l'unité.

10.5 La transformée de Fourier discrète

Ceci étant introduit, notre objectif était d'évaluer rapidement un polynôme $A(x)$ en une série de n valeurs x_k bien choisies. Ces n valeurs seront les n racines $n$ièmes de l'unité [2]. On définit les évaluations y_k, pour $k = 0, 1, ..., n-1$ par :

$$y_k = A(\omega_n^k) = \sum_{j=0}^{n-1} a_j \omega_n^{kj}$$

Le vecteur résultant $y = \langle y_0, y_1, ..., y_{n-1} \rangle$ définit la *transformée de Fourier discrète* (en abrégé de l'anglais : DFT). Les notations suivantes sont courantes :

$$y = DFT(a)$$
$$y_k = DFT_k(a)$$

10.6 La transformée de Fourier rapide

10.6.1 Algorithmes dans le paradigme séquentiel

La transformée de Fourier rapide (FFT) est en fait un algorithme pour calculer la transformée discrète qui exploite les propriétés des racines complexes de l'unité afin d'obtenir une complexité inférieure à l'algorithme naïf. La complexité en temps de l'algorithme, pour une implémentation séquentielle, s'exprime en $O(n \log n)$.

Dans la suite de ce texte, nous supposerons que n est une puissance de 2, c'est-à-dire qu'il existe un nombre naturel k tel que $2^k = n$. Ceci n'est en rien contraignant pour le polynôme $A(x)$ car il suffit d'annuler les nouveaux coefficients de poids supérieurs.

La FFT utilise une méthode *diviser-pour-régner*, en séparant les coefficients d'indice pair et les coefficients d'indice impair du polynôme $A(x)$. Les polynômes résultants, $A^{[0]}(x)$ et $A^{[1]}(x)$ sont d'un degré borné par $\frac{n}{2}$:

$$A^{[0]}(x) = a_0 + a_2 x + a_4 x^2 + ... + a_{n-2} x^{\frac{n}{2}-1}$$
$$A^{[1]}(x) = a_1 + a_3 x + a_5 x^2 + ... + a_{n-1} x^{\frac{n}{2}-1}$$

Nous obtenons l'égalité suivante :

$$A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2) \tag{10.2}$$

Il est ainsi possible de réduire l'évaluation de $A(x)$ en $\omega_n^0, \omega_n^1, ..., \omega_n^{n-1}$ aux étapes suivantes :

- évaluer les deux polynômes de borne deux fois plus petite $A^{[0]}(x)$ et $A^{[1]}(x)$ aux points
$$(\omega_n^0)^2, (\omega_n^1)^2, ..., (\omega_n^{n-1})^2 \tag{10.3}$$

2. Notons que ces valeurs sont bien toutes distinctes, par définition.

$(\omega_n^0)^2$	ω_n^0	$\omega_{n/2}^0$
$(\omega_n^1)^2$	ω_n^2	$\omega_{n/2}^1$
$(\omega_n^2)^2$	ω_n^4	$\omega_{n/2}^2$
...	...	...
$(\omega_n^{\frac{n}{2}-1})^2$	ω_n^{n-2}	$\omega_{n/2}^{\frac{n}{2}-1}$
$(\omega_n^{\frac{n}{2}})^2$	$\omega_n^{n \bmod n} = \omega_n^0$	$\omega_{n/2}^0$
$(\omega_n^{\frac{n}{2}+1})^2$	$\omega_n^{n+2 \bmod n} = \omega_n^2$	$\omega_{n/2}^1$
...	...	...
$(\omega_n^{n-2})^2$	$\omega_n^{2n-4 \bmod n} = \omega_n^{n-4}$	$\omega_{n/2}^{\frac{n}{2}-2}$
$(\omega_n^{n-1})^2$	$\omega_n^{2n-2 \bmod n} = \omega_n^{n-2}$	$\omega_{n/2}^{\frac{n}{2}-1}$

FIGURE 10.2 – Tableau illustrant la correspondance entre l'élévation au carré des racines $n$ièmes de l'unité et les racines $\frac{n}{2}$ièmes de l'unité.

– combiner les résultats obtenus sur base de l'équation 10.2 pour reconstruire l'évaluation de $A(x)$.

Cependant, la liste de valeurs énumérées en 10.3 est composée de doublons. Il s'agit en fait de $\frac{n}{2}$ racines $(\frac{n}{2})$ièmes de l'unité, chacune répétée exactement deux fois. Ceci peut aisément se déduire de la troisième propriété (énoncée en 10.4) des racines complexes de l'unité. Ce fait est illustré par le tableau 10.2.

Il faut donc évaluer les polynômes $A^{[0]}$ et $A^{[1]}$ avec les $\frac{n}{2}$ racines $\frac{n}{2}$ièmes de l'unité, c'est-à-dire $\omega_{n/2}^0, \omega_{n/2}^1, ..., \omega_{n/2}^{\frac{n}{2}-2}, \omega_{n/2}^{\frac{n}{2}-1}$.

Ainsi, les polynômes $A^{[0]}$ et $A^{[1]}$ sont évalués avec les mêmes valeurs. Ceci met en évidence la structure récursive du problème : les problèmes qui consistent à évaluer les polynômes $A^{[0]}$ et $A^{[1]}$ ont exactement la même forme que le problème de base, *i.e.* évaluer le polynôme A, tout en étant moitié moins grands. Cette construction mène à l'élaboration d'un algorithme récursif pour calculer la transformée de Fourier.

```
FFT-Recursive(a)
 1  n ← length(a)

 2  if n = 1
 3       return a

 4  ω_n ← e^{-2πi/n}
 5  ω ← 1
 6  a^[0] ← ⟨a_0, a_2, ..., a_{n-2}⟩
 7  a^[1] ← ⟨a_1, a_3, ..., a_{n-1}⟩
 8  y^[0] ← FFT-Recursive(a^[0])
 9  y^[1] ← FFT-Recursive(a^[1])

10  for k ← 0 to n/2 − 1
11       y_k ← y_k^[0] + ω y_k^[1]
12       y_{k+n/2} ← y_k^[0] − ω y_k^[1]
13       ω ← ω ω_n

14  return y
```

L'algorithme prend en entrée un vecteur a de coefficients complexes de taille n. En sortie, il retourne la transformée de Fourier discrète du vecteur a, représentée par le vecteur y. L'explication complète de l'algorithme récursif peut être déduite des propriétés énoncées ci-dessus. Pour une description complète pas à pas de l'algorithme, le lecteur se référera à [CLRS01].

Il existe une méthode qui permet de retrouver efficacement [3] les coefficients du vecteur a en partant du résultat y de la transformée. Cette méthode s'appelle la transformée de Fourier inverse. Sa description sort du cadre de ce document. Pour plus d'informations, le lecteur se référera également à [CLRS01].

Le temps d'exécution de la fonction récursive est calculé de la façon suivante :

$$\begin{aligned} T(n) &= 2\,T(\frac{n}{2}) + O(n) \\ &= O(n\,log\,n) \end{aligned}$$

D'un point de vue applicatif, ce qui nous intéresse vraiment est une implémentation réellement efficace, c'est-à-dire s'exécutant à la vitesse la plus grande possible, de la transformée de Fourier rapide.

Pour améliorer la vitesse d'exécution de l'algorithme, nous allons commencer par produire une version dérécursifiée, c'est-à-dire itérative, de l'algorithme. Ensuite, nous nous inspirerons de cet algorithme itératif pour créer un circuit théorique résolvant la transformée de Fourier. Il sera enfin très simple de passer de cette spécification théorique à un circuit en RTL.

Au début de chaque itération de la boucle commençant en ligne 10 de la fonction FFT-Recursive, la variable ω contient la valeur ω_n^k. Les lignes 11–12 réalisent

3. Il s'agit d'une interpolation qui fait l'hypothèse que les points x_k sont les racines de l'unité.

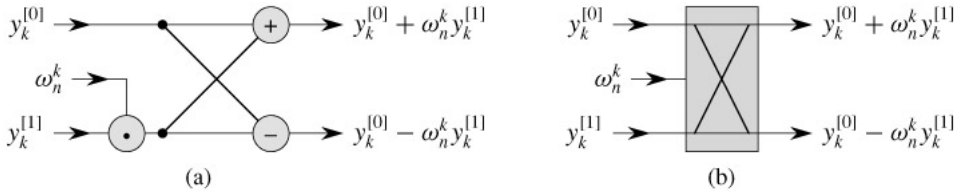


FIGURE 10.3 – L'opération papillon [CLRS01]. En (a) son implémentation et en (b) son schéma abstrait tel qu'il sera utilisé dans le circuit FFT parallèle.

l'évaluation de $A(x)$ conformément à l'équation 10.2. Ces assignations peuvent être groupées ensemble et exécutées en parallèle au sein d'un circuit sous la forme d'une seule opération appelée *opération papillon*. La figure 10.3 illustre ce circuit.

Notons que l'opération papillon ainsi définie réalise, si elle utilise le facteur $\omega_2^0 = 1$, la transformée de Fourier du vecteur à 2 éléments que sont les deux entrées. En effet, si $n = 2$ et $a = \langle a_0, a_1 \rangle$, on a :

$$DFT_k(a) = y_k = \sum_{j=0}^{1} a_j \omega_2^{kj} = a_0 + a_1 \omega_2^k$$

Or $\omega_2^0 = 1$ et $\omega_2^1 = \omega_2 = -1$, ainsi :

$$y_0 = a_0 + a_1$$
$$y_1 = a_0 - a_1$$

Ce qui correspond bien aux instructions ou au circuit de l'opération papillon.

Pour dérécursifier l'algorithme, il faut observer les appels récursifs en profondeur. A chaque appel récursif, l'algorithme groupe les vecteurs en fonction de la parité des bits des indices des éléments restants. Par exemple, pour $n = 8$, la première étape consiste à séparer les éléments $a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7$ en deux vecteurs $\langle a_0, a_2, a_4, a_6 \rangle$ et $\langle a_1, a_3, a_5, a_7 \rangle$. Les éléments sont groupés en fonction de la parité du dernier bit de la représentation binaire de l'indice. A l'appel récursif suivant, l'algorithme va séparer les éléments du vecteur $\langle a_0, a_2, a_4, a_6 \rangle$ en deux vecteurs $\langle a_0, a_4 \rangle$ et $\langle a_2, a_6 \rangle$, groupés selon la parité de l'avant-dernier bit. Et ainsi de suite.

Il est possible de câbler ces groupements à l'aide d'un algorithme très simple. Il mettra côte à côte dans le vecteur les éléments qui sont groupés ensemble. Par exemple, pour $n = 8$, il transformera le vecteur

$$\langle a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7 \rangle$$

en

$$\langle a_0, a_4, a_2, a_6, a_1, a_5, a_3, a_7 \rangle$$

Il suffit, élément par élément, de remplacer l'élément a_i par l'élément a_j, où j est l'entier représenté en binaire par l'image miroir de la représentation de i. La table 10.4 illustre ce principe.

i	bin(i)	inv(bin(i))	j = inv(i)
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

FIGURE 10.4 – Table de regroupement des éléments du vecteur a pour une taille $n = 8$.

La fonction COPY-REVERSE-BITS décrit la procédure pour réarranger le vecteur de cette façon pour un n quelconque. Le vecteur a est le vecteur origine (contenant les éléments dans l'ordre) et le vecteur A est le vecteur destination (qui contiendra les éléments a_k réarrangés après exécution de la fonction).

COPY-REVERSE-BITS(a, A)
1 $n \leftarrow length(a)$
2 **for** $k \leftarrow 0$ **to** $n - 1$
3 $\quad A_{\text{INV}(k)} \leftarrow a_k$

Cet algorithme s'exécute en $O(n \log n)$ instructions. En pratique, on connaît souvent à l'avance le nombre n de points, ce qui permet de câbler une table de correspondance construite à l'aide de cet algorithme. C'est le cas de l'implémentation de la FFT sous forme d'un circuit.

Ayant ainsi réarrangé l'ordre des éléments au sein du vecteur A, nous pouvons appliquer des opérations papillons sur les paires d'éléments adjacentes sur le vecteur afin d'effectuer l'étage le plus profond d'appels récursifs [4]. Ces $\frac{n}{2}$ opérations papillons donnent pour résultat $\frac{n}{2}$ DFT à 2 éléments, adjacentes dans le vecteur de résultats. On itère ensuite $\frac{n}{2}$ opérations papillons $log_2\, n$ fois pour obtenir, étape par étape, $\frac{n}{4}$ DFT à 4 éléments, $\frac{n}{8}$ DFT à 8 éléments, etc jusqu'à obtenir une [5] DFT à n éléments, ce qui n'est rien d'autre que le résultat recherché.

La fonction FFT-ITERATIVE implémente ce mécanisme, en travaillant en place sur le vecteur A. Les étages d'appels récursifs sont modélisés ici par un indice s allant de 1 à $log_2\, n$.

4. Juste avant l'appel récursif correspondant au cas de base.
5. Car, en effet, $\frac{n}{n} = 1$.

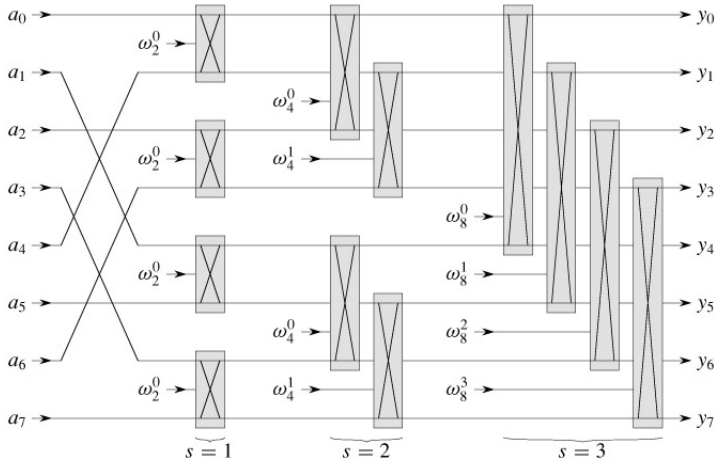


FIGURE 10.5 – Un circuit qui implémente la FFT en parallèle pour $n = 8$ [CLRS01]. Il utilise l'opération papillon.

FFT-ITERATIVE(a)
1 COPY-REVERSE-BITS(a, A)
2 $n \leftarrow length(a)$
3 **for** $s \leftarrow 1$ **to** $log_2\, n$
4 $\quad m \leftarrow 2^s$
5 $\quad \omega_m \leftarrow e^{\frac{-2\pi i}{m}}$
6 $\quad$ **for** $k \leftarrow 0$ **to** $n-1$ **by** m
7 $\quad\quad \omega \leftarrow 1$
8 $\quad\quad$ **for** $j \leftarrow 0$ **to** $\frac{m}{2} - 1$
9 $\quad\quad\quad A_{k+j} = A_{k+j} + \omega A_{k+j+\frac{m}{2}}$
10 $\quad\quad\quad A_{k+j+\frac{m}{2}} = A_{k+j} - \omega A_{k+j+\frac{m}{2}}$
11 $\quad\quad\quad \omega \leftarrow \omega\, \omega_m$

10.6.2 Algorithme dans le paradigme parallèle

En exploitant les caractéristiques de l'implémentation itérative de la transformée de Fourier rapide, un algorithme parallèle peut être décrit. Un tel algorithme est décrit par un circuit, composé de fils (les valeurs des variables) et de blocs de calculs (les opérations réalisées sur ces variables). Le circuit fige le nombre n de coefficients du vecteur d'entrée a à une constante.

L'algorithme est divisé en $1 + log_2\, n$ blocs. Le premier bloc imite la fonction COPY-REVERSE-BITS(a, A). Il ne s'agit en pratique que d'un reroutage des fils de l'entrée vers le bloc suivant. Chacun des $log_2\, n$ suivants est constitué de $\frac{n}{2}$ opérations papillons. Les blocs sont agencés correctement, en respectant les boucles imbriquées de l'algorithme séquentiel itératif FFT-ITERATIVE(a). Chacune des opérations papillons répétées sont exécutées en parallèle. La figure 10.5 représente le circuit pour un nombre de coefficient fixé $n = 8$. La figure 10.6 représente la structure récursive du circuit. Cette structure permet une généralisation de l'algorithme parallèle pour n quelconque puissance de deux.

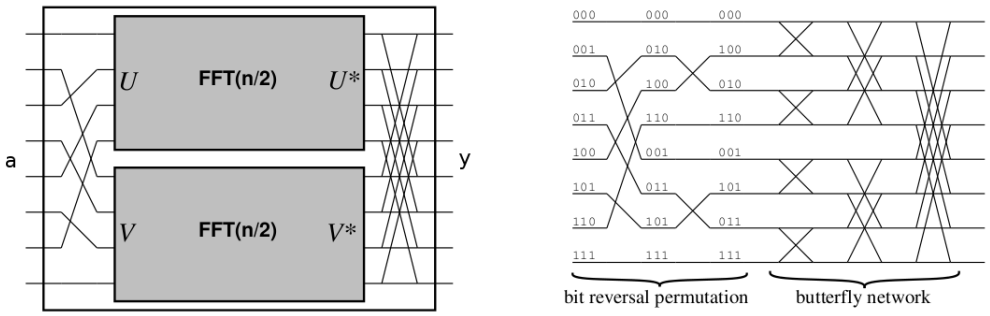


FIGURE 10.6 – Schéma mettant en évidence la structure récursive de l'algorithme FFT pour $n = 8$ [Eri03].

10.7 Conclusion

Nous avons présenté différentes versions de l'algorithme efficace qui permet de calculer la transformée de Fourier discrète : la transformée de Fourier rapide.

En particulier, nous avons décrit une version parallèlisée de la FFT. Cet algorithme parallèle peut être exploité pour la réalisation d'une implémentation RTL de la transformée de Fourier rapide. Dans le chapitre 11, deux implémentations RTL de la transformée de Fourier discrète, utilisant des techniques architecturales différentes, sont utilisées pour la campagne de simulations. Ces implémentations sont cependant basées sur cet algorithme, que nous devons principalement à Cooley et Tukey. A ce propos, voici un paragraphe intéressant tiré de l'ouvrage de Cormen *et al* [CLRS01] :

"*On attribue le plus souvent à Cooley et à Tukey l'élaboration de la FFT dans les années 1960. La FFT a, en réalité, été découverte de nombreuses fois auparavant, mais son importance n'a été pleinement comprise qu'avec l'avènement des ordinateurs. Press, Flannery, Teukolsky et Vetterling attribuent les origines de la méthode à Runge et à König en 1924, mais un article de Heideman, Johnson et Burrus fait remonter l'histoire de la FFT jusqu'à C.F. Gauss en 1805.*"

Il existe également d'autres implémentations de la FFT présentant des caractéristiques différentes en termes d'avantages et d'inconvénients. Nous pouvons citer [11] :

- l'algorithme de Bruun,
- l'algorithme de Rader,
- l'algorithme de Bluestein.

Chapitre 11

Déploiement et résultats

Au sein de ce chapitre, les détails d'une première « preuve du concept » de l'approche présentée au chapitre 9 seront expliqués. Le cas d'étude sélectionné pour celle-ci a été le déploiement d'une transformée de Fourier rapide (comme décrite en détails au chapitre 10) sur cibles FPGA (comme introduites dans la section 3.3.2 du chapitre 3).

Premièrement, afin d'être étudiés, certains choix de conception et certaines métriques de performances seront isolés et formalisés. Les différents degrés de liberté et mesures de performances seront définis méthodiquement. Ensuite, la méthode pour déployer la campagne de simulations sera décrite. Une fois l'objectif de création de modèles bien défini, une analyse sur les données issues des simulations sera effectuée. Enfin, le chapitre concluera sur cette analyse préliminaire, ses limites et les éventuelles améliorations qu'il est possible d'apporter à la méthode générale.

11.1 Cas d'étude et sélection des types de choix de conception et de métriques de performances

Nous allons étudier un cas d'étude simple et tenter d'en extraire un ensemble de modèles de coûts. Le cas d'étude choisi consiste au déploiement d'une transformée de Fourier rapide sur plate-forme reconfigurable de type FPGA.

L'extraction de modèles des performances se déroule en deux phases distinctes :

- exécution d'une campagne de simulations et récolte de données ;
- exploitation des données dans le but de créer des modèles.

Comme défini au chapitre 9, la campagne de simulations consiste à itérer sur les différents choix de design de haut niveau. Cependant, pour des raisons de délai et d'espace, nous allons restreindre l'approche au sein de ce mémoire à un espace de design relativement limité. Les choix sont partitionnés de la façon suivante :

la fonctionnalité : ou *« le design »*, les différentes variantes d'algorithmes et variantes d'implémentation de ces algorithmes, décrites au niveau RTL en langage VHDL.

le support cible : les différentes plates-formes FPGA envisagées pour supporter physiquement la fonction.

l'outil de synthèse : les différents outils utilisés pour synthétiser la fonctionnalité décrite en VHDL sur le support cible FPGA. Notons qu'on distingue la « synthèse » (convertir la description RTL en réseau de portes logiques) du « placement et routage » (réaliser le *mapping* des portes logiques sur les ressources du FPGA, fortement relié au support cible choisi).

