Aide à la conception de systèmes sur puce hétérogènes par l'exploration paramétrable des solutions au niveau système

Yannick Le Moullec

N° d'ordre : 24

THÈSE

présentée devant

L'Université De Bretagne Sud

pour obtenir le grade de

Docteur De
L'Université De Bretagne Sud

Mention :

Électronique et Informatique Industrielle

École Doctorale Pluridisciplinaire

Composante Universitaire :

UFR Sciences
et
Sciences de l'Ingénieur

par

Yannick Le Moullec

Aide à la conception de systèmes sur puce hétérogènes par l'exploration paramétrable des solutions au niveau système

soutenue le 10 Avril 2003 devant la commission d'examen composée de :

M. :	O. Sentieys	Professeur, ENSSAT, Lannion	Rapporteur
M. :	L. Torres	Maître de Conférence, HDR, LIRMM, Montpellier	Rapporteur
M. :	J-L. Philippe	Professeur, LESTER, Lorient	Directeur de thèse
M. :	M. Auguin	Professeur, I3S, Nice	Examinateur
M. :	C. Gamrat	Ingénieur, CEA, Saclay	Examinateur
M. :	J-Ph. Diguet	Maître de Conférence, LESTER, Lorient	Examinateur
M. :	A. Kountouris	Ingénieur, Mitsubishi Electric ITE, Rennes	Invité
M. :	P. Koch	Maître de Conférence, CISS, Aalborg, DK	Invité

Laboratoire d'Électronique des Systèmes TEmps Réel

Remerciements

Ce travail a été effectué au sein du groupe Adéquation Application Système (AAS) du Laboratoire d'Électronique des Systèmes TEmps Réel (LESTER), de l'Université de Bretagne Sud. Je remercie tout d'abord Mr. Eric Martin, directeur du LESTER, de m'avoir accueilli dans son laboratoire et Mr. Jean-Luc Philippe, mon directeur de thèse, pour l'opportunité qu'il m'a donné. Je le remercie aussi pour son accompagnement et ses conseils.

Je remercie ensuite Jean-Philippe Diguet, pour la qualité de son encadrement, sa disponibilité, ses conseils et ses encouragements.

Je tiens ensuite à remercier tous les membres du groupe AAS avec qui j'ai été amené à travailler, parmi lesquels Fabrice Bertrand, Guy Gogniat, Sébastien Bilavarn, Abdenour Azzedine ... Je remercie tout particulièrement Dominique Heller et Thierry Gourdeaux pour leur travaux respectivement sur la structure de donnée et le Parser. Sans eux l'obtention de résultats aurait été impossible.

Je remercie également Peter Koch, Ole Olsen, Jens-Peter Holmegaard, Daniel Lázaro Cuadrado et Rikke Dyg Klemmensen de l'Université d'Aalborg pour leur accueil chaleureux lors d'un séjour de recherche de trois mois au sein du laboratoire "Embedded System Group".

Je remercie aussi tous les membres du LESTER et leur souhaite à chacun de réussir dans leurs projets.

Je remercie ma famille et mes amis pour leur soutien et leurs encouragements. Tack till Elisabet.

Je remercie Messieurs Olivier Sentieys, et Lionel Torres, pour avoir acceptés d'être rapporteurs de ce travail ainsi que le président et les membres du jury.

Résumé

Les travaux présentés dans ce document concernent la conception conjointe matériel/logiciel (*Hardware/Software Co-design*) d'applications orientées multimédia (essentiellement de type "enfouies" ou *embedded*) à un niveau d'abstraction dit *système*. Ces travaux sont menés au sein du L.E.S.T.E.R dans le groupe de recherche Adéquation Architecture Systèmes (AAS).

Les systèmes enfouis sont de plus en plus présents dans la vie quotidienne, que ce soit pour un usage professionnel ou personnel. On peut citer par exemple les téléphones mobiles, les assistants personnels (PDA), les consoles de jeux vidéos portables, les lecteurs multimédias portables (MP3 et consorts). On trouve aussi de plus en plus de systèmes enfouis dans les automobiles, les appareils domestiques "intelligents" etc. Les fonctions qui peuvent être intégrées dans ce type de système peuvent être, par exemple, de type traitement de signal numérique (filtrage, compression-décompression audio-vidéo,...), de type télécommunication (protocole réseau,...) ou bien encore contrôle/commande (domotique...).

La complexité grandissante des applications fait qu'il est nécessaire de pouvoir aborder leurs conceptions à des niveaux d'abstractions élevés. En effet, il est très intéressant de travailler à ces niveaux (par exemple au niveau système) car les gains (en surface/temps/consomation/coût) qu'il est possible d'obtenir par diverses transformations (tant algorithmiques qu'architecturales) sont proportionnels au niveau d'abstraction auquel on se situe. De plus, les décisions prises au niveau système peuvent avoir un impact très important en terme de développement industriel. En effet, une mauvaise adéquation application/architecture (architecture sur/sous-dimensionnée ou mal adaptée aux caractéristiques de l'application) peut imposer, soit de mettre sur le marché un produit trop cher ou peu performant, soit

de relancer un cycle de conception entraînant des délais pouvant être rédhibitoires.

L'ensemble de ces décisions à prendre peut être vu comme un espace de solutions potentielles à parcourir. Celui-ci étant très vaste pour une application (ensemble des couples algorithmes / architectures), il est nécessaire de **l'explorer** et d'effectuer des choix afin de le réduire. On conçoit aisément que cette exploration, lorsqu'elle est effectuée au niveau système, doit présenter un bon compromis vitesse (espace des solutions très vaste) Vs. précision (les choix faits sont lourds en conséquence pour la suite du flot de conception).

Les outils de conception actuels, pour de tels systèmes, sont connus sous le nom d'outils de **codesign** et se situent à des niveaux d'abstractions relativement faibles. En outre, la plupart de ces outils partent d'une architecture cible figée (matériel et logiciel, par exemple un processeur et un ASIC) pour laquelle est choisie l'implantation soit matérielle (sur ASIC ou FPGA), soit logicielle de telle ou telle fonction précaractérisée. Ces outils ne permettent donc pas d'explorer les architectures propres aux différentes fonctions (estimation statique Vs. estimation dynamique). Il y a donc un nouveau pas à franchir, celui de l'exploration système comme moyen de choisir l'architecture cible ou bien encore de fixer les paramètres pour une architecture cible figée mais générique.

La méthode proposée dans cette thèse vise à réduire progressivement l'espace des solutions en permettant au concepteur d'effectuer des compromis entre plusieurs solutions, et ce à chaque niveau d'abstraction en s'appuyant sur un découpage fonctionnel et hiérarchique de l'application qui spécifie progressivement les aspects contrôles, traitements et transferts de données. La méthode est composée des éléments suivant : i) spécification de l'application dans un langage de haut niveau ; ii) caractérisation de l'application par un ensemble de métriques définissant l'orientation transfert mémoire, traitement ou contrôle ainsi que le parallélisme potentiel de ses sous-fonctions ; iii) estimation système dynamique des performances par l'exploration et l'exploitation du parallélisme ; iv) sélection des solutions prometteuses en vue de phases de projections architecturale et logicielle.

Table des matières

Liste des figures

Liste des algorithmes

Introduction

L'électronique numérique a envahi notre quotidien et ce phénomène ne cesse de s'amplifier. En particulier, le marché des applications de type systèmes enfouis *(embedded systems)* a littéralement explosé ces dernières années. Ces applications vont des téléphones mobiles et assistants numériques portatifs (PDA), aux systèmes de commandes et de contrôles dans les automobiles en passant par les appareils audio-visuels (set-top box, lecteur multimedia portatif avec accès internet sans fil). Les fonctionnalités offertes par les systèmes enfouis sont de plus en plus complexes. Cet accroissement permanent en complexité est possible grâce à l'évolution régulière de la technologie qui permet d'intégrer toujours plus de transistors au mm^2. Cette évolution est souvent quantifiée selon "la loi de Moore" [1] qui prévoit le doublement de la densité de transistors sur un circuit intégré tous les deux ans. Ainsi, d'ici 2010 la technologie devrait permettre l'intégration d'un milliard de transistors sur une puce contre quelques dizaines à une centaine de millions actuellement (42 millions pour le processeur Intel "Pentium IV", 125 Millions pour le processeur graphique NVidia "GeForce Fx", en technologie 0.13 μM). La complexité croissante des applications et les nombreux facteurs économiques et techniques dont dépend le succès commercial de tels produits font des méthodes et outils de conception des facteurs décisifs. Alors que la technologie ne cesse d'évoluer, un fossé se creuse entre la productivité des outils de conception actuels et les besoins exprimés par les concepteurs et le marché des systèmes enfouis. Ainsi, la durée de vie d'une génération de produits peut désormais être plus courte que sa durée de conception. Afin de réduire ce fossé, ou du moins éviter qu'il ne s'agrandisse, de nouveaux outils de conception sont nécessaires. Les caractéristiques que ceux-ci devront avoir sont, entre autres, une plus grande rapidité de conception, une meilleure adéquation avec les applications

visées, la possibilité de concevoir plus facilement des produits évolutifs et sûrs. Les travaux présentés dans ce mémoire apportent une contribution visant à réduire ce fossé et à faciliter le travail des concepteurs de tels systèmes en offrant très tôt dans le flot de conception une estimation de l'efficacité des choix architecturaux.

Les systèmes enfouis

Afin de mieux situer le cadre des travaux présentés dans ce mémoire il convient de définir plus précisément ce que sont les systèmes enfouis (également appelés systèmes embarqués). S'il n'existe pas de définition "officielle" il est communément admis que, d'une manière très générale, un système enfoui peut être vu comme un système électronique dont le fonctionnement repose sur un microprocesseur (avec ou sans système d'exploitation), mais qui n'est pas un ordinateur (au sens *Personal Computer*). Pour être plus précis nous pouvons dire qu'un système enfoui est un système électronique dédié à une ou à un ensemble d'applications prédéfinies et dont la mise à jour ne peut être que très limitée (par exemple chargement de nouveaux logiciels ou re-programmation de matériel reconfigurable). De plus, un système enfoui peut être éventuellement multi-domaines, c.a.d analogique et numérique, le domaine analogique comprenant par exemple la partie RF d'un téléphone mobile, le domaine numérique comprenant le traitement numérique du signal et l'interface utilisateur. La partie numérique est souvent hétérogène, à savoir qu'elle fait appel à des composantes logicielles et matérielles. De plus, nous pouvons distinguer les systèmes fixes et les systèmes mobiles. Pour ces derniers la maîtrise de la consommation est un facteur essentiel à leur réussite commerciale ; la recherche sur cet aspect est très dynamique. Enfin, la puissance de calcul des systèmes enfouis peut être très variable. Les systèmes les moins complexes intègrent de simples micro-processeurs et micro-contrôleurs (de nombreux produits font appel aux 8051, 68HC05, ou autres dérivés de 68000) alors que les systèmes plus évolués font appel aux dernières générations de micro-processeurs et de DSP (*Digital Signal Processor*), ainsi qu'à des accélérateurs matériels dédiés.

Comme indiqué plus haut, les systèmes enfouis sont généralement implantés

sous la forme de composantes logicielles et matérielles. Les composantes logicielles peuvent être implantées sur des processeurs à usage général (*GPP*) ainsi que sur des processeurs à usage spécifique (*DSP, ASIP*). Les composantes matérielles sont elles implantées soit sur des composants dédiés (*ASIC*) soit sur des composants reprogrammables (*FPGA*). Il est également important de souligner les cas des systèmes-sur-une-puce (*SoC*), qui intègrent sur une seule puce de silicium les composantes logicielles et matérielles. En effet, bon nombre de processeurs (ou cœur de processeur) sont disponibles sous la forme de "propriétées intellectuelles" (*IP*, spécification, à un niveau d'abstraction donné, d'un composant matériel ou logiciel, et destinée à être re-utilisée par une tierce partie) synthétisables, ce qui permet de les intégrer à coté des parties matérielles, soit sur ASIC soit sur FPGA de dernières générations. Ces derniers intègrent un nombre considérable de portes (1,5 millions pour le modèle Altera EP20K1500E en technologie 0.13 μM), des blocs de traitement du signal numérique (modules matériels de type multiplieur, additionneur-soustracteur, accumulateur, registres pipelinés) comme la famille Stratix chez Altera. On peut enfin citer les FPGA intégrant d'origine des processeurs telle la famille Virtex-II Pro (pouvant disposer de 0 à 4 processeurs de type PowerPc405).

Problématique

Comme nous venons de le voir, les systèmes enfouis font appel à des architectures hétérogènes logicielles/matérielles. Afin de concevoir de tels systèmes, les méthodes de conception conjointe logiciel/matériel (*codesign*) sont utilisées. Elles permettent de définir les sous-ensembles du système à intégrer et d'effectuer leur partitionnement sur les cibles logicielles et matérielles. Comme nous le verrons dans le chapitre 1, de nombreux travaux de recherches et outils commerciaux s'intéressent à cette problématique. Cependant, la complexité sans cesse grandissante de ces systèmes et les diverses contraintes auxquelles doivent faire face leurs concepteurs, font qu'il est nécessaire de faire appel à de nouvelles méthodes de conception, ce que nous allons montrer maintenant. Afin d'assurer le succès commercial de tels systèmes il est primordial que ceux-ci répondent aux attentes des consommateurs parmi les-

quelles nous pouvons citer : nouvelles fonctionnalités, petite taille, poids léger, faible consommation en énergie, simplicité d'utilisation et bien sûr coût acceptable. L'industrie des systèmes enfouis est donc commandée, d'une part, par la pression du délai de mise sur le marché (*time-to-market*) des nouvelles générations de produits, et d'autre part, par la nécessité d'en augmenter la puissance de calcul et d'en diminuer la consommation d'énergie tout en minimisant les coûts de fabrication. Afin d'y parvenir, il est nécessaire d'évaluer un très grand nombre de solutions potentielles. Deux grandes difficultés apparaissent dès lors : d'une part un concepteur n'a pas à sa disposition tous les outils de développement pour toutes les cibles potentiellement intégrables au système, et d'autre part les outils traditionnels de co-design se situent à des niveaux d'abstractions relativement bas, où les modèles utilisés sont détaillés et précis. Ces outils, qui sont arrivés à un degré de maturité assez important restent cependant trop lents, et ne permettent donc pas d'explorer, dans des délais raisonnables, les gigantesques espaces de conception inhérents aux systèmes actuels. La taille de ces espaces de solutions est en effet liée à la liberté de création offerte par l'évolution de la technologie. Les formidables possibilités d'optimisations qui en découlent conduisent en effet à un accroissement important du nombres de solutions et donc à une plus grande difficulté en ce qui concerne les choix que le concepteur devra effectuer.

C'est pourquoi, une nouvelle génération de méthodes a été développée, il s'agit des "méthodes au niveau système". Les avantages des méthodologies au niveau système sont nombreux : *i)* comme représenté sur la figure 1, les décisions prises au niveau système, c.a.d tôt dans le flot de conception, ont un impact déterminant sur les performances (puissance de calcul, consommation, coût) finales du système. *ii)* elles permettent de réduire les coûts de développement grâce à la réutilisation de modules existants et de créer des systèmes fonctionnels et performants du premier coup (*first-time correctness*). Par exemple, dans le domaine des terminaux mobiles [2] il a été montré que 50% des optimisations sur l'architecture peuvent être effectuées en un mois en opérant au niveau système alors qu'il faut un an aux niveaux logique et physique pour n'obtenir que 20% d'amélioration.

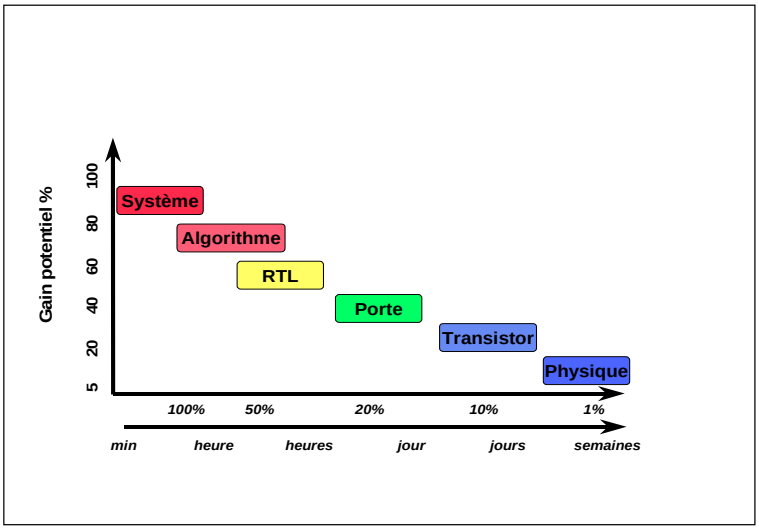

Fig. 1: *Gains potentiels en consommation en fonction du niveau d'exploration. (Source : J.Sproch, ISPLED 97)*

Contribution de la thèse

C'est dans le contexte des méthodes au niveau système que les travaux présentés dans ce mémoire se positionnent.

Tout d'abord, le compromis consommation/surface/performance requis par les systèmes enfouis, notamment pour les télécommunications, exige une utilisation extrêmement optimisée du silicium. Ainsi, les objectifs fixés en termes de rendements performance/surface et consommation/surface ne pourront être atteints qu'à travers l'exploitation massive du parallélisme spatial, l'utilisation d'horloges à fréquences relativement faibles (afin de limiter la consommation énergétique des appareils portables) et le recours plus ou moins intensif à des ressources matérielles spécifiques [3] (les nouvelles architectures de processeurs vont également dans ce sens [4], [5]). La recherche de ce parallélisme et de ce matériel spécifique est une étape à part entière du flot de conception. Elle se situe avant le choix complet de l'architecture cible, à un niveau algorithmique, et peut être qualifiée d'exploration architecturale au niveau système.

Les méthodes et outils permettant une réelle exploration de l'espace de conception dans des temps raisonnables sont encore rares [6]. La plupart des outils de co-design partent d'une architecture cible figée (matériel et logiciel, par exemple

un processeur et un ASIC) pour laquelle est choisie l'implantation soit matérielle (sur ASIC ou FPGA), soit logicielle de telle ou telle fonction pré-caractérisée. Il y a généralement en librairie une version logicielle et une ou deux versions matérielles prédéfinies pour chaque fonction. Ces outils ne permettent donc pas d'explorer les architectures propres aux différentes fonctions.

De plus, dans les méthodes existantes, c'est encore au concepteur de faire une première analyse des spécifications du système afin de définir grossièrement une architecture qui constitue un point de départ à l'exploration de l'espace de conception. La méthode proposée dans ce mémoire vise à guider le concepteur dans ces choix architecturaux en lui permettant de réduire rapidement l'espace de conception pour ne garder qu'un nombre très réduit de solutions prometteuses. La méthode offre la possibilité en effet, de définir progressivement une architecture grâce à un *estimateur système* qui évalue les besoins en ressources de l'application à traiter. L'exploration de l'espace de conception repose sur l'exploration du parallélisme potentiel des fonctions composant l'application. L'exploration consiste à évaluer les différents compromis parallélisme (et donc ressources) vs. nombre de cycles alloués. L'exploration du parallélisme est opérée à tous les niveaux hiérarchiques de l'application à traiter, et est **dynamique** dans le sens ou les fonctions de l'application sont explorées individuellement puis associées entre elles, et non plus pré-caractérisées comme dans la plupart des méthodes existantes. La méthode ne vise pas à obtenir des estimations absolues, mais des estimations relatives afin de comparer différentes solutions et ainsi faciliter les choix du concepteur quant à l'architecture à adopter (adéquation application-architecture). En outre, la méthode proposée permet d'évaluer facilement les choix algorithmiques du concepteur puisque la spécification de l'application se fait en langage de haut niveau. Une fois l'architecture définie, celle-ci peut être affinée par des méthodes de codesign à un niveau de raffinement plus précis mais aussi plus complexe.

Organisation du document

Tout d'abord, le chapitre 1 présente un état de l'art résumant différents travaux réalisés dans le cadre du co-design et de l'exploration architecturale. Le flot de conception proposé est présenté dans le chapitre 2. Ce chapitre présente également les modèles de spécification et d'architectures, notamment ceux que nous avons retenus. Le chapitre 3 présente l'étape de caractérisation macroscopique ; celle-ci extrait des fonctions à implanter, des informations qui les caractérisent selon la nature de leurs comportements (traitement, contrôle ou mémoire). Ensuite, intervient la phase d'estimation intra-fonction permettant d'explorer le parallélisme des fonctions ; celle-ci est détaillée dans le chapitre 4. L'application de la méthode sur des exemples significatifs du domaine et les résultats qui en découlent sont présentés et analysés dans le chapitre 5. Enfin, le dernier chapitre conclut ce mémoire et présente différentes perspectives possibles par rapport au travail mené et aux résultats obtenus.

Chapitre 1

Etat de l'art

L'évolution permanente des techniques et des capacités d'intégration des circuits électroniques permet la réalisation de systèmes toujours plus complexes. Afin d'exploiter au mieux les possibilités offertes par les évolutions technologiques, de nouvelles méthodes de conception sont apparues. Ces méthodes, dites "conception conjointe logiciel-matériel" (co-design), sont désormais couramment utilisées pour le développement des nouveaux systèmes électroniques.

Dans ce chapitre, nous commençons par rappeler la problématique de la conception conjointe logiciel/matériel. Ensuite un flot générique de conception est introduit et les points caractéristiques que l'on retrouve dans les flots de conception existants sont exposés. Enfin, les principales méthodes et outils existants sont présentés. Le passage en revue de ces méthodes mettra en avant deux points importants. Tout d'abord la nécessité de concevoir les systèmes électroniques à un niveau d'abstraction de plus en plus élevé et ensuite le manque dans toutes ces méthodes d'une réelle exploration système se situant avant le choix d'une cible architecturale.

1.1 Problématique

Les systèmes électroniques modernes intègrent de plus en plus de fonctionnalités. Les méthodes de co-design permettent d'aider le concepteur à faire face à la complexité sans cesse grandissante des parties numériques de ces systèmes. Les parties numériques des systèmes électroniques peuvent être implantées sous deux formes : soit sous une forme logicielle (processeur à usage général, processeur de signal numérique (*DSP : Digital Signal Processor*), processeurs spécifiques (*ASIP : Application Specific Instruction set Processor, ASSP : Application Specific Standard Processor*)), soit sous une forme matérielle (composant dédié ou reprogrammable (ASIC : Application Specific Integrated Circuit, FPGA : Field Programmable Gate Array)). D'un côté, l'implantation logicielle autorise une plus grande flexibilité (programmation) et un coût généralement faible mais ne permet pas toujours d'obtenir les meilleures performances. De l'autre côté, l'implantation matérielle dédiée, grâce aux optimisations qu'elle autorise, doit permettre de s'assurer que les contraintes en temps/surface/consommation soient respectées. Malheureusement, l'implantation matérielle est souvent plus coûteuse que l'implantation logicielle. En effet, le cycle de conception de composants matériels dédiés est relativement long (comparé à l'écriture d'un programme pour processeur) et de plus, il faut compter avec des coûts de fonderie importants pour les séries relativement faibles. Une troisième est cependant en train d'émerger, il s'agit des architectures reconfigurables dynamiquement [7] [8] [9] [10]. Malgré l'évolution importante des composants programmables dynamiquement tels que les *FPGA*, il n'existe pas encore de composant miracle offrant les avantages des deux formes d'implantations. Quand bien même une plate-forme idéale existerait la question du choix de l'architecture à y implanter resterait entière.

Il apparaît donc clairement que le concepteur de systèmes électroniques numériques doive trouver les bons compromis entre les deux types d'implantations afin de converger vers une bonne adéquation architecture/système. Au delà du problème du partitionnement logiciel/matériel se pose la question de l'architecture programmable ou configurable la mieux adaptée en terme de granularité et de parallélisme. On le voit, l'espace de conception auquel le concepteur doit faire face est immense. Afin de réduire cet espace et d'automatiser les tâches du cycle de conception, de

nombreuses méthodes et outils ont vus le jour.

1.2 Diversité des flots de codesign

1.2.1 Introduction

L'évolution rapide de l'électronique numérique et plus particulièrement des systèmes enfouis a poussé de nombreux universitaires et industriels à s'intéresser au codesign. Ainsi de nombreux travaux ont été effectués dans ce domaine. La diversité des visions et de l'usage que les auteurs de ces travaux peuvent avoir du codesign ont conduit à une diversité toute aussi importante des méthodes et outils de codesign, et donc des flots de conception. De plus, la terminologie rencontrée dans ce domaine et dans les flots de conception prend plusieurs sens selon le niveau d'abstraction auquel on se situe. Aussi, nous proposons dans le paragraphe suivant un flot de conception générique afin d'y rattacher et de définir les concepts, notions et caractéristiques présents dans les flots existants.

1.2.2 Flot générique de conception

Le flot générique de conception que nous proposons est présenté sur la figure 1.1. Celui-ci est composé de trois grandes parties : a) flot d'estimation indépendant de toute cible architecturale (estimation système), b) flot dépendant d'une cible architecturale (estimation architecturale) et c) flot d'implantation (outils de CAO *backend*). La partie "a)" n'existe pas dans tous les flots de conception (ceux qui l'intègrent sont encore au stade de la recherche), c'est pourquoi il y a deux entrées principales représentées sur la figure 1.1.

Le flot d'estimation indépendant de toute cible architecturale a pour vocation de valider, de caractériser et d'explorer la spécification. Ces étapes permettent de fournir au concepteur des informations sur les caractéristiques de l'application (métriques) et de ces possibilités d'implantation. Ces dernières concernent par exemple le parallélisme potentiel, les possibilités de déroulage des boucles, l'organisation mémoire... Ce sont des éléments importants permettant de respecter les contraintes en temps, surface, consommation... Ces différentes étapes permettent de le guider

dans ses premières décisions relatives à l'implantation de l'application. À ce niveau, la cible architecturale n'est pas encore définie ou ne l'est que très partiellement. C'est pourquoi le modèle architectural ne peut être qu'abstrait (ou générique) : de nombreuses informations sont encore manquantes ou inexistantes puisqu'il s'agit justement de les trouver. Le modèle architecturale abstrait doit pouvoir couvrir une gamme importante de cible allant des ASIC au processeurs en passant par les DSP, les FPGA et les architectures reconfigurables. Ainsi à ce niveau l'objectif est de chercher à comparer les solutions entre elles et à discriminer celles qui semblent les plus prometteuses en fonction des critères de sélection du concepteur.

L'exploration de la spécification passe notamment par des modifications fonctionnelles mettant par exemple en œuvre des transformations algorithmiques, l'utilisation de structures de données (hiérarchie mémoire) plus ou moins complexes etc. Cette exploration peut également conduire à une pré-spécialisation du modèle architectural. En effet les paramètres précédemment cités peuvent faire ressortir qu'il peut paraître opportun d'implanter des éléments caractéristiques (tels que certains opérateurs, par exemple un bloc "papillon"). Une fois que cette étape a été suffisamment itérée, le concepteur dispose d'informations pour introduire un modèle architectural plus précis.

L'entrée dans l'étape "b) flot dépendant d'une cible architecturale (estimation architecturale)" peut se faire de deux manières : soit le concepteur utilise un modèle architectural qu'il a pu définir grâce à l'étape précédente, soit il utilise un modèle déjà existant (réutilisation d'un *design* existant par exemple). Dans cette étape de nouveaux éléments apparaissent, tel que l'allocation de ressources et le raffinement qui concerne le modèle architectural. Un exemple possible de raffinement du modèle architectural est présenté sur la figure 1.1 : partant d'un modèle uniquement fonctionnel on y ajoute ensuite les mécanismes de communications (synchronisation, taille des buffers...), puis le temps (modèle *"cycle accurate"*). Le modèle devient alors assez précis pour supporter une co-simulation complète. Le modèle architectural se précisant il est à présent possible de prendre des décisions concernant l'implantation définitive de l'application.

Enfin, lorsque le partitionnement a été complètement effectué et que toutes les

informations nécessaires à l'implantation sont disponibles il est possible de passer à l'étape "c)" du flot, c.a.d la synthèse "bas niveau" pour la partie matérielle et la compilation pour la partie logicielle. Cette étape est effectuée avec les outils *backend* de CAO et de compilation.

Trois points importants permettent de distinguer les différents flots existants :

1. Beaucoup de méthodes et outils existants ne possèdent pas la première partie "a)", en effet ceux-ci ont été conçus pour une cible particulière (souvent un microprocesseur et un accélérateur matériel). À notre connaissance très peu de méthodes commencent par une exploration sans cible architecturale prédéfinie. Lorsqu'elle existe c'est le modèle d'exécution (partie spécification) qui est utilisé comme support pour effectuer l'exploration de l'espace de conception.
2. Les possibilités de raffinement. Certaines méthodes ne permettent pas d'effectuer de raffinement car les modèles utilisés pour spécifier l'application et l'architecture ne sont pas assez génériques.
3. La priorité donnée aux opérations de traitements, d'accès mémoires et de contrôles. En effet, l'ordre dans lequel ces différents types d'opérations sont gérées peut conduire à des résultats plus ou moins optimaux (selon que l'application est orientée traitement, accès mémoire ou contrôle).

1.2.3 Éléments caractéristiques des flots de codesign et définitions

Cette partie à pour objectif de présenter les éléments que l'on retrouve dans les deux parties du flot générique. En fonction du niveau d'abstraction les caractéristiques de ces éléments différent. Ces différences sont également expliquées dans les paragraphes suivants.