Les outils de synthèse génèrent le *bitstream*, le code permettant de reconfigurer une plate-forme FPGA physique, mais également une série de rapport permettant d'extraire les paramètres de performances du système produit par la synthèse. Les paramètres qui seront extraits afin de faire des modèles prédictifs de ceux-ci sont les suivants :

la latence : ou temps de réponse, la durée en nanosecondes entre la première donnée entrée et la dernière donnée sortie. Dans notre cas d'étude, il s'agit du temps pour prendre un ensemble de points sur l'input, calculer sa FFT et placer le résultat sur l'output [1].

le nombre de LUT utilisées : le système déployé occupe une certaine surface sur le FPGA. Cette surface peut être caractérisée de plusieurs façons et l'une d'elle consiste à compter le nombre de LUT que le système occupe sur le FPGA.

la surface relative occupée : elle correspond au nombre de LUT utilisées par le système divisé par le nombre de LUT disponibles sur le support cible [2]. Une condition nécessaire pour que l'implémentation RTL puisse être déployée sur le support cible est que ce nombre soit inférieur à 1.

la puissance totale : la puissance totale consommée par le circuit résultant en milliwatts.

Les types de choix et de performances considérés, au même titre que le choix du cas d'étude, sont assez simples. Le but de la démarche au sein de ce mémoire n'est pas d'atteindre l'exhaustivité mais plutôt d'illustrer comment la démarche fonctionne, une sorte d'*illustration du concept*.

11.2 Déploiement de la campagne de simulations

Nous allons maintenant décrire les différentes variantes de choix opérés pour réaliser la campagne de simulations.

11.2.1 Choix de la fonctionnalité

Sur cinq versions étudiées de l'implémentation RTL de la FFT, deux ont été retenues comme fonctionnelles :

1. Le débit, qui serait ici caractérisé par le nombre de FFT réalisable par secondes, représente une autre mesure de performances. Comme expliqué en 3.2, elle n'est pas égale à l'inverse de la latence si l'architecture exploite des techniques de type *pipeline*. Il a été choisi de ne pas évaluer le débit dans le cadre de ce cas d'étude.

2. Cette définition ne s'applique que dans le cadre de ce cas d'étude. Elle a été choisie pour des raisons de facilité et il est possible de définir la surface relative de nombreuses façons plus ou moins précises.

fft_v1 : une version [7] réalisant l'algorithme de Cooley et Tukey, comme présenté en 10.6.2. Il s'agit d'une *« radix-2 »*, c'est-à-dire que l'opération de base est une opération papillon sur 2 entrées, exactement comme présentée en 10.6.2. Des degrés de liberté sont offerts au niveau de la taille des points (8, 16 ou 18 bits) et du nombre n de points à traiter (256, 512, 1024, 2046 ou 4096).

fft_v5 : une version [8] réalisant également l'algorithme de Cooley et Tukey, mais basée sur l'utilisation d'un coeur de calcul CORDIC [10]. Cette version-ci est une *« radix-4 »*. Le cas de base n'est plus une opération papillon à 2 entrées, mais à 4 entrées : il s'agit en fait d'une transformée de Fourier sur 4 points. Pour réaliser ce module, il suffit de « dérouler » le circuit parallèle pour $n = 4$, comme décrit en 10.6.2. Le même type de degré de libertés sont offerts dans cette version, en termes de taille des points (8, 16 ou 32 bits) et de nombres de points à traiter (64, 256, 1024, 4096, 16384). Notons que l'opération papillon en base 4 impose une contrainte plus forte sur le nombre n de points à traiter que l'algorithme Cooley et Tukey de base : il est nécessaire que ce nombre soit une puissance de 4.

Dans les deux cas, les nombres réels (les points) sont représentés en *virgule fixe*.

Afin d'analyser de façon neutre les premiers modèles de coûts générés, pas plus d'informations sur les choix architecturaux internes n'ont été analysées.

Les itérations au niveau des choix de technologies de calculs se feront donc pour le type d'implémentation, la taille des points et le nombre de points.

11.2.2 Choix de la cible

Trois plates-formes FPGA ont été prises en considération pour la campagne de simulations :

xc3s400-4ft256 : un *Xilinx Spartan3*, un circuit FPGA composé de 7168 LUT et ayant une fréquence d'horloge maximale de 50 MHz.

xc4vlx100-12ff1148 : un *Xilinx Virtex4*, un circuit FPGA composé de 98304 LUT et ayant une fréquence d'horloge maximale de 500 MHz.

xc5vlx330-2ff1760 : un *Xilinx Virtex5*, un circuit FPGA composé de 207360 LUT et ayant une fréquence d'horloge maximale de 500 MHz.

Notons que la famille des *Spartan* est commercialisé à un prix bien plus démocratique que celle des *Virtex*.

Les itérations au niveau des choix de support physique se feront sur ces trois cibles.

11.2.3 Choix de l'outil de synthèse

Au laboratoire *BEAMS*, essentiellement deux outils de synthèse sont disponibles : *Xilinx ISE* et *Mentor Graphics Precision*. Cependant, malgré de nombreux essais, des problèmes techniques ont rendu l'installation fonctionnelle de *Precision* impossible. Le seul outil de synthèse qui sera donc utilisé pour les simulations est *ISE*. Notons

qu'il sera également utilisé pour réaliser le placement et routage. L'information de puissance consommée est reccueillie à l'aide de l'outil *XPower Analyzer*.

Ce degré de liberté est donc fixé et il ne s'agit pas d'une variable sur laquelle nous pouvons itérer pour les simulations.

11.2.4 Récolte des données

Pour un ensemble de choix $\langle fonction, cible \rangle$ fixé, l'outil de synthèse (ici ISE) génére un ensemble de rapports contenant, notamment, les informations suivantes :

« Clock max delay (ns) » : la durée minimale théorique de la période de l'horloge du système. Elle est affichée dans le « Clock Report » d'ISE. Cette valeur est calculée en fonction de la fonctionnalité RTL synthétisée et du type de ressources présentes sur la cible FPGA. Cependant, elle ne tient pas compte de la limite de dimensionnement de l'horloge de la cible. Ainsi, si $target\_freq\_max$ dénote la fréquence maximale de la cible et $clock\_max\_delay$ dénote cette valeur minimale issue du rapport, la véritable période d'horloge du système est définie de la façon suivante :

$$T = \max\left(\frac{1}{target\_freq\_max}, clock\_max\_delay\right)$$

La véritable fréquence d'horloge du système s'obtient en inversant cette valeur :

$$f = \frac{1}{T}$$

« Total number of input LUTs » : le nombre total de *LookUp Tables* utilisées par la fonction. Il est affiché dans le « Device Utilization Summary » d'ISE. Il est également possible de récupérer dans le même rapport le nombre total de LUT disponible sur la cible FPGA choisie. La fraction d'aire utilisée sur la cible est calculée en divisant la première valeur par la deuxième.

« Total estimated power consumption (mW) » : la puissance totale consommée par le circuit. Elle est récupérée dans un rapport de *XPower Analyzer*. Cette valeur est calculée sur base d'une simulation sur un modèle de coût relativement abstrait : le calcul se base sur une moyenne de sollicitations aléatoires des ressources du système. Une valeur plus précise peut être calculée en procédant à une simulation avec *« back annotation »*. Pour des raisons de simplicité, la première méthode a été adoptée.

Si le nombre de LUT [3] et la puissance peuvent être récupérés directement, la période de l'horloge isolée ne signifie pas grand chose du point de vue du concepteur industriel. Afin de récupérer la latence (ou temps de réponse), il faut également connaître le nombre de cycles d'horloge nécessaires entre l'entrée du premier point et la

3. Le nombre d'« équivalents-portes » constitue une mesure plus générale de la surface que le nombre de LUT, dont la nature peut varier d'un type de plate-forme FPGA à l'autre. Malheureusement, cette mesure n'est pas disponible avec la version utilisée de l'outil de synthèse ISE.

Fonction	Nombre de points	Nombre de cycles d'horloge
fft_v1	256	1450
fft_v1	512	3119
fft_v1	1024	6708
fft_v1	2048	14393
fft_v1	4096	30782
fft_v5	64	265
fft_v5	256	1289
fft_v5	1024	6153
fft_v5	4096	28681
fft_v5	16384	131081

FIGURE 11.1 – Nombre de cycles d'horloge nécessaires pour réaliser une Transformée de Fourier complète en fonction du type d'implémentation considérée et du nombre de points à traiter.

sortie du dernier point [4]. Afin d'extraire cette valeur, des simulations fonctionnelles ont été exécutées à l'aide du logiciel *ModelSim*. Il a été mis en évidence que dans le cas des deux versions RTL de la FFT, cette valeur dépend uniquement du nombre de points à traiter. Le tableau 11.1 présente ces informations.

Pour obtenir le temps de réponse en nanosecondes, il suffit donc de multiplier la période d'horloge (le temps pour réaliser un cycle d'horloge complet) par le nombre de cycles d'horloge nécessaires pour calculer la FFT.

Afin de récolter les différentes données issues des rapports de simulation, des programmes, écrits dans le langage *Python*, intéragissent avec l'outil ISE utilisé en mode console. Des commandes pour pouvoir faire varier les paramètres génériques (le nombre de points et la taille des points) ont été prévues ainsi que des *parsers* pour extraire l'information des rapports en XML ou en texte brut. Le lancement de la campagne de simulation peut donc être réalisé en exécutant un seul programme en *Python*.

Le résultat de ces simulations est donné sous forme tabulaire dans l'annexe A.

11.3 Formalisation du problème

Appliquons la formalisation comme introduite en 6.1.2 et 9.2 à la sélection particulière des choix et des métriques de performances présentée ci-dessus.

Les variables d'entrées, ou *features*, sont représentées par le vecteur $x = \langle x_1, x_2, x_3, x_4 \rangle$. Certaines variables sont quantitatives (numériques), d'autres sont qualitatives (elles correspondent à des choix dans un ensemble fini) :

x_1 **:** la version de l'implémentation RTL de la FFT. On a $x_1 \in \{$*fft_v1*, *fft_v5*$\}$.

x_2 **:** la cible FPGA. On a $x_2 \in \{$*xc3s400-4ft256*, *xc4vlx100-12ff1148*, *xc5vlx330-2ff1760*$\}$.

4. Dans les deux versions RTL de la FFT étudiées, les données sont envoyées en série, point par point, coup d'horloge après coup d'horloge.

x_3 : la taille des points en nombre de bits. On a $x_3 \in \{8, 16, 18, 32\} \subset \mathbb{N}$. Ce paramètre peut donc être quantifié et ordonné. Cependant tous les éléments de l'ensemble ne sont pas accessibles par tous les designs RTL (x_1).

x_4 : le nombre de points à traiter. On a $x_4 \in \{64, 256, 512, 1024, 2048, 4096, 16384\} \subset \mathbb{N}$. Ce paramètre peut donc être quantifié et ordonné. A nouveau, tous les éléments de l'ensemble ne sont pas accessibles par les différents designs RTL (x_1).

Les variables de sorties sont représentées par le vecteur $y = \langle y_1, y_2, y_3, y_4 \rangle$. Ces variables sont toutes numériques :

y_1 : la latence, exprimée en nanosecondes.

y_2 : le nombre de LUT.

y_3 : la surface relative occupée par le design sur la cible. On a :

$$y_3 = \frac{y_2}{nb\_lut(x_2)}$$

La valeur de la fonction $nb\_lut(.)$ pour les différentes cibles est définie dans la section 11.2.2. Si le *mapping* du design sur la cible n'a pas échoué, on a $y_3 \in [0, 1]$.

y_4 : la puissance consommée, exprimée en milliwatts.

L'outil de synthèse établit une relation déterministe [5] entre le vecteur x des choix et le vecteur y des performances. En effet, deux synthèses effectuées avec les mêmes paramètres de fonctionnalité (code VHDL, taille et nombre des points identiques) et de plate-forme (cible identique) donneront toujours les mêmes performances. Le calcul effectué par l'outil ISE pour évaluer les performances est donc déterministe. Ce lien déterministe entre x et y peut être modélisé par la fonction $f(.)$ telle que $y = f(x)$, ou en séparant les différentes métriques de performances :

$$\begin{cases} y_1 &= f_1(x_1, x_2, x_3, x_4) \\ y_2 &= f_2(x_1, x_2, x_3, x_4) \\ y_3 &= f_3(x_1, x_2, x_3, x_4) \\ y_4 &= f_4(x_1, x_2, x_3, x_4) \end{cases}$$

Nous pouvons également écrire de façon plus compacte :

$$y_l = f_l(x_1, x_2, x_3, x_4) \qquad l = 1, 2, 3, 4$$

Après la campagne de simulations, nous disposons d'une série de réalisations (ou *records*) des variables $\langle x, y \rangle$. Cet ensemble de données est appelé *dataset*.

La recherche d'un modèle des performances consiste alors à définir les fonctions $\hat{f}_l(.)$ telles que $\hat{y}_l = \hat{f}_l(x)$. Pour que le modèle ait un certain pouvoir prédictif, l'évaluation des fonctions $\hat{f}_l(.)$ en les points x du *dataset* doit donner des valeurs $\hat{y}_l$ qui soient proches des valeurs y_l correspondantes du *dataset*. L'écart entre $\hat{y}_l$ et y_l est appelé l'erreur de modélisation. Il existe plusieurs façons d'agréger cette erreur

5. Les outils de synthèse étant des logiciels propriétaires, ce fait n'a pu être vérifié que de façon empirique.

en une valeur unique commune à tout le *dataset*. Cette valeur est alors utilisée pour estimer la qualité du modèle produit.

Le travail qui sera effectué ici est essentiellement une analyse préliminaire de la faisabilité de l'approche. En d'autres termes, nous regardons s'il est possible d'appliquer la méthodologie de l'inférence de modèles propre au domaine du *machine learning* à la conception microélectronique et donc aux paramètres et outils qui lui sont reliés. C'est pourquoi nous restreindrons notre approche à l'inférence de *modèles linéaires*, qui est l'approche la plus évidente pour commencer à explorer cette idée. Ceci signifie que les différentes fonctions $\hat{f}_l(.)$ seront en définitive toutes réduites à une combinaison linéaire des entrées[6]. L'étude de l'applicabilité de techniques d'inférence de modèles non-linéaires pourrait éventuellement faire l'objet de travaux ultérieurs à ce mémoire.

11.4 Analyse des données et inférence de modèles

Une fois les données récoltées, nous pouvons tenter d'inférer un modèle visant à prédire les différents critères de performances sur base d'une ou plusieurs des variables d'entrée.

Afin de réaliser cette analyse, le langage *R* a été utilisé. Les scripts utilisés sont fournis au sein de l'annexe C

Nous commencerons par observer le tableau de données pour éventuellement retirer certaines valeurs aberrantes. Ensuite, nous expliquerons notre démarche d'analyse sur base de la première métrique de performances sélectionnée : la latence. Les résultats pour les autres critères seront ensuite présentés. Enfin, nous conclurons sur cette démarche en précisant sa limite et les éventuels travaux ultérieurs envisageables.

11.4.1 Filtrage des données

Certaines implémentations ont échoués lors du *mapping* du RTL synthétisé sur la plate-forme cible. Ces échecs ont généralement pour cause une surface du circuit trop grande, dépassant la totalité des ressources de la cible FPGA. Lorsqu'une synthèse échoue, il est impossible de déterminer la latence et la puissance consommée étant donné qu'il n'a pas été possible de créer le système final.

Les *records* correspondant aux implémentations ayant échoué ont les champs correspondants à la latence et à la puissance fixés à la valeur *NaN* (Not a Number). En termes de métriques, il ne reste donc plus que l'information relative à la surface.

Lorsqu'on observe le tableau de données récoltées (disponible en annexe A), nous pouvons observer une explosion du nombre de portes lorsque les synthèses échouent. Afin d'éviter d'introduire une trop grande erreur de modélisation au sein des modèles linéaires, nous retirerons les *records* des synthèses échouées du *dataset* (ceux-ci représentent pour la plupart des valeurs extrêmement élevées).

La question de déterminer comment intégrer l'information de réussite ou d'échec de la synthèse au sein des modèles reste ouverte.

6. Par exemple, dans le cas où le modèle ne prendrait qu'une variable d'entrée, le modèle serait représenté par une droite.

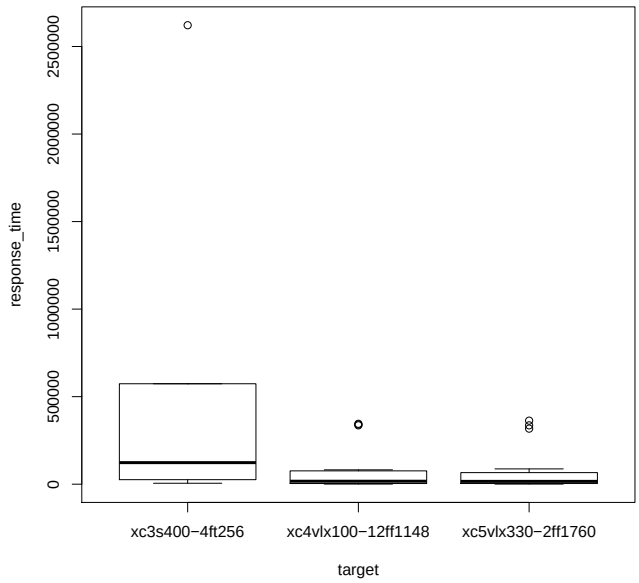


FIGURE 11.2 – Statistiques descriptives séparant les différentes cibles.

11.4.2 Analyse descriptive

Une première chose à faire lorsqu'on est confronté à un ensemble de données statistiques est de se familiariser avec les ordres de grandeurs de celles-ci.

Les informations agrégées de base sur l'ensemble des *records* peuvent être obtenues facilement :

```
> summary(dataset$response_time)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  609.2  5300.0 18030.0 113600.0 79480.0 2622000.0
```

Pour observer ces informations en fonction du design RTL ou de la plate-forme cible FPGA, nous pouvons utiliser les graphiques *boxplots* [9] des figures 11.2 et 11.3. La médiane est représentée par le trait en gras, le premier (resp. troisième) quartile par la limite inférieure (resp. supérieure) de la boîte. Les valeurs minimales et maximales sont représentées par la limite des prolongements inférieurs et supérieurs de la boîte. Certaines valeurs aberrantes peuvent être identifiées et représentées par des points isolés sur le graphique.

Pour se donner un peu plus d'intuitions sur la comparaison fonction/plate-forme, nous pouvons observer les graphiques d'interactions des figures 11.4 et 11.5. Ici, seule la moyenne est représentée par un point, mais les comparaisons entre les différentes variables qualitatives d'entrée (design VHDL et cibles FPGA) sont aisées.

Notons que la *fft_v1* n'apparaît pas sur le Spartan3 : en effet, la synthèse a échoué pour toutes les variations en termes de nombres de points et de tailles de points.

Jusqu'ici, nous n'avons pas considéré les différences en termes des variables quantitatives, c'est-à-dire le nombre de points et la taille des points.

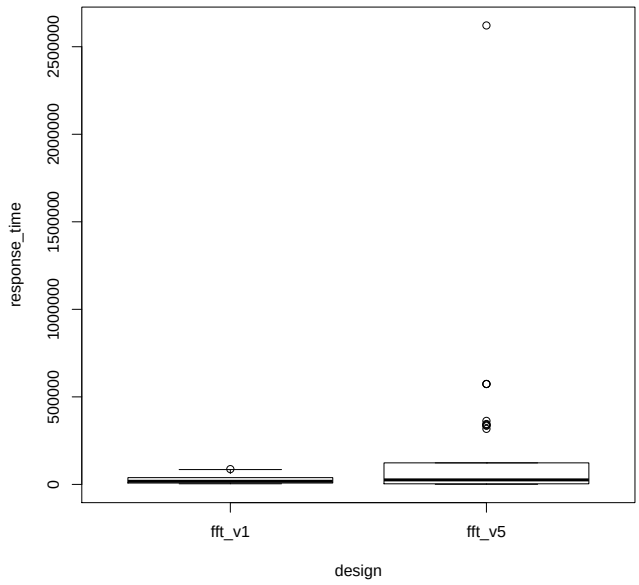


FIGURE 11.3 – Statistiques descriptives séparant les différents designs VHDL.

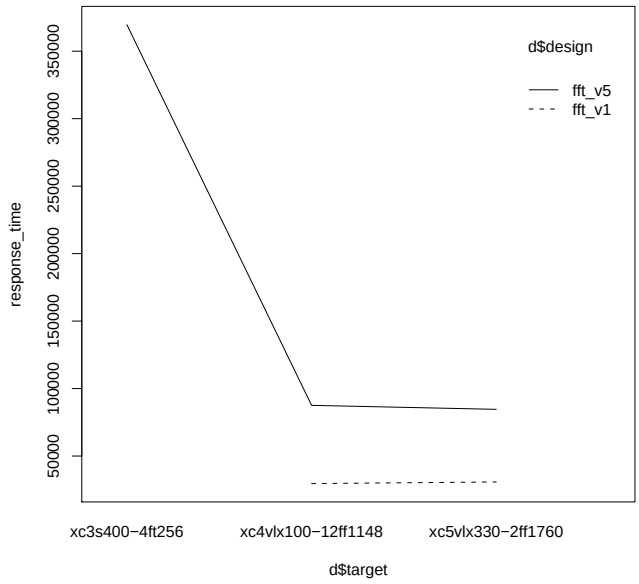


FIGURE 11.4 – Diagramme d'interactions entre les variables qualitatives, point de vue des cibles FPGA.