Spécification

La spécification de l'application est un point important du flot de conception dont le choix peut avoir un fort impact sur les résultats [11]. Les modèles utilisés lors de

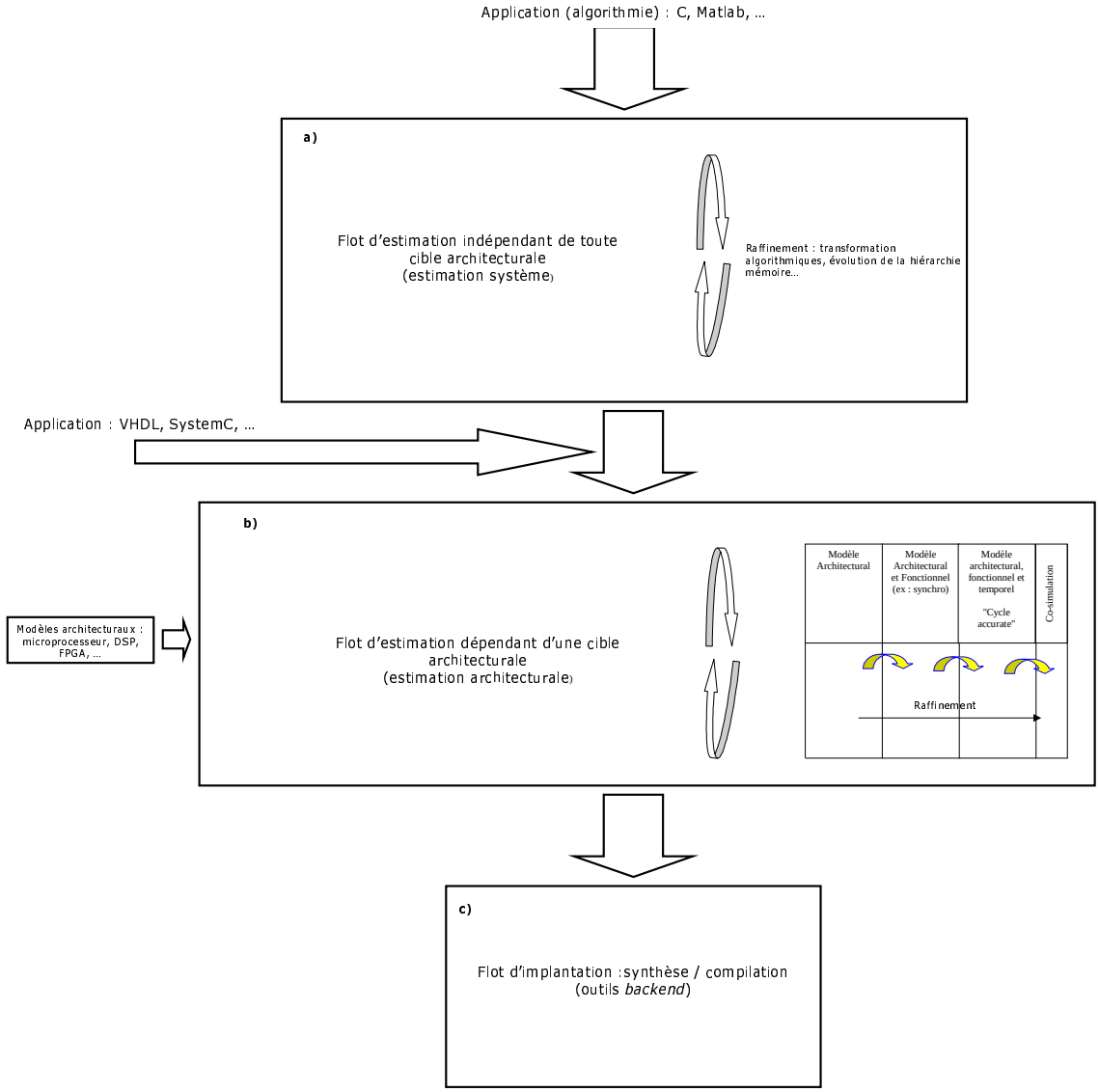

Fig. 1.1: *Flot générique de conception : a) flot indépendant de toute cible architecturale, b) flot dépendant d'une cible architecturale et c) flot d'implantation. Les éléments constitutifs de ces flots sont présentés dans les pages suivantes.*

la spécification doivent permettre d'exprimer les mécanismes d'évolution et/ou de réaction du système à spécifier (modèle d'exécution). Les modèles sont composés d'un ensemble d'objets et de règles animant ces objets. De plus ils doivent réduire le plus possible la sur-spécification et être en adéquation avec le type d'applications considérées. Ainsi les caractéristiques des modèles font qu'ils sont par exemple, plus ou moins adaptés pour décrire les structures (schémas blocs), le contrôle (réseau de Petri, FSM, CDFG), le traitement de type flots de données (DFG), etc. D'autre part les modèles représentent des vues conceptuelles plus ou moins riches des fonctionnalités du système à spécifier ou concevoir. Les vues conceptuelles permettent d'exprimer des notions telles que : la concurrence, la hiérarchie, les communications, la synchronisation, le temps, ...

De nombreux langages et modèles de représentations internes existent. On peut ainsi citer les langages synchrones tels que Esterel [12], Lustre [13] et Signal [14] qui permettent la description de systèmes réactifs à événements discrets mono ou multi-horloges (Signal). En ce qui concerne les modèles nous pouvons citer les CFSMs (*Codesign Finite State Machine*) qui sont des réseaux synchrones de machines d'états finis réactifs utilisés notamment dans l'environnement Polis [15]. La spécification des systèmes intégrant un système d'exploitation peut se faire soit par des graphes flot de données [16] [17] soit par des graphes de tâches [18] [19] [20]. Ces derniers correspondent à une granularité relativement élévée. Un niveau de granularité moyen (CDFG) ou faible (DFG) correspond généralement à une fonction ou suite d'instructions séquentielles. Enfin, on trouve un niveau de granularité faible dans les modèles orientés contrôle (réactifs) qui gèrent tous les événements du système.

Le langage 'C' est plutôt adapté à la description de séquences d'instructions. Ce langage étant très utilisé dans l'industrie des variantes de celui-ci ont été crées afin de pouvoir spécifier des applications complexes, notamment le parallélisme : C^x [21], HardwareC [22], SpecC [23] et surtout SystemC [24] choisi par un certain nombre d'industriels et universitaires (Synopsys, IMEC, ...).

La profusion des langages et les domaines auxquels ils se limitent ont conduit à la création d'environnements multi-langages [25], [26] ou encore de langages plus généraux tel que Rosetta [27].

Simulation / validation

La simulation se retrouve à plusieurs niveaux d'abstraction dans le flot de conception. Le niveau de détails est le plus important au niveau d'abstraction le plus bas et il diminue au fur et à mesure que l'on remonte vers le niveau système. Bien entendu, plus le niveau de détails est élevé plus les temps de simulations sont longs. Au niveau système la simulation est de type fonctionnelle, elle permet de vérifier que le système est fonctionnellement correct sans se préoccuper des détails d'implantations. On trouve à ce niveau des environnements tels que Polis [15], puis Vcc [28] ainsi que CoWare [29]. La vérification peut aussi se faire grâce à des approches du type preuve formelle [30], néanmoins les temps de calcul nécessaires sont prohibitifs pour des applications complexes.

Afin d'augmenter la précision de la simulation il est nécessaire de descendre au niveau d'abstraction inférieur, c.a.d aux niveaux instructions et RTL comportemental pour le logiciel et le matériel respectivement. À ce niveau l'exécution logicielle est approximée par un modèle du processeur de type "instruction" (c.a.d décrivant le processeur par son jeu d'instructions), associé à un simulateur ISS (*Instruction Set Simulator*). En ce qui concerne la partie matérielle un simulateur approprié est utilisé. À ce niveau nous pouvons citer les outils Seamless [31], Eagle [31], Coware [29]...

Pour encore améliorer la précision il est possible de passer au niveau cycle (*cycle accurate*). Ici la partie matérielle est décrite au niveau RTL et la partie logicielle en langage assembleur. À ce niveau on retrouve Seamless [31], Coware [29], ...

Enfin le niveau portes (ou logique) correspond à la réalisation du système. La partie matérielle est issue de la synthèse RTL ; la partie logicielle peut quant à elle correspondre au niveau macro-instructions voir même au niveau portes du processeur. À ce niveau les temps de simulations sont très longs (voir prohibitifs) et ne peuvent donc être utilisés que pour les vérifications finales.

Type d'architecture

La modélisation des architectures permet d'évaluer les performances et les coûts d'implantation de la spécification et ainsi de guider les choix de partitionnement.

Les premières méthodes de codesign considéraient des modèles d'architecture composées d'un processeur et d'un accélérateur matériel dédié (ASIC) [21], [32]. Les méthodes plus récentes permettent de cibler des architectures hétérogènes composées de plusieurs types de processeurs et d'accélérateurs ainsi que des hiérarchies de mémoires [17] [33] [34] [35]. Enfin, il existe des méthodes permettant de gérer les architectures reconfigurables (souvent à base de FPGA) [36], [37], [38], [39], [40] ainsi que les approches où l'architecture est générique, c.a.d décrite à l'aide d'un modèle. On peut citer par exemple l'outil Calife [41] qui utilise le langage de description de processeurs Armor [42].

Synthèse pré-exploratoire

La synthèse pré-exploratoire a pour objectifs de fournir des informations permettant d'évaluer les performances et les coûts des composants présents en librairie et à partir desquels l'architecture sera conçue. La majorité des méthodes de co-design font appel à des librairies statiques dans lesquelles sont stockées, pour chaque "fonction", les performances et coûts d'une ou plusieurs implémentations logicielles et matérielles. Ceci demande un important effort de synthèse pour alimenter ces bibliothèques. La synthèse logicielle consiste en la compilation de la fonction pour le processeur considéré et en l'obtention des chiffres de performances (temps d'exécution) et éventuellement de consommation. La synthèse matérielle quant à elle consiste en la synthèse soit architecturale soit logique des fonctions et en l'obtention des chiffres de performances, de taille et de consommation.

Exploration

L'exploration de l'espace de conception peut s'effectuer à plusieurs niveaux d'abstraction. On parle alors d'exploration au niveau système, architectural voir logique. Un point important est l'exploration du parallélisme de l'application. La recherche du parallélisme peut se faire soit au niveau HCDFG, CDFG (boucle) [43] ou DFG.

Partitionnement

L'exploration de l'espace de conception passe généralement par une étape de partitionnement. Que celle-ci soit manuelle ou automatique, son but est de répartir les "fonctions" de l'application sur les parties logicielles et matérielles de l'architecture cible. Ce processus est ré-itéré jusqu'à ce qu'une solution ou un ensemble de solutions satisfaisantes ait été trouvé. Le problème du partitionnement est très complexe (NP-Complet) et de nombreuses approches ont été élaborées. Dans [21] un algorithme de recuit simulé est utilisé. Dans [32] le partitionnement utilise l'algorithme PACE [44] basé sur une méthode de programmation dynamique. La programmation ILP (*Integer Linear Programming*) a également été utilisée comme dans [45]. On trouve également des approches basées sur les algorithmes génétiques [46]. Enfin, de nombreuses méthodes font appel à des algorithmes d'ordonnancement [16] [17] [47] [48] combinées avec des étapes de sélection de composants.

Stockage des données

Plusieurs techniques ont été imaginées pour optimiser la gestion des accès mémoires et le stockage des données.

La première de ces techniques consiste à réduire la bande passante requise en lissant les accès mémoire au niveau des données scalaires [49], au niveau tableaux multidimensionnels pour les applications dominées par les transferts de données [50, 51] ou encore au niveau tâche [52]. Une autre voie est celle de la répartition horizontale des données dans les banques à un niveau de hiérarchie donné ou dans les différents niveaux hiérarchiques afin d'optimiser consommation, vitesse et surface [49, 53, 54]. Les travaux portant sur la réutilisation de l'espace mémoire (*In-place*) [55, 56, 57] sont liés aux techniques de répartition des données citées plus haut. Lors de la conception d'un système sur silicium, le but est de définir la taille minimale de la hiérarchie mémoire en fonction de la localité des données [57]. Dans le cas d'architectures cibles prédéfinies le but est d'optimiser le placement des données afin de minimiser l'exploitation du cache [58]. Le contrôleur de mémoire Impulse [59] permet d'optimiser l'utilisation du cache et du bus en réorganisant l'accès aux données irrégulièrement réparties en mémoire. Ce type de méthode est à rappro-

cher des techniques de réorganisation des données en mémoire toujours dans le but d'optimiser l'exploitation du cache [60]. Enfin, des techniques de parallélisation sont développées pour la conception des compilateurs orientés mémoire. L'objectif est l'optimisation de la bande passante pour une architecture mémoire donnée (*memory aware ILP compilers* au niveau instruction [61] et au niveau tâche et instruction [62]), ou la réduction de la taille mémoire et du nombre de transferts requis (d'où un impact également sur la consommation).

Synthèse post-exploratoire

Une fois qu'un partitionnement satisfaisant a été trouvé et que l'architecture a été définie l'étape de synthèse post-exploratoire permet de générer le système. Il y a ici deux étapes, d'une part la synthèse matérielle dont le but est de générer soit une *netlist* pour la création d'un ASIC soit le code à télécharger dans un composant programmable de type FPGA, et d'autre part la synthèse logicielle (compilation) dont le but est de générer le code qui sera exécuté par un ou plusieurs microprocesseurs du système.

Guidage

Les étapes du processus d'exploration et plus particulièrement le partitionnement, peuvent être guidées par des métriques caractérisant l'application. Ces métriques permettent de guider le concepteur et les outils quant aux choix de l'architecture et du partitionnement. Dans [63] des métriques sont, entre autres, définies pour retrouver les motifs récurrents d'un algorithme. Dans [64] des métriques de rapprochement (*closeness metrics*) pour guider le partitionnement sont proposées. Elles apportent des renseignements sur le partage de ressources, de données et de canaux de communications. Dans [65] un estimateur probabiliste fait appel à des métriques permettant de mesurer les liens existants entre opérateurs (les connexions se faisant via des registres ou des mémoires). Enfin, dans [66] des métriques relativement fines sont définies pour caractériser l'affinité entre les fonctions d'une application et trois types de cibles architecturales : processeur à usage général (GPP), processeur de signal numérique (DSP) et ASIC. Les métriques, qui résultent de l'analyse

du code 'C' de l'application, permettent de repérer les séquences d'instructions qui peuvent être soit orientées DSP (buffer circulaire, opérations de type MAC,...), orientées ASIC (instructions de niveau bit,...) ou orientées GPP (structure de test, ratio d'instructions d'entrées/sorties,...). Ces métriques servent ensuite à guider un outil de partitionnement logiciel/matériel.

Espace de solutions

Le résultat de l'exploration de l'espace de conception peut prendre la forme, soit d'une solution unique, soit d'un ensemble de solutions. Les méthodes qui fournissent à l'utilisateur une solution unique à partir d'un sous-ensemble restreint de possibilités, visent le plus souvent à trouver la solution optimale au problème alors que les méthodes qui fournissent un ensemble de solutions visent en général à trouver des solutions qui respectent l'ensemble des contraintes sans forcément être optimales. En effet, à un niveau élevé d'abstraction la précision n'est pas suffisante pour mesurer avec certitude qu'une solution est optimale (car trop de détails d'implantation sont inconnus). Il est ainsi plus judicieux de conserver un ensemble de solutions "prometteuses" qui seront ensuite estimées à un niveau inférieur, conduisant à un ensemble de solutions plus réduit. La ré-itération de ce principe permet ainsi de converger vers une solution unique.

1.3 Quelques outils de co-design de type partitionnement

Apparue au début des années 90 la synthèse de haut niveau [67] (encore appelée synthèse comportementale ou *High Level Synthesis*) a permis de travailler à un niveau d'abstraction supérieur à celui de la synthèse logique et ainsi de réduire considérablement le temps nécessaire à l'exploration de l'espace de conception. En effet, au lieu de spécifier l'application au niveau RTL (niveau transfert registre), il est devenu possible de la spécifier avec un langage de haut niveau ('C', VHDL comportemental, Silage, etc...) et ainsi de s'affranchir des détails d'implantation durant la phase d'exploration de l'espace de conception. L'apparition de la synthèse de

haut niveau s'est ensuite accompagnée du développement des méthodes de codesign, celles-ci intégrant la synthèse de haut niveau pour les parties matérielles. Les outils et méthodes présentés dans cette section ont pour élément commun une phase de partitionnement sur laquelle repose l'exploration de l'espace de conception.

1.3.1 COSYMA

COSYMA [21] est un environnement pour l'exploration du processus de co-synthèse. L'architecture cible visée consiste en un processeur RISC (un modèle d'architecture SPARC est fourni), une mémoire programme et donnée et un coprocesseur dédié généré automatiquement. Le flot inclut également l'utilisation d'un outil de synthèse RTL (Synopsys DC).

Les points d'entrées sont les suivants :

- L'application est décomposée en processus communicant entre eux sous contrainte de temps. Ils sont décrits en langage 'C^x' (une extension du langage 'C' permettant de décrire le parallélisme inter processus).
- Un fichier de contraintes et de directives spécifiées par l'utilisateur.

Le flot se compose des étapes suivantes :

1. La spécification est traduite dans un graphe de syntaxe dérivée de SUIF [68]. Ce graphe est ensuite transformé en un ESG (*Extended Syntax Graph*) tout en modifiant dynamiquement la granularité du graphe [69].

2. Une simulation et un *profiling* ou analyse symbolique des processus sont effectués avec un modèle de niveau RTL du processeur. Cette étape permet d'obtenir les informations temporelles d'exécution pour chaque processeur cible.

3. Dans le cas d'applications mono-processus le flot passe directement à l'étape de partitionnement (3). Dans le cas d'applications multi-processus, une première option passe par une phase d'ordonnancement permettant de sérialiser les processus. Une seconde option combine le partitionnement et l'ordonnancement.

4. L'étape de partitionnement a pour entrées le graphe ESG avec les informations de profiling et le fichier de contraintes et de directives. Les directives

incluent le nombre et le type d'unités fonctionnelles à mettre en œuvre pour le coprocesseur dédié. Ces informations sont fournies par le concepteur. Le partitionnement part d'une solution "tout logiciel" et extrait des modules qui sont implantés en matériel jusqu'à ce que les contraintes de temps soient respectées. Les objectifs par ordre décroissant du partitionneur sont de : 1) respecter les contraintes de temps, 2) minimiser le coût matériel et 3) minimiser les temps de calcul. L'algorithme pour le partitionnement est du type recuit simulé (*simulate annealing*) et se guide à l'aide de métriques [70]. Les communications sont analysées durant le partitionnement et insérées lorsque le graphe ESG est retraduit en langage 'C' pour la synthèse logicielle et dans un langage de description de matériel (xHDL) pour la synthèse matérielle.

5. Pour la partie matérielle la *netlist* finale est générée par l'outil Synopsis Design Compiler. Pour la partie logicielle un compilateur C standard est utilisé.
6. Enfin une étape d'analyse à l'exécution (*run time analysis*) permet de valider les résultats d'un point de vue temporel. Cette étape consiste en une analyse formelle qui utilise une simulation logicielle et les résultats de l'ordonnancement de la partie matérielle.

L'exploration de l'espace de conception consiste à réitérer le processus pour plusieurs microprocesseurs puis à comparer les résultats obtenus pour ceux-ci. Cette méthode est donc très complète "verticalement" mais limitée d'une part par son modèle d'architecture et d'autre part par la librairie statique utilisée pour les temps d'exécution sur les processeurs (un seul temps d'exécution par processeur).

1.3.2 Lycos

Lycos [32], est un environnement de codesign qui permet l'exploration de l'espace de conception de systèmes composés d'un microprocesseur et d'un accélérateur matériel. L'environnement se situe avant le niveau RTL. Les points d'entrées de l'environnement sont les suivants :

- L'application qui est spécifiée en langage 'c' ou VHDL.
- Un jeu de données test représentatif pour l'application (en vue de l'étape de *profiling).*

– Une spécification des composants matériels (avec ACE, *Architectural Construction Environment* [71]).

Le flot se compose des étapes suivantes :

1. L'application est traduite dans une représentation interne nommée "Quenya". Celle-ci consiste en des CDFGs hiérarchiques.
2. Recherche des blocs de base pour l'ordonnancement (*BSBs*), ce sont ces blocs qui pourront être déplacés du logiciel vers le matériel.
3. *Profiling* permettant de connaître le nombre d'exécutions d'un BSB et le nombre d'accès aux variables.
4. Création de librairies statiques indiquant pour chaque BSB ses coûts pour une implantation matérielle (temps, surface) et logicielle (temps). Le temps d'exécution pour une implantation matérielle est obtenu rapidement par un ordonnancement par liste, pour une implantation logicielle il est obtenu par *mapping* des nœuds du CDFG vers un jeu d'instruction générique. De plus le coût des communications est pris en compte comme décrit dans [72] et [73] avec par exemple la modélisation des protocoles pour les bus PCI et USB.
5. L'étape suivante est le partitionnement matériel/logiciel. Le cœur du partitionneur est l'algorithme PACE [44], développé à cet effet. Celui-ci permet de gérer les communications et de favoriser la réutilisation des ressources matérielles. Le principe du partitionneur est d'évaluer les gains (ou pertes) engendrées par la migration de séquences de BSBs du logiciel vers le matériel. L'algorithme PACE détecte que les BSBs adjacents peuvent se partager des ressources matérielles (car ils sont mutuellement exclusifs) et peuvent communiquer directement ensemble sans passer par la partie logicielle.

L'exploration de l'espace de conception est obtenue par répétition de la phase de partitionnement pour différentes architectures cibles. Il est alors possible de comparer ces solutions en utilisant des critères qui ne sont pas pris en compte dans l'étape de partitionnement tels que la consommation, le prix, etc.

Cette méthode qui offre plus de possibilités que Cosyma quant à l'architecture cible possède la même limitation en ce qui concerne les librairies d'estimation qui

sont aussi statiques. De plus, le choix des composants de l'architecture initiale reste à la charge du concepteur.

1.3.3 SpecSyn

SpecSyn [33] [74] est un environnement de codesign qui se situe avant la synthèse logicielle/matérielle. Le cœur de la méthodologie est le paradigme "SER" (*Specify-Explore-Refine*).

Les points d'entrées sont les suivants :

- Une spécification fonctionnelle de l'application sous la forme de PSM (*Program-State Machine*) permettant de décrire des machines d'états hiérarchiques/concurrentes, un programme séquentiel ou des processus séquentiels communicants.
- Une liste de composants (microprocesseurs, ASIP, ASICS, bus, etc.) caractérisés dans une librairie.

Le flot est le suivant :

1. La spécification est convertie dans une représentation interne SLIF (*Specification-Level Intermediate Format*). Dans cette représentation les nœuds des graphes représentent soit des comportements (processus ou procédure) soit des variables. Les arcs représentent des canaux de communications (appel de procédure, lecture/écriture d'une variable, passage de messages).
2. L'exploration commence par une phase d'allocation. Celle-ci consiste à ajouter des composants au système. Le nombre de composants n'est pas limité, mais cette phase reste être entièrement manuelle.
3. La deuxième étape de l'exploration est le partitionnement. Celui-ci consiste à répartir les variables, comportements et canaux de communications entre les composants (logiciels ou matériels) de l'architecture. Le partitionneur a pour entrées une structure de données, une procédure de mise à jour de cette structure, des fonctions de coûts. Ces dernières sont fonctions d'informations issues d'une phase de pré-estimation dans laquelle chaque comportement, variable et canal de communications est annoté avec des informations telles que la taille de code pour un certain processeur, le nombre d'accès à un canal de communication, etc... Ensuite, de nombreuses heuristiques intégrées au cœur du

partitionneur (appelé GPP) permettent de procéder au partitionnement automatique. Il est néanmoins possible de procéder à un partitionnement manuel sous le contrôle du concepteur.

4. Les différents alternatives produites grâce à l'étape de partitionnement sont comparées entre elles. Pour ce faire, une nouvelle phase d'estimation est nécessaire. Celle-ci nommée "estimation en ligne" reprend les informations de la phase de pré-estimation et les combine en expressions complexes pour obtenir des métriques pour une allocation et un partitionnement particulier. Ces métriques comprennent la performance (temps d'exécution), la surface (cas du matériel) et la taille du code (cas du logiciel).
5. L'étape de raffinement (*refinement*) génère une nouvelle spécification des composants du système une fois qu'une allocation et qu'un partitionnement à priori satisfaisants ont été trouvés. L'étape de raffinement inclut la génération d'interfaces (taille de bus, protocole), l'assignation des variables aux mémoires, le remplacement des accès aux variables par des références aux emplacements mémoire adéquats et la génération de ressources d'arbitrage si nécessaire.
6. Enfin, la description du système partitionné sert de point d'entrée aux étapes de synthèse logicielle et matérielle.

Ici encore les librairies de coûts issues de la pré-estimation sont statiques, la granularité du parallélisme est figée et la définition de l'architecture initiale reste à la charge du concepteur.

1.3.4 POLIS

POLIS [15] est un environnement de codesign qui part de la spécification du système et descend jusqu'à la synthèse logique et la synthèse logicielle. Le processus passe par les étapes suivantes :

1. La spécification en langage haut niveau (Esterel, FSM, sous-ensemble de VHDL...) est traduite en CFSMs (*Concurrent Finite State Machines*).
2. La spécification est vérifiée par étape de vérification formelle qui fait appel à des outils externes à POLIS.

3. L'étape de partitionnement inclut le partitionnement logiciel/matériel mais aussi la sélection de l'architecture cible et de l'ordonnanceur. Les choix effectués lors de cette étape restent à la charge du concepteur. Pour l'aider dans ces choix un mécanisme de retour d'expérience lui permet d'obtenir des informations issues des étapes de vérification formelle et de co-simulation.
4. Les sous-ensembles de CFSMs destinés à une implantation matérielle sont synthétisés au niveau logique en utilisant des outils extérieurs.
5. La synthèse logicielle inclut la génération de code 'C' pour les CFSMs destinées à une implantation logicielle ainsi que la génération d'un système d'exploitation spécifique à l'application.
6. Les interfaces entre les parties logicielles et matérielles sont automatiquement générées. Les communications peuvent se faire à travers des ports d'entrées/sorties disponibles sur le micro-contrôleur ou à travers des entrées/sorties de mémoires.

1.3.5 PICO

PICO (Program In, Chip Out) [75] est un environnement de co-design qui permet de générer des systèmes composés d'un processeur VLIW ou EPIC dédié à l'application visée, d'une hiérarchie de mémoires caches et d'un accélérateur non-programmable (de type systolique). L'accélérateur a pour but d'accélérer les parties critiques (du point de vue traitement) de l'application. Le module PICO-VLIW génère le processeur et PICO-NPA [5] génère l'accélérateur non-programmable.

Le flot de conception est le suivant :

1. L'application est analysée et les parties critiques sont identifiées manuellement
2. la fonctionnalité *spacewalker* du module PICO-NPA recherche les points Pareto dans l'espace de conception relatifs aux parties critiques de l'application. Pour chaque partie critique (typiquement des cœurs de boucles), le compilateur NPA génère une architecture appropriée après avoir effectué des transformations sur la partie considérée afin d'en exposer le parallélisme. L'architecture est exprimée au niveau RTL et ses performances/coûts estimés lui sont associés.

Le code des parties à implémenter sur l'accélérateur remplace le code d'origine de la spécification.

3. La fonctionnalité *spacewalker* du module PICO-VLIW génère l'architecture et la microarchitecture du processeur VLIW (niveau RTL). Un ensemble de modèle de processeurs est inclu dans PICO, et des heuristiques sont utilisées pour réduire l'espace de conception. De plus, une description compatible avec le compilateur re-ciblable ELCOR [76] est produite. Celui-ci est reciblé pour le nouveau processeur et le code modifié de l'application est compilé.

4. La fonctionnalité *spacewalker* du module de mémoire cache spécifie et évalue des configurations de hiérarchies de caches.

5. La fonctionnalité *spacewalker* de niveau système de PICO compose à partir du processeur VLIW, de l'accélérateur et de la hiérarchie de cache un système complet. Il détermine également quelles parties critiques sont à implémenter sur l'accélérateur plutôt que sur le processeur. La recherche des points Pareto se fait à l'aide de métriques telles que la surface et le temps d'exécution.

Cette méthode nous apparaît très intéressante, néanmoins l'architecture cible est restreinte au modèle processeur VLIW + accélérateur et la recherche des parties critiques qui sont passées à l'outil reste entièrement manuelle. Ces travaux sont donc plus un complément qu'une approche concurrente.

1.4 Vers des outils de co-design niveau système

Afin de toujours raccourcir les temps de développement et favoriser la réutilisation de designs pré-existants l'utilisation de méthodes faisant appel à des composants de plus gros grains sont en train de se développer. À ce niveau il est ainsi possible d'utiliser des composants tels que des microprocesseurs, DSP, accélérateurs matériels pré-définis. Ces méthodes se rapprochent donc de la partie "a)" du flot générique proposé en 1.2.2, sans pour autant y être complètement intégrées.

1.4.1 CODEF

Codef [34] [77], est un outil pour l'exploration de l'espace de conception. L'exploration est automatique et/ou interactive et repose sur un algorithme de partitionnement récursif suivi d'une étape de synthèse des communications.

Les entrées de l'outil sont :

- Un modèle de l'application sous la forme d'un BDF (*Boolean-controlled Data Flow*).
- Une librairie modélisant les composants (logiciels et matériels) de l'architecture cible. Celle-ci comprend le nombre et le type de processeurs, la taille et les temps d'accès aux mémoires, la contribution de coprocesseurs ou d'accélérateurs sur les performances et le coût, les protocoles de communications entre les composants, ...
- Un ensemble de contraintes ainsi qu'une librairie de modèles d'implantations de tâches ou processus sur les composants de l'architecture.
- Eventuellement un système prédéfini, dans ce cas l'outil sert à affiner et paramètrer ce système.

Le principe du flot de l'outil est le suivant :

1. Une étape de partitionnement/ordonnancement permet de construire l'architecture à partir des composants définis dans la librairie. Le partitionnement commence par les parties les plus critiques du graphe de la spécification. L'étape de partitionnement/ordonnancement vise à minimiser soit les temps d'exécution, soit la surface ou une combinaison de ces deux critères. L'exploration de l'espace de conception se fait en fonction de seuils variables (réglés automatiquement ou manuellement). Ces seuils s'appliquent à des paramètres concernant la ré-utilisation et l'instantiation.
2. La synthèse des communications intervient après l'étape de partitionnement / ordonnancement. Celle-ci vise à minimiser les inter-connexions en utilisant au maximum des transferts synchrones (grâce à des mécanismes de rendez-vous ou d'interruptions). La synthèse des communications étant réalisée après le partitionnement, il est possible de synchroniser les transferts asynchrones afin de ne pas violer les contraintes de temps et de minimiser la surface totale due

aux communications.

3. L'outil acceptant un système préconçu en entrée, il est ainsi possible de procéder au raffinement de celui-ci par itérations successives.

1.4.2 TCM / Matador

L'approche TCM (*Task Concurrency Modeling*, aussi connue sous le nom Matador) [78], [79] de l'IMEC vise à minimiser dynamiquement la consommation d'un système en fonction de la charge de travail effective de celui-ci. L'idée de base est de modéliser l'application visée sous la forme d'un ensemble de tâches concurrentes. Les applications dynamiques sont caractérisées par le fait que le nombre de tâches et leurs dépendances changent au cours du temps. Lors de la conception du système des *mappings* d'implantations sont réalisés pour chaque tâche. Les plus intéressants sont sélectionnés grâce à l'utilisation d'une courbe de type Pareto. Celle-ci décrit le compromis optimal entre le temps d'exécution et la consommation en énergie. Ces courbes sont ensuite utilisées par un ordonnanceur (de type *run-time*) de faible complexité. Cet ordonnanceur gère les différentes tâches de l'application dynamique de façon à minimiser la consommation totale tout en respectant la contrainte de temps.

Flot de TCM / Matador

Le flot est composé d'étapes séquentielles et commence par la description du système pour arrivé à l'ordonnanceur de type *run-time*.

1. La première étape est de trouver un modèle de type "boite noire"de l'application. Ce modèle doit contenir suffisamment d'informations afin que l'ordonnanceur puisse prendre des décisions optimales ; dans le même temps et pour que la complexité reste à un niveau acceptable, le modèle ne contient pas tout les détails d'implantations.

2. Les étapes suivantes concernent l'exploration au niveau tâche des transferts et du stockage des données. Celles-ci vise à réduire les ressources nécessaires aux communications entre tâches et au stockage des données. À ce niveau le code

optimisé obtenu et les informations concernant les transferts et le stockage des données sont intégrés au modèle "boite noire".

3. Ensuite, la concurrence globale entre tâches est améliorée et optimisée (élimination de la redondance). Il est important que la concurrence utile soit clairement exhibée car un niveau plus élevé implique plus d'opportunités pour l'approche TCM/Matador de trouver des compromis pour chaque tâche entre la consommation globale et le temps d'exécution.

4. L'étape suivante à pour rôle de trouver, pour chaque tâche, une série de points optimaux (c.a.d une courbe de type Pareto) qui représente une combinaison optimale entre un coût (consommation en énergie) et le temps d'exécution. Chaque point correspond à un ordonnancement spécifique des sous-tâches d'une tâche. Ceci est obtenu en changeant par exemple la tension d'alimentation du processeur ou bien encore en utilisant un bus mémoire plus rapide mais plus gourmand. Une boite "grise" est utilisée à ce niveau, elle fait apparaître plus détails que la "boite noire" sans pour autant tout montrer ("boite blanche").

5. Enfin, le code pour l'ordonnanceur est compilé. Grâce à une méthode heuristique les courbes Pareto sont combinées de manière à minimiser la consommation totale tout en respectant les contraintes de temps.

Cette approche qui utilise aussi des courbes de compromis optimales (courbes Pareto) vise surtout à minimiser la consommation. L'exploration consiste essentiellement à faire varier la tension d'alimentation en fonction de la charge du système.

1.4.3 ARTEMIS/SPADE

Artemis [35] est un environnement de modélisation et de simulation permettant l'exploration de l'espace de conception des systèmes enfouis. Il repose sur la méthodologie SPADE [80]. Grâce à une modélisation de l'architecture cible à différents niveaux d'abstraction il est possible d'évaluer un grand nombre de solutions en utilisant une modélisation "gros grains" puis d'affiner quelques solutions prometteuses en utilisant des modèles d'architecture plus précis. Au niveau le plus haut il

est ainsi possible de construire une architecture cible à partir de microprocesseurs, co-processeurs, ...

Les entrées de SPADE sont les suivantes :

- La spécification du système se fait sous la forme de réseaux de processus de Kahn (*KPN Kahn Process Network*) qui permettent d'exprimer le parallélisme et de rendre les communications explicites. Dans ce modèle les processus communiquent au travers de mécanismes du type FIFO illimités en taille. Chaque processus s'exécute séquentiellement.
- L'architecture cible est construite à partir de blocs de base génériques. Ceux-ci modélisent les différents types de ressources de l'architecture tels que les ressources de traitement, de communications et de mémoires. Il suffit d'instancier et de connecter ces blocs pour modéliser l'architecture. Pour chaque ressource une liste indique les instructions symboliques qu'elle peut exécuter ainsi que leurs latences. Ces informations sont issues, soit de modèles plus détaillés des ressources, soit d'outils d'estimations, soit estimées par le concepteur.

La méthodologie de SPADE est la suivante :

1. Le concepteur étudie l'application, fait des calculs préliminaires et propose une architecture initiale. Pour une étude à un haut niveau d'abstraction celle-ci peut être composée de microprocesseurs, DSPs etc. [81].
2. Le modèle de la spécification est analysé (avec l'outil YAPI) afin d'en déterminer la charge de travail en traitements et en communications. Cela correspond à une sorte de *profiling* qui indique pour chaque processus le nombre d'invocations de ses sous-fonctions et pour les communications le nombre de jetons pour chaque donnée transférée entre deux processus.
3. Un partitionnement manuel (appelé *mapping* par les auteurs) des charges de travail sur les ressources est effectué. Dans le cas ou il apparaît qu'un processus doit être assigné à plus d'une ressource le concepteur doit ré-écrire la spécification afin de casser le processus en sous-processus.
4. Le modèle de l'application et celui de l'architecture sont simultanément simulés. La simulation de l'application génère des "traces" qui pilotent la simulation de l'architecture. La simulation de l'architecture utilise l'environnement Pa-

mela qui permet d'exécuter les processus. L'architecture est quant à elle simulée avec l'outil TSS de Philips [82]. Les résultats de l'estimation sont récupérés pour chaque ressource. Ceux-ci représentent les temps d'occupation des ressources de traitement passés à faire des calculs et à gérer les entrées/sorties, les temps d'attente de données et les temps d'inactivité. Pour les ressources de communications les informations sont les temps d'activité et le nombre de données transférées. Les différents temps collectés sont exprimés en nombre de cycles.

5. Enfin, les résultats de la simulation sont manuellement analysés par le concepteur qui peut identifier les ressources sous-utilisées, les goulets d'étranglement (contraintes de temps non respectées). Après analyse le concepteur modifie l'architecture et réitère le processus. Ainsi par raffinements il peut converger vers l'architecture la mieux adaptée à l'application.

1.4.4 Platune

Dans [83] une méthode intéressante pour la configuration de SOCs paramètrés est présentée. Celle-ci vise à rendre automatique l'exploration de l'espace de *configuration* pour une gamme restreinte d'architectures. Le modèle d'architecture présenté par les auteurs comprend un processeur MIPS associé à des caches données et instructions, une mémoire principale, un UART et un codec DCT, le tout relié par des bus. Les éléments de ce modèle sont en partie paramétrables (taille mémoire, associativité des caches, voltage/fréquence du processeur)

La méthodologie se compose de deux étapes principales. Tout d'abord les performances et la consommation de l'application à traiter sont obtenues pour le modèle d'architecture. Ensuite, à l'aide de ces informations, la recherche des points Pareto est effectuée. Pour ce faire, un graphe de dépendances entre les différents paramètres de l'architecture est établi. Ensuite, le graphe est mis sous la forme de *clusters* (sous-ensemble de dépendances entre paramètres). Il est ainsi possible de procéder soit à une recherche exhaustive des points pareto au sein des clusters (dont le nombre de nœuds est assez faible), soit à une recherche en utilisant une heuristique dans le cas des clusters complexes (cas assez rares d'après les auteurs). Une fois que les confi-

gurations optimales ont été trouvées dans les clusters, ceux-ci sont combinés deux à deux et les points Pareto dans la combinaison sont recherchés. La procédure est réitérée jusqu'à ce qu'il ne reste plus qu'un seul cluster dont les points Pareto représentent la configuration optimale (du point de vue performance/consommation) du système.

1.4.5 Coware

CoWare [29] est une suite d'outils commerciaux issue de travaux menés à l'IMEC [84]. CoWare permet de concevoir des SoCs et des plate-formes au niveau système à partir des langages C et SystemC. CoWare N2C Design System (DS) est composé des outils suivants : Advanced System Designer (ASD), System Designer (SD) et Hdl Import.

System Designer

System Designer est un environnement de simulation qui permet la simulation de SoCs complexes à de hauts niveaux d'abstraction. L'environnement de simulation permet de gérer les descriptions logicielles/matérielles en langages basés sur le C (tel que SystemC) ainsi qu'en HDL. Le cœur du simulateur, pour la partie logicielle, repose sur des *packages* de description de processeurs. Grâce au système "Coware aware Processor Support Package" il est possible d'intégrer le simulateur de jeu d'instructions (ISS) du processeur ainsi que la chaîne compilateur-éditeur et liens-debugger des processeurs composant le système. Le simulateur pour la partie matérielle gère la plupart des simulateurs HDL.

Hdl Import

Hdl Import est un outil permettant la conversion de descriptions HDL (VHDL ou Verilog) en descriptions basées sur le C (SystemC,...). Il est ainsi possible d'intégrer au niveau système des IPs disponibles en HDL.

Advanced System Designer

Advanced System Designer est un environnement permettant de créer et modifier rapidement des plate-formes fixes ou évolutives. Pour cela, il intègre des outils de modélisation, synthèse et évaluation. La modélisation consiste à assembler des IPs et à les interconnecter via des bus de type AMBA ou STBUS. Il semble que

de nombreuses IPs compatibles avec l'outil soient disponibles. Le module Interface Synthesis (seconde génération) permet la synthèse automatique des interconnexions. Le partitionnement logiciel/matériel est manuel. Un ensemble d'outils de vérification et de simulation (extension de System Designer, en particulier l'ajout de l'analyse de bus complexes) permet d'analyser les choix du concepteur et ainsi de procéder à l'exploration architecturale. Finalement l'outil permet la génération du code HDL du système pour attaquer des outils externes de synthèse logique.

1.4.6 Platform based design / metropolis

Dans [85] et [86] une définition de la "conception basée sur des plate-formes" (*Platform Based Design*) ainsi qu'une méthodologie générique de codesign sont présentées. Le but de la méthodologie est de favoriser la réutilisation de design préexistants afin de faire face aux coûts de développements sans cesse grandissants. Dans cette approche les différents niveaux d'abstraction sont appelés plate-formes. Deux plate-formes de niveaux d'abstraction successifs plus les outils permettant de passer de l'une à l'autre sont appelés pile de plate-forme (*platform stack*).

La méthodologie générique présentée par les auteurs commence à être implantée dans divers projets dont Metropolis [87] (lié au programme Gigascale [88]). Il nous semble cependant que l'ensemble reste encore très conceptuel à l'heure de la rédaction de ce document.

1.5 Bilan

Comme nous venons de le voir, de nombreuses méthodes de co-design existent. Les premières d'entre elles se situent à des niveaux d'abstraction relativement bas et sont souvent limitées à une architecture cible particulière. De plus, l'utilisation de librairies statiques ne permet pas une exploration approfondie du parallélisme potentiel. Les méthodes plus récentes se situent à des niveaux d'abstraction plus élevés et permettent de traiter tout type d'architecture cible durant l'exploration de l'espace de conception. Cependant, la définition de l'architecture cible initiale reste à la charge du concepteur, qui doit alors tirer profit de son expérience ou partir d'un

design pré-existant. Or, cette tâche qui s'avère cruciale dans le flot de conception est loin d'être triviale. Il y a donc un "gouffre" qui sépare la phase de spécification de l'application et les premières étapes d'exploration de l'espace de conception dans les méthodes présentées plus haut. Nous proposons donc une nouvelle étape dont le but est d'aider le concepteur à définir une architecture initiale. De plus la séparation entre logiciel et matériel commence à s'estomper, notamment au niveau système [89]. À cet effet, une nouvelle méthodologie reposant sur une exploration dynamique et hiérarchique du parallélisme a été développée. Opérant au niveau système, celle-ci est décorélée de toute hypothèse architecturale. Elle s'inscrit dans le contexte plus large des travaux de recherche sur le codesign menés au LESTER. Ce contexte et cette méthodologie sont présentés dans le chapitre suivant.

Chapitre 2

Flot de conception

Dans ce chapitre nous présentons tout d'abord le contexte dans lequel se situent les travaux que nous avons développé ; nous introduisons à cet effet les travaux situés soit en amont soit en aval de notre méthode. Ensuite, le flot de conception de la méthode développée est présenté. Les différentes étapes qui le composent sont définies. Ensuite, les modèles de spécifications de l'application et du modèle architectural de niveau système sont présentés.

2.1 Contexte de l'étude - approche codesign du LESTER - DesignTrotter

Toutes les méthodes présentées dans le chapitre précédent reposent sur le principe de partitionnement, qui consiste à décomposer la spécification dans le but de réaliser l'intégration du système sur plusieurs composants. Le partitionnement constitue alors le cœur du codesign et s'appuie sur une architecture pré-définie. Les fonctions qui composent le système sont réparties sur les différentes unités du modèle de l'architecture cible. Ensuite, grâce à des fonctions de coûts, des estimateurs ou bien encore une vérification après implantation, la qualité de ce partitionnement est évaluée. La qualité de l'intégration repose donc essentiellement sur celle du partitionnement et finalement, la définition de l'architecture ne tient pas compte directement des propriétés intrinsèques de l'application. L'approche originale développée au LESTER cherche au contraire à guider la construction de l'architecture en fonction de ces propriétés. L'architecture est définie itérativement en précisant progressivement les hypothèses architecturales à mesure que l'on progresse dans l'analyse du système.

À notre connaissance peu d'approches font intervenir les estimations avant le choix de l'architecture. Cependant, l'utilisation d'estimateurs peut se faire très tôt dans le cycle de conception, juste après la phase de spécification de l'application et peut éventuellement servir à modifier celle-ci en guidant les modifications à apporter sur la description algorithmique (plusieurs écritures ou algorithmes pour une même fonction). L'utilisation d'estimateurs à ce niveau permet donc de guider la définition de l'architecture en adéquation avec l'application. La définition de l'architecture passe par un raffinement progressif de l'analyse de la spécification au moyen d'estimations opérant à différents niveaux d'abstraction. Dans cette approche il n'y a plus de modèle d'architecture prédéfini, ni de notion de partitionnement tel qu'elle a été précédemment définie. Ainsi, grâce à l'utilisation d'estimateurs très tôt dans le flot de conception il est possible d'explorer des espaces de conception plus importants, en adéquation avec la liberté de création offerte par les possibilités technologiques actuelles. L'originalité de l'approche développée au LESTER tient dans la stratégie qui permet une construction incrémentale de l'architecture et une exploration de

l'espace de conception à partir d'un très haut niveau d'abstraction. Les différents travaux menés sur ce sujet au sein du LESTER ont été intégrés dans un flot de conception global nommé DesignTrotter. Afin de mieux cerner le contexte des travaux présentés dans cette thèse nous commençons par présenter une vue globale de DesignTrotter. Ensuite la partie "flot d'estimation" est introduite. Les différentes étapes du flot d'estimation sont ensuite détaillées dans les chapitres 3 et 4.

Design Trotter est un environnement de conception de systèmes électroniques enfouis (ou embarqués). Les domaines d'application couverts sont le multimédia, les télécommunications, ... Le flot de conception DesignTrotter est issu des différents travaux de recherche menés au sein du LESTER et s'inscrit dans le cadre de coopérations avec d'autres Universités et des industriels comme nous le verrons un peu plus loin dans cette partie. Ce flot est présenté dans la figure 2.1. L'environnement Design Trotter dispose de nombreuses fonctionnalités relatives au codesign et aux estimations. Dans toute cette partie nous allons voir quelles sont ces fonctionnalitées et comment les travaux présentés dans cette thèse s'y rattachent. L'originalité de notre approche se situe dans une méthode en 2 étapes : i) estimation du parallélisme et ii) projections avec des modèles matériels et logiciels suivies de deux types de partitionnement (approche temps réel et approche inter-fonctions). Les travaux de thèse sont intégrés dans la boite "Flot Estimation Design Trotter" (point 7 sur la figure 2.1). Ce point est détaillé sur la figure 2.2.

Les entrées de l'environnement Design Trotter se présentent sous la forme de fonctions décrites en langage de haut niveau (le C pour le moment). Ces fonctions peuvent être issues soit de la décomposition en une machine d'état de l'application (comme par exemple à partir du langage Esterel dans le cadre du projet Epicure), soit à partir d'un graphe de tâches (fournie par la méthode Radha Ratan dans le cadre de la thèse de A. Azzedine, doctorant au sein du projet). L'environnement Design Trotter fournit en sortie différents types d'estimations et d'informations visant à guider le concepteur de systèmes embarqués. Parmi celles-ci nous pouvons citer les estimations systèmes (objet de cette thèse), les estimations sur architectures reconfigurables (thèse L. Bossuet, projet Epicure), les estimations sur processeur (thèse A. Azzedine) etc. Ces différentes sorties sont détaillées dans le descriptif des

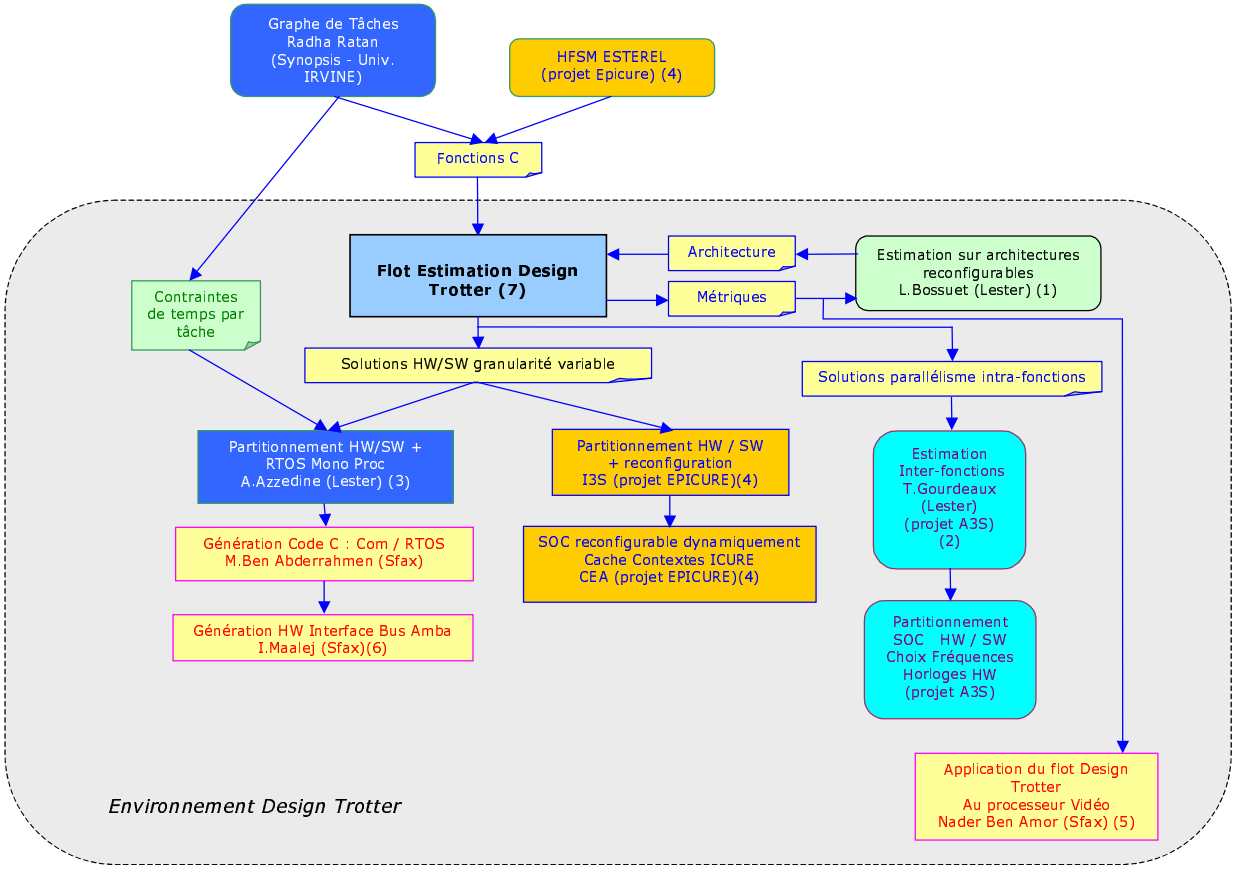

Fig. 2.1: *L'environnement "Design Trotter".*

modules de Design Trotter qui suit.

Projection sur architectures reconfigurables

Le flot d'exploration proposé dans la thèse de L. Bossuet (point 1 sur la figure 2.1) vise à évaluer très tôt dans le flot de conception différentes architectures afin de comparer leurs performances relatives et d'identifier les solutions à étudier plus finement. Cette étude se concentre sur les architectures reconfigurables n'intégrant pas de cœur de processeur. Afin de pouvoir étudier ces architectures une modélisation est proposée. Celle-ci est de type fonctionnelle, c'est à dire que exprimant ce que l'architecture est capable de réaliser et avec quelles performances, et non pas comment elle est physiquement réalisée. Cette modélisation s'articule autour de deux types d'éléments : les éléments fonctionnels représentant les ressources de l'architecture (blocs logiques, opérateurs, mémoires, entrées-sorties) et les éléments

hiérarchiques représentant les connexions entre les éléments fonctionnels. Il est à noter que dans cette modélisation les ressources de routage de l'architecture ne sont pas précisément modélisées, mais leurs impacts sont est pris en compte dans le processus d'estimation. L'approche fonctionnelle permet de décrire de nombreuses architectures avec des granularités différentes tant aux niveaux des ressources (de traitements ou de mémorisations) que des interconnexions, et avec une diversité importante de ressources possibles.

L'estimation relative (travaux de L. Bossuet) est composée de trois étapes dont la première, l'étape de projection permet de mettre en correspondance les ressources nécessaires à l'application avec les ressources de l'architecture (ressources de traitement et de mémorisation). Une fois la projection réalisée, les ressources allouées de l'architecture sont assignées aux ressources du graphe. Dans l'étape type de composition, les ressources supplémentaires nécessaires à l'ordonnancement de l'application (multiplexeur, registre, machine d'états) sont estimées et réservées sur l'architecture. Enfin, l'étape d'estimation permet de caractériser les performances de l'application spécifiée sur l'architecture décrite en termes de vitesse et de consommation. Ces estimations prennent en compte, d'une part les coûts statiques décrits dans le modèle de l'architecture (coûts de connexions à l'intérieur des éléments hiérarchiques et coûts d'utilisation des éléments fonctionnels) et d'autre part les coûts dynamiques qui représentent les caractéristiques du graphe (chemin critique et communication inter-opérateurs). À la suite des trois étapes de l'estimation relative, quatre fichiers seront édités : trois détaillent les résultats de chacune des étapes (projection, composition et estimation) et le dernier donne un résultat synthétique indiquant l'adéquation de l'architecture décrite avec l'application spécifiée en fonction des contraintes qu'elle doit tenir. Une fois le flot exécuté sur toutes les architectures décrites par l'utilisateur, celui-ci disposera d'un ensemble d'informations lui permettant de choisir l'architecture la plus en adéquation avec l'application qu'il développe. Il est à noter que l'utilisateur peut, pour le même panel d'architectures, réaliser le flot avec plusieurs applications. Il dégagera ainsi l'architecture la plus en adéquation avec les applications qu'il souhaite développer (ces applications pouvant êtres configurées dynamiquement sur l'architecture si celle-ci le permet). Fort des informations

ainsi obtenues, l'utilisateur pourra raffiner la description de l'architecture choisie, avec une modélisation par exemple du type physique (contenant entre autre une description précise du routage et des ressources), afin d'utiliser un outil de synthèse générique.

Le lecteur intéressé trouvera de plus amples détails dans [39] et [40].

Partitionnement logiciel/matériel

Les résultats obtenus durant l'étape d'estimation sont ensuite combinés afin de répartir les besoins en ressources et en bande passante pour l'application complète et afin de proposer un partitionnement logiciel/matériel (point 2 et 3 sur la figure 2.1). Pour cela deux voies sont explorées : i) une approche de type flot de données à granularité élevée appelée "estimation inter-fonction" (un nœud du graphe = une fonction) et ii) une approche de type temps réel consistant en un partitionnement en tâches (une tâche = un graphe de fonction). Pour chacune des tâches de l'application on obtient alors une courbe de compromis parallélisme / temps d'exécution. Ainsi, en fonction du temps alloué à la tâche, le parallélisme potentiel est plus ou moins exploité ce qui conduit à un coût d'implantation différent. Les points de la courbe d'estimation (appelée estimation dynamique) correspondent ainsi à différentes implantations possibles : logicielles, logicielles avec co-processeurs et totalement matérielles.

- **Partitionnement des applications à comportement statique (estimation inter-fonctions) :**

  Cette partie décrit les travaux de thèse de T. Gourdeaux (point 2 sur la figure 2.1). Le problème général d'optimisation traité par l'estimation inter-fonctions (les fonctions sont elles-même estimées grâce aux travaux présentés dans cette thèse), est d'assigner un nombre de cycles (c.a.d choisir une des solutions issuent de l'estimation système) et une fréquence à chaque fonction et de décider de la date de début de l'ordonnancement de la fonction afin de réduire au minimum les ressources. Les résultats que nous fournissons à l'étape d'estimation inter-fonctions ont pour unité de temps un nombre de cycles abstraits qui doivent maintenant être convertis en temps. En considérant une architecture

hétérogène, différentes périodes d'horloge sont considérées. Ainsi, chaque horloge correspond à une partie du SOC à l'intérieur de laquelle le partage des ressources est possible. Notons que ces contraintes sur la disponibilité des ressources peuvent être introduites dans le but de modéliser une surface SOC associée à un noyau de processeur. À chaque fonction f et à chaque ressource r requise est associée une courbe de compromis qui représente, pour un nombre de cycles donné, la quantité de ressources nécessaires. Chaque point de la courbe de compromis fournit un quadruplet qui spécifie : le nombre de cycles, la quantité disponible pour chaque ressource r, la cible du composant SOC (cœur de processeur, cœur de processeur + coprocesseur ou matériel seul) et un profil d'ordonnancement de la ressource. Afin d'assigner un nombre de cycles et une fréquence à chaque fonction et de décider de la date de début de l'ordonnancement, un ordonnanceur basé sur une approche de type génétique est proposée. Le lecteur intéressé pourra se référer à [90] pour plus de détails.

- **Partitionnement des applications à comportement dynamique (SE temps réel) :**

  Cette partie décrit les travaux de thèse d'A.Azzedine (point 3 sur la figure 2.1).

  Étant donné i) un graphe de tâches apériodiques et périodiques multi-cadences, ii) un ensemble de contraintes temps réel et des bornes d'implantation (surface maximum autorisée, énergie disponible), iii) un modèle d'architecture basé sur un processeur avec une granularité matérielle variable, iv) un graphe HCDFG (*Hierarchical and Control Data Flow Graph*, cf.2.3.2) décrivant le parallélisme de chaque tâche (information fournie par l'estimation système) et v) une fonction de coût (surface, énergie), il s'agit de trouver l'ordonnancement temps réel et le partitionnement logiciel/matériel qui minimise la fonction de coût. Le lecteur intéressé peut se référer à [91] pour plus d'informations.

Projet EPICURE

Le projet EPICURE [92] (points 4 sur la figure 2.1) vise à étudier une nouvelle méthodologie de conception pour les architectures re-configurables. Ces nouvelles ar-

chitectures de calcul associent un processeur classique avec un élément de logique reconfigurable. L'objectif est de réaliser une optimisation dynamique des performances de l'application par l'exécution sélective de tâches sur le processeur (logiciel) et/ou sur l'élément reconfigurable (matériel). Ainsi, l'exécution d'une tâche est réalisée par la ressource de traitement (logicielle ou matérielle) la plus adaptée en fonction des contraintes du "moment" (rapidité d'exécution, consommation, utilisation des ressources). Ce type de structure permet d'envisager, outre un gain en évolutivité et en performance, une réduction du cycle de conception des applications ainsi qu'une augmentation de la pérennité des plates-formes matérielles.

L'implantation d'une application sur un calculateur reconfigurable ne peut être envisagée avec des méthodes de conception qui ne prennent pas en compte l'aspect dynamique du partitionnement qui leur est inhérent.