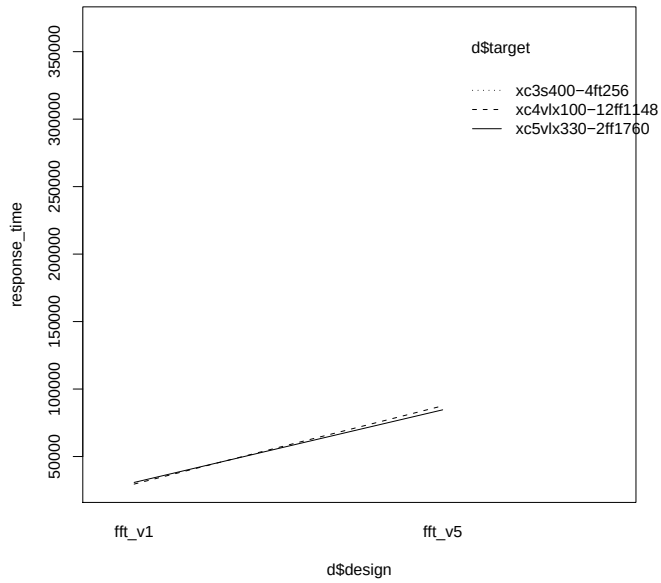


FIGURE 11.5 – Diagramme d'interactions entre les variables qualitatives, point de vue des designs VHDL.

Comme nous sommes confrontés à certaines variables purement qualitative [7] (x_1 et x_2), nous ne pouvons pas appliquer directement les techniques de régression linéaires. Pour pallier à ce problème, nous avons envisagé deux approches différentes : l'inférence de modèles par stratification des variables d'entrées d'une part, et par arbres de régression d'autre part. De plus, les modèles pour les arbres de regression seront créés progressivement à l'aide d'une procédure de *feature selection* (ou sélection de variables de l'entrée). Chaque génération de modèle linéaire exploite la technique bien connue de régression aux moindres carrés, qui consiste à minimiser la somme des carrés des erreurs des différentes prédictions. Pour rappel, l'erreur de modélisation d'une prédiction $\hat{y}$ en un point du *dataset* s'obtient en calculant la différence entre cette prédiction et la valeur du paramètre y du *dataset*.

11.4.3 Inférence de modèles par stratification des variables d'entrées

La première approche pour analyser les données consiste à *stratifier* le *dataset*, c'est-à-dire à grouper les données. Les données seront séparées en groupes en fonction de la valeur des variables catégorielles. Pour chacun de ces sous-ensembles du *dataset*, un modèle de régression linéaire indépendant sera construit. Chacun de ces modèles linéaires sera une fonction $g : \mathbb{R}^2 \mapsto \mathbb{R} : g(x_3, x_4) = \beta_0 + \beta_1 x_2 + \beta_2 x_4$ telle que $\forall i = 0, 1, 2 : \beta_i \in \mathbb{R}$.

Il existe différentes façons de stratifier les données :

– Par design et par plate-forme, un modèle linéaire pour chaque combinaison

7. Ou catégorielles.

Type de stratification	NMSE
Par designs et par cibles	0.01491196
Par cibles uniquement	0.01488387
Par designs uniquement	0.8545365

FIGURE 11.6 – Tableau permettant la comparaison des différentes techniques de stratification pour la latence.

d'RTL et de cibles. En l'occurrence, une de ces combinaisons ayant échoué (fft_v1 sur Spart3), cette stratification permettra l'inférence de 5 modèles.

- Par plate-forme, les modèles générés faisant alors abstraction de l'implémentation RTL en mélangeant les données issues des différents designs.
- Par design, les modèles générés faisant alors abstraction de la plate-forme FPGA cible en mélangeant les données synthétisées sur les différents designs.

Cette approche permet de déterminer si la connaissance du design ou de la plate-forme (de façon indépendante) permet une bonne prédiction ou si les deux informations sont nécessaires pour obtenir une prédiction correcte.

Pour s'en convaincre, nous pouvons :

- observer les graphiques,
- utiliser un critère basé sur l'évaluation de l'erreur. Dans les programmes R d'analyse de données est calculée, pour chaque type de stratification, une valeur globale d'erreur afin de les comparer. Cette valeur est appelée NMSE [8]. Plus cette valeur sera proche de 0, plus l'erreur globale sera faible et, par conséquent, meilleur sera le modèle en termes de capacité de prédiction. Si cette valeur s'écarte de 0, la prédiction sera plus mauvaise. Si elle dépasse 1, le modèle sera de très mauvaise qualité. Nous pouvons ainsi comparer les différents types de stratification à l'aide de cette valeur. Le tableau 11.6 illustre ces valeurs pour les différents types de stratification donnés ci-dessus dans le cadre de l'étude de la latence.

Dans le cas de la latence, nous observons que c'est la cible considérée de façon indépendante qui permet la meilleure qualité de prédiction. Par contre, dans ce cas, le design à lui seul ne permet qu'une très mauvaise prédiction de la latence. Ces considérations basées sur l'argument numérique de la valeur de la NMSE sont confirmés par la lecture de graphiques d'exemples sur les figures 11.7 et 11.8. Ce type de graphique se lit de la façon suivante : les prédictions, issues des modèles linéaires, pour les différents *records* en abcisse, sont représentées par les boules vides. Les valeurs réelles des points, issues des *records* correspondants, sont représentés par les points reliés par les lignes noires. Nous observons donc sur ces exemples que les écarts entre la prédiction et les *records* sont plus petits dans le cas de la stratification par cibles que dans le cas de la stratifcation par designs.

8. Pour *« Normalized Mean Square Error »*, la valeur moyenne des erreurs de prédiction normalisée par rapport à la variance des données de sortie y. Ici, l'erreur ponctuelle est calculée en *leave-one-out*, c'est-à-dire l'écart entre $\hat{y}$ et y si le point $\langle x, y \rangle$ n'avait pas été considéré pour la construction du modèle.

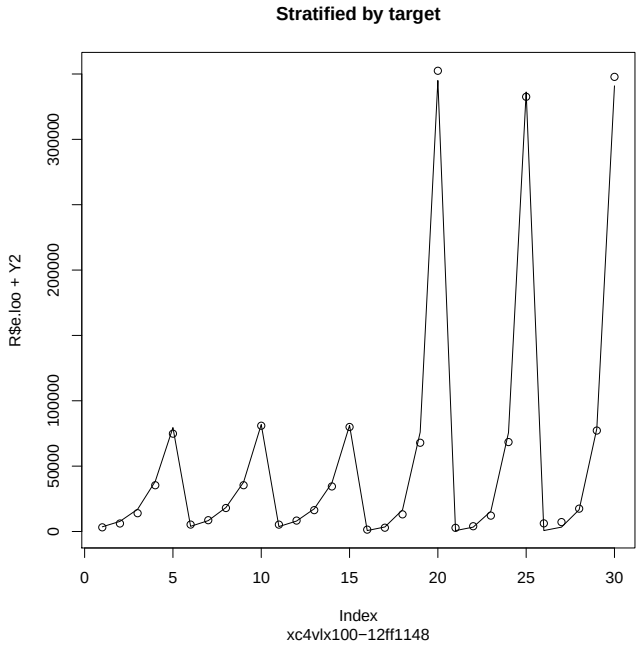


FIGURE 11.7 – Graphique illustrant l'écart entre les données réelles de temps de latence et les données prédites par le modèle stratifié par cibles pour le Virtex4.

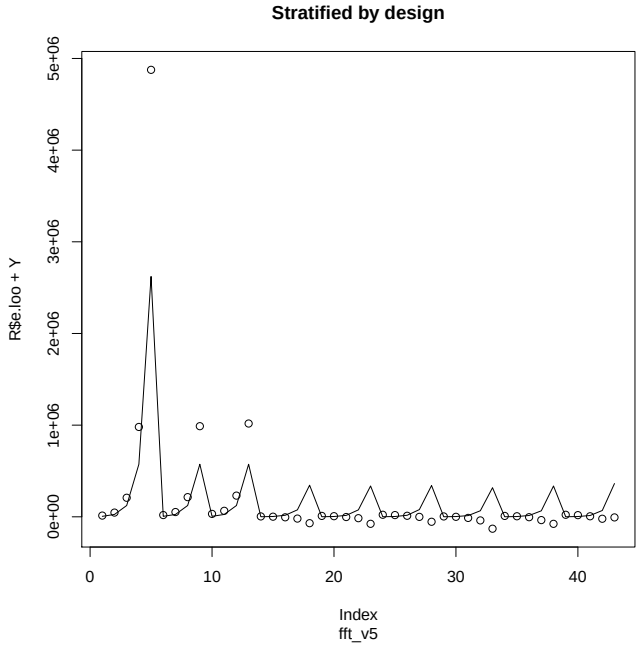


FIGURE 11.8 – Graphique illustrant l'écart entre les données réelles de temps de latence et les données prédites par le modèle stratifié par designs pour la fft_v5.

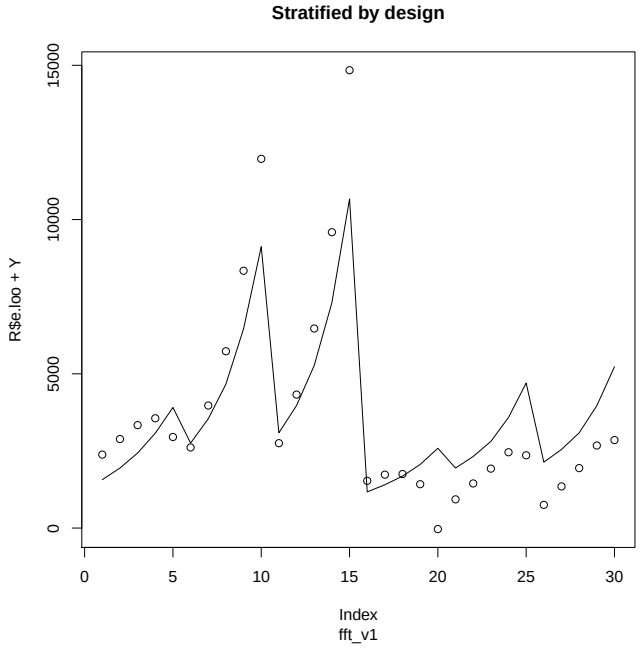


FIGURE 11.9 – Graphique illustrant l'écart entre les données réelles de nombre de LUT et les données prédites par le modèle stratifié par designs pour la fft_v1.

Par contre, les résultats au niveau du nombre de portes logiques indiquent des informations opposées : le design semble être le facteur permettant le plus de bonnes prédictions. Les figures 11.9, 11.10 et 11.11 l'illustrent graphiquement. Cependant, dans ce cas-ci, la combinaison des deux informations, cible et design, permet de prédire plus précisément que lorsqu'on stratifie selon un seul facteur. Enfin, remarquons que globalement, la précision est moins bonne que dans l'analyse de la latence. En effet, la meilleure valeur de NMSE pour le nombre de LUT est de 0.1438274 (pour la stratification par designs et par cibles) alors qu'elle est de 0.01491196 pour la latence (dans le même cas de stratification).

Les conclusions que nous pouvons tirer sur l'occupation relative de la surface et sur la puissance consommée sont du même ordre :

- la surface relative est assez bien prédite par la cible uniquement (NMSE = 0.2220138), et très mal prédite par le design uniquement (NMSE = 1.021376) ;
- la combinaison des deux informations, dans le cas de la surface relative, améliore la qualité de la prédiction (NMSE = 0.1798977) ;
- la puissance consommée est relativement bien prédite par la cible uniquement (NMSE = 0.5542299) et mal prédite par le design (NMSE = 1.126122) ;
- à nouveau, la combinaison des deux informations permet une amélioration de la qualité de prédiction de la puissance consommée (NMSE = 0.2585833).

En fait, le seul cas où l'utilisation des deux informations dans la stratification dégrade la qualité du modèle est la latence (la NMSE passe de 0.01488387 pour la cible uniquement à 0.01491196 pour la cible et le design). Cependant, cette variation est très faible : elle est inférieure à 1%. Notons qu'ici, la NMSE caractérise la qualité

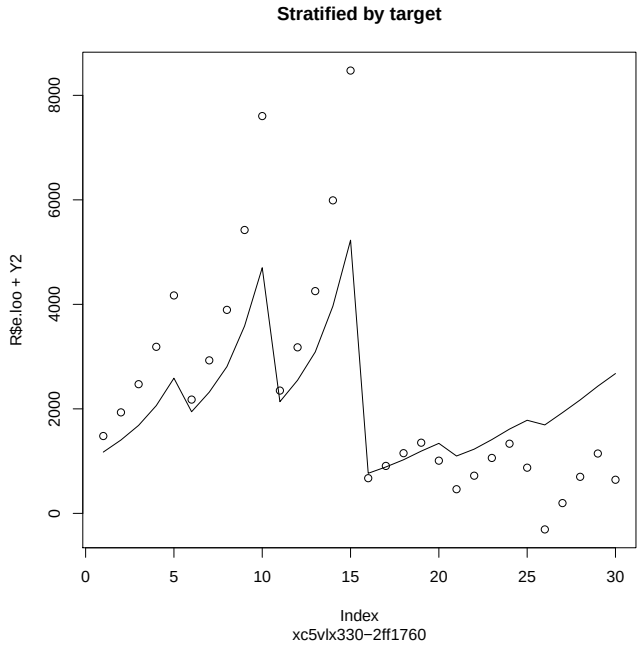


FIGURE 11.10 – Graphique illustrant l'écart entre les données réelles de nombre de LUT et les données prédites par le modèle stratifié par cibles pour le Virtex5.

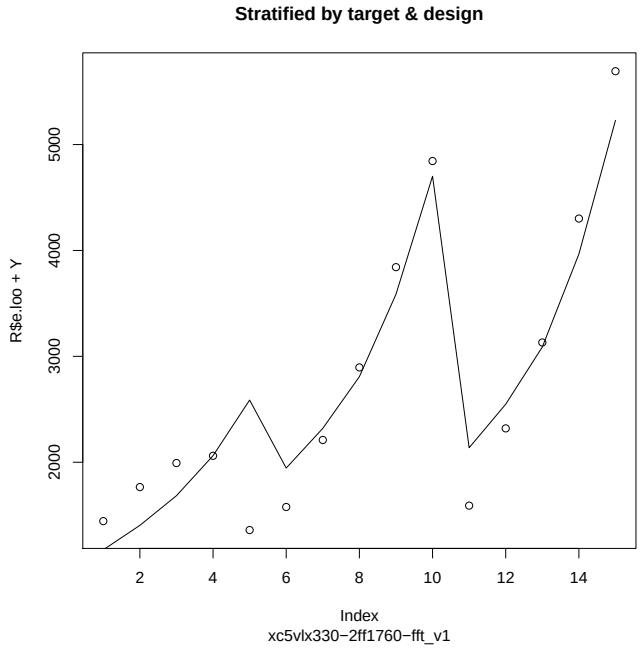


FIGURE 11.11 – Graphique illustrant l'écart entre les données réelles de nombre de LUT et les données prédites par le modèle stratifié par cibles (Virtex5) et par designs (fft_v1).

d'un type de stratification pour un critère de sortie donné, pas seulement d'un modèle isolé.

Le lecteur intéressé par réaliser une analyse plus fine peut se référer à l'annexe B, qui contient le tableau de valeurs des NMSE pour les différentes métriques ainsi que tous les graphiques générés pour l'analyse descriptive sommaire et l'analyse par stratifications pour toutes les métriques de performances $(y_1, ..., y_4)$ étudiées.

11.4.4 Inférence de modèles par arbres de régression

Lors de l'analyse par stratification, nous avons évalué la pertinence des variables catégorielles (design et cible) en termes de qualité de prédiction des différentes métriques des performances sans remettre en question la pertinence des variables numériques (nombre de points et taille de ces points). La seconde technique présentée ici permet de mettre toutes les variables x_i sur un même pied d'égalité. Nous allons construire un modèle généralisant les notions de stratifications présentées au point précédent.

La première étape de la construction d'un tel modèle consistera en une *feature selection* (ou « sélection de variables d'entrée »). Cette procédure consiste à choisir quels sont, parmi les 4 variables x_i, celles qui contiennent le plus d'informations à propos de l'output y_l considéré. En d'autres termes, elle consiste à sélectionner le sous-ensemble des variables menant au modèle de régression linéaire ayant la NMSE minimale. Une méthode naïve consisterait à énumérer tous les sous-ensembles de l'ensemble des variables x_i, et pour chaque sous-ensemble, inférer un modèle linéaire, déterminer la NMSE associée et sélectionner le sous-ensemble ayant la NMSE minimale. Cependant, le temps pris par cette méthode augmente exponentiellement avec le nombre de variables (il y a exactement 2^n sous-ensembles pour n variables). Nous avons ainsi choisi d'adopter une heuristique ne conduisant pas forcément à la solution optimale mais ayant des résultats satisfaisants. Cette heuristique est nommée *forward selection*. Elle consiste à ajouter une à une les variables d'entrée dans le modèle linéaire, en sélectionnant à chaque itération la variable qui fait baisser le plus la NMSE du modèle. Si lors de l'ajout d'une variable, la NMSE monte [9] ou varie de façon négligeable, la procédure s'arrête.

Pour gérer simultanément les variables numériques et catégorielles, nous utiliserons un *arbre de régression*. Un arbre de régression permet la division de l'espace d'entrée (l'ensemble des valeurs possibles du vecteur x) en régions mutuellement exclusives. Dans le cas où nous sélectionnons toutes les variables, il s'agit d'énumérer les solutions possibles des variables catégorielles x_1 et x_2. Les noeuds internes effectuent la classification des variables catégorielles, tandis que les noeuds feuilles (il y en a 6 ici, incluant un modèle vide pour la synthèse ayant échoué) contiennent chacun un modèle linéaire à 2 dimensions (corespondantes aux variables numériques x_3 et x_4), exactement comme dans l'analyse par stratifications.

Pour généraliser, pour un sous-ensemble donné de variables sélectionnées, le modèle est construit de la façon suivante :

- pour l'ensemble des variables catégorielles sélectionnées, toutes les valeurs possibles de cet ensemble de variables sont énumérées au sein de l'arbre de régression.

9. Dans ce cas, l'overfitting se manifeste.

A chaque niveau de l'arbre est décidé la valeur d'une des variables catégorielles ;
- pour chaque combinaison des variables catégorielles sélectionnées, un modèle linéaire est créé avec les variables numériques sélectionnées.

Ainsi construit, le modèle par arbre de régression permet de prédire la performance de n'importe quelle réalisation de la variable x. Ceci généralise bien les modèles construits par stratification de l'espace de valeurs de x. Notons que lorsqu'on sélectionne les valeurs de la même façon que dans les modèles stratifiés[10], nous retrouvons les mêmes valeurs de NMSE (les modèles étant identiques).

Le résultat de l'exécution du script R permettant la sélection de variables et l'inférence de modèles par arbre de régression figurent sur le tableau de la figure 11.12. Une croix dans la dernière case indique le modèle de meilleure qualité en termes de prédiction, et donc celui à choisir pour minimiser l'erreur de modélisation. Les résultats du tableau confirment les observations de l'analyse par stratification, tout en généralisant le raisonnement à l'ensemble des variables d'entrée.

11.4.5 Interprétation physique des résultats

Nous allons maintenant discuter l'importance des variables pour les différentes métriques de performances, et tenter d'interpéter, autant que faire se peut, la pertinence des variables sélectionnées pour prédire les différentes métriques de performances.

La latence : le nombre de points conditionne de façon déterministe le nombre de cycles horloges qu'utilise le système pour calculer une FFT. Il est donc logique qu'il soit noté comme premier critère. Les fréquences maximales des cibles FPGA étant fortement différentes (de 50 à 500 MHz), il est intuitif d'imaginer que la cible conditionne fortement et permet de prédire cette latence. Le design est listé comme avant dernier critère. Etant donné qu'il n'y a que deux designs différents testés, que les algorithmes sous-jacents à la description RTL sont relativement proche (tous les deux issus de l'algorithme de Cooley et Tukey), et que, par définition de la latence, nous ne considérons pas les spécificités du design en termes de *pipeline*, il est normal que le design influe peu sur les performances.

Le nombre de LUT : ici par contre, le design est en première place en termes de pouvoir de prédiction. Ceci s'explique par le fait que le nombre absolu de ressources utilisées est fortement influencé par le degré de parallélisation de la description RTL du circuit. Les deux versions du design ont sans doûte des spécificités qui les différencie suffisamment à ce niveau. La taille des points joue le rôle secondaire : en effet, la taille des points conditionne la taille des unités de traitement et de mémoire à prévoir pour accueillir la fonctionnalité. Ces unités de traitement et de mémoire sont reconverties en LUT lors de la synthèse.

la surface relative : la cible influence de façon majeure la capacité à prédire surface du circuit. A nouveau, la surface relative est construite sur base d'un paramètre intrinsèque à la cible FPGA (le nombre total de LUT), cette conclusion est donc intuitive.