Les intervenants suivants sont impliqués dans le projet EPICURE :

- La société Esterel Technologies fournit l'outil de saisie de spécifications Esterel-Studio basé sur le langage synchrone Esterel et le formalisme graphique des SyncCharts.
- Le LCEI du CEA-LIST offre l'interface générique ICURE permettant l'abstraction des communications HW/SW/reconfigurable.
- Le LESTER pour les outils d'estimation système et architecturale.
- Le laboratoire I3S pour les algorithmes et stratégies de partitionnement logiciel/matériel.
- La société THALES-Communications qui, agissant en tant qu'utilisateur potentiel de la technologie, fournit une application et valide la méthode.

L'idée générale du flot de conception est de générer des solutions de partitionnement matériel/logiciel en partant d'une description fonctionnelle et structurelle des spécifications écrites dans le formalisme Esterel. Chaque proposition de partitionnement conduit ensuite à la génération d'une part d'un flot logiciel constitué du graphe de contrôle principal et des appels de fonctions et de contextes, et d'autre part d'un flot matériel constitué de plusieurs contextes à implanter sur l'élément de traitement reconfigurable. Dans la structure proposée, le contrôle des contextes matériels (reconfiguration) est intégré dans le flot logiciel et masqué par l'interface ICURE qui

agit comme une sorte de cache de contextes. Ainsi, la gestion de la machine d'états est implantée sur un processeur, les fonctions appelées par cette machine sont quant à elles implantées sur le processeur ou sur une unité matérielle dédiée. L'écriture des spécifications est effectuée avec l'outil Esterel-Studio en utilisant une méthode synchrone classique. À ce niveau, il est possible de vérifier des propriétés et d'utiliser des outils de preuve formelle. Le graphe de contrôle issu de cette phase est ensuite analysé afin d'extraire les appels de fonctions (typiquement écrites en C dans les spécifications).

C'est ici que le travail présenté dans cette thèse est utilisé : les caractéristiques de ces fonctions sont estimées avec la méthode et l'outil Design Trotter que nous avons développé. Les résultats de ces estimations sont ensuite utilisés par l'outil de partitionnement développée par l'I3S et qui effectue une classification des tâches selon l'unité de traitement optimale pour leur exécution (logiciel/matériel). Les fonctions classées matériel sont regroupées en contextes destinés à l'unité de traitement reconfigurable (un contexte correspond ainsi à une configuration de la ressource de calcul reconfigurable). S'ensuit une phase de génération de code (langage C pour le contrôle et les fonctions classées logicielles, SystemC ou VHDL pour les fonctions marquées matérielles).

Thèses en cotutelle

L'approche Design Trotter englobe également des travaux effectués dans le cadre de thèses en co-tutelles (points 5 et 6 sur la figure 2.1). Nous les introduisons rapidement dans ce qui suit.

- **Processeur Vision (Thèse de Nader Ben Amor) :**

  Ce sujet de thèse vise à définir sur la base de l'architecture PACM (Processeur Avec Co-processeur Matériel), un processeur de vision embarqué. Pour cela, outre la définition de la topologie de l'architecture et celles des unités de calcul, ce travail nécessite de formaliser le choix de l'architecture mémoire optimisée (FIFO, RAM simple ou double port, etc.). Dans un deuxième temps, l'étude s'orientera vers les méthodes de codesign logiciel/matériel permettant d'orienter la réalisation de ces primitives soit sous la forme d'instructions logicielles

dédiées (ASIP), soit par l'instanciation dans le processeur de modules matériels spécifiques vus comme unités fonctionnelles de l'ASIP (co-processeur ou accélérateurs).

- **Synthèse des communications (Thèse de Issam Maalej) :** Une des phases critique du flot de codesign logiciel/matériel correspond à l'étape de synthèse des communications. L'approche que nous souhaitons intégrer dans l'outil Design Trotter positionne la synthèse des communications après l'étape de partitionnement afin de minimiser la complexité des algorithmes à mettre en œuvre. Trois niveaux de synthèse des communications sont considérés : deux niveaux correspondent à de la synthèse d'interfaces et un troisième niveau à de la synthèse de communications. En effet, l'architecture visée repose sur l'utilisation d'unités qui sont composées d'un cœur de processeur, de coprocesseurs et d'accélérateurs. Pour ces différentes unités il est essentiel de définir des interfaces permettant de connecter aisément au bus processeur les accélérateurs considérés.

Comme nous venons de le voir l'environnement Design Trotter propose tout un panel de fonctionnalitées qui tournent autour de la partie "Flot Estimation Design Trotter". Celui-ci fait l'objet de cette thèse et est maintenant présenté plus en détail.

2.2 Flot de conception de l'estimateur système

Comme indiqué dans le chapitre précédent le but de ce travail est d'apporter une aide au concepteur lors de la phase initiale de la conception d'un système. Cette aide se matérialise sous la forme d'informations issues de l' **estimateur système** développé dans le cadre de cette thèse. Ces informations vont permettre de renseigner le concepteur sur les besoins en ressources de l'application ce qui lui permettra ainsi de définir une architecture initiale. Une vue globale de l'estimateur est représentée dans la figure 2.2.

L'estimation système permet tout d'abord de **caractériser** grâce à des **métriques d'orientation**(cf.3.1.3) un ensemble de fonctions issues de l'application visée et de les classer par ordre de **criticité** (cf.3.1.4). Le processus de caracté-

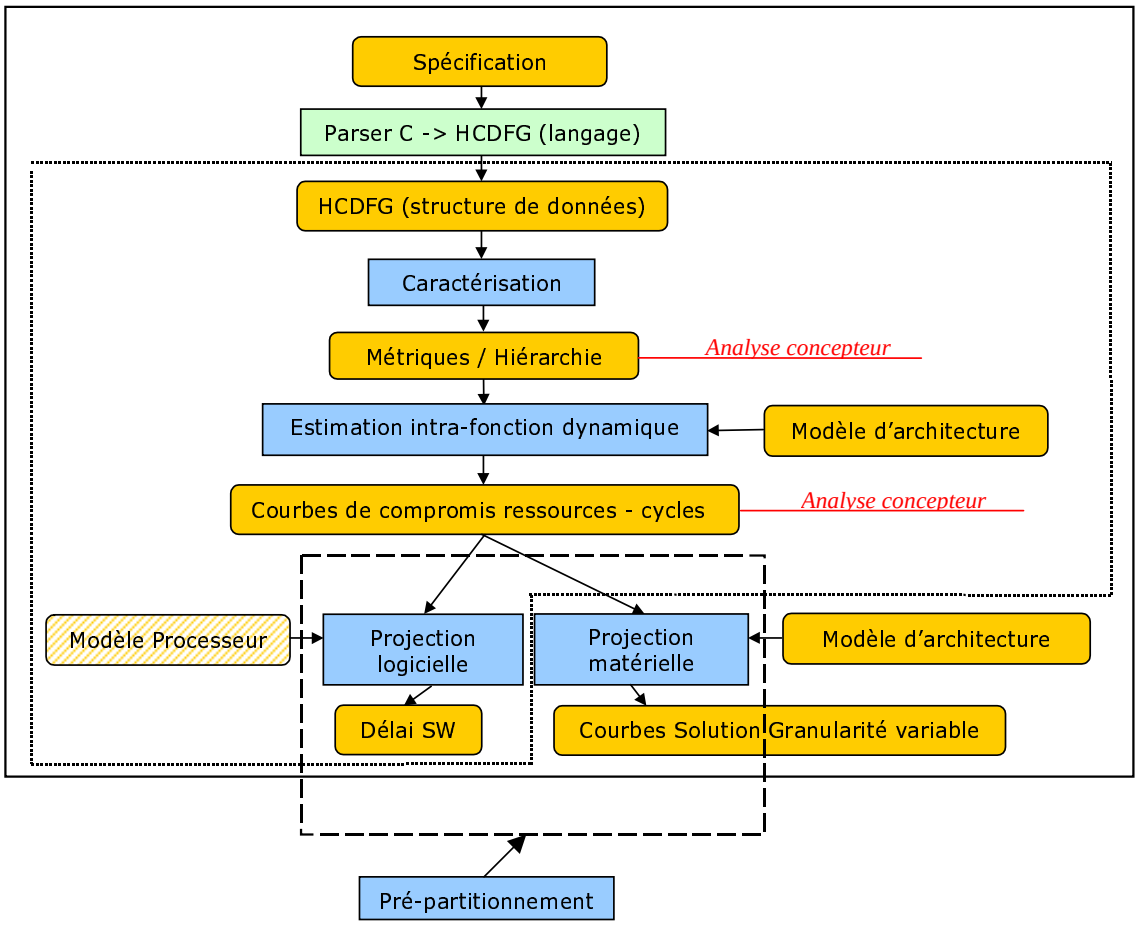

Fig. 2.2: *Flot d'estimation de Design Trotter. La partie dans le cadre pointillé correspond au travaux développés dans cette thèse.*

risation est décorrélé de toute hypothèse architecturale, c'est à dire que seule la spécification comportementale est prise en compte pour cette analyse. Le parallélisme et les performances de chaque fonction sont ensuite **estimés** en commençant par les fonctions les plus critiques. La dernière étape de l'estimation système consiste en une **projection** de chacune des fonctions de l'application sur une cible modélisée (processeur, FPGA,...) afin d'effectuer un prépartitionnement définissant ainsi les fonctions candidates à une implantation logicielle, les fonctions pressenties pour une implantation matérielle et enfin celles pour qui le choix ne peut être fait à ce niveau d'abstraction. Ainsi, le concepteur dispose en entrée de l'étape de partitionnement d'une décomposition logicielle / matérielle à partir de laquelle il peut affiner la solution finale.

Le paragraphe qui suit a pour objectifs de présenter rapidement les entrées/sorties et les étapes du flot de conception de l'estimateur système. Ces points seront présentés de manière plus détaillée dans la suite du document (chapitres 3 et 4).

Les entrées du flot d'estimation sont les suivantes :

- Une description en langage de haut niveau de l'application. Au moment de la rédaction le langage 'C' est géré par l'outil Design Trotter mais des passerelles vers d'autres langages tels que le VHDL comportemental sont à l'étude. Le choix du langage 'C' est justifié par le fait qu'il soit le langage couramment utilisé pour la description des normes.
- Des informations relatives à la construction de l'architecture et des directives relatives à l'exploration de l'espace de conception. Toutes ces informations sont contenues dans un fichier nommé UAR *User Abstract Rules*. Celui-ci est détaillé dans la partie 2.3.3.

Les étapes du flot sont les suivantes :

1. La *spécification* en langage de haut niveau est traduite dans la représentation interne HCDFG grâce au parser C->HCDFG qui a été développé au sein du LESTER par Thierry Gourdeaux. Cette représentation interne est commune aux différents modules de l'outil DesignTrotter (estimation système, estimation architecturale [93], estimation inter-fonctions [90], estimation relative [39], partitionnement temps réel [91]).

2. Une étape de **caractérisation** permet de connaître l'*orientation* et la *criticité* des fonctions de l'application. L'orientation d'une fonction indique quel est le type d'opération dominant de celle-ci. Les types considérés sont le traitement (calcul), le contrôle et la mémoire (TCM par la suite). La criticité d'une fonction indique le degré de parallélisme moyen disponible pour son exécution. Ces deux informations vont être utilisées lors de l'étape suivante afin de i) choisir le bon type d'algorithme d'estimation à appliquer aux fonctions (orientation) et ii) classer les fonctions afin de procéder à l'estimation des fonctions par ordre de criticité décroissante. En effet, comme les fonctions les plus critiques ont un potentiel de parallélisme spatial important, elles se verront allouées un certain nombre de ressources. Il est alors possible d'envisager la réutilisation de ces

ressources par les fonctions moins critiques.

3. L'étape d'**estimation intra-fonction dynamique** a pour but d'explorer le parallélisme intrinsèque des fonctions de l'application. L'estimateur que nous avons développé est de type dynamique (contrairement à une majorité de méthode qui utilise des estimations statiques, cf.1.5). L'estimation consiste à utiliser un algorithme d'ordonnancement à temps contraint pour *plusieurs* contraintes de temps. L'ordonnancement permet, pour chacune des contraintes de temps, de trouver le nombre minimum de ressources nécessaires à l'exécution de la fonction dans le temps imparti (le nombre de ressources n'est pas limité mais l'ordonnanceur cherche à le minimiser). À cet effet, l'information d'orientation obtenue dans l'étape précédente permet de choisir au mieux le type d'algorithme d'ordonnancement à appliquer (trois types sont considérés : mémoire prioritaire, traitement prioritaire et mixte). De plus, les possibilités de déroulage de boucles sont gérées afin d'explorer au mieux le parallélisme. Le résultat de l'étape d'estimation intra-fonction est une courbe de compromis temps d'exécution/ressources reflétant les possibilités d'implantation de la fonction. Les bornes minimum et maximum de cette courbe correspondent à d'une part l'exécution de la fonction en un temps égal à son chemin critique (exécution avec un haut degré de parallélisme), et d'autre part à l'exécution de la fonction dans le temps nécessitant le nombre de ressources minimum (exécution quasi séquentielle). Bien entendu il n'est pas nécessaire d'effectuer une recherche exhaustive sur tout l'espace ainsi défini. Il est en effet possible, d'une part de limiter la recherche à une gamme de valeurs définies autour d'un temps nominal pour la fonction et d'autre part, de définir un "pas" indiquant l'espace entre deux points de la courbe.

4. La flexibilité offerte par le mode de représentation de l'architecture cible nous permet d'intégrer une étape de projection de chacune des fonctions de l'application sur une cible modélisée plus précisément par rapport à l'étape d'estimation système. Deux types de projections sont possibles dans l'outil DesignTrotter : la **projection logicielle** (cible de type microprocesseur) et la **projection matérielle** (cible de type composant programmable FPGA).

La **projection logicielle** a pour but d'estimer la performance des fonctions de l'application sur une cible de type microprocesseur. Pour ce faire, les ressources du processeur sont décrites dans le fichier UAR et, pour tenir compte des limitations du processeur, un ordonnanceur à ressources contraintes à été développé à partir de celui utilisé lors de l'étape d'estimation intra-fonction. Signalons qu'une extension possible de ce travail est l'utilisation d'un langage de spécification de processeur (par exemple ARMOR [42]) permettant une description plus précise du processeur considéré. Ce extension, commencée lors d'un séjour de recherche dans le laboratoire "Embedded System Group" de l'Université d'Aalborg au Danemark, s'y poursuivra par un séjour post-doctoral.

La **projection matérielle** intervient afin d'évaluer les performances des fonctions candidates à une implantation matérielle et celles pour qui le choix n'est pas encore établi. L'étape de projection matérielle a donc pour rôle de déterminer une solution d'implantation pour toutes les fonctions pouvant être implantées sur une cible matérielle, en l'occurrence, ici un FPGA. La projection matérielle sort du cadre de cette thèse, le lecteur intéressé pourra se référer à la thèse de Sébastien Bilavarn [93] pour de plus amples renseignements.

5. Les résultats obtenus lors des étapes d'estimations vont permettre de procéder à un **prépartitionnement**. Il est ainsi possible de classer les fonctions selon le type d'implantation qui leur semble le plus approprié à priori. Trois catégorie sont considérées : les fonctions potentiellement implantables sous forme matérielle, logicielle et celles où il est encore trop tôt pour décider d'un type d'implantation.

De ce flot résulte donc plusieurs types d'informations qui seront utiles au concepteur : i)un ensemble de métriques caractérisant les fonctions de l'application, ii)des courbes de compromis temps/ressources exhibant le parallélisme de ces fonctions et iii) un prépartitionnement logiciel/matériel pour ces fonctions, ce prépartitionnement étant issu des phases de projection matérielle et logicielle.

Nous allons maintenant détailler les modèles de spécifications de l'application et de l'architecture abstraite (fichier "UAR"). Nous reviendrons en détails sur les

étapes du flot dans les chapitres 3 et 4.

2.3 Spécifications de l'application et représentation interne

La spécification de l'application est un point essentiel dans les méthodes de codesign de niveau système. De nombreux langages existent à ce niveau (C, VHDL comportemental, Esterel, ...). Les représentations internes sont encore plus nombreuses ; en effet pratiquement chaque environnement de codesign dispose de sa propre représentation interne (cf.1.2.3). L'environnement développé au LESTER n'échappe pas à la règle. La nécessité de développer un nouveau type de représentation interne est essentiellement due au fait que plusieurs outils d'estimation (estimation système, estimation architecturale [93], estimation relative [39], estimation inter-fonctions [90]) coexistent au sein de l'environnement DesignTrotter. Les contraintes imposées par ces divers outils ont conduit les membres des différents groupes de travail à chercher un modèle de représentation interne. Comme aucun autre modèle disponible ou facilement utilisable n'apportait les caractéristiques souhaitées, nous avons été amené à définir une nouvelle représentation interne offrant la flexibilité nécessaire au développement des différents estimateurs. La définition de cette représentation interne fait partie des travaux menés dans le cadre de cette thèse. Ainsi est né le modèle "HCDFG" qui est à la base d'une représentation du système à l'aide de trois vues (*System/Process/Function*) présenté en 2.3.1. Il est à noter que ce modèle sert également lorsqu'une vue du système sous forme de graphe de tâches est considéré (cf.2.3.1). Le modèle HCDFG est quant à lui décrit en 2.3.2.

2.3.1 Utilisation du modèle HCDFG

L'objectif du modèle HCDFG est de représenter une fonction afin d'appliquer des méthodes d'exploration architecturale et d'estimation (incluant les propriétés de séquentialité, de parallélisme, de mutuelle exclusion et de hiérarchie). Cette notion de fonction est exploitée différemment suivant la manière qu'à le concepteur de modéliser l'application. Au sein du projet nous distinguons deux types de représentation,

le premier repose sur la notion de machine d'état hiérarchique (HFSM), la seconde est du type graphe de tâches (GT). Dans les deux cas de figure, la séparation HFSM ou /fonction(HCDFG) est du ressort du concepteur. Ce choix est important dans le processus du partitionnement logiciel/matériel puisque c'est à ce niveau que se décide le degré le plus élevé de la granularité. Aux niveaux inférieurs, la hiérarchie est disponible au sein de la fonction (HCDFG).

HFSM

Ce type de représentation est le modèle original du projet *Design Trotter* dit SPF (*System, Process, Function*) [94] cf. Fig.2.3. Il s'agit également du modèle utilisé dans le projet RNTL EPICURE [92]. La décomposition hiérarchique au niveau système est réalisée à l'aide de l'outil ESTEREL Studio offrant notamment des fonctionnalités du type preuve formelle et simulation fonctionnelle. Lors de l'étape de spécification, le concepteur insère des appels à des fonctions C au sein des états du système lorsque que celui considère que le mode de spécification par machine d'états n'est plus adapté. Au regard du modèle SPF, la spécification atteint alors la vue "processus". Le mode de représentation propre à cette vue est un graphe de fonctions pour un état, autorisant la séquentialité, la mutuelle exclusion et le parallélisme. Dans le cadre du langage ESTEREL, cela traduit la possibilité d'insérer des appels à des fonctions C indépendantes ou des tests sur l'exécution de ces dernières. Enfin, la vue fonction correspond à la description d'une fonction particulière sous la forme d'un HCDFG. Dans le projet EPICURE le partitionnement logiciel/matériel est réalisé pour chaque état du système. Un état du système étant représenté par un graphe de fonctions, le partitionnement consiste à décider pour chaque fonction de son implantation logicielle ou d'une version d'implantation matérielle fournit par l'outil Design Trotter en considérant le temps de reconfiguration.

Au sein de l'environnement Design Trotter, le graphe de fonction (vue processus) est le point d'entrée de l'étape dite **inter-fonctions** [90], tandis qu'une fonction esseulée relève de l'estimation intra-fonction. L'objectif de cette étape est de répartir le parallélisme en choisissant le nombre de cycle alloué à chacune des fonctions à l'aide des courbes de compromis(ressources/nombre de cycles) fournit par l'estima-

tion intra-fonction.

GT

Le second type de représentation d'une application est hérité d'un mode de spécification classique dans le domaine du temps-réel, il s'agit d'un graphe de tâches. Nous utilisons cette specification, au sein du projet pour réaliser un partitionnement logiciel/matériel sur une cible pourvue d'un exécutif temps réel devant gérer des tâches périodiques et apériodiques [91]. Le modèle complet est décrit sur la figure 2.4, celui-ci comprend quatre niveaux : le niveau système définit les contraintes sur les entrées sorties, le graphe de tâches, le graphe de fonctions et le niveau HCDFG pour la spécification de chacune des fonctions. Outre la représentation sous forme de graphe de tâches[1], la différence avec le modèle SPF réside dans une restriction au niveau du graphe de fonctions. Il s'agit de la contrainte classique dans le monde du temps réel qui impose qu'aucun parallélisme ne soit considéré au sein d'un tâche [95]. Enfin, il est à noter que comme pour le niveau fonction, le niveau processus dans le modèle SPF peut-être éliminé si à une tâche est associée une seule fonction qui peut alors être de granularité élevée.

2.3.2 Le modèle HCDFG

Introduction

Le modèle HCDFG correspond à la vue fonction. Les fonctions de l'application visée, décrites sous la forme de HCDFGs, constituent le point d'entrée des travaux concernant l'estimation système présentée dans cette thèse.

Le modèle HCDFG (*Hierarchical Control and Data Flow Graph)* permet de représenter sous la forme d'un graphe hiérarchique des fonctions contenant des structures de contrôles (boucles, conditions), des traitements et manipulant des données du type scalaire ou tableau. Ce graphe a été défini dans le souci de proposer une représentation intermédiaire qui soit adaptée aux méthodes d'estimations. C'est pourquoi les informations concernant les fonctions et qui sont nécessaires pour les étapes

[1]Le terme tâche est similaire au terme processus utilisé dans la représentation HFSM, il s'agit de mieux marqué la différence entre les deux modes de représentation.

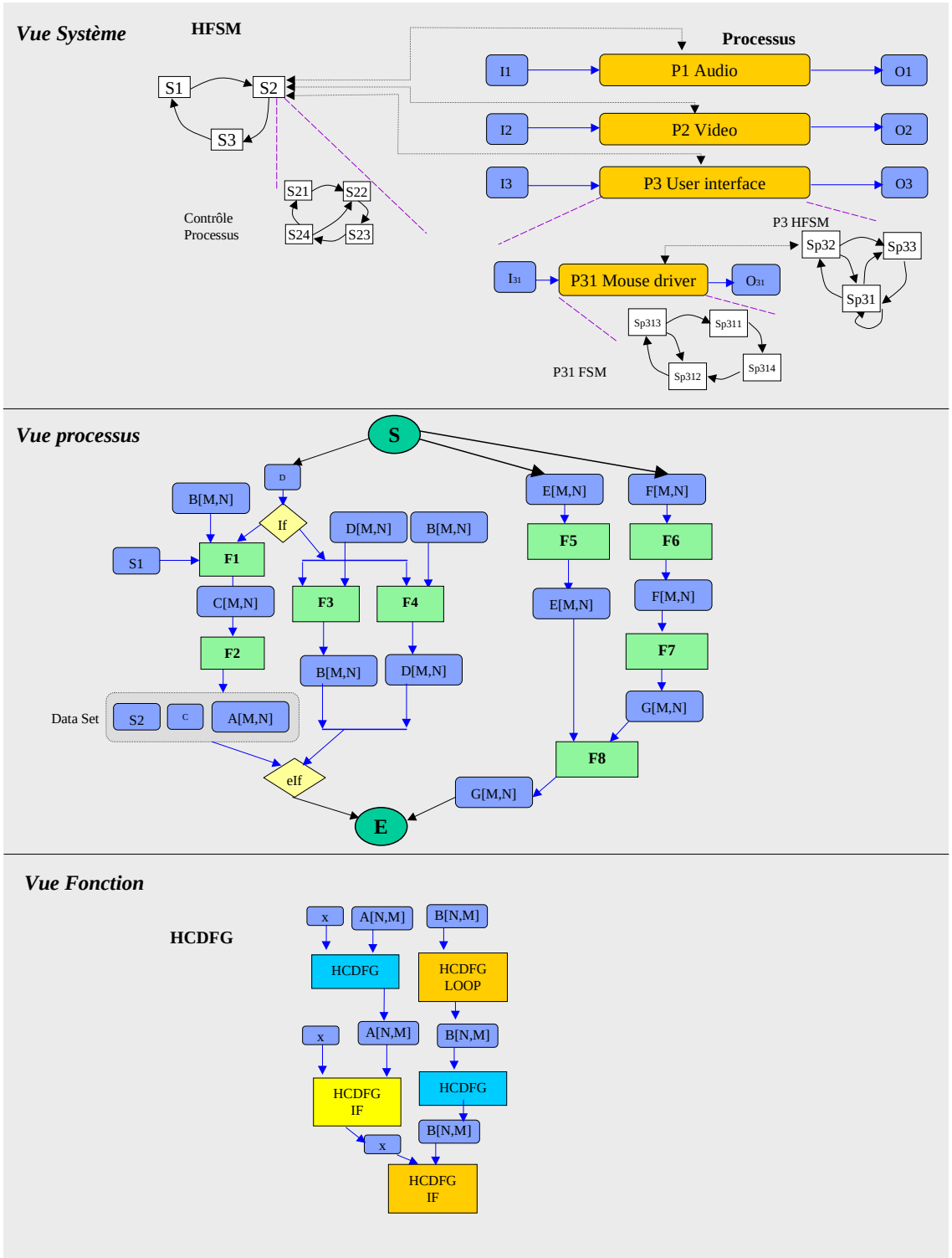

Fig. 2.3: *Exemple de spécification avec le modèle SPF.*

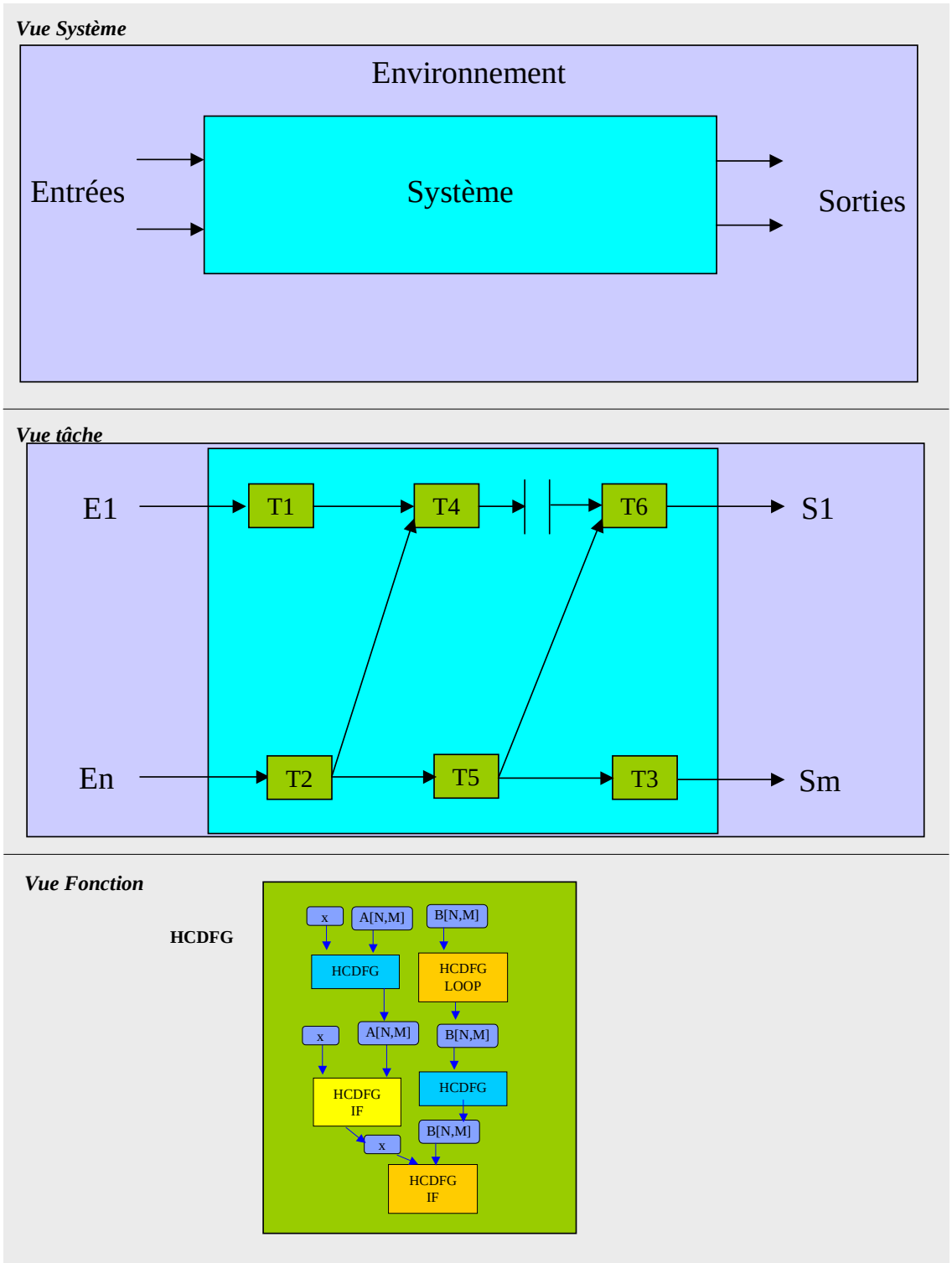

Fig. 2.4: *Spécification avec le modèle RT-SPF.*

d'estimations sont intégrées dans le graphe sous forme d'attributs. Deux attributs communs à tous les nœuds sont leurs dates d'ordonnancement ASAP (*As Soon As Possible*) et ALAP (*As Last As Possible*). D'autres attributs sont, par exemple, le niveaux de hiérarchie pour les nœuds mémoires, le type d'accès (lecture/écriture) pour les transferts, le type d'opération pour les nœuds traitements, ...

Chaque fonction écrite dans le langage de haut niveau ('C') est transformée en un HCDFG. La hiérarchie d'un HCDFG est composée des éléments suivants (par ordre de granularité croissant) : nœuds élémentaires, DFG et CDFG. Ces différents éléments sont détaillés dans ce qui suit.

DFG

Un **DFG** est un graphe qui ne contient que des nœuds mémoire et traitement élémentaires ; il représente une séquence d'opérations non conditionnelles. Il y a trois types de nœuds élémentaires (c.a.d non hiérarchiques) dont la granularité est en relation avec le modèle d'architecture décrit en 2.3.3. Un **nœud traitement** représente soit une opération arithmétique ou logique (grain fin) soit une opération plus complexe (gros grain). Des opérations du type addition, multiplication sont de grain fin alors que des opérations du type MAC ou papillon sont de grain plus gros. Dans ce cas l'opérateur est vu comme une boîte noire qui peut éventuellement déjà avoir été estimée (et peut donc disposer d'une courbe de compromis). Un **nœud mémoire** représente un transfert de donnée. Les données manipulées sont donc explicitement représentées par des nœuds dans le graphe et ne sont pas associées comme dans beaucoup de représentations aux arcs. Ceci a pour avantage de ne pas dupliquer les informations concernant les nœuds mémoires et donc de ne pas surcharger la représentation. Dans le cas des données multidimentionnelles (tableaux, vecteurs), les mécanismes d'adressage sont explicitement représentés dans le graphe à l'aide de DFG d'indices. Les paramètres principaux des nœuds mémoires sont le format de la donnée, le niveau hiérarchique de stockage (cf.2.3.3) qui peut être défini par le concepteur et le sens du transfert (lecture/écriture). Enfin un attribut permet de stocker la valeur des nœuds valués.

CDFG

Un **CDFG** est un graphe hiérarchique (un HCDFG particulier). Les CDFGs de base correspondent à des modèles de contrôles prédéfinis. Ils sont tous associés à un graphe d'évaluation. La structure For est en plus associée à un graphe d'évolution. Ces graphes de condition produisent une donnée unique, conditionnant l'exécution de la structure conditionnelle. Cette donnée pointe sur le modèle de contrôle considéré via un arc de dépendance d'ordre. Par ailleurs, le modèle de contrôle pointe sur les CDFG ou DFG de niveau hiérarchique inférieur via des arcs de dépendance d'ordre.

Un CDFG est composé des éléments suivants : deux **nœuds contrôles** (début et fin) qui indiquent le type de la structure (*if, switch-case, while, do-while et for*), un graphe d'évaluation et en plus, dans le cas d'un FOR, un graphe d'évolution et enfin un ou plusieurs H/CDFGs qui représente(nt) le/les traitements conditionné(s) par le nœud contrôle.

Graphe d'évaluation : les graphes d'évaluations produisent la donnée booléenne orientant le choix du traitement à exécuter. Un graphe d'évaluation correspond au calcul d'une condition. Le nœud donnée représentant la variable booléenne est relié au nœud contrôle qui est en tête de la structure de contrôle par un arc de dépendance d'ordre.

Graphe d'évolution : les graphes d'évolutions sont présents dans les structures de type For. Ils représentent le mécanisme d'incrément des indices de boucles (seuls les mécanismes d'incréments linéaires sont autorisés). Un graphe d'évolution correspond au calcul d'incrément de l'indice. Le nœud donnée représentant l'indice de la boucle est relié au nœud contrôle qui est en tête de la structure de contrôle par un arc de dépendance d'ordre.

HCDFG

Un **HCDFG** est un graphe qui contient des nœuds de type HCDFG, CDFG et DFG. Il permet de représenter la hiérarchie de l'application, c.a.d l'imbrication des structures de contrôles et des graphes s'exécutant en serie ou en parallèle.

Arcs

Les arcs représentent trois types de dépendances. Une **dépendance de contrôle** indique une dépendance d'ordre entre les opérations sans transfert mémoire (par exemple une opération de calcul d'indice avant l'accès à un tableau). Une **dépendance de donnée scalaire** entre deux nœuds A et B indique que le nœud A utilise un scalaire produit par le nœud B. Une **dépendance de donnée multi-dimensionnelle** est une dépendance de données dans laquelle la donnée produite n'est pas un scalaire mais un tableau. Par exemple un tel arc est créé entre un CDFG de type boucle qui lit un tableau transformé par un autre CDFG de type boucle.

Construction d'un HCDFG

En ce qui concerne la création des graphes, le principe de décomposition est relativement simple. La structure de donnée issue du *parser* 'C' est parcourue avec un algorithme de type *depth-first search*. Un HCDFG ou CDFG est créé lorsqu' un nœud conditionnel est trouvé dans le niveau de hiérarchie supérieur. Lorsqu'il n'y a plus de nœud conditionnel, un DFG est construit.

Un point important est que le modèle HCDFG couvre l'ensemble de l'application. Ainsi, les calculs d'adresses, ceux des index d'évolution de boucles et les tests sont représentés par des graphes. Un exemple de HCDFG est représenté sur la figure 2.5

Une description plus détaillée du modèle SPF-HCDFG peut être trouvée dans [94].

2.3.3 UAR - spécification de l'architecture

Introduction

Afin de pouvoir lancer le processus d'estimation, il est nécessaire d'avoir un minimum d'informations sur les ressources disponibles dans l'architecture abstraite que le concepteur désire construire et explorer. Pour ce faire, le concepteur définit un ensemble de règles, *UAR (User Abstract Rules)*. Les UAR caractérisent l'unité de traitement et la hiérarchie mémoire. La grammaire que nous avons définie pour les UAR permet de spécifier des architectures avec plus ou moins de précision se-

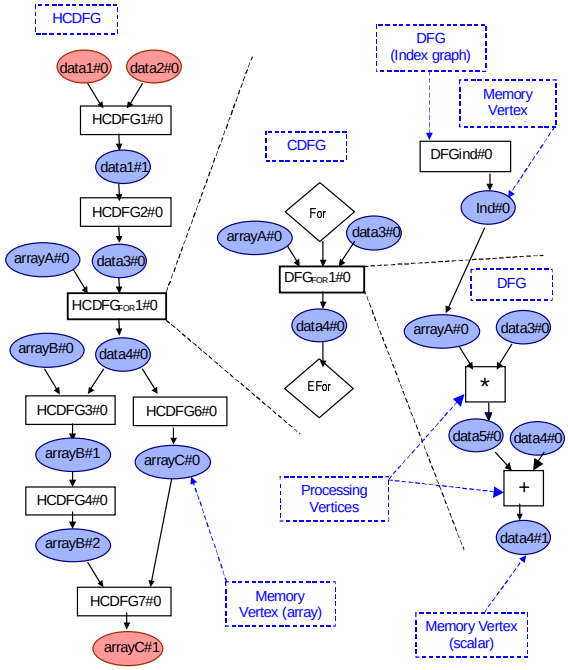

Fig. 2.5: *Élements d'un HCDFG.*

lon le niveau d'abstraction auquel sont effectuées les estimations (niveau système, projection matérielle, projection logicielle). Ainsi au niveau système le concepteur indique les types de composants qu'il souhaite utiliser (l'outil Design Trotter lui indiquera si certaines opérations ne sont pas réalisables avec les composants qu'il indique afin qu'il en ajoute). De plus, à ce niveau les UAR reposent sur une unité de temps abstraite (exprimée en cycles). La figure 2.9 présente un exemple de fichier UAR au niveau système. Lors des projections les UAR (exemple sur la figure 2.10) contiennent des information plus détaillées sur l'architecture cible ainsi que la quantité de ressources disponibles (pour l'estimation système c'est l'outil DesignTrotter qui indiquera au concepteur la quantité de ressources nécessaires). Ainsi, les UAR permettent d'apporter le niveau de précision nécessaire et suffisant à un endroit donné du flot de conception.

Ressources de traitement

Pour la partie traitement le concepteur doit définir le type de ressources disponibles : UAL, MAC, ... et le type d'opérations qu'elles peuvent réaliser. À chaque type d'opération est associé soit un nombre de cycles (estimation système) soit un

temps en secondes (projection matérielle) qui correspond à la latence de l'opérateur. D'autres attributs peuvent également être indiqués tels que le débit (dans le cas des opérateur pipeline), la consommation, la surface etc...

Ressources de mémorisation

En ce qui concerne la partie mémoire, le concepteur définit le nombre de niveaux (L_i) de la hiérarchie et le nombre de cycles associés à chaque type de transfert (lecture/écriture). Dans la figure 2.6 quatre niveaux sont considérés. Le niveau registre L_0[2], deux niveaux de cache (ou de *scratchpad memory*) L_1 et L_2 et une mémoire principale L_3.

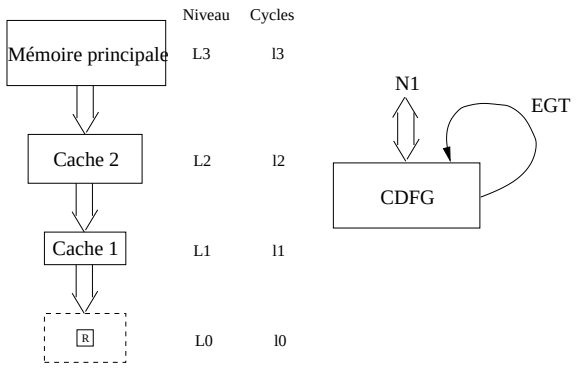

Fig. 2.6: *Modèle de hiérarchie mémoire.*

Dans la suite nous appelons "bus mémoire" les mécanismes architecturaux (mémoires multi-ports, etc.) permettant d'effectuer des transferts mémoire. Par exemple si 4 bus sont disponibles, cela revient à pouvoir effectuer 4 transferts mémoire simultanément entre deux niveaux L_{i-1} et L_i.

Nous pouvons distinguer plusieurs types de nœuds mémoire pour chaque CDFG. Le type d'un nœud est stocké sous la forme d'un attribut de ce nœud :

- N1 : les entrées/sorties, sont identifiées comme les données qui entrent et sortent des CDFGs que l'on traite.
- N2 : les données temporaires (produites lors des traitements internes).
- N3 : les données ré-utilisables (nœuds en entrée ré-utilisés), sont identifiées grâce à leur numéro d'utilisation (les variables étant à assignation unique).

[2]Bien que conceptuellement les registres ne fassent pas partie de la mémoire.

Ces trois premiers types sont détectés automatiquement par l'outil Design-Trotter.

– N4 : les données que l'on souhaite imposer en tant qu'accumulation, sont notées par un pragma lors de la spécification (*register* en C).

Nous qualifions un nœud mémoire de local si celui-ci est interne au CDFG, nous le qualifions de global si celui-ci est externe au CDFG. Les données de type N1 sont toujours globales, celles de type N4 toujours locales, alors que celles de type N2 et N3 sont initialement locales mais peuvent générer des transferts supplémentaires dans le cas où la mémoire locale est pleine (EGT, *Extra Global Transfers*).

Des exemples de fichier UAR sont présentés sur les figures 2.9 et 2.10.

2.3.4 Structure de données JAVA

Ce paragraphe a pour objectif de présenter succintement la structure de données JAVA qui implante les modèle HCDFG et UAR.

C → HCDFG Le passage d'un fichier 'C' à un fichier 'HCDFG' est réalisé par un *parser*, développé au sein du LESTER. Il ne s'agit pas ici de détailler ce *parser* mais de présenter quelques informations importantes :

1. Assignation unique : cette règle stipule que les assignations de données sont uniques, c.a.d que chaque écriture peut être différenciée ceci afin de s'affranchir des fausses dépendances de données et donc de faire apparaître tout le parallélisme. Le nom de la donnée est conservé mais se voit étendu de deux informations : les caractères "♯ x" indiquent qu'il s'agit de la $x^{\text{ième}}$ écriture et les symboles "% y" indiquent qu'il s'agit de la $y^{\text{ième}}$ lecture consécutive à la $x^{\text{ième}}$ écriture. Par exemple le code C

$$toto = a * b; b = toto + 2; c = toto;$$

donnera naissance aux nœuds *AccessVertex* suivants :

```
a#0%0 , b#0%0 , toto#1 ; toto#1%0 , b#1 , toto#1%1 et c#1
```

2. Tout est représenté par des graphes : naissance de graphes d'évaluation (IF, FOR, ...), d'évolution (FOR) et d'index (calcul d'indice de tableau).

3. Analyse des dépendances : elle consiste à chercher les dépendances vraies dans le code C et à créer les arcs correspondants dans le HCDFG. Ceci permet de conserver l'ordre dans lequel les graphes doivent s'enchaîner.

   Un exemple de passage du C vers un HCDFG est présenté sur la figure 2.7.

Fig. 2.7: *Passage du C au HCDFG.*

Structure de données : partie application En ce qui concerne la partie spécification de l'application nous trouvons les éléments qui suivent.

TreatmentVertex : ils représentent les nœuds de traitement. *MemoryVertex* : représentent les données. *AccessVertex* : représentent les accès en lecture/écriture sur les nœuds. *CompositeVertex* : permettent la hiérarchisation et la spécification des

structures de contrôle. Edge : arc de base. *OrderEdge* : arcs entre nœuds qui indiquent une dépendance d'ordre d'exécution. *BoundEdge* : arcs pour les bornes d'évolution des indices. *ConditionalEdge* : arcs conditionnels.

Un exemple passage du HCDFG vers la structure interne (en fait l'arbre de celle-ci) est présenté sur la figure 2.8.

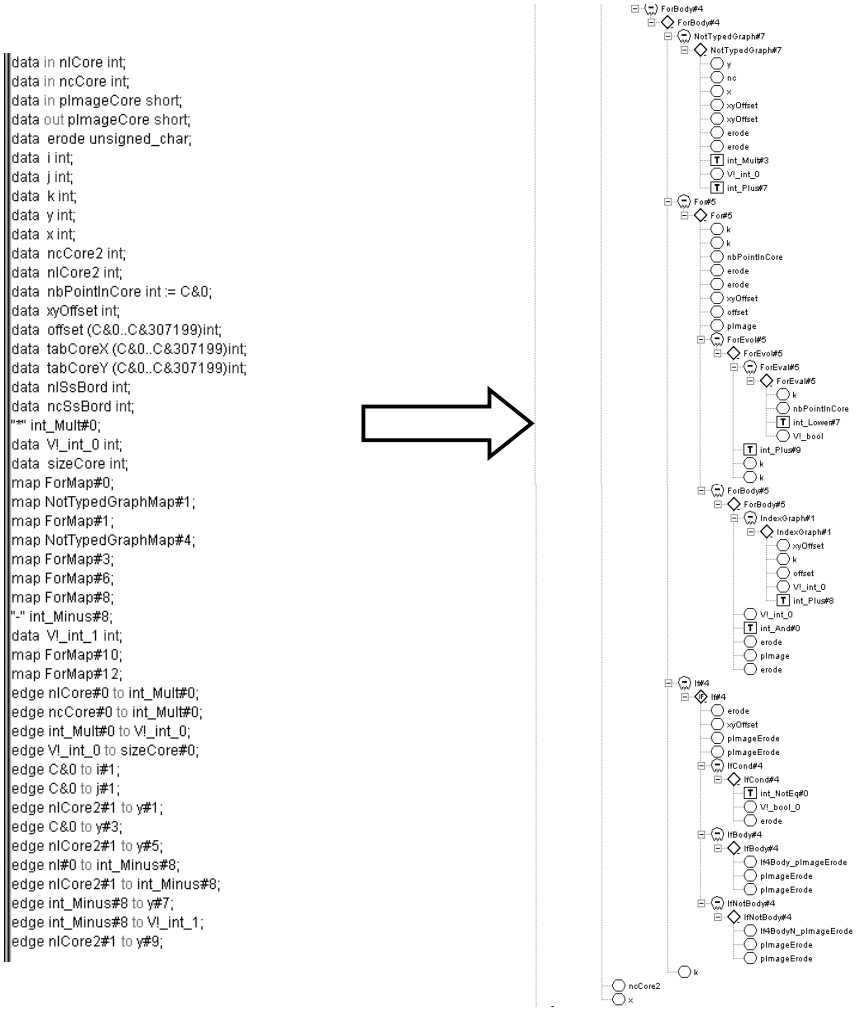

Fig. 2.8: *Passage du HCDFG à la structure de données.*

Fichier UAR → Structure de données

Exemples d'UAR Les figures 2.9 et 2.10 permettent d'illustrer les concepts présentés précédemment. La figure 2.9 présente un fichier UAR pour l'estimation système et la figure 2.10 le fichier UAR pour le FPGA Xilinx V400.

```
    <LIBRARY> systeme
    <OPERATOR> Mul
        <OPERATIONS> "*"
        <ENDOPERATIONS>
        <ATTRIBUTES>
            latency:cycle:= 1

        <ENDATTRIBUTES>
    <ENDOPERATOR>
    <OPERATOR> Alu
        <OPERATIONS> "+","-","/","%","<","<=","=","!=",">=",">","||","&&","!","<<",">>","^"
        <ENDOPERATIONS>
        <ATTRIBUTES>
            latency:cycle:= 1

        <ENDATTRIBUTES>
    <ENDOPERATOR>
    <OPERATOR> Mac
        <OPERATIONS> "*+"
        <ENDOPERATIONS>
        <ATTRIBUTES>
            latency:cycle:= 1

        <ENDATTRIBUTES>
    <ENDOPERATOR>
    <OPERATOR> But
        <OPERATIONS> "#"
        <ENDOPERATIONS>
        <ATTRIBUTES>
            latency:cycle:=1

        <ENDATTRIBUTES>
    <ENDOPERATOR>

/* MEMORY */
    <MEMORY> RAM_DP
        <ATTRIBUTES>
            access_mode:=rw
            latency_read:cycle:=1
            latency_write:cycle:=1
        <ENDATTRIBUTES>
    <ENDMEMORY>

    <MEMORY> ROM
        <ATTRIBUTES>
            access_mode:=readonly
            latency_read:cycle:=1
        <ENDATTRIBUTES>
    <ENDMEMORY>
<ENDLIBRARY>
```

Fig. 2.9: *UAR de niveau système. Chaque opérateur dispose de la liste des opérations qu'il peut effectuer. et le temps requis pour les effectuer (temps exprimé en nombre de cycles). Les ressources mémoires sont quant à elle caractérisées par le(s) mode(s) disponibles (lecture/écriture) et leur(s) latence(s).*

```
/* ARCHITECTURE */

<LIBRARY> V400EPQ240-7:virtex
    <UNITS>
        CL:="SLICE"
        CD:="BRAM"
    <ENDUNITS>
    <ATTRIBUTES>
        NB_CL:INT:=4800
        NB_CD:INT:=40
        NB_tristate:INT:=4800
        NB_ES:INT:=158
    <ENDATTRIBUTES>

/* RESSOURCES */
...
    <OPERATOR> add_16
        <OPERATIONS> "+"
        <ENDOPERATIONS>
        <ATTRIBUTES>
            implementation:=CL
            latency:time:=5.726 ns
            area:INT:=8
            datawidth:INT:=16
            ctllines:INT:=1
        <ENDATTRIBUTES>
    <ENDOPERATOR>

    <OPERATOR> mul_16
        <OPERATIONS> "*"
        <ENDOPERATIONS>
        <ATTRIBUTES>
            implementation:=CL
            latency:time:=19.129 ns
            area:INT:=140
            datawidth:INT:=16
            ctllines:INT:=1
        <ENDATTRIBUTES>
    <ENDOPERATOR>
.....
/* MEMORY */
    <MEMORY> RAM_DP
        <ATTRIBUTES>
            implementation:=CD
            access_mode:=rw
            bit_per_cell:INT:=4096
            latency_read:TIME:=7.2 ns
            latency_write:TIME:=7.2 ns
        <ENDATTRIBUTES>
    <ENDMEMORY>
...

<ENDLIBRARY>
```

Fig. 2.10: *Extrait du fichier UAR du FPGA Xilinx V400. Comparé au fichier de niveau système de la figure 2.9 de nouvelles informations relatives à la cible architecturale sont apparues : la quantité de ressources disponibles, le type d'implantation (cellules logiques ou dédiées), le temps en ns etc.*

Structure de données : partie architecturale Les éléments suivant de la structure de données concernent la partie architecturale (UAR) :

MemoryAccess : ressources d'accès mémoires, associées d'une part aux *AccessVertex* de la partie spécification de l'application et d'autre part aux attributs des parties mémoires du fichier UAR (temps accès, mode lecture/écriture, ...).

Operator : ressources de traitements, associées d'une part aux *TreatmentVertex* de la partie spécification de l'application et d'autre part aux attributs des parties traitements du fichier UAR (latence, ...).

2.4 Bilan

Dans ce chapitre nous avons présenté le flot de conception de l'estimation système. À partir d'une description en langage de haut niveau et d'une description plus ou moins affinée d'une architecture cible (UAR), l'estimation système fournit au concepteur un ensemble d'informations qui vont lui permettre de spécifier une architecture initiale. Le modèle de représentation interne et le fichier UAR ont été présentés. Les étapes du flot sont détaillées dans les chapitres qui suivent.

Chapitre 3

Caractérisation Macroscopique

Ce chapitre a pour objectif de présenter l'étape de caractérisation macroscopique. L'objectif de celle-ci est d'analyser les fonctions de l'application afin d'en déterminer deux paramètres : d'une part leur orientation Traitement, Contrôle ou Mémoire (TCM) et d'autre part leur criticité (parallélisme moyen disponible). À cet effet un ensemble de métriques permettant d'obtenir ces informations est défini. Après avoir décrits les différents types d'opération rencontrées dans une fonction nous présentons les différentes métriques.

3.1 Métriques

Les fonctions qui composent les applications à estimer peuvent présenter des caractéristiques très différentes que ce soit au niveau de leur **orientation** (traitement, contrôle ou mémoire noté TCM) ou de leur niveau de **parallélisme** potentiel. La caractérisation des fonctions selon ces critères a deux objectifs : d'une part guider le concepteur dans ses choix architecturaux et d'autre part guider l'étape d'estimation intra-fonction afin d'y utiliser les algorithmes les plus adaptés aux caractéristiques des fonctions.

La caractérisation des fonctions est réalisée sans aucune directive architecturale, c.a.d sans la plupart des informations présentes dans le fichier UAR. Les résultats de cette étape permettent d'esquisser une architecture cible qui servira à effectuer les premières estimations. Les métriques permettent de mettre en avant le style d'architecture adaptée à une fonction. Ceci inclut le compromis *wider/deeper*, c.a.d le degré de parallélisme spatial versus le parallélisme temporel, la nécessité de mécanismes de contrôles complexes, les besoins en termes de mémoires locales et de bande passante ou bien encore le besoin en ressources dédiées pour la génération des adresses.

L'étape de caractérisation consiste à calculer un ensemble de **métriques**. Le calcul de ces métriques repose sur un dénombrement des différents types d'opérations (TCM) rencontrées dans les fonctions à caractériser.

Nous avons défini les métriques suivantes :

- Métriques d'orientation : métrique d'orientation mémoire : *Memory Orientation Metric (MOM)*, métrique d'orientation contrôle : *Control Orientation Metric (COM)*.
- Métrique de criticité : γ.
- Métriques de réutilisation des données : Data Reuse Metric (DRM) et Hierarchical Data Reuse Metric (HDRM).

D'autres métriques seront programmées telles que :

- Métrique de calcul d'adresses : *Address Orientation Metric (AOM)*.
- Histogramme du format des données (nombre de bits).

3.1.1 Définitions

Avant de donner les formules des différentes métriques nous définissons quelques termes et abréviations qui seront utilisés par la suite.

Nous distinguons trois types d'opérations : traitement, contrôle et mémoire. Nous proposons les "définitions" suivantes :

- **Traitement** : le traitement comprend les opérations de calcul (ALU, MAC...), les calculs d'adresses (ceux-ci pouvant être exécutés sur des ALUs ou sur des unités spécialisées de type AGU *Adress Generator Unit*) [96], les tests de type "déterministes", c.a.d les tests du type "i <= constante ?" (qui sont essentiellement rencontrés dans les boucles). Ce sont des tests "prévisibles" qui peuvent être éliminés si par exemple on déroule une boucle, ils ne posent donc pas de problème de branchement conditionnel. Dans un premier temps nous ferons l'hypothèse que toutes ces opérations nécessitent un temps de cycle. Plus tard on pourra affiner cette valeur en introduisant des coefficients correcteurs en fonction de connaissances supplémentaires sur l'architecture cible. Par exemple des opérations de types ALU et MAC ne nécessiteront qu'un seul cycle alors qu'un accès en mémoire globale en prendra 2 ou 3.
- **Contrôle** : nous comptabilisons comme contrôle les tests indéterministes (pour lesquels la condition est de type *data-dependant*, c.a.d que la condition n'est connue qu'au cours de l'exécution (ou simulation)).
- **Mémoire** : pour l'aspect mémoire les opérations prises en compte sont les accès en lecture et en écriture aux données. Nous pouvons distinguer leur taille, leur(s) temps d'accès et le nombre d'accès en lecture/écriture simultanés sur celles-ci. De plus, la mémoire peut-être de type hiérarchique. Dans un premier temps nous ferons une distinction entre la mémoire "globale" et la mémoire "locale". La mémoire dite globale comprend les mémoires ram, rom, les disques durs, ... La mémoire locale comprend essentiellement les unités de types caches et les registres internes. Enfin, la stratégie de gestion de la mémoire locale pourra être définie (algorithme de type FIFO, LRU...) par l'utilisateur. Lors du déroulement du flot de conception nous affinerons les données relatives aux mémoires, en prenant un modèle d'architecture ou de processeur pour lequel

nous connaîtrons, par exemple, la taille du cache. Ceci permettra de faire la comparaison entre le modèle "idéalisé" et le cas avec un modèle plus précis.

Voici la liste des abréviations utilisées dans la suite de ce chapitre :

- Nt : nombre d'opérations de traitement.
- Nc : nombre d'opérations de contrôle non déterministes (tests).
- Nm : nombre d'accès à la mémoire globale.

  avec Nb_t = nombre d'opérations de type ALU, Macs,...+ les calculs d'adresses + tests déterministes.

- vrai/faux : graphes des branches mutuellement exclusives dans une structure "IF-THEN-ELSE".
- p_{vrai}, p_{faux} : probabilités d'exécutions des branches vrai et faux.
- i : graphe d'une branche dans une structure de type "SWITCH-CASE".
- p_i : probabilité d'exécution d'une branche i.
- evaluation : graphe d'évaluation (présent dans les toutes les structures de contrôle)
- evolution : graphe d'évolution (présent dans les structures "FOR")
- for : graphe du cœur de boucle d'une structure "FOR".
- while : graphe du cœur de boucle d'une structure "WHILE" ou "DO-WHILE".
- N_{it} : nombre d'itérations dans les structures répétitives.

3.1.2 Principe de calcul des métriques pour une fonction (c.a.d pour un HCDFG)

Le calcul des métriques est fait de façon hiérarchique avec une approche de type ascendante. Les métriques sont d'abord calculées pour les graphes feuilles (DFG) et ensuite pour les graphes de contrôle (CDFG) et hiérarchiques (HCDFG) en suivant des règles de combinaisons de types exclusive (CDFG), parallèle et séquentielle (CDFG, HCDFG). Durant les combinaisons ce ne sont pas les valeurs des métriques propres aux graphes concernés qui sont utilisées mais les dénombrements des différents types d'opérations des graphes (par exemple $\mathrm{MOM}_{graphe1,graphe2}$ est différent de $\mathrm{MOM}_{graphe1}$ + $\mathrm{MOM}_{graphe2}$), cf.3.1.3, 3.1.3, 3.1.4 et 3.1.4.

3.1.3 Métriques d'orientation

Métrique d'orientation mémoire MOM

La métrique MOM indique la fréquence des accès mémoire dans un graphe. Les valeurs de MOM sont normalisées dans l'intervale [0 ;1]. Comme introduit précédemment, au niveau système la taille de la mémoire locale associée à l'unité de traitement n'est pas encore définie (la caractérisation à un rôle de guidage). Il faut donc définir une notion générale de localité liée à l'application et non à l'architecture. Aussi avons nous pris le parti de considérer la différence entre accès locaux et globaux de la manière suivante. Un accès global est un transfert de donnée qui est définie à l'extérieur du graphe considéré. Un accès local est un transfert de donnée au sein de ce graphe (transfert du type registre-registre d'un point de vue architectural). Des valeurs de métrique MOM élevées indique que les traitements sont appliqués à des données nouvelles (par opposition aux données produites par des calculs précédents et qui peuvent rester en mémoire locale en fonction de sa taille). Plus MOM tend vers 1, plus la fonction est dominée par les accès mémoire (si MOM = $\frac{3}{4}$) alors un traitement requière en moyenne trois accès mémoire). Par exemple dans le cas où les contraintes temporelles sont serrées, une valeur de MOM proche de 1 indique que des mémoires ayant de bonnes performances (bande-passante élevée, double ports, ...) ainsi qu'une exploitation efficace d'une hiérarchie mémoire et de la localité des données sont nécessaires [57].