10. C'est-à-dire sélectionner les variables x_3 et x_4 ainsi que l'une, l'autre ou les deux variables x_1,x_2.

Métrique	Sélection	NMSE	Choix
Latence	Nb. points	0.8345224	
Latence	Nb. points Cible	0.01372921	×
Latence	Nb. points Cible Design	0.01379779	
Latence	Nb. points Cible Design Taille pts.	0.01491196	
LUT	Design	0.7134792	
LUT	Design Taille pts.	0.5327016	
LUT	Design Taille pts. Nb. points	0.3114987	
LUT	Design Taille pts. Nb. points Cible	0.1438274	×
Surface relative	Cible	0.1789132	
Surface relative	Cible Taille pts.	0.06980224	
Surface relative	Cible Taille pts. Design	0.05029981	×
Surface relative	Cible Taille pts. Design Nb. points	0.1798977	
Puissance consommée	Cible	0.05723294	
Puissance consommée	Cible Nb. points	0.05632752	×
Puissance consommée	Cible Nb. points Taille pts.	0.5542299	
Puissance consommée	Cible Nb. points Taille pts. Design	0.2585833	

FIGURE 11.12 – Tableau illustrant les étapes de la procédure *forward selection* et choix final du sous-ensemble de variables utilisées inférant un modèle modèle ayant la plus petite NMSE parmi les sous-ensembles explorés.

la puissance consommée : elle est conditionnée principalement par la cible. Nous avons discuté plus haut de la précision de la simulation pour le calcul de puissance. Nous pourrions éventuellement cerner des informations plus précises en réalisant une simulation se basant sur un modèle plus précis du système (*back annotation*).

11.5 Conclusion

Au long de ce chapitre, nous avons réalisé une campagne de simulations afin de récolter un ensemble de données liant les choix de conception haut niveau avec les évaluations de différentes métriques de performances issues de l'outil de synthèse.

Afin de créer un modèle de haut niveau, nous avons appliqué une des techniques les plus simples d'apprentissage supervisé : la modélisation linéaire par approximation aux moindres carrés. Afin de gérer les variables catégorielles que représentaient le choix des designs RTL et des cibles FPGA dans un ensemble fini de valeurs incomparables, nous avons, pour créer des modèles, d'une part stratifié les données, d'autre part associé les modèles linéaires avec un arbre de régression.

Nous sommes arrivé à d'intéressants résultats en termes d'erreur de modélisation. Pour chaque métrique de performances analysée, nous sommes parvenu à inférer un modèle tel que la NMSE est inférieure à 0.15. Ceci en sachant que l'erreur en un point est toujours calculée en *« leave-one-out »*, c'est-à-dire en calculant le modèle tel qu'il aurait été s'il avait été généré avec l'ensemble du *dataset* sauf ce point dont on calcule l'erreur.

Un concepteur peut, grâce à ces modèles, faire le lien entre l'espace de conception et l'espace des performances où apparaissent les courbes de Paréto présentées en 6.1. Pour donner un exemple, la figure 11.13 représente un graphique où est affiché l'ensemble des points du *dataset* en mettant en vis à vis la latence du système et la surface en termes de nombre de LUT. Nous observons le nuage des points des solutions en fonction de deux critères de performances. Notons que cette représentation des données fait abstraction des choix de design. Il ne s'agit pas d'un graphique directement adapté à l'optimisation multicritère, étant donné que certains choix de design sont conditionnés par la spécification du cahier de charges (*e.g.* le nombre de points).

Clairement, en l'état, les modèles ne sont pas utilisables tels quels, que ce soit en vue d'une intégration dans Nessie ou par une utilisation directe des concepteurs. L'exercice réalisé au sein de ce chapitre visait plutôt une illustration du concept présenté au chapitre 9 qu'une application directe. Pour les rendre exploitables, plusieurs choses doivent encore être travaillées.

Une première remarque concerne les variables catégorielles. Jusqu'ici, nous avons considérés qu'un « choix de conception » consistait à prendre une valeur dans un ensemble fini, comme par exemple l'une des deux versions RTL considérées de la FFT. Dans une optique de prédiction de performances, il serait préférable de pouvoir quantifier en termes de variables numériques les choix de conception. Par exemple, un outil de *profiling* pourrait être exécuté sur le code RTL au niveau de la simulation fonctionnelle. Cet outil, qui prendrait le code en entrée, fournirait un ensemble de paramètres numériques caractérisant les choix de conception et permettant d'inférer

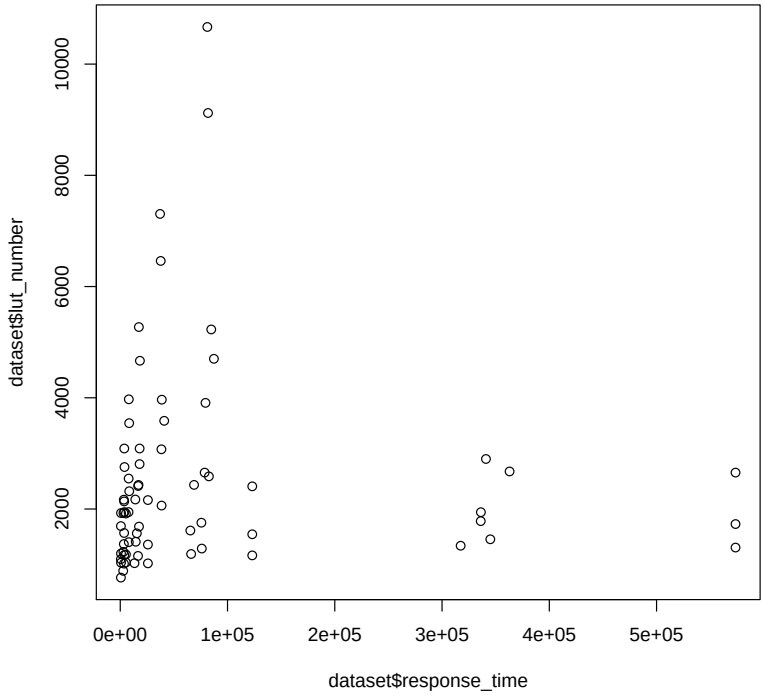


FIGURE 11.13 – L'ensemble du *dataset* affiché en mettant en vis à vis latence et surface occupée.

des modèles sans passer par des techniques de stratification ou d'arbres de régression. Cette méthode présenterait l'avantage suivant : si un nouveau design se présente (par exemple, fft_v6), nous pourrions lancer l'outil de *profiling* sur celui-ci afin d'extraire les paramètres numériques et utiliser ensuite les modèles déjà construits dans le but de prédire la performance sans passer par la campagne de simulations. La méthode de *profiling* a déjà été exploitée dans le cadre d'analyse des performances d'algorithmes de traitement vidéo déployés sur processeurs DSP [BK02].

Une deuxième remarque concerne l'interprétation physique tirée du processus de sélection de variables. Les conclusions qui ont pu être tirées ici sont essentiellement qualitatives et évidentes [11]. Il pourrait être intéressant de pouvoir, sur base des modèles, tirer des conclusions quantitatives et difficiles, voire impossible, à extraire sans les modèles [12]. Il est également nécessaire de raffiner les mesures effectuées, en tenant compte de critères comme le débit d'exécution.

Ensuite, il est important de rappeler qu'une analyse statistique ne peut se faire sans un nombre considérable de données à disposition, notamment si cette analyse est réalisée dans le but d'inférer un modèle. Le *dataset* présenté ici est relativement limité, et il sera nécessaire, si des travaux sont poursuivis dans le domaine, de collecter encore de nombreuses données issues de synthèses d'outils différents, d'algorithmes et de variantes différentes, de cibles différentes, etc. Plus les données seront nombreuses, plus les modèles seront d'une grande qualité de prédiction.

Enfin, la « procédure d'apprentissage » ne peut se limiter à un seul type de modèles. En effet, pour trouver le modèle qui prédit le mieux les données, il est nécessaire d'envisager de nombreuses approches et techniques différentes. En particulier, il existe également des méthodes non-linéaires qu'il pourrait être intéressant d'explorer. Un bon processus d'apprentissage typique automatise l'exploration d'une vaste gamme de modèles afin de mettre en évidence celui qui sera de meilleure qualité en termes de prédiction. Des travaux postérieurs à ce mémoire peuvent envisager la création d'un tel processus appliqué à la conception de systèmes embarqués.

Néanmoins, l'exercice réalisé dans ce chapitre nous permet d'affirmer que cette démarche, bien que devant être affinée et perfectionnée, fait sens. Bien qu'une erreur de modélisation soit présente [13], il a été possible de donner un sens à ce « nuage de points » liant l'espace de conception à l'espace des performances. De plus, comme illustré au sein des parties I et II, cette approche s'inscrit dans une époque où les concepteurs industriels et les chercheurs manquent de modèles permettant d'évaluer les performances de leur système selon de multiples critères. Bien que nous avons pu montrer au sein de ce mémoire qu'il existe bel et bien un lien entre choix de conception haut niveau et performances du système, les travaux qui consistent à rechercher la méthodologie visant à produire et à valider sérieusement des modèles est un sujet de recherche à part entière.

11. Ces conclusions auraient très bien pu être tirées par un concepteur industriel sans effectuer l'entièreté de l'exercice d'inférence de modèles.

12. Par exemple, le concepteur pourrait apprendre qu'en passant d'une cible FPGA à une autre, il gagnerait 20% de la surface du circuit.

13. Elle le sera toujours.

Chapitre 12

Conclusion

La conclusion du mémoire est divisée en deux parties : le résumé de la contribution apportée accompagné d'une discussion de l'approche et la présentation des travaux ultérieurs nécessaires dans le domaine.

12.1 Résumé de la contribution et discussion de l'approche

Au sein de ce mémoire, nous avons présenté le défi que représente la conception de systèmes embarqués sous fortes contraintes de performances. Les métriques telles que, notamment, le temps d'exécution, la surface occupée, l'énergie consommée sont conditionnées par les décisions prises par le concepteur. Si les choix des niveaux d'abstraction inférieurs sont aujourd'hui gérés par des outils de synthèse automatique, les choix de haut niveau sont laissés à l'appréciation de l'expert humain. Or ces décisions de haut niveau conditionnent toutes celles qui sont prises aux niveaux inférieurs et influent donc de façon critique sur les performances du système final.

Il existe de nombreux travaux de recherche visant à établir des flots de conception de haut niveau (*« system-level »*) pour simplifier la tâche des concepteurs. Nous avons présenté deux d'entre eux, organisés autour des outils Nessie et Gaspard2.

Néanmoins, que ce soit les concepteurs dans un cadre industriel ou les chercheurs dans la conception des flots, tous ont besoin de modèles permettant de caractériser l'impact des choix de conception sur les performances finales du système. Le concepteur désire pouvoir prédire de façon sommaire l'impact de ses choix de conception sur les performances du système. Si un outil comme Nessie permet d'estimer les performances du système sur base des modèles des composants fondamentaux constitutifs de celui-ci, il n'empêche que ces modèles doivent être définis de façon précise.

C'est pourquoi nous avons proposé une démarche permettant la création de tels modèles sur base de techniques de *machine learning*. Elle consiste à générer, dans un premier temps, un large ensemble de données issues de simulations. Lors de ces simulations (qui consistent à synthétiser des blocs comme la FFT sur FPGA), le lien entre les choix de conception et les performances résultantes est enregistré point après point. Dans un second temps, des techniques d'inférence de modèles sont utilisées afin de créer un modèle permettant une prédiction des performances approchant les résultats réels.

L'idée principale sous-jacente est de comparer le processus d'acquisition de données, précis mais long, au processus d'évaluation à l'aide du modèle ainsi construit, moins précis mais très rapide. Lors de la campagne de simulations, un ensemble de N solutions est simulé afin de créer un modèle. Ces simulations prennent un temps très long (*cf.* dans le cas d'étude simple, l'annexe A pour les temps de synthèse). La question est de savoir si nous pouvons présenter un $(N+1)$-ème cas au modèle créé et obtenir une réponse assez précise et proche du résultat réel, sans passer par le long processus de synthèse.

Afin de tester notre approche, nous avons étudié au sein du mémoire un cas d'étude simple : la transformée de Fourier rapide sur plate-forme FPGA. Deux implémentations de la FFT furent synthétisées sur trois cibles FPGA différentes. Ensuite, un modèle linéaire a été inféré de façon à prédire les données. Les résultats de l'exercice sont assez concluants, et bien que souffrant d'une erreur non négligeable dans certains cas, encouragent un approfondissement de la démarche.

En plus de fournir un premier exemple d'application de la démarche, ce mémoire a été le premier pas dans une formalisation mathématique du problème de la prédiction de performances pour la synthèse des circuits numériques. Cette formalisation est passée par une série de choix : différents types de décisions de conception et de métriques de performances ont été sélectionnés afin de permettre une première étude.

Cette approche est complexe, de par la pluralité disciplinaire qu'elle nécessite. En effet, elle est au carrefour des compétences de l'électronicien, pour la capture de données ayant un sens et l'interprétation des résultats, et du statisticien, pour la création de modèles ayant une bonne qualité de prédiction. En combinant les deux approches, nous avons pu observer des résultats intéressants.

Une question légitime que nous pouvons poser est la suivante : les techniques de *machine learning* sont-elles adaptées à ce problème ? Pour tenter d'y répondre, il est important de rappeler les difficultés que permettent de gérer les techniques de *machine learning* [Bon11] :

- minimiser l'erreur de prédiction,
- modéliser sur base d'un nombre fini de données,
- modéliser sachant que nous n'avons pas d'information d'*a priori* sur les données.

Nous pouvons voir ici que ces trois objectifs répondent bien à notre problème : nous voulons prédire un processus sur lequel nous n'avons que peu d'informations [1] avec un nombre fini de données [2] en minimisant l'erreur de prédiction. L'idée de commencer par les techniques les plus simples (linéaires) pour obtenir de l'information sur les données était donc une bonne approche, quitte à complexifier les modèles dans des études ultérieures.

1. Les logiciels de synthèse et les architectures des FPGA sont propriétaires et fermés ou trop complexes pour être finement modélisés.
2. A cause de la lenteur du processus de synthèse.

12.2 Travaux ultérieurs

Comparé à l'ensemble des travaux qu'il est possible de réaliser, ce mémoire ne constitue qu'une petite entrée en matière. En effet, le nombre d'hypothèses simplificatrices qui ont été appliquées sur le cas d'étude et la taille relativement limitée de l'ensemble des données analysées ouvrent la voie à de vraies généralisations ainsi qu'à une campagne sérieuse de collecte de données.

Premièrement, il est nécessaire d'établir un état de l'art complet des techniques de *machine learning* et de leurs applications dans un cadre microélectronique.

Deuxièmement, il est nécessaire d'explorer l'approche présentée dans toute sa profondeur, en multipliant le nombre de cas d'études analysés. En multipliant les algorithmes, les variantes d'implémentation RTL, les types de cibles FPGA et les outils de synthèse, le nombre de données potentielles pour construire des modèles devient très vite gigantesque.

Sur base de ce large ensemble de données, plusieurs techniques d'inférence de modèles peuvent être utilisées. En particulier, une procédure automatique de *sélection de modèles* (linéaires ou non linéaires) pourrait être appliquée au design microélectronique.

A plus long terme, une fois les modèles validés en théorie, de nombreuses applications de ceux-ci peuvent être imaginées.

Il est possible d'envisager la création d'un outil de prédiction de performances basé sur :

- une bibliothèque de modèles des principaux algorithmes de traitement du signal ou de leurs composantes, permettant d'évaluer a priori l'impact des choix du concepteur (aux niveaux algorithme, cible matérielle et outils) sur les performances finales du système,
- une méthodologie d'identification rapide de tels modèles, incluant en particulier une définition des expériences à mener et des traitements mathématiques à opérer pour obtenir rapidement les modèles correspondant à un nouvel algorithme et/ou une nouvelle cible hardware.

Nous pouvons également exploiter les modèles créés dans le cadre de problématiques connexes intervenant dans la conception de systèmes embarqués, comme :

- l'optimisation multicritère dans le cadre de grands espaces de conception,
- la possibilité d'exploitation des modèles dans un système à reconfiguration dynamique [BNH06].

La première perspective viserait l'intégration des modèles au sein d'outils comme Nessie ou Gaspard2.

La seconde perspective viserait l'exploitation des modèles dans un environnement en exécution, comme le projet de système d'exploitation temps réel sur plate-forme reconfigurable dynamiquement [1]. Le système devant optimiser les tâches sous contraintes fortes de surface disponibles et de puissance de calculs, des modèles multicritères exploitables en temps réel pourraient représenter un atout considérable.

Pour conclure le mémoire et insister sur les perspectives, nous terminerons sur une série de questions qui nous apparaissent essentielles dans la poursuite des travaux dans le domaine de la prédiction de performances pour la conception de systèmes embarqués :

- Est-il possible d'extraire, avec les méthodes citées ci-dessus, des modèles permettant une prédiction efficace de la performance pour les algorithmes considérés ?

- Dans l'affirmative, quel est le niveau de granularité et quels sont les moyens les plus efficaces de production de tels modèles ?

- Dans quelle mesure ces modèles peuvent s'avérer utiles pour un concepteur humain de systèmes embarqués ? En particulier, est-il possible de composer ces modèles entre eux afin d'obtenir une prédiction de la performance du système final, dans sa totalité ?

- Est-il possible de concevoir une méthodologie d'identification rapide de modèles, c'est-à-dire de formaliser les étapes de caractérisation des algorithmes et des cibles matérielles ?

- Ces modèles sont-ils exploitables dans les flots de conception haut niveau actuellement émergents, basés sur des standards et outils comme SystemC, MARTE, GASPARD2 ou NESSIE ?

Annexes

Annexe A

Données récoltées

Cet annexe reprend les données récoltées lors de la synthèse sous forme tabulaire.

Les simulations fonctionnelles pour récupérer le nombre de cycles d'horloge pour réaliser une FFT ont été executées avec *ModelSim SE* v6.6b (2010). Les synthèses ont été réalisées avec l'outil *ISE Xilinx* v11.1 (2009).

Lorsqu'une entrée dans un *record* est à *NaN*, ceci signifique que l'outil de synthèse n'est pas parvenu à *mapper* le design VHDL sur la cible FPGA. C'est le cas pour les *records* de la version 1 de la FFT sur le Spartan3 (xc3s400-4ft256), pour les variations faisant intervenir les plus grands paramètres de taille des points et de nombre de points à traiter. De plus, quatre synthèses ont été interrompues car le temps de calcul pour celles-ci a explosé.