- **MOM d'un DFG**

  Pour un DFG la métrique d'orientation mémoire MOM se calcule ainsi :

$$MOM = \frac{\text{Nb accès mémoire globale}}{\text{Nb accès mémoire globale} + \text{Nb opération traitement}}$$

  soit

$$MOM = \frac{Nm}{Nm + Nt}$$

- **MOM d'une structure IF-THEN-ELSE**

$$MOM_{IF} = \frac{Nm_{vrai} * p_{vrai} + Nm_{faux} * p_{faux} + Nm_{evaluation}}{\sum_{x=t,c,m}(Nx_{vrai} * p_{vrai} + Nx_{faux} * p_{faux} + Nx_{evaluation})}$$

– **MOM d'une structure SWITCH**

$$MOM_{SWITCH} = \frac{\sum_i Nm_i * p_i + Nm_{evaluation}}{\sum_{x=t,c,m}[\sum_i (Nx_i * p_i) + Nx_{evaluation}]}$$

– **MOM d'une structure FOR**

$$MOM_{FOR} = \frac{Nm_{evaluation} + Nmfor + Nm_{evolution}}{\sum_{x=t,c,m}(Nx_{evaluation} + Nx_{for} + Nx_{evolution})}$$

– **MOM d'une structure WHILE et DO-WHILE**

$$MOM_{WHILE,DO-WHILE} = \frac{N_{it} * Nm_{while} + (N_{it} + 1) * Nm_{evaluation}}{\sum_{x=t,c,m}(N_{it} * Nx_{while} + ((N_{it} + 1) * Nx_{evaluation}))}$$

en supposant N»1 on obtient alors :

$$MOM_{WHILE,DO-WHILE} \simeq \frac{Nm_{while} + Nm_{evaluation}}{\sum_{x=t,c,m}(Nx_{while} + Nx_{evaluation})}$$

– **MOM d'un HCDFG**

$$MOM_{HCDFG} = \frac{\sum_{\text{sous-graphes j}} Nm_j}{\sum_{\text{sous-graphes j, x=t,c,m}} Nx_j}$$

Métrique d'orientation contrôle

La métrique COM indique la fréquence des opérations de contrôle indéterministes (c.a.d qui ne peuvent être éliminées lors de la compilation). Cette métrique est nulle pour les DFGs puisque ceux-ci, par définition, ne peuvent pas inclure d'opérations de contrôle ou de la hiérarchie. Les valeurs de la métrique COM sont normalisées dans l'intervale [0 ;1]. Plus la valeur de COM est proche de 1, plus la fonction est dominée par des opérations de contrôle (si COM = $\frac{3}{4}$ alors une opération du graphe sur quatre est de type test nondéterministe). Ceci indique qu'une architecture adaptée à cette

fonction possède des mécanismes de gestion du contrôle évolués et robustes et que l'utilisation d'un pipeline de longueur trop importante n'est sans doute pas efficace.

Dans le cas d'un CDFG la métrique d'orientation contrôle se calcule avec la formule générale suivante :

$$MOC = \frac{Nb\ d'opérations\ de\ contrôle\ non\ déterministe}{Nb\ d'opérations\ de\ traitement\ +\ Nb\ d'opérations\ de\ contrôle\ non\ déterministe\ +\ Nb\ d'accès\ à\ la\ mémoire\ globale}$$

soit

$$MOC = \frac{Nc}{Nt + Nc + Nm}$$

- **MOC d'un DFG**

  Cette métrique est nulle pour les DFGs puisque ceux-ci n'incluent ni d'opérations de contrôle ni de hiérarchie.

- **MOC d'une structure IF-THEN-ELSE**

$$MOC_{IF} = \frac{Nc_{vrai} * p_{vrai} + Nc_{faux} * p_{faux} + Nc_{evaluation}}{\sum_{x=t,c,m}(Nx_{vrai} * p_{vrai} + Nx_{faux} * p_{faux} + Nx_{evaluation})}$$

- **MOC d'une structure SWITCH**

$$MOC_{SWITCH} = \frac{\sum_i Nc_i * p_i + Nc_{evaluation}}{\sum_{x=t,c,m}[\sum_i (Nx_i * p_i) + Nx_{evaluation}]}$$

- **MOC d'une structure FOR**

$$MOC_{FOR} = \frac{Nc_{evaluation} + Ncfor + Nc_{evolution}}{\sum_{x=t,c,m}(Nx_{evaluation} + Nx_{for} + Nx_{evolution})}$$

- **MOC d'une structure WHILE et DO-WHILE**

$$MOM_{WHILE,DO-WHILE} = \frac{N_{it} * Nc_{while} + (N_{it} + 1) * Nc_{evaluation}}{\sum_{x=t,c,m}(N_{it} * Nx_{while} + ((N_{it} + 1) * Nx_{evaluation}))}$$

en supposant N»1 on obtient alors :

$$MOM_{WHILE,DO-WHILE} = \frac{Nc_{while} + Nc_{evaluation}}{\sum_{x=t,c,m}(Nx_{while} + Nx_{evaluation})}$$

– **MOC d'un HCDFG**

$$MOC_{HCDFG} = \frac{\sum_{\text{sous-graphes j}} Nc_j}{\sum_{\text{sous-graphes j, x=t,c,m}} Nx_j}$$

3.1.4 Métrique de criticité

L'approche adoptée dans notre méthodologie consiste à estimer les fonctions les plus critiques en priorité (les fonctions les moins critiques pouvant potentiellement ré-utiliser les ressources allouées aux fonctions les plus critiques). La criticité est définie par la métrique γ telle que

$$\gamma = \frac{\text{NB opérations de type transfert de données et calcul}}{\text{Chemin Critique}}$$

Le chemin critique au sein d'un DFG est défini comme la plus longue chaîne d'opérations séquentielles dans celui-ci (en nombre de cycles). Le chemin critique d'une fonction est obtenu hiérarchiquement par combinaison des chemins critiques des HCDFGs qui la composent.

γ indique le parallélisme moyen disponible à un niveau de hiérarchie donné : considérons un HCDFG contenant 5 DFGs parallèles. Si chaque DFG s'exécute séquentiellement alors γ pour chacun des DFGs a pour valeur 1 mais la métrique γ a pour valeur 5 pour le HCDFG.

Une fonction dont la métrique γ est élevée pourra tirer bénéfice d'une architecture offrant un degré de parallélisme important. Par contre, une fonction dont la métrique γ est faible (c.a.d se rapprochant de 1) a une exécution plutôt séquentielle, l'accélération de cette fonction passe donc par l'exploitation du parallélisme temporel (c.a.d une longueur de pipeline assez importante) car d'un point de vue consommation, une fonction avec un fort parallélisme offre l'opportunité de réduire la fréquence d'horloge en exploitant le parallélisme spatial.

En ce qui concerne les structures de contrôle la métrique de criticité est calculée ainsi :

γ d'une structure IF-THEN-ELSE

$$\gamma_{IF} = (\text{proba gV}*\frac{\text{Nb opérations gV}}{\text{Chemin Critique gV}})+(\text{proba gF}*\frac{\text{Nb opérations gF}}{\text{Chemin Critique gF}})+(\frac{\text{Nb opérations gCond}}{\text{Chemin Critique gCond}})$$

γ d'une structure SWITCH

$$\gamma_{SWITCH} = \frac{\text{Nb opérations gCond} + \sum_i \text{proba g}Case_i * \text{Nb opérations g}Case_i}{\text{chemin critique gCond} + \sum_i \text{chemin critique g } Case_i}$$

γ d'une structure FOR

$$\gamma_{FOR} = \frac{\text{Nb opérations gEvol + Nb opérations gFor + Nb opérations gCond}}{\text{Chemin Critique gEvol + Chemin Critique gFor + Chemin Critique gCond}}$$

γ d'une structure WHILE et DO-WHILE

$$\gamma_{WHILE} = \frac{\text{Nb iter (Nb opérations gWhile) + (Nb iter+1)(Nb opérations gCond)}}{\text{Nb iter (Chemin Critique gWhile) + (Nb iter+1)(Chemin Critique gCond)}}$$

en supposant N»1 on obtient alors :

$$\gamma_{WHILE} = \frac{\text{Nb opérations gWhile + Nb opérations gCond}}{\text{Chemin Critique gWhile + Chemin Critique gCond}}$$

γ de HCDFGs

$$\gamma_{\text{série}} = \frac{\sum_{\text{sous-graphes j, x=t,c,m}} Nx_j}{\sum_{\text{sous-graphes j}} CP_j}$$

$$\gamma_{\text{parallèle}} = \frac{\sum_{\text{sous-graphes j, x=t,c,m}} Nx_j}{MAX_{\text{sous-graphes j}}(CP_j)}$$

3.1.5 Métriques DRM et HDRM

Métrique DRM

La gestion de l'équilibrage entre les traitements et les accès mémoire est un point critique qui permet de faire face au goulet d'étranglement que constitue la

bande passante limitée des mémoires par rapport aux performances des cœurs de processeurs. Cet équilibrage peut être obtenu par l'utilisation d'algorithmes d'ordonnancement adaptés à l'orientation des fonctions. Afin de nous aider dans l'analyse des fonctions et dans le choix des algorithmes d'ordonnancement adaptés à celles-ci, nous avons défini une métrique : *DRM (Data Reuse Metric)*. Cette métrique tient compte de la taille mémoire locale dont nous devons effectuer une première estimation. Étant donné le niveau d'abstraction auquel nous nous situons, nous n'avons pas besoin d'estimations précises au niveau temporel : seule la répartition par cycle nous intéresse. De plus, comme nous voulons obtenir les estimations rapidement, les méthodes de type coloriage de cliques sont exclues. Nous utilisons donc la durée de vie moyenne des données pour estimer le nombre moyen de données vivantes à chaque cycle. Les durées de vie minimum et maximum d'une donnée d sont définies comme suit :

$$MinDL(d) = ASAP(d^n) - ALAP(d^1) + 1$$

$$MaxDL(d) = ALAP(d^n) - ASAP(d^1) + 1$$

avec $ASAP$ la date d'ordonnancement au plus tôt et $ALAP$ la date d'ordonnancement au plus tard, d^1 et d^n respectivement la première et la dernière lecture de la donnée d, pour une contrainte de temps donnée. De là nous en déduisons la durée de vie moyenne de la donnée d :

$$AvDL(d) = \frac{1}{2}\left(MinDL(d) + MaxDL(d)\right)$$

Enfin, nous obtenons le nombre moyen de données vivantes par cycle (AAD) comme suit :

$$AAD = \frac{1}{T}\sum_{d} AvDL(d)$$

avec T le nombre de cycles alloués à la fonction estimée. Le nombre de transferts locaux se transformant en transferts globaux (*Extra Global Transfers*) en raison de la taille mémoire insuffisante est donné par :

$$EGT = \begin{cases} (AAD - UM)T \; si \; AAD > UM \\ 0 \; sinon \end{cases}$$

avec UM la taille mémoire locale en nombre de données. Celle-ci peut être re-définie par l'utilisateur en fonction de ses connaissances éventuelles sur l'architecture. Sa valeur par défaut est égale à AAD.

La métrique DRM donne le rapport entre le nombre d'accès globaux et locaux, plus fine que la métrique MOM, la DRM est calculée lors de l'ordonnancement des DFG. Elle requière pour cela les dates ASAP-ALAP des nœuds et la taille de la mémoire locale. Un accès local qui produit un conflit de mémoire (mémoire locale pleine) implique une lecture et une écriture globale, donc le nombre d'accès globaux supplémentaires est $\beta * EGT$ avec β le nombre de cycles moyen nécessaire pour un accès à la mémoire globale (mémoire principale avec ou sans cache). Suivant le degré d'abstraction considéré, β est extrait d'un modèle plus ou moins affiné de cette mémoire. La recherche de β et son ajout dans le fichier UAR restent à la charge du concepteur puisqu'il s'agit de choix d'implantation. Dans le cas d'une seule mémoire qui nécessite uniquement un cycle par accès, $\beta = 2$. Si nous considérons la hiérarchie mémoire de la figure 2.6 avec les paramètres suivants : L1 : $l1 = 1$, L2 : $l2 = 2$, L3 : $l3 = 3$ et si MR_i est le taux d'échec du cache de niveau i, alors :

$$\beta = 1.(1 - MR_1) + 2.MR_1.(1 - MR_2) + 3.MR_1.MR_2$$

Si nous généralisons à K niveaux de hiérarchie nous obtenons :

$$\beta = \sum_{k=1}^{K} \left(l_k + (1 - MR_k). \prod_{j=1}^{K-1} MR_j \right)$$

Finalement la métrique DRM s'obtient comme suit :

$$DRM = \frac{N1 + \beta.EGT}{N1 + \beta.EGT + N2 + N3 + N4} = \frac{\text{Nb accès mémoire globale}}{\text{Nb total accès mémoire}}$$

Lorsque $DRM \rightarrow 1$ la fonction est orientée transfert mémoire, et lorsque $DRM \rightarrow 0$ la fonction est orientée traitement. L'utilisation de la métrique DRM est illustrée dans le chapitre suivant, lors de la présentation de l'ordonnancement.

Métrique HDRM

La métrique HDRM (*Hierarchical Data Reuse Metric*) étend la métrique DRM à la réutilisation de données inter-HCDFG. Son calcul est général et utilisé pour

toutes les estimations hiérarchiques. Le principe de ce calcul, qui repose sur un regroupement deux à deux de HCDFG, est illustré par la figure 3.1. A1 et A2 sont les quantités de données lues par HCDFG1 et HCDFG2 respectivement. A12 représente les données lues par les deux graphes et A3 représente les données résultats produites par le premier graphe et transmises au second. La métrique HDRM donne le ratio de données réutilisées entre les deux HCDFG qui peuvent s'exécuter soit parallèlement (dans ce cas A3=0), soit séquentiellement ou bien encore par une combinaison de ces deux cas.

La métrique HDRM est donc calculée comme suit :

$$HDRM = \frac{A_3 + A_{12}}{A_1 + A_2 + A_3 + A_{12}}$$

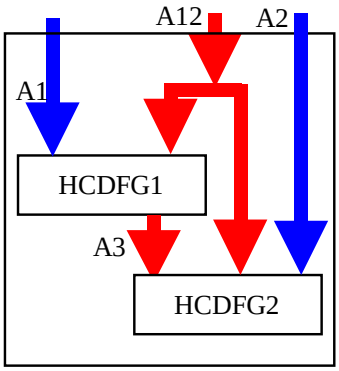

Fig. 3.1: *Illustration de la métrique HDRM.*

Lorsque HDRM tend vers 1 cela indique que le ratio de réutilisation est maximum : toutes les données lues par HCDFG2 sont partagées/produites avec/par HCDFG 1. Cela indique également qu'une mémoire locale peut être implantée efficacement. Au contraire, lorsque HDRM tend vers 0 cela indique qu'il n'y pas ou très peu de données réutilisées et donc que les possibilités d'optimisation sont faibles. Dans les applications de type traitement de signal (typiquement des boucles) la réutilisation des données peut avoir un impact important sur les performances et le coût car les possibilités d'optimisation sont essentiellement centrées sur les aspects mémoires des nœuds de boucles. Nous pouvons utiliser la métrique HDRM dans ce genre de situation car les graphes HCDFG1 et HCDFG2 peuvent être vus comme deux itérations successives d'une boucle. L'implantation de la métrique HDRM dans l'outil Design Trotter est en cours, cependant nous présentons un exemple théorique

permettant d'illustrer son concept. Nous avons considéré l'algorithme bien connu d'estimation de mouvement de la compression MPEG. Nous avons calculé la métrique HDRM pour plusieurs niveaux de boucles : colonne et ligne d'un bloc $n * n$ (avec n=8), colonne et ligne de la fenêtre de référence $2m + n - 1 * 2m + n - 1$ (avec m=16) et colonne et ligne d'une *frame* $W * H$ (au format Qcif : w=174, H=144). Le tableau 3.1 présente les valeurs prises par les données A_i et la métrique DRM pour chaque niveau hiérarchique de boucle (avec OF : Old Frame, NF : New Frame). La dernière colonne indique une taille mémoire potentielle pour stocker l'ancienne *frame*. Nous pouvons voir que les meilleures opportunités de réutilisation des données sont présentes pour les cas suivants : colonne de la fenêtre de référence (HDRM = 88%), ligne de la fenêtre de référence (HDRM = 81%) et ligne d'une *frame* (HDRM=87Ceci indique donc que l'utilisation d'une mémoire had hoc pour stocker localement ces données ré-utilisées doit permettre d'obtenir de bonne performances et également de réduire la consommation.

Niveau	A1-OF	A1-NF	A2-OF	A2-NF	A12-OF	A12-NF	A3	HDRM	Mémoire OF
colonne bloc	1	1	1	1	0	0	1	0,20	1
ligne bloc	8	8	8	8	0	0	1	0.03	n
colonne fenêtre	8	0	8	0	56	64	0	0,88	$n * n$
ligne fenêtre	39	0	39	0	273	64	0	0,81	$(2m + n - 1) * n$
colonne frame	312	312	312	312	1209	1209	0	0,66	$(2m + n - 1)^2$
ligne frame	1408	1408	1408	1408	5456	25336	0	0,87	$W * (2m + n - 1)$

TAB. 3.1 – *Résultats théoriques pour la métrique HDRM sur l'exemple de l'estimation de mouvement MPEG.*

3.1.6 Bilan

Dans ce chapitre nous avons présenté l'étape de caractérisation macroscopique. Celle-ci a pour but d'analyser les fonctions de l'application afin d'en extraire deux caractéristiques : l'orientation et la criticité. Á cet effet, un ensemble de métriques a été défini. Ces métriques vont guider d'une part l'ordre dans lequel les fonctions vont être estimées, et d'autre part le choix des algorithmes d'ordonnancement qui seront utilisés lors de l'étape d'estimation intra-fonction. Cette étape est présentée dans le chapitre suivant.

Chapitre 4

Estimation intra-fonction

Dans ce chapitre l'étape d'estimation intra-fonction est présentée. Cette étape a pour objectif de produire des courbes de compromis "ressources (parallélisme) / nombre de cycles" pour les fonctions composant l'application à estimer, et ce pour tous les niveaux de granularité de la spécification. Nous commençons donc par présenter les techniques d'estimation pour les graphes feuilles (DFGs), puis les techniques utilisées pour l'estimation des structures de contrôle (CDFGs) et enfin les méthodes de combinaison utilisées pour l'estimation des HCDFGs.

4.1 Introduction

Le but de l'estimation intra-fonction est de produire une "base de données" d'architectures potentielles pour les différentes fonctions d'une application dans un but exploratoire. Le résultat de cette étape se présente sous la forme de courbes de compromis "ressources (parallélisme) / nombre de cycles" qui reflètent les possibilités d'exploitation du parallélisme de chacune des fonctions composant l'application. Ces courbes sont produites pour tous les niveaux hiérarchiques des fonctions (fonctions et combinaison de fonctions (HCDFG)).

L'estimation d'une fonction comprend plusieurs aspects. Un premier aspect de la méthode d'estimation est que celle-ci vise à trouver le nombre minimum de ressources (de type traitement et accès mémoire) nécessaires à l'exécution d'une fonction et ce pour chacune des contraintes de temps de la courbe de compromis. Un second aspect concerne la technique utilisée pour effectuer l'estimation. Celle-ci est de type hiérarchique et ascendante ; elle commence par les graphes feuilles (DFG) et se poursuit ensuite par les graphes de contrôle (CDFG) et hiérarchiques (HCDFG). L'estimation des DFGs repose sur un ordonnanceur rapide de type *List-Scheduling*, celle des CDFGs et HCDFGs sur des règles de combinaisons de type exclusive (CDFG), parallèle et séquentielle (CDFG, HCDFG).

Comme vu précédemment, les fonctions à traiter ont été caractérisées. Cette caractérisation va permettre de guider l'ordonnanceur afin de le rendre plus efficace. La phase de caractérisation nous apporte deux informations : tout d'abord elle nous informe de l'orientation (TCM) des fonctions. Ensuite elle nous permet d'effectuer un classement des fonctions par ordre de criticité. En combinant ces deux informations nous en déduisons comment traiter les fonctions (ainsi pour une fonction orientée traitement on ordonnancera d'abord la partie traitement puis la partie mémoire) et dans quel ordre les ordonnancer (les fonctions les plus critiques étant traitées en premier).

4.2 Ordonnancement des DFGs

4.2.1 Algorithmes d'odonnancement

De nombreux algorithmes d'ordonnancement ont été développés dans le cadre de la synthèse d'architecture et du codesign. Des classifications de ceux-ci ont été proposées dans [67] et [97]. Il est ainsi possible de distinguer les algorithmes à ressources contraintes et ceux à temps contraint.

Ordonnancement ASAP et ALAP

Les algorithmes d'ordonnancement ASAP (*As Soon As Possible*) et ALAP (*As Late As Possible*) permettent de connaître les dates limites (dates au plus tôt et au plus tard) théoriques d'ordonnancement des nœuds du graphe. L'ordonnancement ASAP consiste à placer les nœuds (en commençant par les nœuds racines) dans la première période d'horloge disponible tout en tenant compte des dépendances de données issues de l'ordonnancement de leurs prédécesseurs. L'ordonnancement de type ALAP nécessite tout d'abord la définition d'une contrainte de temps. Celle-ci peut être soit le nombre de cycles (encore appelé pas de contrôle) nécessaire à l'ordonnancement ASAP (c.a.d le chemin critique) ou une contrainte fixée par ailleurs. Une fois la contrainte de temps fixée, l'algorithme consiste à placer les nœuds (en considérant d'abord les nœuds feuilles) dans les cycles (pas de contrôle), en commençant au plus tard, c.a.d celui de la contrainte de temps. Le placement des prédécesseurs des nœuds ordonnancés se fait de la même façon, c.a.d au plus tard, tout en tenant compte des dépendances de données.

Connaissant les dates ASAP et ALAP des nœuds il est possible de calculer leur mobilité "absolue" de la façon suivante :

$$\text{mobilité nœud i} = \text{date ALAP nœud i - date ASAP nœud i}$$

La mobilité "absolue" est une information qui est souvent utilisée comme paramètre de classement des nœuds dans les algorithmes d'ordonnancement basés sur des listes. Notons que cette définition correspond à la mobilité "avant ordonnancement". En cours d'ordonnancement un deuxième type de mobilité est à considérer :

il s'agit d'une mobilité relative. Celle-ci peut être par exemple la différence entre le cycle (pas de contrôle) "courant" et la date ALAP des nœuds.

List-Scheduling

L'algorithme d'ordonnancement par liste, [98], est une extension de l'ordonnancement de type ASAP. Ce type d'algorithme utilise une liste dans laquelle sont classés les nœuds. Il existe plusieurs variantes de ce type d'algorithme. La liste peut être soit statique (construite une fois pour toute au début) soit dynamique (mise à jour à chaque nouveau cycle (pas de contrôle)). Généralement la création de la liste consiste à effectuer un classement des nœuds en fonction de leur date ASAP et de leur mobilité "absolue". Dans le cas des algorithmes d'ordonnancement à liste dynamique, une fois le processus d'ordonnancement lancé, la liste des nœuds est gérée dynamiquement, c.a.d que celle-ci est mise à jour en fonction des nœuds ordonnançables au prochain cycle (pas de contrôle). L'ordonnançabilité d'un nœud peut dépendre de plusieurs critères dont : sa priorité (fonction de la mobilité relative), données à consommer disponibles, ressources disponibles (cas d'un ordonnancement à ressources contraintes), ...

L'avantage de l'algorithme d'ordonnancement par liste est sa complexité relativement faible ($O(NC)$, avec N le nombre de nœuds et C le nombre de cycles (pas de contrôle)).

Force Directed Scheduling

La méthode heuristique *force directed scheduling* [99] vise à minimiser le nombre de ressources sous contrainte de temps. Pour cela, l'algorithme cherche à distribuer uniformément les opérations de même type dans les cycles (pas de contrôle). Le calcul des dates ASAP et ALAP permet de déterminer les bornes des dates d'ordonnancement de chaque opération. On suppose que chaque opération o_i possède une probabilité d'ordonnancement uniforme à l'intérieur de l'encadrement calculé précédemment et une probabilité nulle en dehors. Ainsi pour un pas de contrôle s_j tel que $E_i \leq j \leq L_i$ (où E_i et L_i représentent les dates au plus tôt et au plus tard), la probabilité que l' opération o_i soit ordonnancée dans l'état s_j vaut $p_j(o_i) = 1/(L_i - E_i + 1)$.

Le cœur de la méthode consiste, à partir d'un ordonnancement O_k, à déplacer des opérations dans des cycles (pas de contrôle) différents, aboutissant à un ordonnancement O_{k+1}. Il s'agit ensuite de choisir les mouvements qui minimisent le coût de la solution $k+1$. A chaque opération correspond une barre qui représente le coût en opérateurs (EOC) pour chaque état. Le coût attendu pour l'état s_j de l'opération de type k est donné par $EOC_{j,k} = c_k * \sum_{i,s_j \in mrange(o_i)} p_j(o_i)$, où o_i est une opération de type k et c_k le coût de l'unité fonctionnelle réalisant l'opération de type k. Comme les unités fonctionnelles peuvent être partagées entre les états, la valeur maximale du coût en opérateurs (EOC) atteinte sur tous les états donne une mesure du coût d'implantation pour les opérations de ce type. Ce calcul du coût est répété pour tous les autres types d'opérations du graphe. La force (du nom de l'algorithme) représente la différence de coût entre deux ordonnancements. Celle-ci permet donc d'identifier l'ordonnancement le plus prometteur (force la plus négative).

L'algorithme *Force Directed Scheduling* permet l'obtention de bons résultats (mais pas forcément optimaux). Néanmoins, sa complexité ($O(N^2C)$, avec N le nombre de nœuds et C le nombre de cycles (pas de contrôle)) est trop importante par rapport aux graphes des applications visées.

Path Based Scheduling

Les algorithmes *Path Based Scheduling* [100] visent à minimiser le nombre de cycles (pas de contrôle) nécessaires à l'exécution du chemin critique d'une représentation de type CDFG (applications orientées contrôle). Chaque chemin d'exécution est extrait et ordonnancé séparément. Ensuite, l'algorithme regroupe les différents ordonnancements afin d'obtenir l'ordonnancement final. Différentes stratégies ont été envisagées, de manière générale l'objectif est d'aboutir à une machine d'états avec un nombre minimum d'états et un chemin critique minimum par état (pour maximiser la l'horloge de la machine d'état).

Recuit simulé

L'algorithme du recuit simulé (*simulated annealing*) [101] repose sur le procédé physique du recuit simulé. Le recuit simulé est utilisé de manière générale pour

la résolution heuristique de problèmes complexes de décision, notamment celui de l'ordonnancement. Ce procédé débute par une montée en température du matériau telle que les molécules le constituant acquièrent une grande liberté de mouvement. Le matériau est ensuite refroidi lentement, le mouvement des molécules décroît graduellement jusqu'à ce qu'elles atteignent une position fixe. Grâce au refroidissement lent, l'énergie du matériau est alors minimale. Dans le cadre de l'ordonnancement le but est de trouver un algorithme permettant de sortir des minimums locaux. L'analogie se fait sur les éléments suivants : l'énergie devient une fonction de coût, le mouvement des molécules devient le passage d'une solution à une autre (déplacement de l'ordonnancement de certaines opérations) et la température devient un paramètre T qui contrôle la probabilité d'acceptation du passage d'une solution vers une autre. Le principe de l'algorithme est le suivant : un mouvement conduisant d' une solution k à une solution k+1 à plus faible coût est toujours accepté quelque soit T. Par contre un «mauvais mouvement» correspondant à un accroissement du coût n'est pas forcément refusé. Il est accepté avec la probabilité : $e^{-\frac{\delta c}{T}}$ avec $\delta c = cot(k) - cot(k+1)$ et T diminuant. L'algorithme s'arrête lorsque T est proche de zéro ou lorsque aucune amélioration n'a été observée pendant un certain nombre d'itérations.

L'une des difficultés de cet algorithme est de régler les différents paramètres : critère d'arrêt, mise à jour de T, ...

Algorithmes génétiques

Les algorithmes évolutionnaires dont font partie les algorithmes génétiques [102] [103], utilisent des modèles de calculs imitant les mécanismes de la nature. Dans le cas des algorithmes génétiques ce mécanisme est l'évolution. L'idée de base est de faire *évoluer* itérativement les individus (solutions, représentées sous la forme de chromosomes) d'une population (groupe de solutions) afin de converger vers une population incluant une solution satisfaisante. L'évolution passe par la sélection des individus les plus robustes (c.a.d les mieux adaptés à leur environnement).

Le schéma générique d'un algorithme génétique est le suivant : une population initiale est créée (au hasard ou issue d'un autre algorithme), les individus sont en-

suite évalués afin d'en sélectionner une partie. Ces individus sont ensuite soumis à diverses opérations génétiques (mutations, crossover, etc...) afin de générer une nouvelle population qui est à son tour évaluée. Ce genre d'algorithme permet d'obtenir de bons résultats mais souffre de plusieurs handicaps. Tout d'abord il faut réussir à représenter les solutions sous la forme de chromosomes, ensuite il faut régler de nombreux problèmes tels que la définition de la fonction d'évaluation (coût), le choix de la procédure de sélection , le choix des opérations génétiques et leurs paramètres et enfin définir les critères d'arrêts de l'algorithme.