Les colonnes du tableau permettent d'interpréter les *records* de la façon suivante :

i : l'index du record, utile pour pouvoir remettre ensemble les différents morceaux du tableau.

design : la version de l'implémentation RTL de la FFT (v1 ou v5).

target : la cible FPGA.

tool : l'outil de synthèse. Cette case avait été prévue pour d'éventuels autres outils de synthèse qu'ISE.

point_width : la taille des points en nombre de bits.

points_number : le nombre de points à traiter.

max_delay : le dimensionnement théorique minimal de la période d'horloge fourni par l'outil de synthèse (en nanosecondes).

ticks_number : le nombre de cycles d'horloge pour réaliser une FFT.

response_time : la latence d'exécution d'une FFT en nanosecondes.

lut_number : le nombre de LUT qu'exploite le système.

area_fraction : la surface relative (au niveau LUT) occupée par le système sur la cible FPGA.

total_power : la puissance consommée par le système. Cette information est fournie par l'outil *XPower Impact*.

synth_time : la durée de la synthèse en secondes pour cette implémentation, donnée à titre indicatif.

i	design	target	tool	point_width	points_number
1	fft_v1	xc3s400-4ft256	ISE	8	256
2	fft_v1	xc3s400-4ft256	ISE	8	512
3	fft_v1	xc3s400-4ft256	ISE	8	1024
4	fft_v1	xc3s400-4ft256	ISE	8	2048
5	fft_v1	xc3s400-4ft256	ISE	16	256
6	fft_v1	xc3s400-4ft256	ISE	16	512
7	fft_v1	xc3s400-4ft256	ISE	16	1024
8	fft_v1	xc3s400-4ft256	ISE	16	2048
9	fft_v1	xc3s400-4ft256	ISE	18	256
10	fft_v1	xc3s400-4ft256	ISE	18	512
11	fft_v1	xc3s400-4ft256	ISE	18	1024
12	fft_v1	xc4vlx100-12ff1148	ISE	8	256
13	fft_v1	xc4vlx100-12ff1148	ISE	8	512
14	fft_v1	xc4vlx100-12ff1148	ISE	8	1024
15	fft_v1	xc4vlx100-12ff1148	ISE	8	2048
16	fft_v1	xc4vlx100-12ff1148	ISE	8	4096
17	fft_v1	xc4vlx100-12ff1148	ISE	16	256
18	fft_v1	xc4vlx100-12ff1148	ISE	16	512
19	fft_v1	xc4vlx100-12ff1148	ISE	16	1024
20	fft_v1	xc4vlx100-12ff1148	ISE	16	2048
21	fft_v1	xc4vlx100-12ff1148	ISE	16	4096
22	fft_v1	xc4vlx100-12ff1148	ISE	18	256
23	fft_v1	xc4vlx100-12ff1148	ISE	18	512
24	fft_v1	xc4vlx100-12ff1148	ISE	18	1024
25	fft_v1	xc4vlx100-12ff1148	ISE	18	2048
26	fft_v1	xc4vlx100-12ff1148	ISE	18	4096
27	fft_v1	xc5vlx330-2ff1760	ISE	8	256
28	fft_v1	xc5vlx330-2ff1760	ISE	8	512
29	fft_v1	xc5vlx330-2ff1760	ISE	8	1024
30	fft_v1	xc5vlx330-2ff1760	ISE	8	2048
31	fft_v1	xc5vlx330-2ff1760	ISE	8	4096
32	fft_v1	xc5vlx330-2ff1760	ISE	16	256
33	fft_v1	xc5vlx330-2ff1760	ISE	16	512
34	fft_v1	xc5vlx330-2ff1760	ISE	16	1024
35	fft_v1	xc5vlx330-2ff1760	ISE	16	2048
36	fft_v1	xc5vlx330-2ff1760	ISE	16	4096
37	fft_v1	xc5vlx330-2ff1760	ISE	18	256
38	fft_v1	xc5vlx330-2ff1760	ISE	18	512
39	fft_v1	xc5vlx330-2ff1760	ISE	18	1024
40	fft_v1	xc5vlx330-2ff1760	ISE	18	2048
41	fft_v1	xc5vlx330-2ff1760	ISE	18	4096
42	fft_v5	xc3s400-4ft256	ISE	8	64
43	fft_v5	xc3s400-4ft256	ISE	8	256
44	fft_v5	xc3s400-4ft256	ISE	8	1024
45	fft_v5	xc3s400-4ft256	ISE	8	4096

i	design	target	tool	point_width	points_number
46	fft_v5	xc3s400-4ft256	ISE	8	16384
47	fft_v5	xc3s400-4ft256	ISE	16	64
48	fft_v5	xc3s400-4ft256	ISE	16	256
49	fft_v5	xc3s400-4ft256	ISE	16	1024
50	fft_v5	xc3s400-4ft256	ISE	16	4096
51	fft_v5	xc3s400-4ft256	ISE	16	16384
52	fft_v5	xc3s400-4ft256	ISE	32	64
53	fft_v5	xc3s400-4ft256	ISE	32	256
54	fft_v5	xc3s400-4ft256	ISE	32	1024
55	fft_v5	xc3s400-4ft256	ISE	32	4096
56	fft_v5	xc3s400-4ft256	ISE	32	16384
57	fft_v5	xc4vlx100-12ff1148	ISE	8	64
58	fft_v5	xc4vlx100-12ff1148	ISE	8	256
59	fft_v5	xc4vlx100-12ff1148	ISE	8	1024
60	fft_v5	xc4vlx100-12ff1148	ISE	8	4096
61	fft_v5	xc4vlx100-12ff1148	ISE	8	16384
62	fft_v5	xc4vlx100-12ff1148	ISE	16	64
63	fft_v5	xc4vlx100-12ff1148	ISE	16	256
64	fft_v5	xc4vlx100-12ff1148	ISE	16	1024
65	fft_v5	xc4vlx100-12ff1148	ISE	16	4096
66	fft_v5	xc4vlx100-12ff1148	ISE	16	16384
67	fft_v5	xc4vlx100-12ff1148	ISE	32	64
68	fft_v5	xc4vlx100-12ff1148	ISE	32	256
69	fft_v5	xc4vlx100-12ff1148	ISE	32	1024
70	fft_v5	xc4vlx100-12ff1148	ISE	32	4096
71	fft_v5	xc4vlx100-12ff1148	ISE	32	16384
72	fft_v5	xc5vlx330-2ff1760	ISE	8	64
73	fft_v5	xc5vlx330-2ff1760	ISE	8	256
74	fft_v5	xc5vlx330-2ff1760	ISE	8	1024
75	fft_v5	xc5vlx330-2ff1760	ISE	8	4096
76	fft_v5	xc5vlx330-2ff1760	ISE	8	16384
77	fft_v5	xc5vlx330-2ff1760	ISE	16	64
78	fft_v5	xc5vlx330-2ff1760	ISE	16	256
79	fft_v5	xc5vlx330-2ff1760	ISE	16	1024
80	fft_v5	xc5vlx330-2ff1760	ISE	16	4096
81	fft_v5	xc5vlx330-2ff1760	ISE	16	16384
82	fft_v5	xc5vlx330-2ff1760	ISE	32	64
83	fft_v5	xc5vlx330-2ff1760	ISE	32	256
84	fft_v5	xc5vlx330-2ff1760	ISE	32	1024
85	fft_v5	xc5vlx330-2ff1760	ISE	32	4096
86	fft_v5	xc5vlx330-2ff1760	ISE	32	16384

i	max_delay	ticks_number	response_time
1	NaN	1450	NaN
2	NaN	3119	NaN
3	NaN	6708	NaN
4	NaN	14393	NaN
5	NaN	1450	NaN
6	NaN	3119	NaN
7	NaN	6708	NaN
8	NaN	14393	NaN
9	NaN	1450	NaN
10	NaN	3119	NaN
11	NaN	6708	NaN
12	2.476000	1450	3590.2
13	2.479000	3119	7732.001
14	2.535000	6708	17004.78
15	2.666000	14393	38371.738
16	2.582000	30782	79479.124
17	2.730000	1450	3958.5
18	2.667000	3119	8318.373
19	2.732000	6708	18326.256
20	2.622000	14393	37738.446
21	2.658000	30782	81818.556
22	2.596000	1450	3764.2
23	2.566000	3119	8003.354
24	2.588000	6708	17360.304
25	2.579000	14393	37119.547
26	2.637000	30782	81172.134
27	2.564000	1450	3717.8
28	2.558000	3119	7978.402
29	2.606000	6708	17481.048
30	2.678000	14393	38544.454
31	2.684000	30782	82618.888
32	2.687000	1450	3896.15
33	2.688000	3119	8383.872
34	2.688000	6708	18031.104
35	2.847000	14393	40976.871
36	2.837000	30782	87328.534
37	2.512000	1450	3642.4
38	2.512000	3119	7834.928
39	2.691000	6708	18051.228
40	2.691000	14393	38731.563
41	2.756000	30782	84835.192
42	1.034000	265	5300.0
43	1.034000	1289	25780.0
44	1.035000	6153	123060.0
45	1.034000	28681	573620.0

i	max_delay	ticks_number	response_time
46	1.035000	131081	2621620.0
47	1.035000	265	5300.0
48	1.035000	1289	25780.0
49	1.035000	6153	123060.0
50	1.035000	28681	573620.0
51	NaN	131081	NaN
52	1.035000	265	5300.0
53	1.035000	1289	25780.0
54	1.035000	6153	123060.0
55	1.035000	28681	573620.0
56	NaN	131081	NaN
57	2.687000	265	712.055
58	2.682000	1289	3457.098
59	2.693000	6153	16570.029
60	2.651000	28681	76033.331
61	2.632000	131081	345005.192
62	2.621000	265	694.565
63	2.590000	1289	3338.51
64	2.512000	6153	15456.336
65	2.638000	28681	75660.478
66	2.565000	131081	336222.765
67	2.743000	265	726.895
68	2.589000	1289	3337.221
69	2.713000	6153	16693.089
70	2.744000	28681	78700.664
71	2.601000	131081	340941.681
72	2.299000	265	609.235
73	2.127000	1289	2741.703
74	2.158000	6153	13278.174
75	2.304000	28681	66081.024
76	2.421000	131081	317347.101
77	2.309000	265	611.885
78	2.273000	1289	2929.897
79	2.350000	6153	14459.55
80	2.279000	28681	65363.999
81	2.564000	131081	336091.684
82	2.306000	265	611.09
83	2.326000	1289	2998.214
84	2.284000	6153	14053.452
85	2.399000	28681	68805.719
86	2.769000	131081	362963.289

i	lut_number	area_fraction	total_power	synth_time
1	8818	1.23018973214	NaN	152
2	19606	2.73521205357	NaN	295
3	42288	5.89955357143	NaN	794
4	90229	12.5877511161	NaN	2583
5	24008	3.34933035714	NaN	394
6	49833	6.9521484375	NaN	1018
7	106826	14.9031808036	NaN	3338
8	232232	32.3984375	NaN	13485
9	27587	3.8486328125	NaN	488
10	59467	8.29617745536	NaN	1309
11	123725	17.2607421875	NaN	4316
12	1569	0.0159606933594	873.08	281
13	1945	0.019785563151	902.61	277
14	2433	0.0247497558594	924.08	334
15	3074	0.0312703450521	921.06	346
16	3909	0.0397644042969	692.78	522
17	2755	0.0280253092448	926.34	338
18	3544	0.0360514322917	958.15	388
19	4666	0.0474650065104	996.29	674
20	6461	0.0657246907552	738.41	1345
21	9121	0.092783610026	835.85	1682
22	3088	0.0314127604167	919.96	356
23	3973	0.0404154459635	965.46	436
24	5272	0.0536295572917	1000.78	560
25	7306	0.0743204752604	730.74	1119
26	10668	0.108520507812	820.50	1439
27	1173	0.0056568287037	6313.53	479
28	1404	0.00677083333333	6310.55	460
29	1684	0.00812114197531	6311.79	508
30	2062	0.00994405864198	6343.40	524
31	2587	0.0124758873457	3741.26	577
32	1945	0.00937982253086	6354.38	526
33	2320	0.0111882716049	6341.38	576
34	2808	0.0135416666667	6370.06	636
35	3586	0.017293595679	3742.86	727
36	4701	0.0226707175926	3790.02	908
37	2137	0.0103057484568	6357.49	552
38	2548	0.012287808642	6372.11	580
39	3088	0.0148919753086	6361.42	660
40	3966	0.0191261574074	3774.52	782
41	5229	0.0252170138889	3741.10	1020
42	1039	0.144949776786	121.41	87
43	1023	0.142717633929	79.50	82
44	1166	0.162667410714	82.05	90
45	1307	0.182338169643	87.87	90

i	lut_number	area_fraction	total_power	synth_time
46	1444	0.201450892857	95.91	171
47	1183	0.1650390625	152.36	86
48	1361	0.189871651786	83.84	90
49	1546	0.215680803571	90.39	94
50	1730	0.241350446429	98.33	102
51	85464	11.9229910714	NaN	4640
52	1919	0.267717633929	218.65	110
53	2161	0.301478794643	101.81	119
54	2408	0.3359375	110.41	124
55	2655	0.370396205357	102.98	134
56	170429	23.7763671875	NaN	16988
57	1035	0.0105285644531	705.95	195
58	1018	0.0103556315104	719.27	168
59	1157	0.0117696126302	721.70	174
60	1289	0.0131123860677	717.92	178
61	1456	0.0148111979167	746.93	186
62	1200	0.01220703125	764.15	172
63	1370	0.0139363606771	759.47	176
64	1563	0.0158996582031	762.67	183
65	1754	0.0178426106771	795.79	188
66	1943	0.019765218099	824.04	210
67	1928	0.0196126302083	869.09	194
68	2168	0.0220540364583	837.86	198
69	2414	0.0245564778646	850.37	208
70	2655	0.0270080566406	871.96	216
71	2900	0.0295003255208	913.63	252
72	767	0.00369888117284	5213.86	279
73	888	0.00428240740741	5218.18	271
74	1026	0.00494791666667	5223.34	280
75	1191	0.00574363425926	5228.83	291
76	1340	0.00646219135802	5236.80	296
77	1098	0.00529513888889	5236.06	257
78	1229	0.0059268904321	5226.76	283
79	1412	0.00680941358025	5229.14	294
80	1613	0.00777874228395	5228.55	308
81	1782	0.00859375	5232.67	313
82	1693	0.00816454475309	5233.18	316
83	1928	0.00929783950617	5231.94	332
84	2171	0.0104697145062	5235.03	335
85	2434	0.0117380401235	5244.64	344
86	2676	0.0129050925926	5244.76	364

Annexe B

Résultats d'analyses

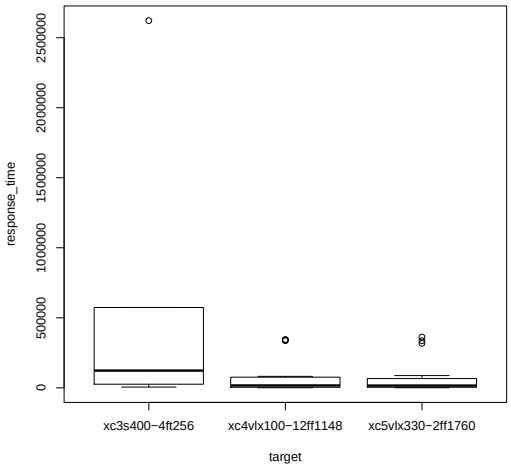
Cet annexe répertorie :

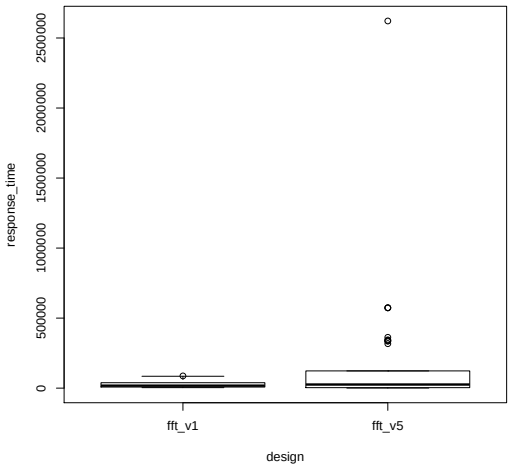
- tous les types de graphiques générés (description et stratification) pour les quatre différentes métriques de performances étudiées au sein du mémoire ;
- le tableau contenant la quantification de l'erreur moyenne (NMSE) pour les différentes métriques et les différents types de stratification (*cf.* tableau ci-dessous).

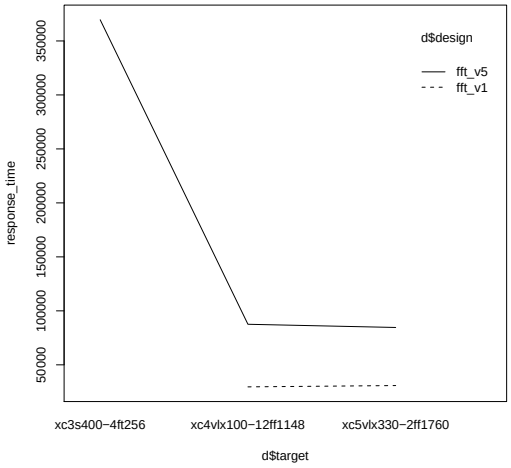
Métrique	Type de stratification	NMSE
Latence	Par designs et par cibles	0.01491196
	Par cibles uniquement	0.01488387
	Par designs uniquement	0.8545365
Nombre de LUT	Par designs et par cibles	0.1438274
	Par cibles uniquement	1.221330
	Par designs uniquement	0.3114987
Surface relative	Par designs et par cibles	0.1798977
	Par cibles uniquement	0.2220138
	Par designs uniquement	1.021376
Puissance	Par designs et par cibles	0.2585833
	Par cibles uniquement	0.5542299
	Par designs uniquement	1.126122

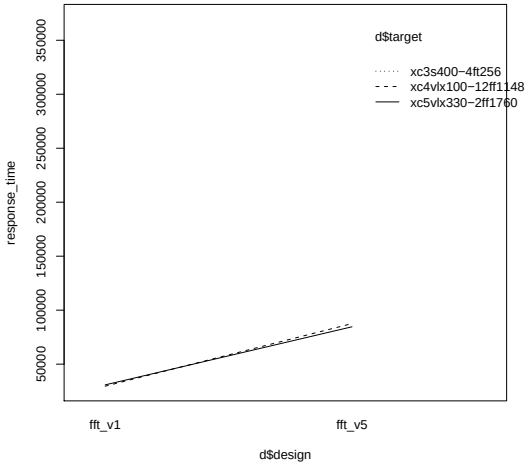
B.1 La latence

B.1.1 Analyse descriptive

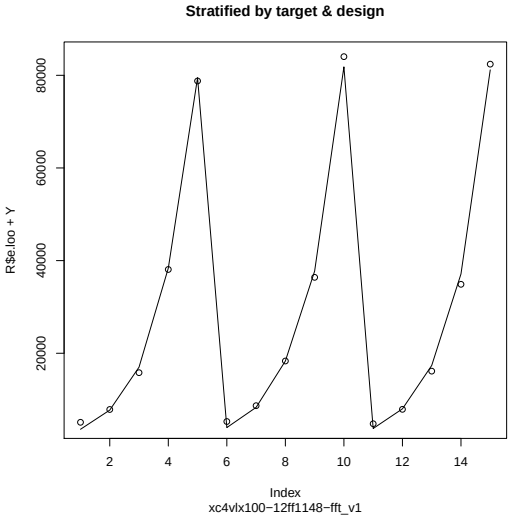


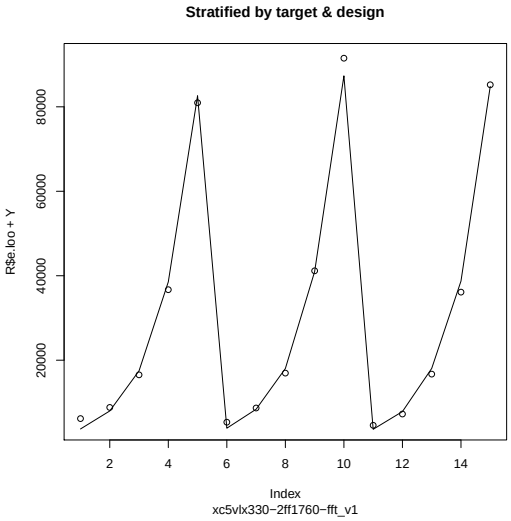


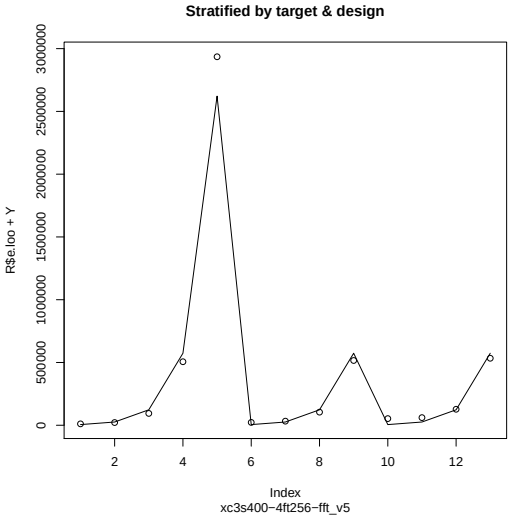

response_time
50000
100000
150000
200000
250000
300000
350000
d$design
fft_v5
fft_v1
xc3s400−4ft256
xc4vlx100−12ff1148
xc5vlx330−2ff1760
d$target

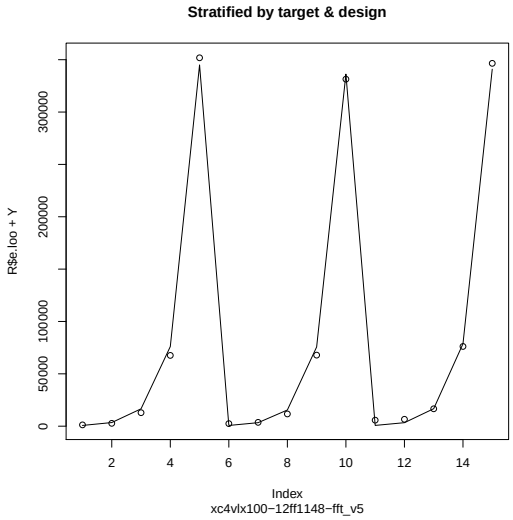

response_time
50000
100000
150000
200000
250000
300000
350000
d$target
xc3s400−4ft256
xc4vlx100−12ff1148
xc5vlx330−2ff1760
fft_v1
fft_v5
d$design