4.2.2 Principes de l'ordonnancement développé

Notre méthode a pour but d'explorer dynamiquement un vaste espace de conception, aussi il est indispensable que les estimations soient réalisées très rapidement. C'est pourquoi une heuristique d'ordonnancement de type *list-scheduling* a été choisie plutôt que des approches de type *force-directed scheduling* ou algorithme génétique. La phase d'ordonnancement des DFG est importante car elle constitue la brique de base de l'estimation. À partir des informations contenues dans le graphe et les règles de l'utilisateur (UAR), le but est de minimiser les besoins en ressources et en bande-passante et ce pour plusieurs contraintes de temps (exprimées en nombre de cycles). Le fait d'obtenir plusieurs courbes de compromis temps/ressources pour une même fonction, nous permettra par la suite de choisir celles qui donnent la meilleure combinaison pour l'application. Pour chaque fonction l'exploration se fait autour de sa contrainte nominale Tn qui est proportionnelle à la criticité de la fonction considérée. La gamme de contraintes de temps ΔT à explorer autour de Tn fait l'objet d'un compromis entre le nombre de solutions et la rapidité d'exploration. Le concepteur peut ainsi fixer lui-même la gamme ΔT à explorer et le "pas" entre deux contraintes successives. De plus, notre outil détecte automatiquement deux bornes d'exploration : d'une part le temps minimum nécessaire à l'exécution de la fonction (chemin critique), et d'autre part la contrainte de temps à partir de laquelle le nombre de ressources de chaque type est égal à 1, ou à partir de laquelle ce nombre ne diminue plus. Enfin, le nombre de ressources initiales de type i allouées au début

de chaque ordonnancement est donné par la borne minimum :

$$\text{Nb moyen ressource type i} = \frac{\text{Nb opération type i}}{T}$$

avec T la contrainte de temps en nombre de cycles.

Dans un ordonnancement de type *list-scheduling*, l'ordre d'ordonnancement est déterminant. Suivant l'orientation des fonctions il faut donc favoriser la bande-passante ou la partie traitement. Des outils tels que Monet (Mentor Graphics), B.C (Synopsys) ou Gaut [104] sont basés sur une approche traitements prioritaires, alors que Atomium [105] et Phideo [106] sont basés sur des approches mémoire prioritaire. C'est pourquoi nous avons opté pour la possibilité de procéder à l'ordonnancement des graphes en deux temps : la partie traitement puis la partie mémoire ou vice-versa. Notre méthode se décompose en trois algorithmes : traitement prioritaire, mémoire prioritaire et mixte. Le fait de traiter en premier lieu les nœuds traitements (fonction orientée traitement) ou les nœuds accès mémoire (fonction orientée transferts mémoires) va permettre de réduire au minimum les ressources nécessaires pour le premier type traité (qui est le plus critique) et ensuite d'imposer des contraintes pour l'ordonnancement du deuxième type. Le type "mixte" correspond à un ordonnancement où le traitement et les accès mémoire sont gérés en même temps sans distinction. La sélection de l'un de ces trois algorithmes est guidée par la métrique DRM. Pour des valeurs élevées de DRM la fonction est orientée transferts mémoire alors que pour des valeurs plus faibles la fonction est orientée traitement. Enfin, les aspects contrôles sont détaillés dans la partie 4.3.

Algorithme traitement prioritaire (*processing first*)

Cet algorithme est utilisé dans le cas où la fonction est orientée traitement. Pour chaque cycle le nombre moyen de ressources nécessaires est calculé et les nœuds de type traitement sont ordonnancés. À la fin de l'ordonnancement des traitements, les dates *ASAP* et *ALAP* des nœuds mémoire sont mises à jour en fonction des dates d'ordonnancement des nœuds traitement. Ensuite les nœuds mémoire sont ordonnancés, toujours en calculant le nombre moyen de ressources nécessaires. Dans le cas où il resterait des cycles non utilisés après l'ordonnancement du traitement, une technique originale dite de la "pioche" (cf. 4.2.3) est utilisée.

Algorithme mémoire prioritaire (*memory first*)

Cet algorithme est le pendant du précédent. Il est appliqué dans le cas où la fonction est orientée transferts mémoire. Ici aussi le nombre moyen de ressources nécessaires est calculé puis les nœuds de type mémoire sont ordonnancés. À la fin de l'ordonnancement mémoire les dates $ASAP$ et $ALAP$ des nœuds traitement sont mises à jours en fonction des dates d'ordonnancement des nœuds mémoire. Ensuite les nœuds de type traitement sont ordonnancés, toujours en calculant le nombre moyen de ressources nécessaires. Dans le cas où il resterait des cycles non utilisés après l'ordonnancement des accès mémoire la technique de la pioche (cf. 4.2.3) est utilisée si besoin est.

Algorithme mixte

L'algorithme "mixte" montre que les approches traditionnelles de synthèse d'architecture (ou d'estimation) pour la conception de systèmes et basées sur la séparation traitement/transfert mémoire ne sont pas forcément adaptées à tous les cas. En effet, il convient dans certains cas de considérer simultanément l'ordonnancement des traitements et des accès mémoires (cas où les contraintes transferts mémoire <-> traitements sont équilibrées). Ce troisième algorithme est utilisé dans le cas où l'on n'arrive pas à bien définir l'orientation de la fonction. On considère alors que les nœuds mémoire et traitement sont à traiter avec la même priorité. De la même manière que dans les deux autres cas, les nœuds sont ordonnancés en fonction de leur mobilité m tel que m = date $ALAP$ - date $ASAP$.

REMARQUE. — Comme nous le verrons par la suite, l'orientation mémoire/traitement d'un DFG peut varier en fonction de la contrainte de temps.

4.2.3 Amélioration de l'utilisation et de la distribution des ressources

Afin de minimiser le nombre de ressources nécessaires, plusieurs techniques ont été employées.

OARC *Online Average Resource Computation*

L'un des inconvénients du *list-scheduling* est le traitement linéaire de l'ordonnancement (c'est aussi ce qui lui permet d'être rapide, sa complexité étant $O(n)$). Si en raison de dépendances de données les ressources sont insuffisantes, de nouvelles ressources sont allouées en fin d'ordonnancement et celles-ci risquent d'être sous-exploitées. Pour minimiser la sur-allocation de ressources nous proposons de calculer le nombre moyen de ressources à chaque cycle. Le calcul du nombre moyen de ressources de type i à chaque cycle est évalué comme suit :

$$\text{Nb moyen ressource type i} = \frac{\text{Nb de nœuds restants utilisant ressource type i}}{\text{Nb de cycles restants}}$$

L'exemple de la figure 4.1 montre comment ce calcul permet d'économiser une ressource de type transfert mémoire dans le cas du calcul de la DCT présenté sur la figure 4.3.

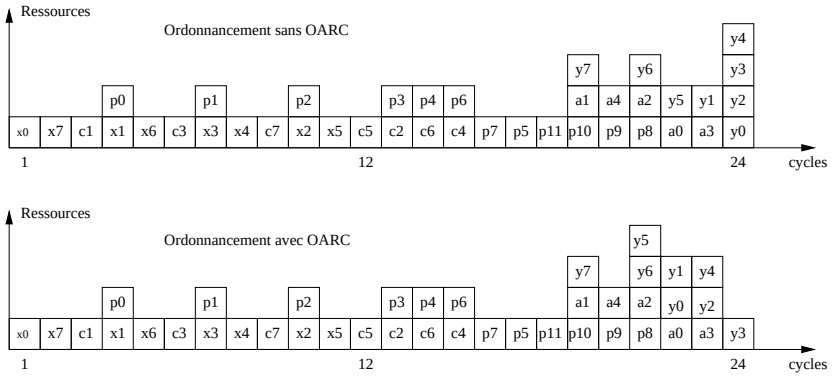

Fig. 4.1: *Exemple d'utilisation du calcul OARC. En l'appliquant dans le cas de la DCT, on peut économiser deux ressources de transfert mémoire. Dans le premier cas, le nombre moyen de ressources de type transfert mémoire calculé au départ est égal à 1 et en fin d'ordonnancement il est nécessaire d'ajouter 3 ressources supplémentaires (pour y2, y3 et y4) qui sont sous-utilisées. En calculant le nombre moyen de ressources à chaque cycle, 2 ressources de transfert mémoire sont allouées plus tôt pendant l'ordonnancement, elles sont ainsi mieux exploitées.*

Ce nombre moyen pouvant s'avérer trop faible, nous autorisons l'ajout de ressources en cours d'ordonnancement pour les nœuds dont la mobilité est arrivée à

zéro. Pour les nœuds dont la mobilité n'est pas encore nulle et pour lesquels le nombre de ressources est insuffisant nous appliquons la technique décrite ci-après.

La pioche

Cette technique s'applique aux algorithmes de type traitement prioritaire et mémoire prioritaire. En effet, après avoir ordonnancé soit les traitements soit les accès mémoire il peut rester des cycles en fin d'ordonnancement non utilisés (par rapport à la contrainte de temps fixée et au nombre de ressources allouées). Or, lors de l'ordonnancement respectivement des accès mémoire ou du traitement, il se peut que certains nœuds ne puissent être ordonnancés pour cause de ressources insuffisantes. Au lieu d'augmenter le nombre de ces ressources, il est préférable de "piocher" dans les cycles non utilisés afin de décaler, si leur mobilité le permet, les nœuds pour lesquels le nombre de ressources est insuffisant. Après l'ordonnancement des traitements (respectivement des accès mémoires), nous imposons que l'ordre ne soit pas modifié lors de l'ordonnancement des accès mémoire (respectivement des traitements). La seule variation possible est de décaler tout ou partie des nœuds ordonnancés en les déplaçant vers la "droite" en tirant profit de la liberté offerte par la pioche. La figure 4.2 illustre le fonctionnement de la technique pour l'exemple de la DCT (présenté sur la figure 4.3). Tout d'abord en A) les accès mémoires (x, c et y) sont ordonnancés et il reste 10 cycles disponibles dans la "pioche". Ensuite en B) les traitements sont ordonnancés sans utiliser la pioche ; 3 additionneurs (additions a0 à a2 en parallèle) et 4 opérateurs papillons (papillons p8 à p11) sont nécessaires. Enfin en C) l'ordonnancement des traitements est effectué en utilisant la pioche. Les nœuds mémoires (y0 - y6) étant mobiles, il est possible de les décaler (ici de 10 cycles), ainsi il ne faut plus que 1 additionneur et 1 opérateur papillon tout en respectant la contrainte de temps.

REMARQUE. — Étant donné que la méthode utilisée est heuristique, il se peut que des sur-estimations surviennent pour certaines contraintes de temps (mauvais choix lors de la gestion des dépendances). Cependant, le fait de produire des estimations dynamiques permet auxiliairement d'en éliminer un certain nombre. Ces cas surviennent lorsque la pente de la courbe de compromis temps/ressources devient

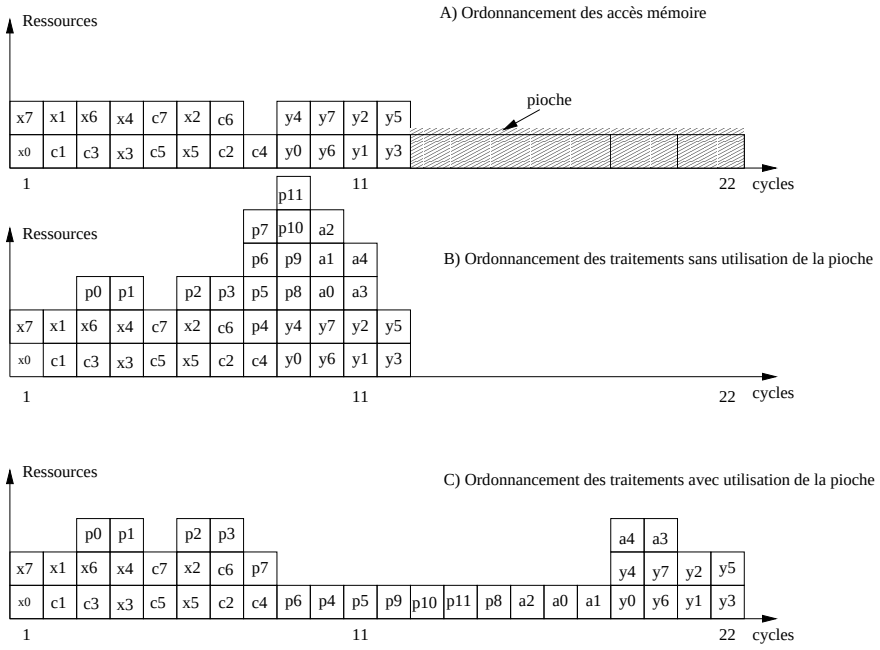

Fig. 4.2: *Illustration de la technique de la pioche sur l'exemple de la DCT avec un ordonnancement de type mémoire prioritaire pour une contrainte de 22 cycles.*

positive alors que la contrainte de temps augmente. Ces erreurs sont éliminées en conservant les résultats obtenus pour les contraintes de temps précédentes.

L'algorithme simplifié mémoire prioritaire est présenté en Alg.1. L'algorithme traitement prioritaire est son symétrique. L'algorithme simplifié mixte est présenté en Alg.2.

4.2.4 Stratégies d'ordonnancement

Choix du type d'algorithme d'ordonnancement

Le choix du type d'ordonnancement à appliquer peut se faire de plusieurs manières. La première option consiste en un choix manuel par le concepteur, qui après analyse des métriques MOM et COM décide d'appliquer un des types d'algorithmes (dans ce cas le même type est utilisé pour tous les sous-graphes de la fonction considérée). Une deuxième option permet l'automatisation du choix et l'application de différents types d'algorithmes pour les sous-graphes de la fonction. Dans ce cas le choix est guidé par la métrique DRM et la contrainte de temps courante. Une première approche consiste à diviser la gamme de valeurs que la métrique DRM peut

```
pour t = nombre de cycles minimum à nombre de cycles maximum, par pas utilisateur faire
  Trier les nœuds de la liste en fonction de leur date ASAP et de leur mobilité
  pour slot = 0 à t, pas + 1 faire
    Faire liste des nœuds ordonnançables
    Calcul OARC mémoire et ajout de ressource si besoin est
    pour tous les nœuds mémoires ordonnançables faire
      Chercher ressource disponible
      si ressource disponible alors
        Ordonnancer le nœud
      sinon si mobilité du nœud égale 0 alors
        Ajouter une ressource
      fin si
      Mettre à jour la liste des nœuds
    fin pour
  fin pour
  pour slot = 0 à t, pas + 1 faire
    Faire liste des nœuds ordonnançables
    Calcul OARC traitement et ajout de ressource si besoin est
    pour tous les nœuds traitements ordonnançables faire
      Chercher ressource disponible
      si ressource disponible alors
        Ordonnancer le nœud
      sinon si pioche = vrai alors
        Décaler ordonnancement mémoire
      sinon si mobilité du nœud égale 0 alors
        Ajouter une ressource
      fin si
      Mettre à jour la liste des nœuds
    fin pour
  fin pour
fin pour
```

Algorithme 1: *Algorithme simplifié pour l'ordonnancement mémoire prioritaire. L'algorithme traitement prioritaire est son symétrique.*

```
pour t = nombre de cycles minimum à nombre de cycles maximum, par pas utilisateur faire
  Trier les nœuds de la liste en fonction de leurs dates ASAP et de leurs mobilités
  pour slot = 0 à t, pas + 1 faire
    Faire liste des nœuds ordonnançables
    Calcul OARC mémoire et traitement, ajout de ressource si besoin est
    pour tous les nœuds ordonnançables, quel que soit leur type (mémoire ou traitement) faire
      Chercher ressource disponible
      si ressource disponible alors
        Ordonnancer le nœud
      sinon si mobilité du nœud égale 0 alors
        Ajouter une ressource
      fin si
      Mettre à jour la liste des nœuds
    fin pour
  fin pour
fin pour
```

Algorithme 2: *Algorithme simplifié pour l'ordonnancement mixte.*

prendre en trois :

si $drm > \frac{2}{3}$ l'algorithme mémoires prioritaires est utilisé, si $\frac{1}{3} < drm <= \frac{2}{3}$ l'algorithme mixte est appliqué et enfin si drm $<= \frac{1}{3}$ c'est l'algorithme traitements prioritaires qui est choisi.

Le réglage de ce paramètre reste accessible au concepteur.

Estimation en ligne de la taille mémoire locale

En ce qui concerne la taille mémoire locale le concepteur dispose de trois options : i) taille mémoire locale infinie, ii) taille mémoire locale pré-définie mais illimitée (c.a.d que l'algorithme d'ordonnancement l'augmente en fonction des besoins, comme pour les ressources de traitement et d'accès mémoire) et enfin taille mémoire locale pré-définie et limitée (si par exemple le concepteur effectue une projection logicielle et connaît cette taille). Lorsque les options i) et ii) sont sélectionnées l'outil Design Trotter mémorise la taille mémoire locale maximum utilisée et la fournie comme résultat pour le concepteur. Ces deux options permettent ainsi de réaliser une estimation de la taille mémoire locale nécessaire.

4.2.5 Comparaison des types d'algorithmes d'ordonnancement

Afin de comparer les différents types d'ordonnancement, nous les avons appliqués à plusieurs flots de données, notamment à celui de la transformée en cosinus discrète *DCT* (figure 4.3), pour différentes contraintes de temps. Les résultats (courbes de compromis temps/ressources) présentés sur les figures 4.4 à 4.6, permettent de comparer les quantités de ressources (de types traitement et transfert mémoire) nécessaires (avec une taille mémoire locale de 8 données) pour 4 algorithmes d'ordonnancement. En effet, en plus des 3 algorithmes (mémoire prioritaire, traitement prioritaire et mixte que nous avons développés), nous avons implanté l'algorithme *Force Directed Scheduling (FDS)* dans une version de type "mixte" afin "d'étalonner" notre approche.

La première observation que nous avons faite est que si celui-ci donne de bons résultats, c'est au prix d'un temps de calcul prohibitif dans le contexte de l'exploration de l'espace de conception au niveau système. En effet, alors qu'il n'a fallu

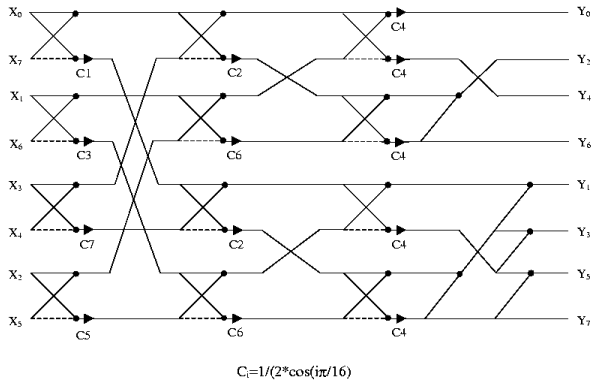

Fig. 4.3: *Algorithme de Lee pour le calcul de la DCT.*

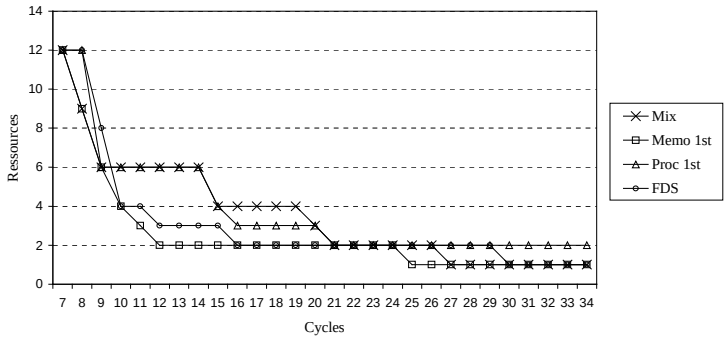

Fig. 4.4: *Courbe de compromis des ressources mémoires obtenue pour une taille mémoire locale de 8 données avec les 4 types d'ordonnancements.*

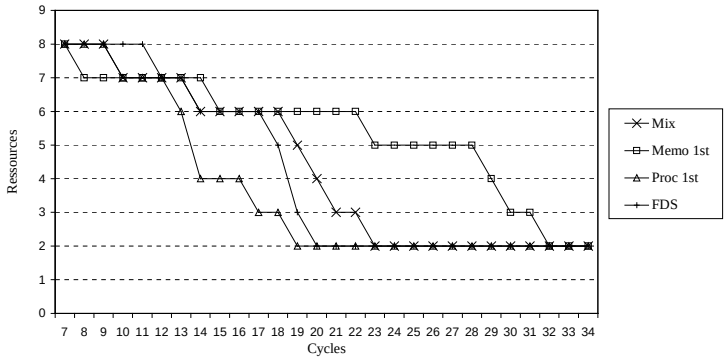

Fig. 4.5: *Courbe de compromis des ressources traitements obtenue pour une taille mémoire locale de 8 données avec les 4 types d'ordonnancements.*

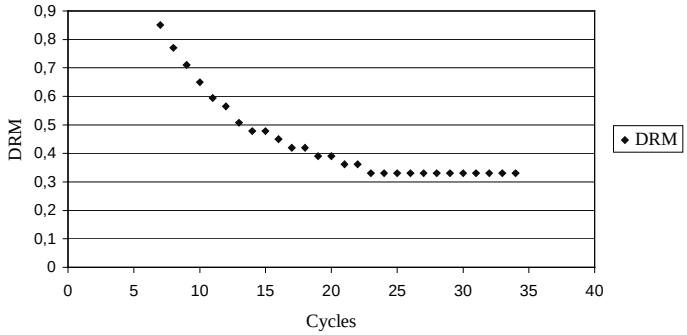

Fig. 4.6: *Évolution de la métrique DRM pour une taille mémoire locale de 8 données et pour différentes contraintes de temps.*

que 0.6 secondes [1] pour estimer 28 points (un point correspond à une solution) avec chacun des trois types d'algorithmes que nous avons présentés, il a fallu 50 minutes pour estimer ces 28 points avec l'algorithme *Force Directed Scheduling*. Étant donné sa lenteur nous considérons donc qu'il n'est pas adapté à l'exploration au niveau système.

En ce qui concerne les 3 autres types d'ordonnancement nous pouvons faire les remarques suivantes à partir des figures 4.4 et 4.5. Nous notons une première zone qui va du chemin critique à une contrainte de temps de 12 cycles. Dans cette zone pour lesquelles les contraintes de temps sont fortes c'est l'algorithme de type mémoire prioritaire qui donne les meilleurs résultats. Ensuite dans une zone allant de 13 à 16 cycles l'algorithme mémoire prioritaire donne encore de bons résultats, néanmoins si l'on considère à la fois les ressources de types traitement et transfert mémoire il devient plus difficile de choisir un des 3 types d'algorithmes. Dans ce genre de situation il est donc nécessaire que le concepteur puisse choisir de minimiser soit les ressources de type traitement soit celles de transfert mémoire (ces dernières étant en général plus coûteuses que celles de traitement). A partir d'une contrainte de 16 cycles c'est l'algorithme de type traitement prioritaire qui devient globalement le plus efficace. Cependant là encore si l'on considère uniquement l'un des types de ressources, le choix devient plus difficile. Enfin, pour des contraintes supérieures à 23 cycles, c'est l'algorithme de type mixte qui devient le plus efficace globalement. Pour des contraintes en temps et en mémoire fortes ce sont les algorithmes à deux

[1] Temps en Java interprété, réduit en Java compilé.

phases (traitement et mémoire prioritaire) qui donnent les meilleurs résultats, alors que pour des contraintes plus faibles c'est l'algorithme mixte. Ceci s'explique par le fait que les nœuds disposent d'une mobilité importante et que l'algorithme de type mixte permet justement de tirer profit de cette mobilité, là où les 2 autres types ajoutent des contraintes sur l'ordonnancement. Afin d'offrir le maximum de flexibilité au concepteur nous avons intégré plusieurs options pour le choix du type d'algorithme. Dans la première option c'est le concepteur qui choisit de privilégier tel ou tel type de ressource. La seconde option consiste à appliquer les 3 types d'algorithmes et à choisir la meilleure solution pour chaque contrainte de temps selon une fonction de coût pour chacun des types de ressources. Enfin, une troisième option permet d'utiliser la métrique DRM (*Data Reuse Metric*) pour choisir le type d'algorithme. À cet effet, l'utilisateur peut redéfinir des gammes de valeur de la métrique à associer aux types d'algorithmes. Par exemple, lorsque $DRM > 0.5$ (du chemin critique à une contrainte de 12) on applique l'algorithme mémoire prioritaire, pour $0.5 >= DRM > 0.333$ (contrainte de temps de 13 à 23) l'algorithme traitement prioritaire et pour $DRM <= 0.333$ (contrainte > 23) l'algorithme mixte. Nous tirons des conclusions à partir de cet exemple car d'une part il montre le type d'utilisation que l'on peut faire de l'estimation et d'autre part la structuration du DFG de la DCT est suffisamment complexe pour être un bon exemple de DFG "type". Il montre également l'importance que le type d'ordonnancement appliqué peut avoir sur les résultats. En effet, le cas de la DCT, qui est souvent vu comme une fonction orientée traitement peut en fait être orientée mémoire si la mémoire locale est trop petite. Cela peut conduire à plus de transferts de données entre les mémoires locale et globale (*spilling*).

4.3 Ordonnancement des aspects contrôles

La gestion du contrôle comprend plusieurs aspects que nous présentons ici. Ètant donné la complexité potentielle des applications à traiter il faut non seulement gérer les flots de données présents au sein des fonctions mais également, au niveau le plus élevé du système, le contrôle dit réactif. Il s'agit à ce niveau de traiter le

découpage et l'ordonnancement des tâches (périodiques et apériodiques). Cet aspect fait actuellement l'objet de recherches au sein de notre équipe [91] et n'est pas détaillé ici. Notre étude porte sur le contrôle au sein des fonctions (représentées sous forme de HCDFGs).

En premier lieu il faut faire la différence entre, d'une part, le contrôle dynamique (les tests non résolubles au moment de la compilation ou de la synthèse [107]) pour lequel il faut faire appel à des solutions soit matérielles (utilisation d'algorithmes de type *path-based*) ou logicielles, et d'autre part le contrôle statique qui comprend les boucles à indices connus et les tests pouvant être éliminés.

4.3.1 Gestion du contrôle dynamique

Il s'agit de traiter les structures de type multi-branches *if-else et switch-case*. Nous avons distingué deux cas : le cas de branches équilibrées (au sens équiprobables) et le cas de branches non-équilibrées. Les probabilités de passage dans les branches peuvent être obtenues par simulation (trace, *profiling*) de l'algorithme avec un jeu de données représentatif de l'application. Dans le cas où il n'est pas possible d'obtenir ces probabilités, on considère que les branches sont équiprobables. La gestion du contrôle consiste ici à savoir comment allouer les ressources pour l'exécution des branches et comment ordonnancer ces branches. Comme pour les autres aspects du flot d'estimation nous avons cherché à minimiser le nombre de ressources tout en proposant au concepteur un nombre suffisant de solutions lui permettant d'explorer l'espace de conception

Branches équiprobables

La méthode proposée pour l'estimation des branches équiprobable est la suivante :

1. Calcul des valeurs moyennes en ressources pour chaque branche et chaque type de ressource (y compris la taille mémoire locale dans le cas où elle n'a pas été fixée par l'utilisateur).
2. Recherche des valeurs moyennes MAX (entre toutes les branches) pour chaque type de ressources.

3. Ordonnancement de la branche la plus critique (criticité évaluée avec la métrique γ).
4. Ordonnancement des autres branches par ordre de criticité décroissante. Dans le cas où le nombre de ressources n'est pas suffisant celui-ci peut être augmenté via les techniques déjà utilisées pour l'ordonnancement du traitement et de la mémoire.

La méthode est résumée par l'algo.3.

si (nœud composé = if-else ou switch-case) ET (branches équiprobables) **alors**
calculer les valeurs moyennes pour chaque type de ressources pour toutes les branches
Estimer la taille mémoire locale dans le cas où l'utilisateur ne l'a pas définie
Rechercher le max des valeurs moyennes entre les branches
Ordonnancer la branche la plus critique (avec l'aide de la métrique γ)
Ordonnancer les autres branches en ré-utilisant les ressources déjà allouées et éventuellement en en ajoutant
fin si

Algorithme 3: *Algorithme pour l'ordonnancement de branches équiprobables.*

Branches non-équiprobables

Le premier problème que l'on rencontre lors de l'estimation des branches non-équiprobables est de définir le seuil de "déséquilibre" entre les branches, ceci est laissé au soin de l'utilisateur. Nous considérons ici que deux branches a et b sont non-équiprobables si Pa est très inférieure à Pb. Nous rappelons également que l'ordonnancement est contrait par le temps. La méthode proposée dans le cas de branches non-équiprobables est la suivante :

1. Commencer par la branche la plus probable (ordonnancement classique en T cycles).
2. Traiter les autres branches par probabilité décroissante (quelle que soit leur criticité). Considérons le cas de deux branches, la deuxième branche est ordonnancée avec les ressources de la première. Des ressources peuvent êtres éventuellement allouées si leur type n'est pas présent dans la première branche. Le résultat est un ordonnancement à ressources contraintes en un temps T'.

Deux cas se présentent alors : *i*) si $T' \leq T$ alors la contrainte de temps est respectée, *ii*) si $T' > T$ il faut donner plus de temps à la deuxième branche (afin de ne pas augmenter les types de ressources qui sont spécifiques à cette branche car sa probabilité de passage est faible, donc ses ressources risquent d'être sous-exploitées).

Pour donner plus de temps à cette branche l'idée originale est de gagner des cycles lors de chaque passage fréquent dans la première branche afin d'en gagner suffisamment pour pouvoir les allouer à la seconde branche le cas échéant. L'objectif est de respecter la contrainte de temps au niveau CDFG. La question est donc : combien de cycles faut-il imposer sur la première branche ? il faut gagner ΔT cycles $= T' - T$ sur le nombre de passages dans la première branche. Prenons $P1, P2$ probabilités de passage respectivement dans la première et la deuxième branche. Le nombre de passages dans la première branche est alors $P1 * 100$. Le nombre de cycles $\Delta T * P1 * 100$ permet d'exécuter $P2 * 100$ fois la seconde branche sans violer la contrainte de temps. L'ordonnancement de la première branche doit donc s'effectuer en

$$T\text{"} = T - \lceil \frac{\Delta T}{P1/P2} \rceil$$

3. Enfin, si le temps ΔT ne peut être gagné alors il n'y a pas de solution respectant globalement la contrainte T.

La méthode est résumée dans l'algorithme 4.