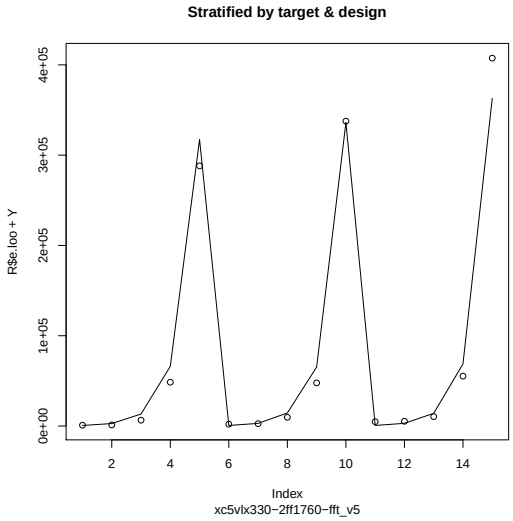
B.1.2 Modélisation linéaire

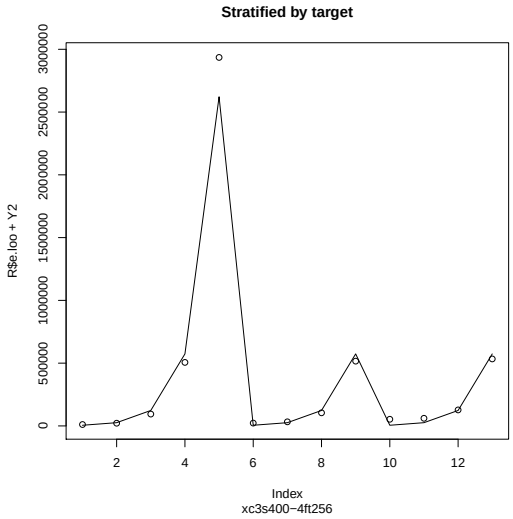


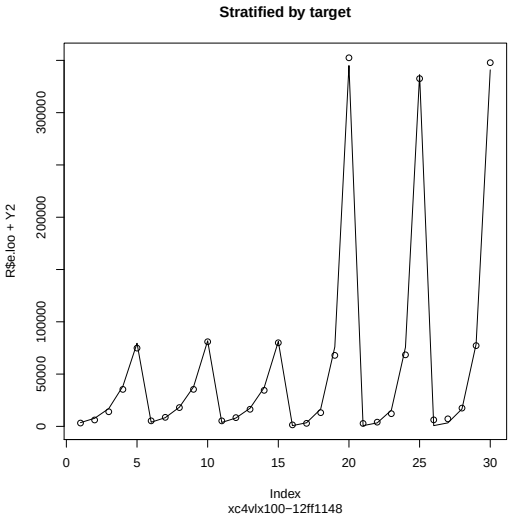


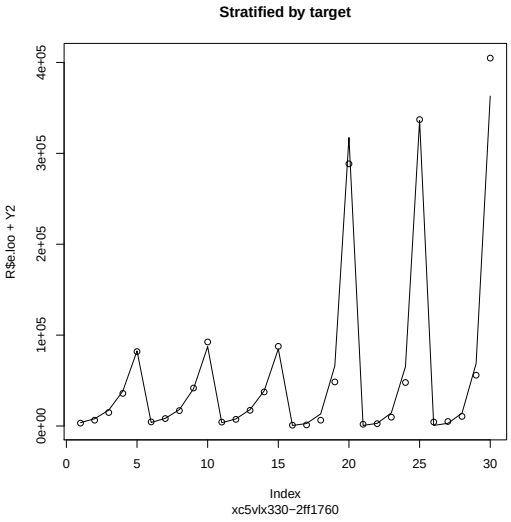


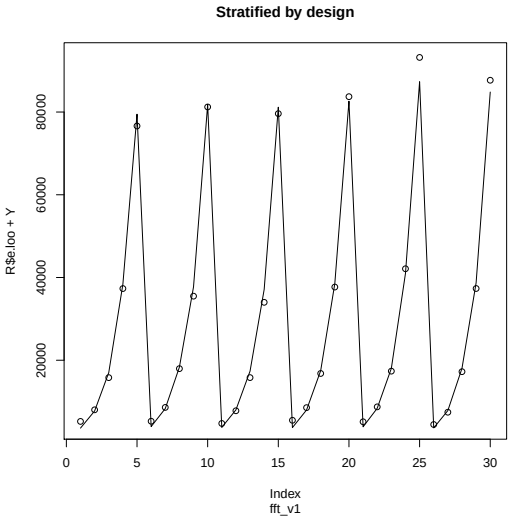

Stratified by target & design
R$e.loo + Y
0
50000
100000
200000
300000
2
4
6
8
10
12
14
Index
xc4vlx100−12ff1148−fft_v5

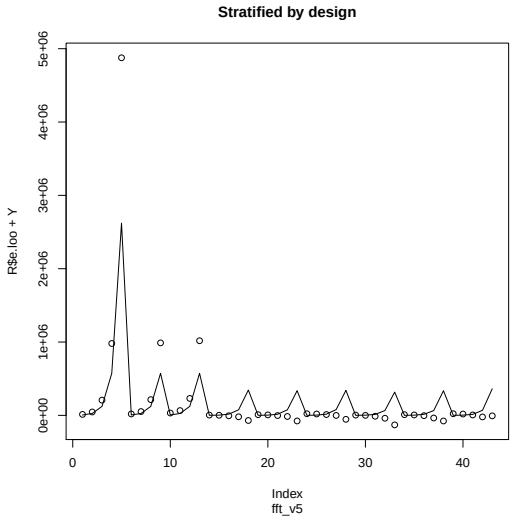

Stratified by target & design
R$e.loo + Y
0e+00
1e+05
2e+05
3e+05
4e+05
2
4
6
8
10
12
14
Index
xc5vlx330−2ff1760−fft_v5


Stratified by target
R$e.loo + Y2
0
500000
1000000
1500000
2000000
2500000
3000000
2
4
6
8
10
12
Index
xc3s400−4ft256


Stratified by target
R$e.loo + Y2
0
50000
100000
200000
300000
0
5
10
15
20
25
30
Index
xc4vlx100−12ff1148

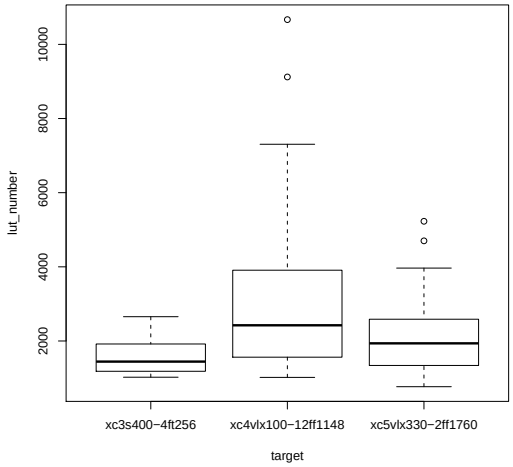

Stratified by target
R$e.loo + Y2
0e+00
1e+05
2e+05
3e+05
4e+05
0
5
10
15
20
25
30
Index
xc5vlx330−2ff1760

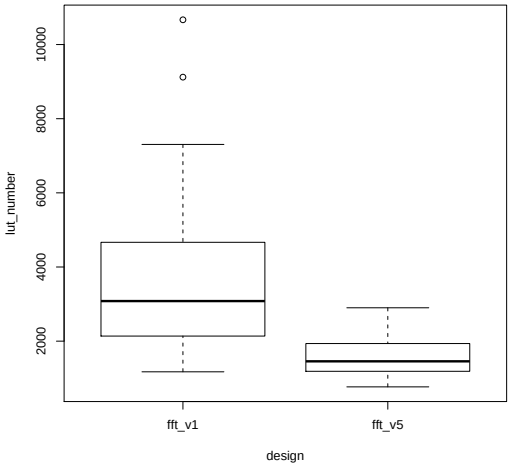

Stratified by design
R$e.loo + Y
20000
40000
60000
80000
0
5
10
15
20
25
30
Index
fft_v1

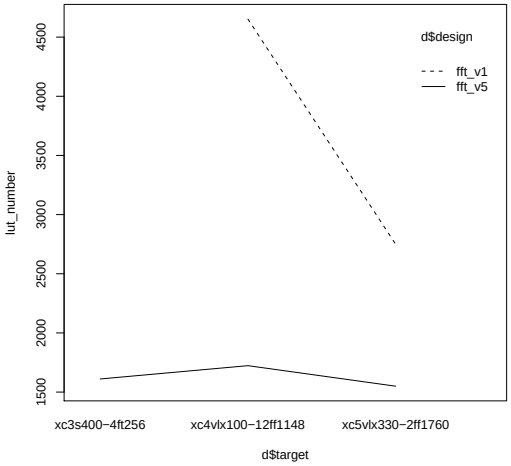


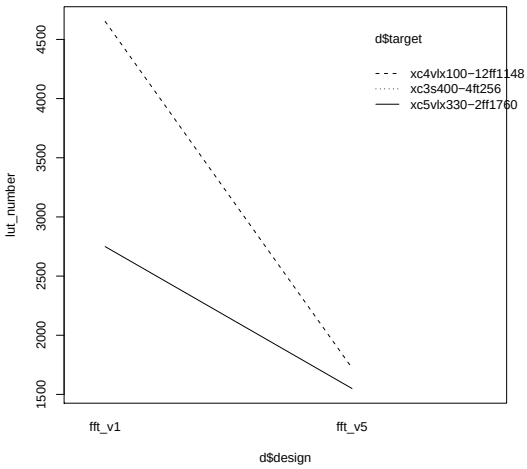
B.2 Le nombre de LUT

B.2.1 Analyse descriptive

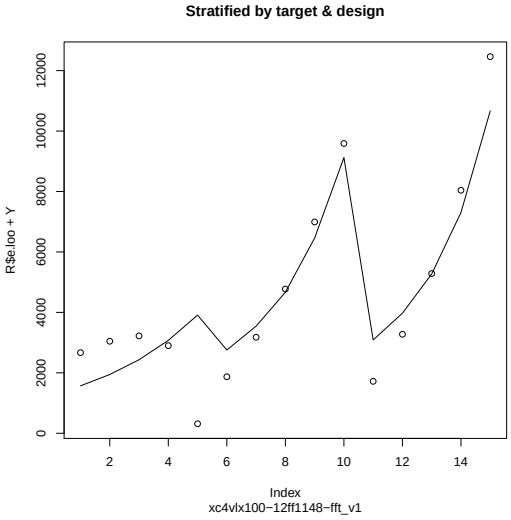


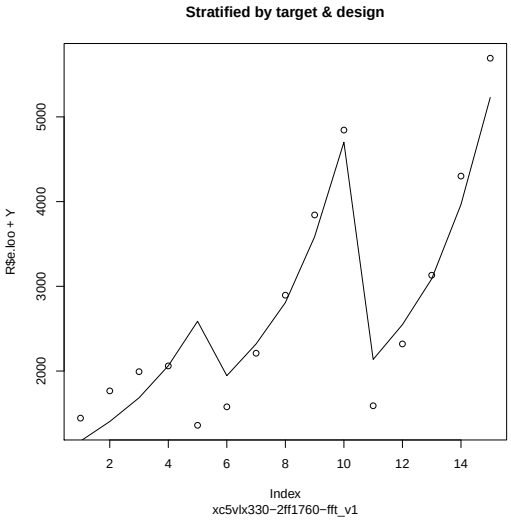

lut_number
2000
4000
6000
8000
10000
fft_v1
fft_v5
design

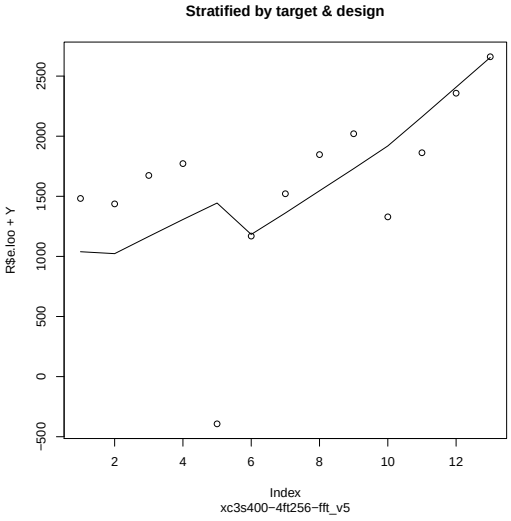

lut_number
1500
2000
2500
3000
3500
4000
4500
d$design
fft_v1
fft_v5
xc3s400−4ft256
xc4vlx100−12ff1148
xc5vlx330−2ff1760
d$target

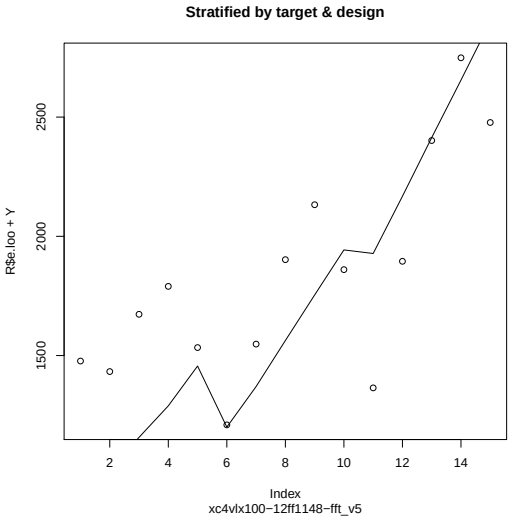

lut_number
1500
2000
2500
3000
3500
4000
4500
d$target
xc4vlx100−12ff1148
xc3s400−4ft256
xc5vlx330−2ff1760
fft_v1
fft_v5
d$design

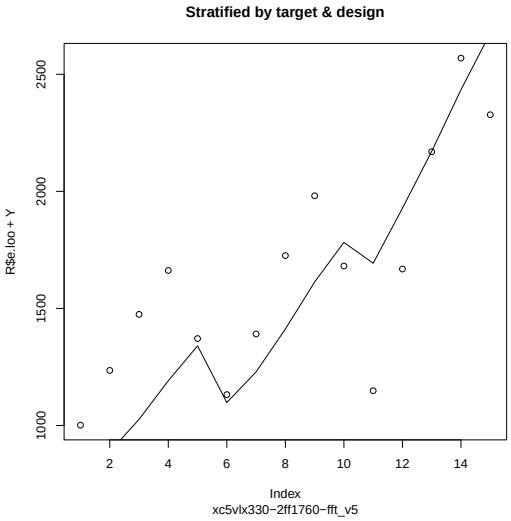
B.2.2 Modélisation linéaire

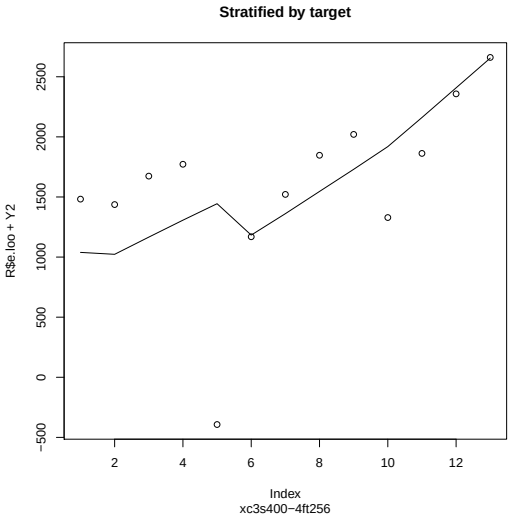


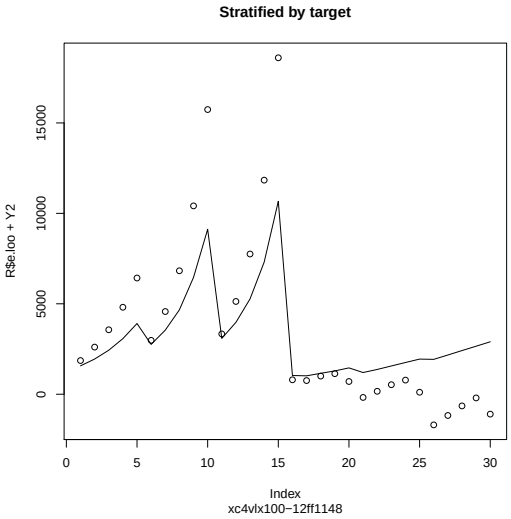


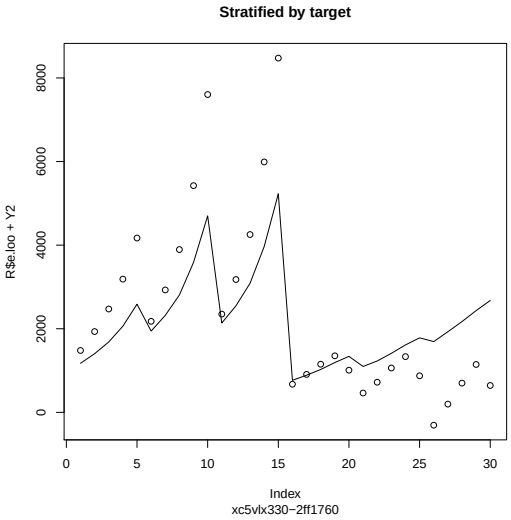


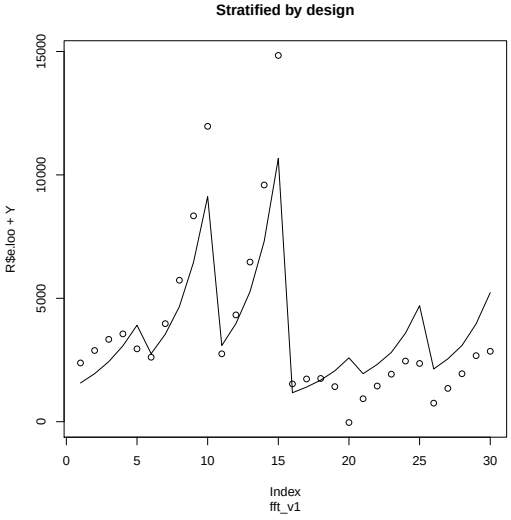

Stratified by target & design
R$e.loo + Y
Index
xc4vlx100−12ff1148−fft_v5

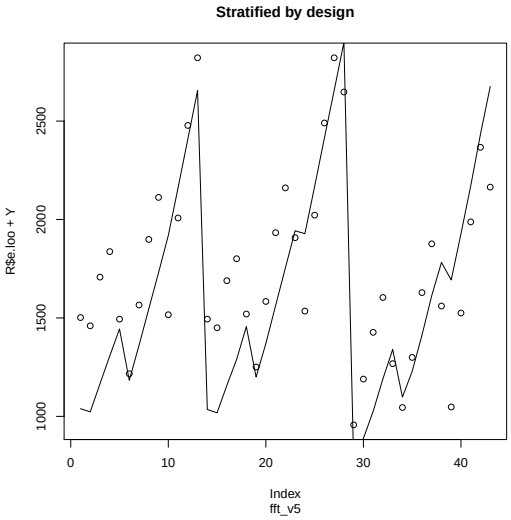

Stratified by target & design
R$e.loo + Y
Index
xc5vlx330−2ff1760−fft_v5


Stratified by target
R$e.loo + Y2
Index
xc3s400−4ft256


Stratified by target
R$e.loo + Y2
0
5000
10000
15000
0
5
10
15
20
25
30
Index
xc4vlx100−12ff1148

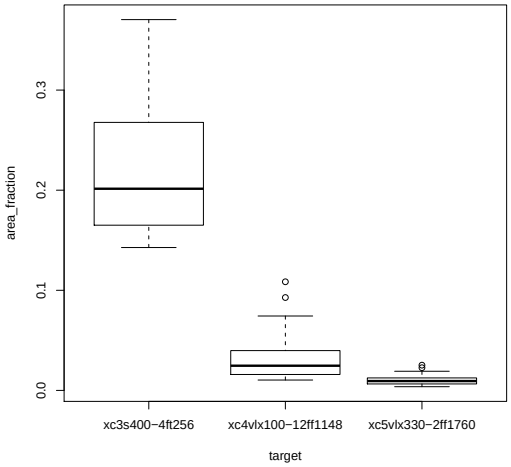

Stratified by target
R$e.loo + Y2
0
2000
4000
6000
8000
0
5
10
15
20
25
30
Index
xc5vlx330−2ff1760

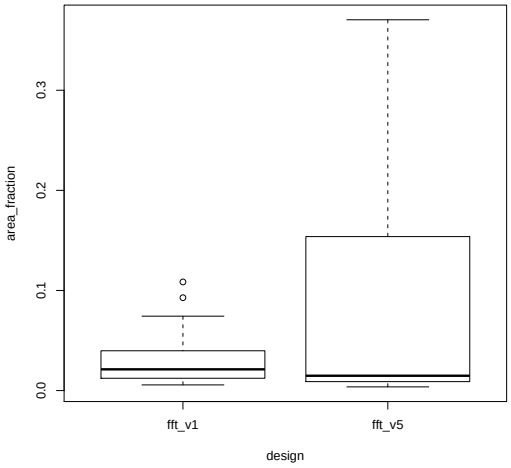

Stratified by design
R$e.loo + Y
0
5000
10000
15000
0
5
10
15
20
25
30
Index
fft_v1

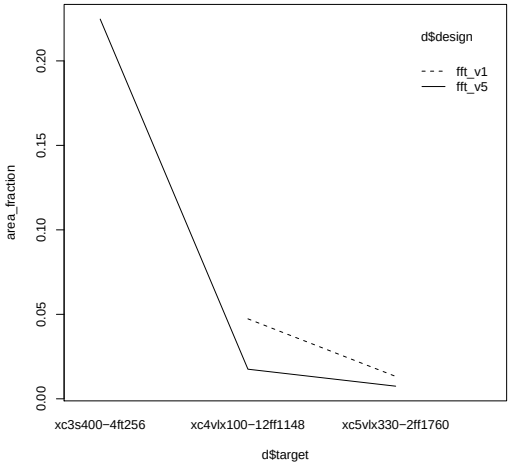


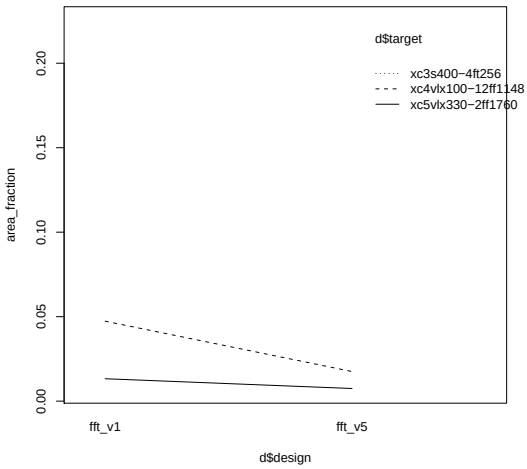
B.3 La surface relative

B.3.1 Analyse descriptive

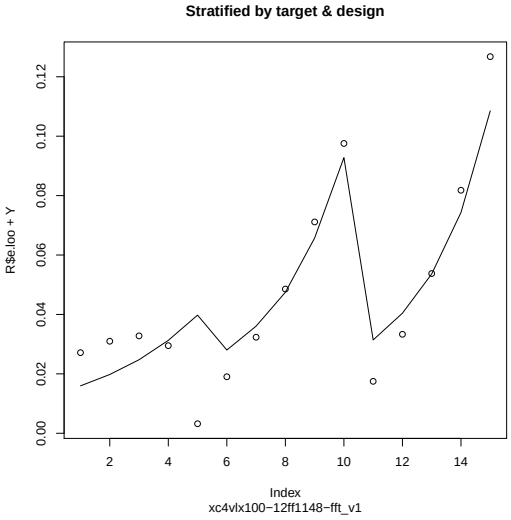


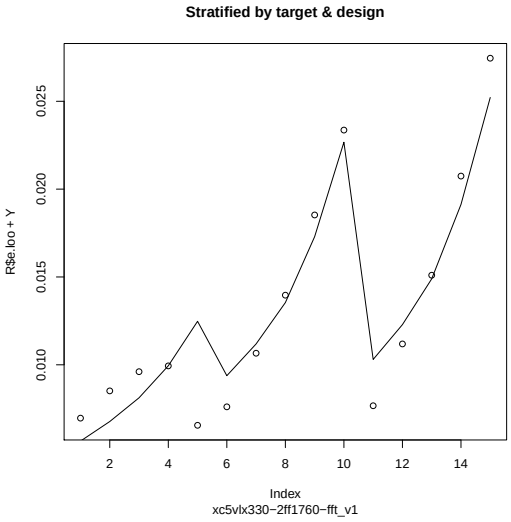

area_fraction
0.0
0.1
0.2
0.3
fft_v1
fft_v5
design

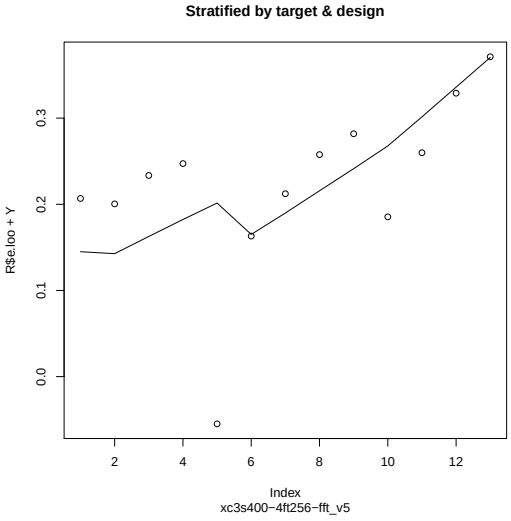

d$design
fft_v1
fft_v5
area_fraction
0.00
0.05
0.10
0.15
0.20
xc3s400−4ft256
xc4vlx100−12ff1148
xc5vlx330−2ff1760
d$target

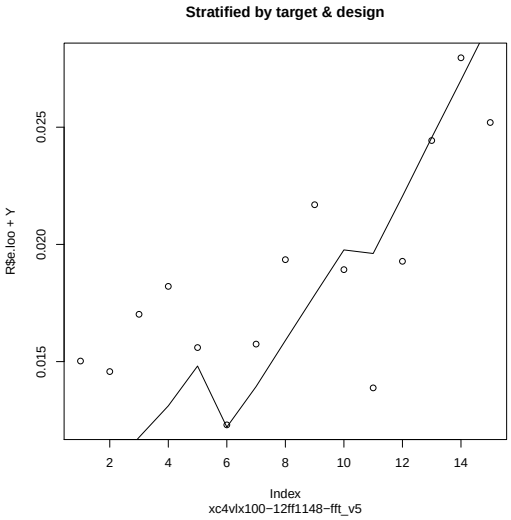

d$target
xc3s400−4ft256
xc4vlx100−12ff1148
xc5vlx330−2ff1760
area_fraction
0.00
0.05
0.10
0.15
0.20
fft_v1
fft_v5
d$design

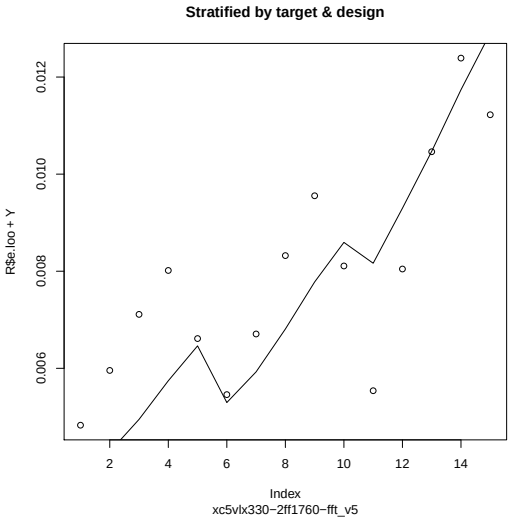
B.3.2 Modélisation linéaire

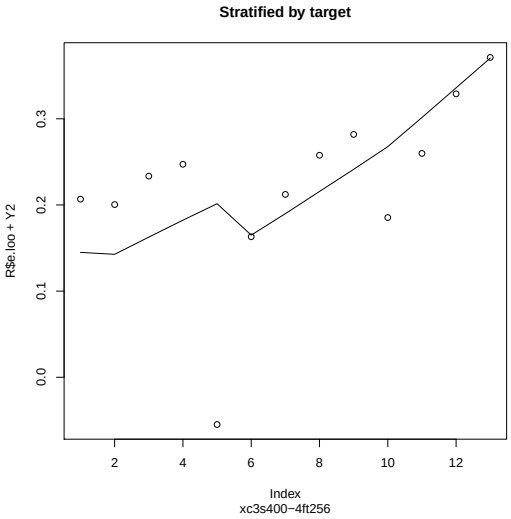


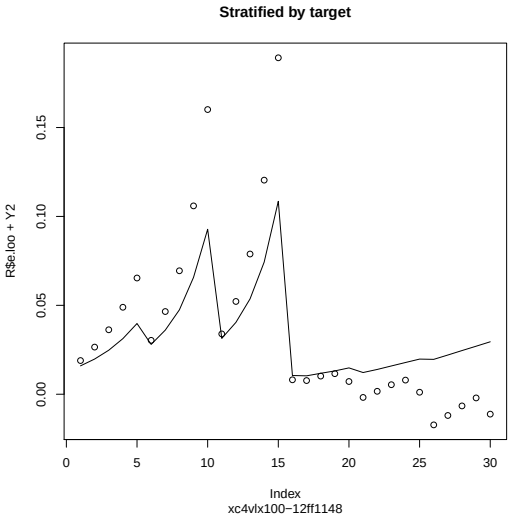


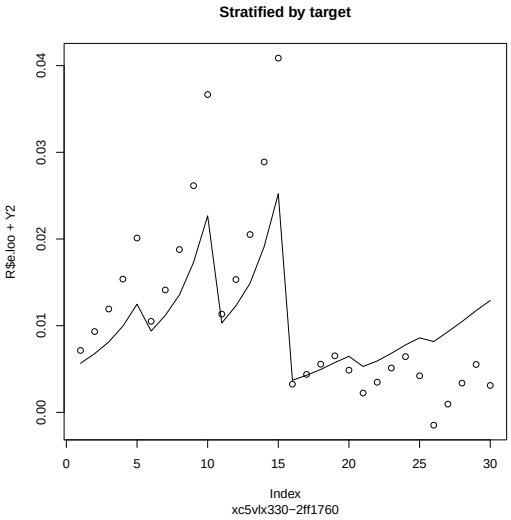


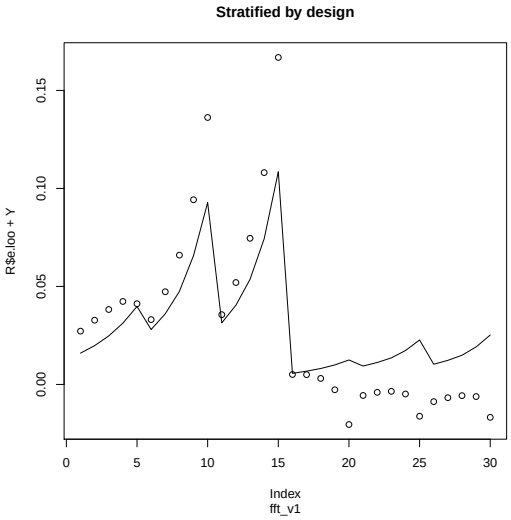

Stratified by target & design
R$e.loo + Y
0.015
0.020
0.025
2
4
6
8
10
12
14
Index
xc4vlx100−12ff1148−fft_v5

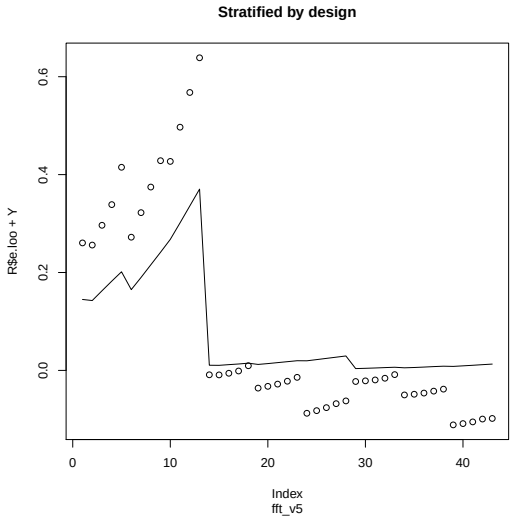

Stratified by target & design
R$e.loo + Y
0.006
0.008
0.010
0.012
2
4
6
8
10
12
14
Index
xc5vlx330−2ff1760−fft_v5


Stratified by target
R$e.loo + Y2
0.0
0.1
0.2
0.3
2
4
6
8
10
12
Index
xc3s400−4ft256


Stratified by target
R$e.loo + Y2
0.00
0.05
0.10
0.15
0
5
10
15
20
25
30
Index
xc4vlx100−12ff1148

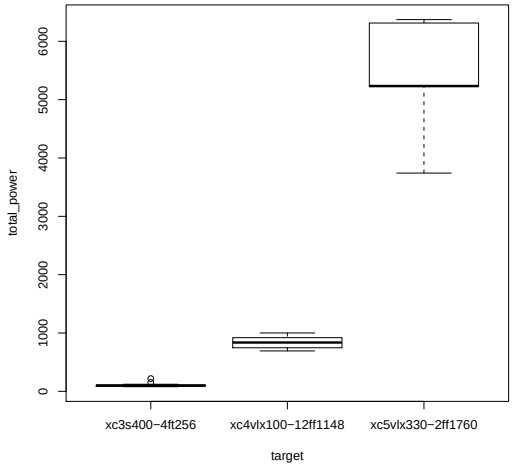

Stratified by target
R$e.loo + Y2
0.00
0.01
0.02
0.03
0.04
0
5
10
15
20
25
30
Index
xc5vlx330−2ff1760

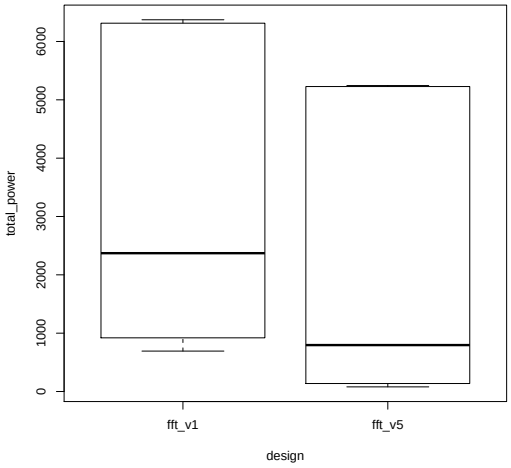

Stratified by design
R$e.loo + Y
0.00
0.05
0.10
0.15
0
5
10
15
20
25
30
Index
fft_v1

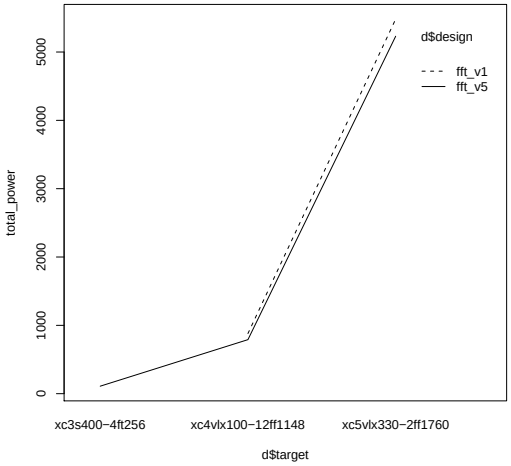


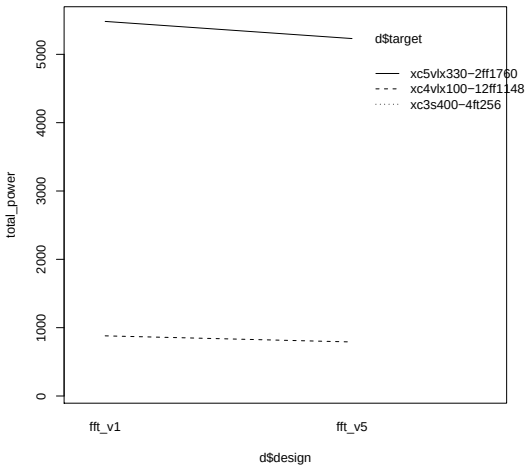
B.4 La puissance consommée

B.4.1 Analyse descriptive

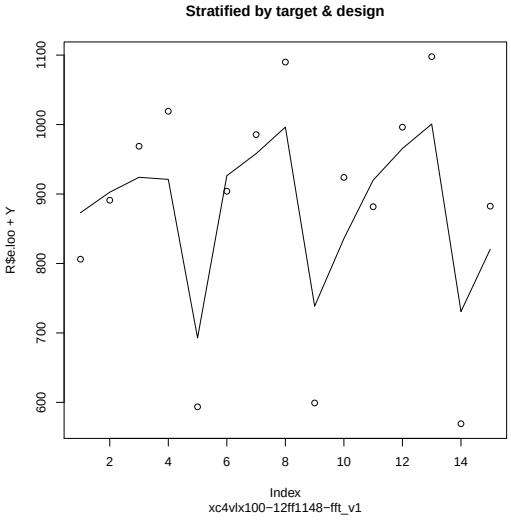


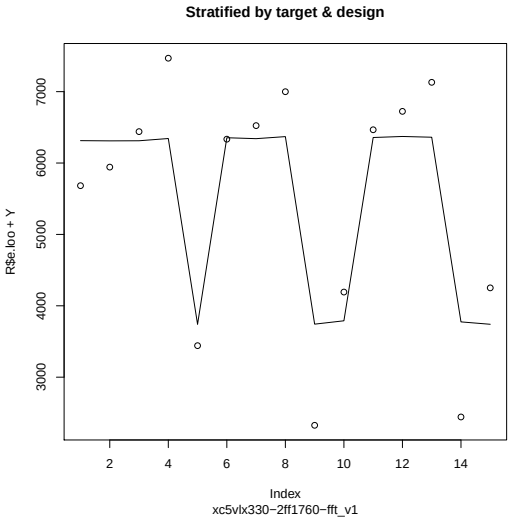

total_power
0
1000
2000
3000
4000
5000
6000
fft_v1
fft_v5
design

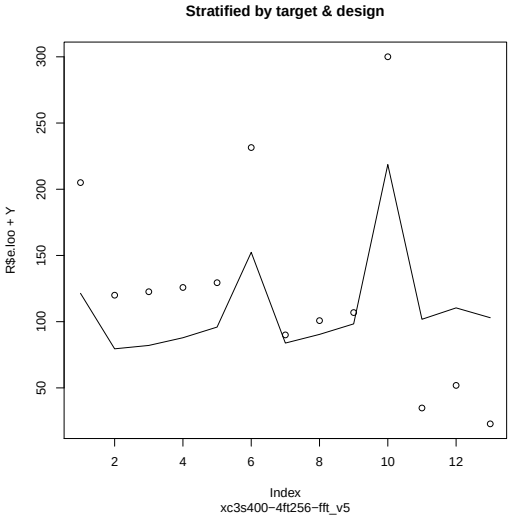

total_power
0
1000
2000
3000
4000
5000
xc3s400−4ft256
xc4vlx100−12ff1148
xc5vlx330−2ff1760
d$target
d$design
fft_v1
fft_v5

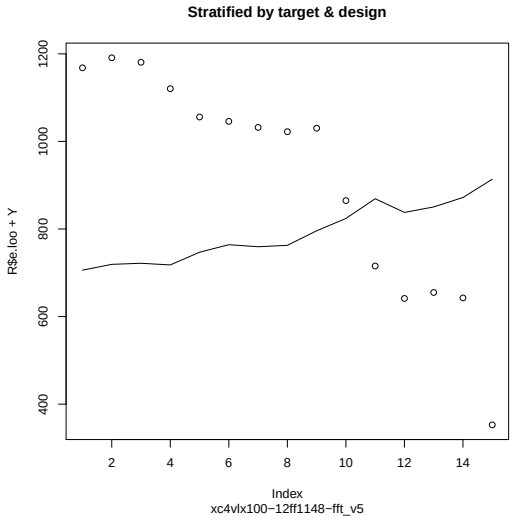

total_power
0
1000
2000
3000
4000
5000
fft_v1
fft_v5
d$design
d$target
xc5vlx330−2ff1760
xc4vlx100−12ff1148
xc3s400−4ft256

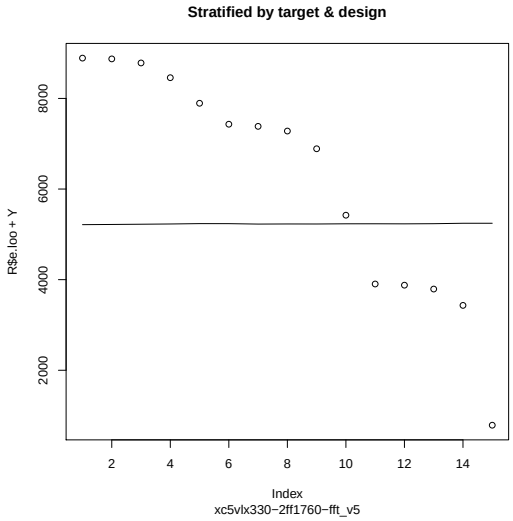
B.4.2 Modélisation linéaire

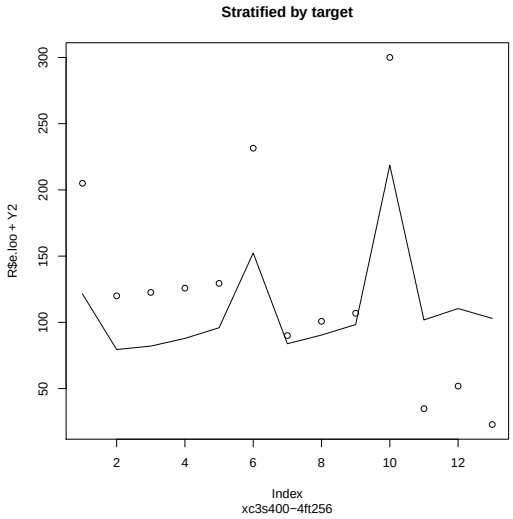


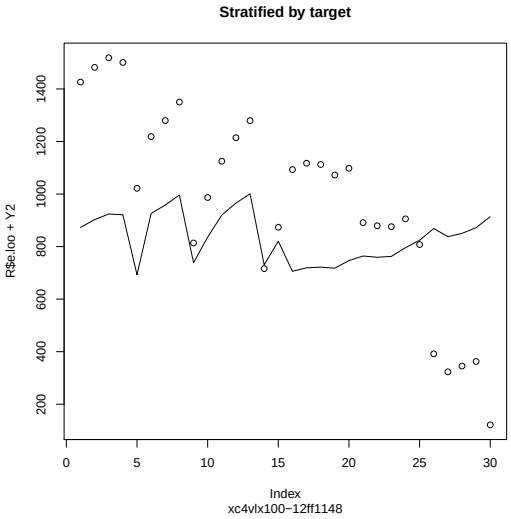


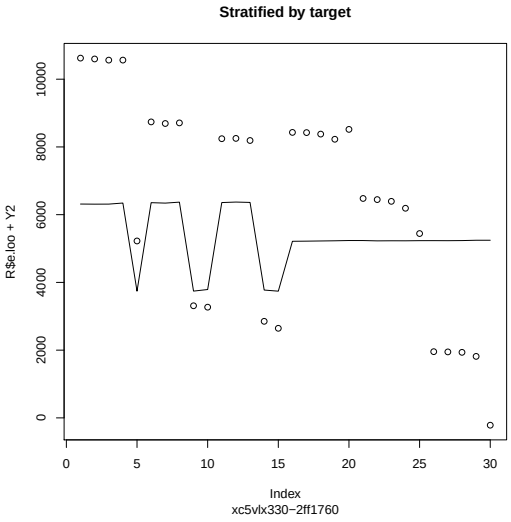


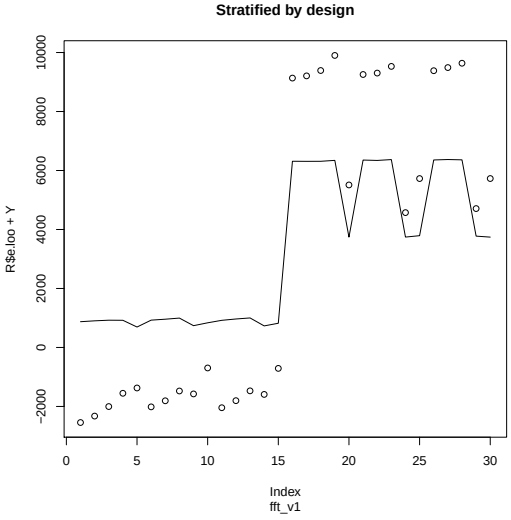

Stratified by target & design
R$e.loo + Y
Index
xc4vlx100−12ff1148−fft_v5

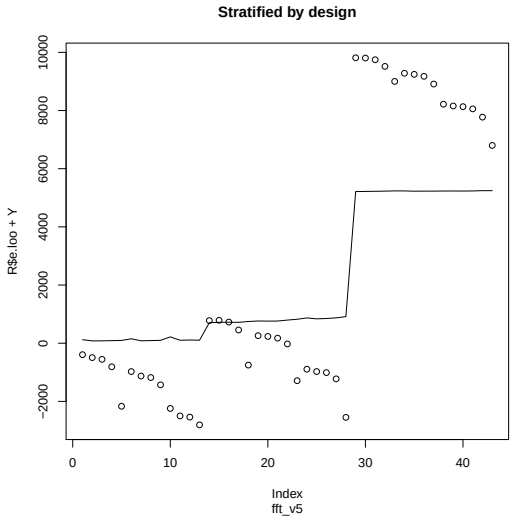

Stratified by target & design
R$e.loo + Y
Index
xc5vlx330−2ff1760−fft_v5


Stratified by target
R$e.loo + Y2
Index
xc3s400−4ft256


Stratified by target
R$e.loo + Y2
Index
xc4vlx100−12ff1148


Stratified by target
R$e.loo + Y2
Index
xc5vlx330−2ff1760


Stratified by design
R$e.loo + Y
Index
fft_v1


Stratified by design
R$e.loo + Y
10000
8000
6000
4000
2000
0
−2000
0
10
20
30
40
Index
fft_v5

Annexe C

Programmes d'analyse

Cet annexe liste les différents scripts R utilisés pour générer les graphiques de l'annexe B à partir des données de l'annexe A.

C.1 Script lin.R

Le premier fichier, « lin.R », est une librairie utilisée par les deux autres fichiers. Il implémente l'inférence de modèles linéaires.

```
library(MASS)

regrlin<-function(X,Y,X.ts=NULL,lambda=1e-3){

  n<-NCOL(X) # number input variables
  p<-n+1
  N<-NROW(X) # number training data

  XX<-cbind(array(1,c(N,1)),X)

  H1<-ginv(t(XX)%*%XX+lambda*diag(p))
  beta.hat<-H1%*%t(XX)%*%Y
  H<-XX%*%H1%*%t(XX)
  Y.hat<-XX%*%beta.hat
  e<-Y-Y.hat
  var.hat.w<-(t(e)%*%e)/(N-p)
  MSE.emp<-mean(e^2)
  e.loo<-e/(1-diag(H))
  MSE.loo<-mean( e.loo^2 )
  NMSE<-mean( e.loo^2 )/(sd(Y)^2)
  Y.hat.ts<-NULL
  if (!is.null(X.ts)){
    N.ts<-NROW(X.ts)
    if (is.vector(X.ts) & n>1 ){
      Y.hat.ts<-c(1,X.ts)%*%beta.hat
```

```
        } else {
        XX<-cbind(array(1,c(N.ts,1)),X.ts)
        Y.hat.ts<-XX%*%beta.hat
      }
     }
   list(e=e,beta.hat=beta.hat,
         MSE.emp=MSE.emp,sdse.emp=sd(e^2),var.hat=var.hat.w,
                  MSE.loo=MSE.loo,sdse.loo=sd(e.loo^2),
         Y.hat=Y.hat,Y.hat.ts=Y.hat.ts,e.loo=e.loo)
}


PRESS <- function(mod) {
   res <- resid(mod)
   hat <- lm.influence(mod)$hat
   list(MSE.loo=mean(  (res/(1-hat))^2  ),MSE.emp=mean(res^2))
}


regrlin.d<-function(X,Y,X.ts=NULL,lambda=1e-5){

   n<-NCOL(X) # number input variables
   p<-n+1
   N<-NROW(X) # number training data
   N.ts<-NROW(X.ts)
   XX<-cbind(array(1,c(N,1)),X)
   G<-XX%*%t(XX)   ## [N,N]
   H1<-ginv(t(XX)%*%XX+lambda*diag(p))

   H<-XX%*%H1%*%t(XX)%*%Y-ginv(G)%*%Y


   alpha.hat<-ginv(G+lambda*diag(N))%*%Y  ##[N,1]
   beta.hat<-t(XX)%*%alpha.hat
   Y.hat<-G%*%alpha.hat

   e<-Y-Y.hat
   var.hat.w<-(t(e)%*%e)/(N-p)

   K<- cbind(array(1,c(N.ts,1)),X.ts)%*%t(XX)  ##[Nts,N]

   MSE.emp<-mean(e^2)



   Y.hat.ts<-K%*%alpha.hat


   list(e=e,beta.hat=beta.hat,alpha.hat=alpha.hat,
```

```
        MSE.emp=MSE.emp, sdse.emp=sd(e^2), var.hat=var.hat.w,
        Y.hat=Y.hat, Y.hat.ts=Y.hat.ts)
}


demo<-function(){

N<-100
n<-10
X<-array(rnorm(N*n),c(N,n))
Y<-sin(X[,1]*X[,2])

R<-regrlin(X,Y,X)

R2<-regrlin.d(X,Y,X,lambda=1e-5)
browser()
}
```

C.2 Script main.R

Le second fichier, « main.R », réalise l'affichage des graphiques de statistiques descriptives et de l'analyse par stratification.

```
rm(list=ls())
source("lin.R")
D<-read.table("rdata_86points.txt",h=TRUE)

N<-NROW(D)
## outputs: response_time, lut_number, area_fraction,
##          total_power, max_delay, ticks_number,

## factor inputs: design (2 levels: "fft_v1" "fft_v5")
##                target (3 levels: "xc3s400-4ft256"
##                                  "xc4vlx100-12ff1148"
##                                  "xc5vlx330-2ff1760")
##                tool (1 level) EXCLUDED
## real inputs  : point_width
##                points_number


##  other: synth_time

par(ask=T) #ask for confirmation
##print(cor(D[,c("response_time","lut_number",
#                "area_fraction","total_power")],
##use="pairwise"))

## lut_number and area_fraction are redundant
```

```
output<-"response_time"
inputs<-c("design","target","point_width","points_number")

I<-which(!is.na(D[,output]) & !is.nan(D[,"total_power"]))
Y<-D[I,output]

X<-D[I,inputs]
n<-NCOL(X)

d<-data.frame(X,Y)
names(d)[1:n]<-inputs
names(d)[n+1]<-"Y"

####### VISUALIZATIONS

interaction.plot(d$target,d$design,d$Y,ylab=output)
interaction.plot(d$design,d$target,d$Y,ylab=output)
plot(Y~target+design,d,ylab=output)

########## FITTING LINEAR MODEL
mod3<-lm(Y~ .,data=d)
print(summary(mod3))

mod1<-lm(Y~target+design,data=d)
print(summary(mod1))

mod2<-lm(Y~ points_number + point_width,data=d)
print(summary(mod2))


readline("Type for stratified analysis")

################################### DES & TARGET
L.design<-levels(D$design)
L.target<-levels(D$target)
Eloo<-NULL
for (ld in L.design){
  for (lt in L.target){
    I<-which(D$design==ld & D$target== lt
       & !is.na(D[,output]) & !is.nan(D[,"total_power"]))
   ##  I<-which( D$target== lt & !is.na(D[,output]))
    D2<-D[I,]

    if (length(I)>0 & sd(D2[,output])>0){
      cat("design=",ld,"target=",lt,"\n")

      Y<-D2[,output]
```

```
      X<-as.matrix(D2[,c("points_number","point_width")])
      n<-NCOL(X)

      R<-regrlin(X,Y)

      Eloo<-c(Eloo,R$e.loo)
      print(Y)
      print(R$e.loo+Y)
      plot(R$e.loo+Y,main=paste("Stratified by target & design"),
           sub=paste(lt,ld,sep="-"))
      lines(Y)
    }
  }
}

cat("Stratified by target & design NMSE=",mean(Eloo^2)/var(D[,
    output],na.rm=T), "\n --------\n")

#####################################  TARGET
 Eloo<-NULL
  for (lt in L.target){
    I<-which( D$target== lt & !is.na(D[,output])
              & !is.nan(D[,"total_power"]))
    D2<-D[I,]

    if (length(I)>0 & sd(D2[,output])>0){
      cat("design=",ld,"target=",lt,"\n")

      Y2<-D2[,output]


      X2<-as.matrix(D2[,c("points_number","point_width")])
      n<-NCOL(X2)

      R<-regrlin(X2,Y2)

      plot(R$e.loo+Y2,main=paste("Stratified by target") ,sub=lt)
      lines(Y2)

      Eloo<-c(Eloo,R$e.loo)
    }
}

cat("Stratified by target NMSE=",mean(Eloo^2)/var(D[,
    output],na.rm=T), "\n --------\n")

#####################################  DESIGN
Eloo<-NULL
for (ld in L.design){
```

```
  I<-which(D$design==ld & !is.na(D[,output])
           & !is.nan(D[,"total_power"]))

    D2<-D[I,]

    if (length(I)>0 & sd(D2[,output])>0){
      cat("design=",ld,"target=",lt,"\n")

      Y<-D2[,output]


      X<-as.matrix(D2[,c("points_number","point_width")])
      n<-NCOL(X)

      R<-regrlin(X,Y)

      plot(R$e.loo+Y,main=paste("Stratified by design"),sub=ld)
      lines(Y)

      Eloo<-c(Eloo,R$e.loo)
  }
}

cat("Stratified by design NMSE=",mean(Eloo^2)/var(D[,
    output],na.rm=T), "\n --------\n")
```

C.3 Script sel.R

Le troisième fichier, « sel.R », réalise la sélection de variable avec la méthode de *forward selection.*

```
rm(list=ls())
source("lin.R")
val<-function(D,Y){

  w.na<-which(!is.na(Y))

  if (is.null(dim(D))){
    D<-D[w.na]
    n<-1
  } else {
    D<-D[w.na,]
    n<-NCOL(D)
  }
  Y<-Y[w.na]

  N<-NROW(D)
```

```
if (n==1){
  if (is.factor(D))
    w.f<-1
  else
    w.f<-NULL
} else {
  w.f<-which(sapply(D,is.factor))
}
w.nf<-setdiff(1:n,w.f)

Eloo<-NULL
if (length(w.f)>0){
  if (length(w.f)==1){
    if (n==1)
      Levs<-levels(D)
    else
      Levs<-levels(D[,w.f])
    cmb<-array(Levs,c(length(Levs),1))
 } else {
   Levs<-sapply(D,levels)
   nLevs<-unlist(lapply((sapply(D,levels)),length))

   cmb<-expand.grid(Levs[w.f])
   cmb<-cmb[,names(Levs)[w.f]]
 }

  for (i in 1:NROW(cmb)){
    I<-1:N
    ##print(cmb[i,])
    for (j in 1:length(cmb[i,]))
      if (n==1){
        I<-intersect(I,which(D==cmb[i,j]))
     } else
        I<-intersect(I,which(D[,w.f[j]]==cmb[i,j]))

    if (length(I)>0)
      if (length(w.nf)>0){

        Eloo<-c(Eloo,regrlin(as.matrix(D[I,w.nf]),Y[I])$e.loo)
      }
      else {
        e<-Y[I]-mean(Y[I])
        NI<-length(I)

        Eloo<-c(Eloo,NI/(NI-1)*e)

      }

  }
```

```
  } else {

    if (is.vector(D))
       Eloo<-c(Eloo,regrlin(D,Y)$e.loo)
    else
      Eloo<-c(Eloo,regrlin(as.matrix(D[,w.nf]),Y)$e.loo)
  }

  sqrt(mean(Eloo^2))
}


########################################
D<-read.table("rdata_86points.txt",h=TRUE)

N<-NROW(D)
##c("response_time","lut_number","area_fraction","total_power")

names.inputs<-c("target","design","points_number","point_width")
names.output<-c("response_time")
I<-which(!is.na(D[,names.output]) & !is.nan(D[,"total_power"]))
X<-D[I,names.inputs]
Y<-D[I,names.output]
n<-NCOL(X)

### FEATURE SELECTION BY FORWARD
selected<-NULL
for (s in 1:4){
  E<-numeric(n)+Inf
  for (i in setdiff(1:n,selected)){
    feat<-c(selected,i)
    E[i]<-val(X[,feat],Y) ## error in leave one of model having
    print(E[i])           ## "feat" as inputs
  }
  selected<-c(selected,which.min(E))
  cat(names.inputs[selected],"->",names.output,
      "E=",min(E)^2/var(Y,na.rm=T),"\n")
}

print(selected) # selected variables
```

Références

Bibliographie

[BK02] G. Bontempi and W. Kruijtzer. A data analysis method for software performance prediction. In *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, pages 971–976. IEEE, 2002.

[BNH06] F. Berthelot, F. Nouvel, and D. Houzet. Partial and dynamic reconfiguration of fpgas : a top down design methodology for an automatic implementation. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 4–pp. IEEE, 2006.

[Bon11] G. Bontempi. *Statistical foundations of machine learning*, 2011. Université Libre de Bruxelles. Support de cours.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition.* The MIT Press, September 2001.

[Doa09] Anh Vu Doan. Analyse multicritère appliquée au problème de la conception des circuits 3d. Master's thesis, Université Libre de Bruxelles, 2009.

[DT01] W.J. Dally and B. Towles. Route packets, not wires : On-chip interconnection networks. In *Design Automation Conference, 2001. Proceedings*, pages 684–689. IEEE, 2001.

[EPTP07] C. Erbas, A.D. Pimentel, M. Thompson, and S. Polstra. A framework for system-level modeling and simulation of embedded systems architectures. *EURASIP Journal on Embedded Systems*, 2007(1) :2–2, 2007.

[Eri03] Jeff Erickson. *Combinatorial Algorithms*, 2003. University of Illinois. Support de cours.

[FGE05] J. Figueira, S. Greco, and M. Ehrgott. *Multiple Criteria Decision Analysis : State of the Art Surveys.* Springer Verlag, Boston, Dordrecht, London, 2005.

[GAGS09] D.D. Gajski, S. Abdi, A. Gerstlauer, and G. Schirner. *Embedded System Design : Modeling, Synthesis and Verification.* Springer Verlag, 2009.

[GK97] D.D. Gajski and J. Kleinsmith. *Principles of digital design*, volume 42. Prentice Hall, 1997.

[Gum10] Nastassia Gumuchdjian. Feasibility study for integrating a model driven engineering approach to circuit design. Master's thesis, Université Libre de Bruxelles, 2010.

[HNG09] B. Holland, K. Nagarajan, and A.D. George. Rat : Rc amenability test for rapid performance prediction. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 1(4) :1–31, 2009.

[HTF01] T. Hastie, R. Tibshirani, and J. Friedman. The elements of statistical learning. 2001.

[KB05] Souha Kamoun and Pierre Boulet. Spécification des propriétés dans un processus d'aide à l'optimisation. 2005.

[LDA+06] Y. Le Moullec, J.P. Diguet, N.B. Amor, T. Gourdeaux, and J.L. Philippe. Algorithmic-level specification and characterization of embedded multimedia applications with design trotter. *The Journal of VLSI Signal Processing*, 42(2) :185–208, 2006.

[Led10] Sébastien Leduc. Design de plateforme électronique via uml. Technical report, Université Libre de Bruxelles, 2010.

[Mil11] D. Milojevic. *Architectures numériques avancées*, 2011. Université Libre de Bruxelles. Support de cours.

[MP02] S. Mohanty and V.K. Prasanna. Rapid system-level performance evaluation and optimization for application mapping onto soc architectures. In *ASIC/SOC Conference, 2002. 15th Annual IEEE International*, pages 160–167. IEEE, 2002.

[Red10] Mohamed Akli Redjedal. Optimisation multicritère pour le placement d'applications intensives sur système-sur-puce. Master's thesis, Université de Lille, 2010.

[Ric10] A. Richard. *Development and validation of NESSIE : a multi-criteria performance estimation tool for SoC/Développement.* PhD thesis, Université Libre de Bruxelles, 2010.

[Van09] Alexis Vander Biest. *Developing Multi-Criteria Performance Estimation Tools for Systems-on-Chip.* PhD thesis, Université Libre de Bruxelles, 2009.

[VG02] F. Vahid and T. Givargis. *Embedded system design : a unified hardware/-software introduction.* Wiley, 2002.

Webographie

[1] Integrating, evaluating and characterising real-time multiprocessors systems on reconfigurable platforms. `http://parts.ulb.ac.be/page.php3?listing=AY.html`. [En ligne ; Page disponible le 29 mai 2011].

[2] Interview de Pierre Boulet à propos du MARTE. `http://www.inria.fr/actualites/inedit/mars2010/marte-langage.fr.html`. [En ligne ; Page disponible le 25-mai-2010].

[3] Présentation du projet NESSIE. `http://beams.ulb.ac.be/beams/research/projects/current/119.html`. [En ligne ; Page disponible le 25-mai-2010].

[4] Site de l'équipe BEAMS. `http://beams.ulb.ac.be/`. [En ligne ; Page disponible le 25-mai-2010].

[5] Site de l'équipe DaRT à l'INRIA Lille Nord Europe. `http://www.inria.fr/recherche/equipes/dart.fr.html`. [En ligne ; Page disponible le 25-mai-2010].

[6] Wiki de l'omg sur marte. `http://www.omgwiki.org/marte/`. [En ligne ; Page disponible le 8 septembre 2010].

[7] OpenCores. Confluence radix 2 fft core. `http://opencores.org/project,cf_fft`, 2011. [Online ; accessed 26-May-2011].

[8] OpenCores. Radix 4 fft core. `http://opencores.org/project,cfft`, 2011. [Online ; accessed 26-May-2011].

[9] Wikipedia. Box plot — wikipedia, the free encyclopedia, 2011. [Online ; accessed 29-May-2011].

[10] Wikipedia. Cordic — wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=CORDIC&oldid=430739998`, 2011. [Online ; accessed 26-May-2011].

[11] Wikipedia. Fast fourier transform — wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Fast_Fourier_transform&oldid=428261485`, 2011. [Online ; accessed 29-May-2011].

[12] Wikipedia. Moore's law — wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Moore%27s_law&oldid=429603897`, 2011. [Online ; accessed 21-May-2011].

[13] Wikipédia. Dhrystone — wikipédia, l'encyclopédie libre. `http://fr.wikipedia.org/w/index.php?title=Dhrystone&oldid=59135702`, 2010. [En ligne ; Page disponible le 11-mai-2011].

[14] Wikipédia. Optimum de pareto — wikipédia, l'encyclopédie libre. `http://fr.wikipedia.org/w/index.php?title=Optimum_de_Pareto&oldid=59929278`, 2010. [En ligne ; Page disponible le 24-mai-2011].

[15] Wikipédia. Réseau de Petri — Wikipédia, l'encyclopédie libre. `http://fr.wikipedia.org/w/index.php?title=R%C3%A9seau_de_Petri&oldid=52094831`, 2010. [En ligne ; Page disponible le 25-mai-2010].

[16] Wikipédia. Complementary metal oxide semi-conductor — wikipédia, l'encyclopédie libre. `http://fr.wikipedia.org/w/index.php?title=Complementary_metal_oxide_semi-conductor&oldid=62383365`, 2011. [En ligne ; Page disponible le 24-mai-2011].