Afin d'illustrer ce concept voici un exemple (alg.5) issu d'une de nos applications tests (ICAM, cf. 5.3) :

Les probabilités de passage dans les deux branches ont été obtenues par *profiling*. La branche prise quand la condition est vraie est appelée a, l'autre b. Les probabilités de passage dans les branches a et b sont $Pa = 0.984$ et $Pb = 0.016$ respectivement. Comme $Pa \ll Pb$, la méthode ordonnance la branche a avec une contrainte de temps $Tc = 7$ cycles. Ensuite elle ordonnance la branche b, mais celle-ci requiert 14 cycles pour être ordonnancée. Pour "donner" plus de temps à la branche b la branche a est re-ordonnancée avec un nouvelle contrainte de temps donnée par :

si (nœud composé = if-else ou switch-case) ET (branches non-équiprobables) **alors**
Calculer les valeurs moyennes pour chaque type de ressources de la branche la plus probable
Estimer la taille mémoire locale dans le cas où l'utilisateur ne l'a pas définie
Ordonnancer la branche la plus probable en un temps T
pour toutes les autres branches par probabilité décroissante **faire**
Ordonnancer la branche i + 1 en un temps Ti en ré-utilisant les ressources déjà allouées et éventuellement en ajouter
si Ti>T **alors**
Ré-ordonnancer la branche la plus probable en $T" = T - \lceil \frac{\Delta T}{P1/P2} \rceil$
si les ressources maintenant nécessaires à la branche la plus probable sont inférieures aux ressources nécessaires à la branche i **alors**
Garder cette solution
sinon
Garder la première solution
fin si
sinon
i++
fin si
fin pour
fin si

Algorithme 4: *Algorithme pour l'ordonnancement de branches non-équiprobables.*

```
pour y=0 à nombre de lignes (nL) faire
  pour x=0 à nombre de colonnes (nC) faire
    ...
    si ((CharLabel[xyOffset-1]==0)and(CharLabel[xyOffset-nC]==0)) alors
      CharLabel[xyOffset]=newLabel;
      newLabel=newLabel+5;
    sinon
      CharLabel[xyOffset]=TabEqui[CharLabel[xyOffset-1]];
      TabEqui[CharLabel[xyOffset-Image.nC]]=CharLabel[xyOffset];
    fin si
    ...
  fin pour
fin pour
```

Algorithme 5: *Extrait d'une structure de contrôle de la fonction ic_etiquet de l'application ICAM.*

$T" = 7 - \lceil \frac{14-7}{\frac{0.984}{0.016}} \rceil = 6$. Dans ce cas il est possible de "sauver" $N \times 0.984 \times 6$ cycles qui peuvent être utilisés pour compenser les $N \times 0.016 \times (14-7)$ cycles (avec N le nombre d'itérations). Comme il est possible d'ordonnancer la branche a en 6 cycles cette solution est choisie.

Implantation

Une fois que les différentes branches ont été ordonnancées en suivant les méthodes précédemment décrites il reste à estimer la structure complète. Cette étape est réalisée par combinaisons des résultats d'estimation des graphes des branches puis du résultat de cette combinaison avec le graphe d'évaluation représentant le test. Deux options sont disponibles : le pire cas et le cas "moyen". L'option pire cas consiste à prendre le maximum entre les ressources des différents graphes alors que l'option "moyenne" consiste à multiplier les nombres de ressources nécessaires par les probabilités de passage dans les branches.

La méthode utilisée pour combiner les branches des structures de contrôle de type IF est résumée en alg.6. La figure 4.7 présente les principaux cas rencontrés : chevauchement temporel nul (cas 1), partiel (cas 2) et recouvrant (cas 3).

4.3.2 Gestion des boucles

Nous traitons ici les structures mono-branche *for, while et do-while.*

Déroulage et dépendances

L'intérêt du déroulage de boucles est qu'il permet d'augmenter le parallélisme potentiel et donc de réduire le chemin critique. La contre-partie est l'augmentation du nombre d'instructions (cas du logiciel) ou du nombre d'états (cas du matériel) nécessaires à l'exécution de la boucle une fois déroulée. Il y a donc un compromis à trouver. Outre la disponibilité de ressources, il y a une autre condition pour pouvoir dérouler une boucle : il faut que les dépendances de données puissent le permettre (ce sont elles aussi qui limitent le facteur de déroulage). Nous considérons les dépendances vraies (ou de flot) : dépendances de données inter-itérations, par exemple si le calcul de A[i] nécessite le résultat de A[i-4].

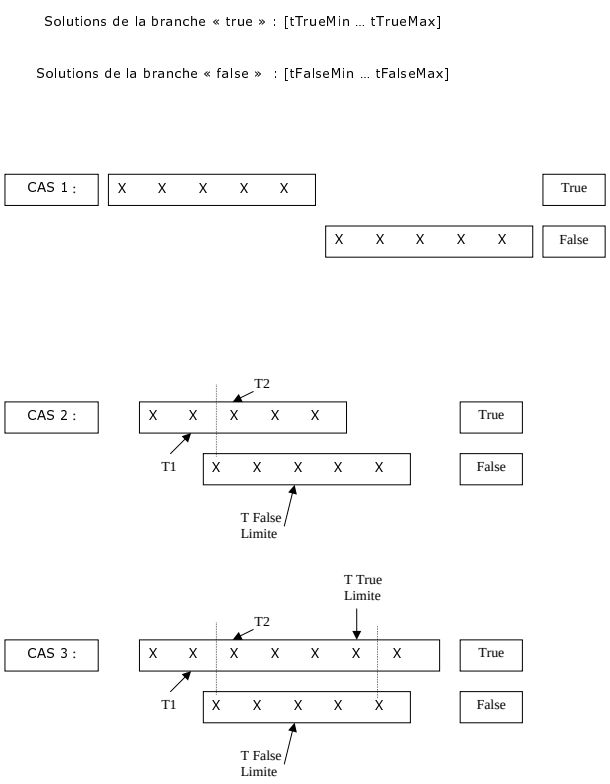

Fig. 4.7: *Cas rencontrés lors de la combinaison de branches d'une structure de contrôle de type IF. Les "x" représentent les solutions pour chacune des branches. Cette figure est associée à l'algorithme 6.*

Dépendances vraies

L'analyse et la résolution des dépendances vraies inter-itérations peuvent s'avérer complexes. Les recherches sur ce problème se concentrent habituellement sur le facteur de déroulage α nécessaire pour obtenir le débit optimal. L'originalité de notre approche consiste à chercher le facteur α minimum qui permette de satisfaire la contrainte de temps, tout en minimisant le parallélisme, c.a.d en limitant le nombre de ressources nécessaires.

Dans [108] il est montré que le facteur α permettant d'obtenir le débit optimal peut être obtenu par :

$$\alpha_{mgpr} = \lceil \frac{T_{max} PGCD(\Delta_{cr}, T_{cr})}{T_{cr}} \rceil \frac{\Delta_{cr}}{PGCD(\Delta_{cr}, T_{cr})}$$

avec T_{max} la latence de l'opération la plus lente, Δ_{cr} le nombre de délais (repérés par les indices de tableaux dans le HCDFG) et T_{cr} la latence cumulée du cycle critique. Dans notre cas, le problème est différent, nous devons trouver le facteur

```
tTrueMax : temps max de la branche "vraie"
tTrueMin :temps min de la branche "vraie"
tFalseMax : temps max de la branche "faux"
tFalseMax : temps max de la branche "faux"
si tTrueMax<tFalseMin alors
  Utiliser méthode 1
fin si
si tTrueMin<tFalseMin<tTrueMax et tTrueMax<tFalseMax alors
  Utiliser méthode 2
fin si
si tTrueMin<tFalseMin et tFalseMax<tTrueMax alors
  Utiliser méthode 3
fin si
*** METHODE 1 ***
pour t=tFalseMin à tFalseMax faire
  Combiner solutionFalse(t) et solutionTrue(tTrueMax)
fin pour
*** METHODE 2 ***
Combiner solutionFalse(tFalseMin) et solutionTrue(t1)
pour tFalse = tFalseMin à tLimite faire
  pour tTrue = t2 à tTrueMax faire
    combiner solutionTrue(tTrue) et solutionsFalse(tFalse)
    si tFalse > tTrue alors
      tCombinaison = tFalse
      tFalse = temps du prochain point False
    sinon
      tCombinaison = tTrue
      tTrue = temps du prochain point True
    fin si
  fin pour
fin pour
pour tFalse = temps du point suivant de tLimite à tFalseMax faire
  combiner solutionFalse(tFalse) et solutionTrue(tTrueMax)
fin pour
*** METHODE 3 ***
Combiner solutionFalse(tFalseMin) et solutionTrue(t1)
pour tTrue = t2 à tLimite faire
  pour tFalse = tFalseMin à tFalseMax faire
    combiner solutionTrue(tTrue) et solutionsFalse(tFalse)
    si tFalse > tTrue alors
      tCombinaison = tFalse
      tFalse = temps du prochain point False
    sinon
      tCombinaison = tTrue
      tTrue = temps du prochain point True
    fin si
  fin pour
fin pour
pour tTrue = temps du point suivant de tLimite à tTrueMax faire
  combiner solutionTrue(tTrue) et solutionFalse(tFalseMax)
fin pour
```

Algorithme 6: *Algorithme pour la combinaison des branches dans les structures de contrôle de type IF. Cet algorithme est associé à la figure 4.7.*

de déroulage minimum pour ordonnancer une boucle avec une contrainte de temps T. Il peut-être reformulé comme suit : si une boucle nécessite T" cycles pour être ordonnancée sans déroulage, nous devons trouver le facteur de déroulage permettant de gagner $G = T" - T$ cycles. On peut montrer que le nombre de cycles obtenus après un déroulage avec un facteur α est :

$$T" = \frac{T + (\alpha - 1)d_{min}}{\alpha}$$

avec

$$d_{min} = \lceil max_{cycles}(\frac{T_{cr}}{\Delta_{cr}}) \rceil$$

Finalement, le facteur de déroulage α pour gagner G cycles est donné par :

$$\alpha = \frac{1}{1 - \frac{G}{T - d_{min}}}$$

Dépendances de mémoire

Il s'agit d'un cas particulier de dépendances qu'il est facile de "casser" car elles ne sont dues qu'au style d'écriture de la spécification, comme illustré par l'exemple suivant (algorithme 7).

y(k) = 0
pour i = 0 à N - 1 **faire**
  y(k) = y(k) + a_ix(k - i)
fin pour
	peut s'écrire :	y1(k) = 0
		y2(k) = 0
		pour i = 0 à $\frac{N}{2} - 1$ **faire**
		  y1(k) = y1(k) + a_ix(k - i)
		  y2(k) = y2(k) + $a_{i+\frac{N}{2}}$x(k - (i + $\frac{N}{2}$))
		fin pour
		y(k) = y1(k) + y2(k)

Algorithme 7: *Exemple d'élimination de dépendances de mémoire.*

Cette transformation correspond à une mise en arbre [109]. Le problème consiste à trouver la décomposition minimum en fonction de la contrainte de temps. Soit une boucle dont le chemin critique sans déroulage CC = N. Le nouveau chemin critique CC_L de cette boucle pour un déroulage de L est donné par :

$$CC_L = \lceil \frac{N}{L} \rceil + \lceil Log_2(L) \rceil$$

Log_2 venant des additions supplémentaires, comme illustré sur la figure 4.8.

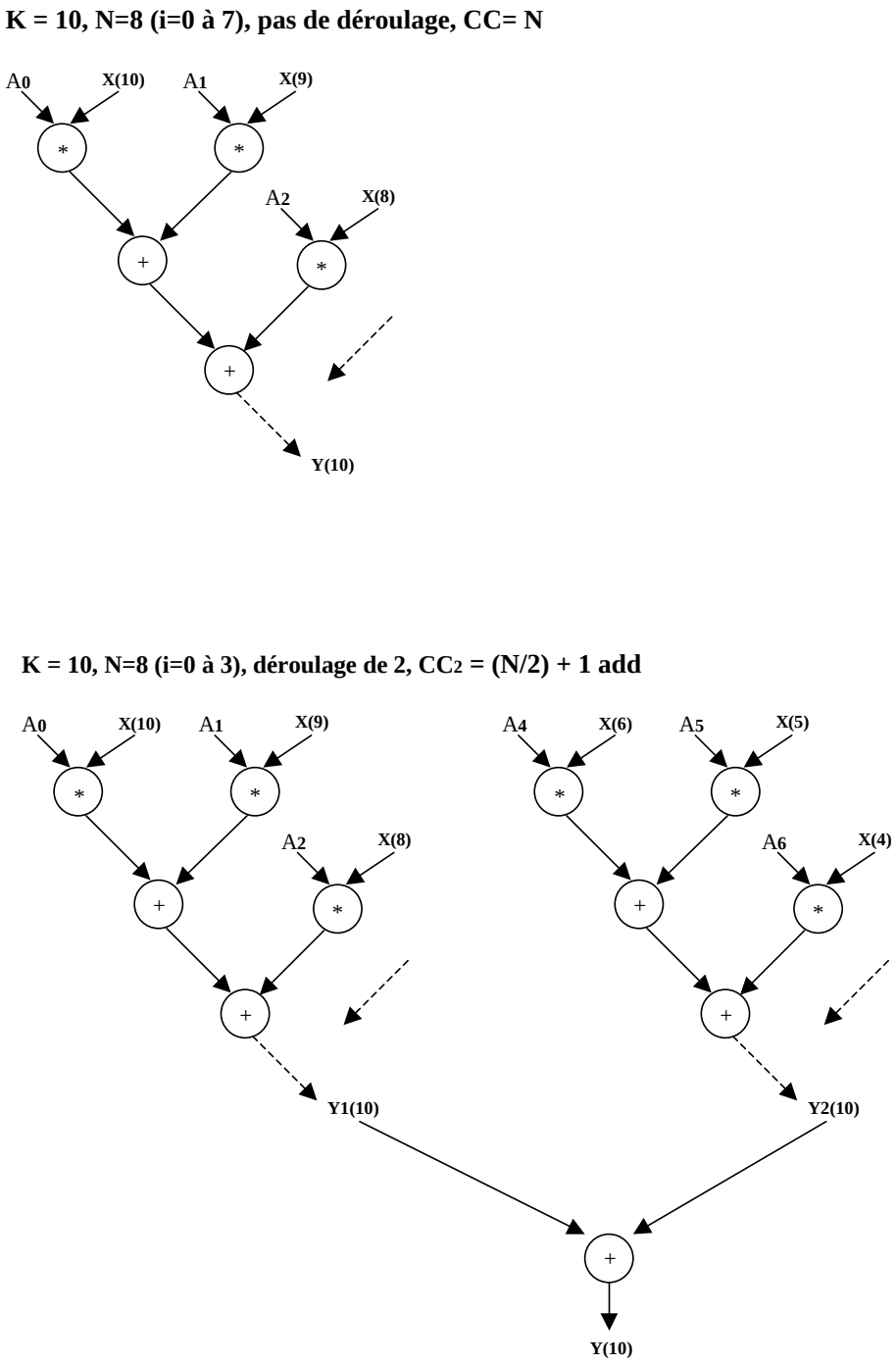

Fig. 4.8: *Mise sous la forme d'arbre d'une boucle dans le cas de dépendances de flot.*

4.3.3 Implantations et stratégie

Dans ce paragraphe nous expliquons comment les éléments précédents ont été implantés dans l'outil Design Trotter et quelles sont les stratégies qui ont été choisies.

Le déroulage de boucle est lancé lorsque l'ordonnanceur n'arrive pas à trouver de solutions respectant la contrainte de temps pour une structure de type FOR. La stratégie suivie consiste en les étapes suivantes : i) détection et calcul des dépendances ii) calcul du décalage minimum entre itérations (décalage correspondant à dmin), iii) calcul du facteur de déroulage (α) iv) ré-ordonnancement et v) sélection de la solution la moins coûteuse.

- Détection des dépendances : il s'agit de chercher les tableaux qui sont à la fois en lecture et en écriture au sein du cœur de boucle. Les tableaux forment ainsi un ensemble de "paires" de lecture-écritures $\{W, R\}$.
- Calculs des dépendances : pour chacune de ces "paires" on cherche les valeurs min et max que peut prendre l'indice.
- Calcul de dmin : le calcul de dmin est concrètement réalisé en utilisant les dates ASAP et ALAP des nœuds mémoires, ainsi $T_{cr} = |ASAP(W) - ASAP(R)$ et $\delta_{cr} = indiceW - indiceR$
- calcul du facteur de déroulage : celui-ci correspond à celui défini en 4.3.2.
- Réordonnancement : une fois le facteur de déroulage α trouvé, nous effectuons $\alpha - 1$ duplications du graphe de base (qui peut être un simple DFG ou un graphe hiérarchique) pour constituer le nouveau graphe. Une fois le graphe de base dupliqué, et avant d'être inséré dans le nouveau graphe, les dates asap / alap de ces nœuds sont décalées de dmin comme illustré sur la figure 4.9.
- Enfin, le nouveau graphe est ré-ordonnancé.

Grâce au déroulage il y a donc une nouvelle partie de l'espace de conception qui est explorée : de la réalisation "parallèle déroulée" (dont le temps est égal à : nombre de cycles minimum du cœur de boucle + nombre de cycles minimum du graphe d'évolution + (2*nombre de cycles minimum du graphe d'évaluation)) à la réalisation "parallèle non déroulée" (premier point sur la courbe de compromis avant déroulage). De plus le réordonnancement peut s'effectuer pour plusieurs points de la courbe de solutions du cœur de boucle (la solution la moins coûteuse au final est

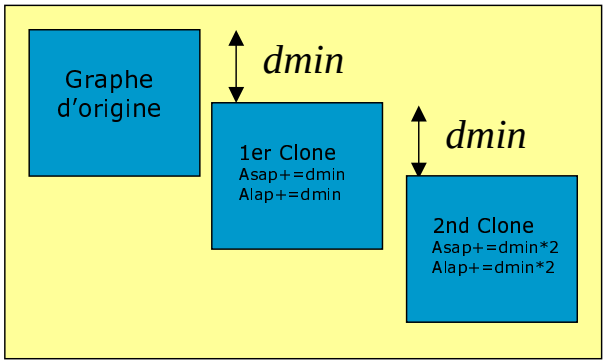

Fig. 4.9: *Déroulage d'un graphe avec un facteur $\alpha = 3$. Le graphe d'origine est dupliqué deux fois et les dates asap/alap de ces nœuds sont mises à jour en fonction de dmin.*

sélectionnée). Il est ainsi possible de paramétrer cette exploration par d'une part le nombre de solutions du cœur de boucle à utiliser et d'autre part le pas d'exploration entre la solution "parallèle déroulée" et la solution "parallèle non déroulée". Bien évidemment un pas faible et l'utilisation d'un nombre important de solutions du cœur de boucles se traduiront par un ralentissement certain du processus d'exploration mais permettront d'aboutir à de plus nombreuses solutions.

Limites

Les limites d'implantation du déroulage sont la forme que peuvent prendre les indices : nous nous limitons à la forme aI+b avec a et b constantes.

4.4 Combinaison de graphes

Une fois que les CDFGs ont été estimés individuellement (parties 4.2.2 et 4.3), il faut estimer les HCDFGs (c.a.d les fonctions). Ceci est réalisé en combinant les CDFGs qui s'exécutent séquentiellement et parallèlement (le cas des exclusions mutuelles a été décrit en 4.3.1). Comme dans le reste de la méthode, il s'agit ici de favoriser la réutilisation des ressources afin d'en minimiser le nombre. L'estimation est faite hiérarchiquement, elle commence par les DFGs, ensuite les CDFGs et finalement les HCDFGs. N'oublions pas que nous voulons obtenir les estimations rapidement afin d'évaluer un grand nombre de solutions, c'est pourquoi nous ne

pouvons pas appliquer une recherche exhaustive [93] des combinaisons à coût minimum des courbes de compromis temps/ressources. Au lieu de cela, nous utilisons une technique basée sur les points particuliers (PP) des courbes de compromis. Un point particulier correspond à une contrainte de temps pour laquelle le nombre de ressources diminue.

4.4.1 CDFGs séquentiels

La méthode pour estimer les CDFGs séquentiels est présentée sur la figure 4.10 où la courbe de compromis du graphe CDFG1-2 est construite en combinant les courbes des graphes CDFG1 et CDFG2.

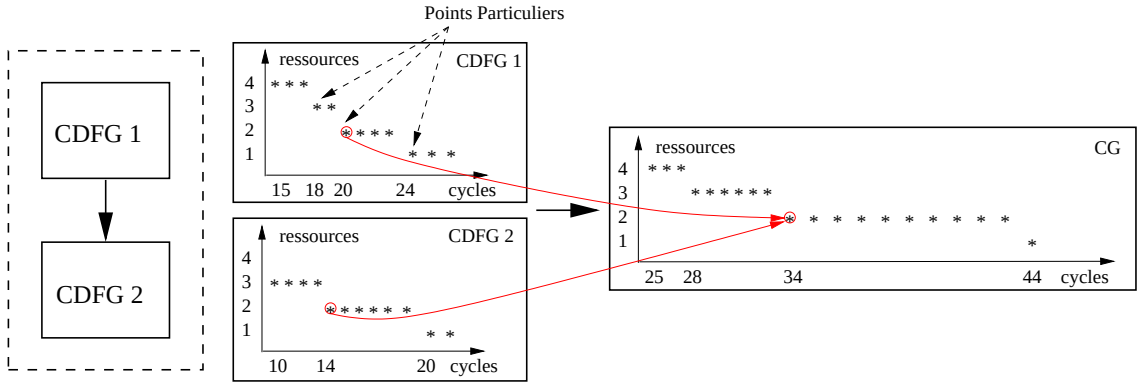

Fig. 4.10: *Combinaison de CDFGs séquentiels.*

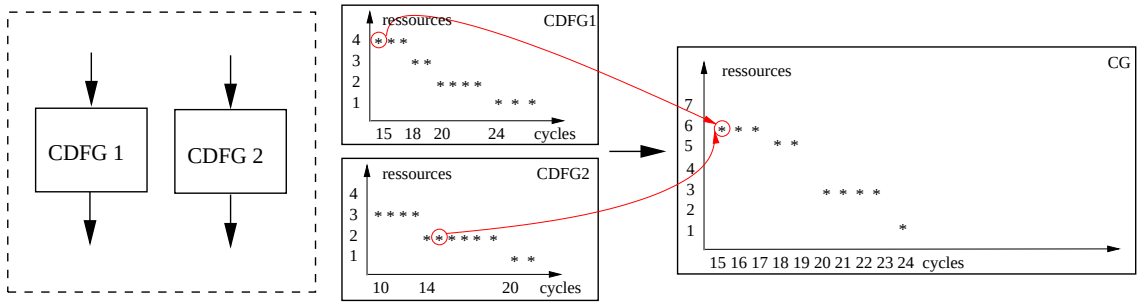

Fig. 4.11: *Combinaisons de CDFGs parallèles sans fusion de profils.*

Le temps minimum d'exécution de la séquence [CDFG1 puis CDFG2] est donné par $t_{min} = t_{min}(CDFG1) + t_{min}(CDFG2) = 25$. L'algorithme sélectionne d'abord

le CDFG dont le coût est le plus élevé en terme de ressources (ressources qui sont suffisantes pour être ré-utilisées par le deuxième CDFG). Ensuite le point particulier du graphe sélectionné est traité et le deuxième point du CDFG résultat est créé pour une contrainte de temps qui est dans cet exemple : $t_2 = t_{2^n dpp}(CDFG1) + t_{min}(CDFG2) = 18 + 10 = 28$. Ensuite, le processus est réitéré jusqu'à ce que la borne minimum (1 ressource ou bien un chiffre fixé par le concepteur dans le fichier UAR) soit atteinte. La méthode est résumée par l'algorithme 8, sa complexité est O(n) avec n le nombre de points particuliers.

4.4.2 CDFGs parallèles

La méthode pour estimer des CDFGs parallèles est présentée sur la figure 4.11. Dans le cas de CDFGs concurrents, un point important est que les CDFGs peuvent partager des ressources dans un cycle donné. Le nombre de cycles minimum est $t_{min} = MAX[PPG1[1]; PPG2[1]]$. Le principe de l'algorithme d'estimation est, pour une itération donnée, de sélectionner le CDFG avec le coût maximal et d'incrémenter sa contrainte de temps jusqu'au prochain point particulier. Là, une fusion des profils d'ordonnancement (un point de la courbe de compromis correspond à un ordonnancement particulier) des deux CDFGs est réalisée et la valeur maximum donne le nombre de ressources correspondant au point de la courbe de compromis du graphe résultat. Ainsi, l'algorithme passe des points particuliers d'un graphe à ceux de l'autre en sélectionnant le CDFG avec le coût le plus élevé. Suivant le compromis optimisation/vitesse de calcul choisi, la fusion se fait soit cycle par cycle, soit en suivant les valeurs maximums.

Remarque — Suite à la combinaison parallèle de deux graphes, ceux-ci sont également combinés séquentiellement. En effet, les deux graphes n'ayant pas forcément les même dates de "départ" leurs exécutions peuvent plus ou moins se chevaucher, le cas extrême étant que l'un des deux graphes commence après l'arrêt du premier, ce qui correspondant à une exécution séquentielle de ceux-ci.

La méthode est résumée par l'algorithme 9.

PPG1[1..maxIndex1] Points Particuliers du graphe 1
PPG1[1..maxIndex2] Points Particuliers du graphe 2
PPCG[1..maxIndexG] Points Particuliers du graphe combiné
PPCG[1]← PPG1[1]+PPG2[1]
indexCG← 1 index1← 1 index2← 1
t1← PPG1[1]
t2← PPG2[1]
tmax← PPG1[maxIndex1]+PPG1[maxIndex2]
t← PPCG[1]
tant que t≤Tmax **faire**
  indexCG++
  si G1[PPG1[index1]]>G2[PPG2[index2]] **alors**
    t1← PPG1[index1+1]
    index1++
  sinon si G1[PPG1[index1]]<G2[PPG2[index2]] **alors**
    t2← PPG2[index2+1]
    index2++
  sinon
    si G1[PPG1[index1+1]]>G2[PPG2[index2+1]] **alors**
      t1← PPG1[index1+1]
      index1++
    sinon
      t2← PPG2[index2+1]
      index2++
    fin si
  fin si
  CG[t1+t2]← $MAX_{j=1;2}[G_j$[PPGi][ij]]
  PPCG[indexCG← t1+t2]
fin tant que

Algorithme 8: *Algorithme de combinaison de graphes séquentiels.*

```
PPG1[1..maxIndex1] Points Particuliers du graphe 1
PPG2[1..maxIndex2] Points Particuliers du graphe 2
PPCG[1..maxIndexG] Points Particuliers du graphe combiné
Gi[k,1..k] profile du graphe i (de 1 à k cycles) pour la courbe de compromis de contrainte k
tmin← MAX[PPG1[1];PPG2[1]]
tmax← MAX[PPG1[maxIndex1];PPG1[maxIndex2]]
t←tmin
indexCG← 1 index1← 1 index2← 1
tant que t≤Tmax faire
  max← 0
  mobility,j ←extension d'index
  pour c=1 TO t faire
    CG[indexCG,c]← G1[t,c]+G2[t,c]
    si CG[indexCG,c]>max alors
      max← CG[indexCG,c]
    fin si
  fin pour
  CG[t]← max
  PPCG[indexCG]← t
  si PPG1[index1+1]<PPG2[index2+1] alors
    t← PPG1[index1+1]
    index1++
  sinon
    t← PPG2[index2+1]
    index2++
  fin si
  indexCG++
fin tant que
coût← CG[indexCG-1]
si coût>1 alors
  t← nombre d'opération
fin si
tant que coût>1 faire
  max← 0
  pour c=1 TO t faire
    CG[indexCG,c]← G1[t,c]+G2[t,c]
    si CG[indexCG,c]>max alors
      maw← CG[indexCG,c]
    fin si
  fin pour
  coût← max
  t++
fin tant que
CG[indexCG]← 1
PPGC[indexCG]← t
Combinaison série de graphe 1 et graphe 2

/****************************************/
/************Extension d'index************/
si ∃ PPG_k[indexk]<tmin avec k∈1,2 alors
  si ∃ j>indexk tel que PP_k[j]<tmin alors
    index_k ←j
  fin si
fin si
/****************************************/
```

Algorithme 9: *Algorithme de combinaison de CDFGs parallèles.*

4.4.3 Méthode de parcours pour les combinaisons

Dans les partie 4.4.1 et 4.4.2 les méthodes pour combiner les graphes deux à deux, soit de manière séquentielle soit de manière parallèle ont été présentées. Ce paragraphe décrit la méthode utilisée pour l'application des deux méthodes de combinaisons.

À titre d'exemple observons le cas présenté sur la figure 4.12. Les nœuds HCDFG2 et HCDFG3 sont combinés de manière parallèle pour former le nœud résultat HCDFG23, qui est ensuite lui-même combiné avec le nœud HCDFG4 pour former le nœud HCDFG234. Ensuite, ce sont les nœuds HCDFG234, HCDFG6 et HCDFG1, HCDFG5 qui sont traités. Une analyse des dépendances entre graphes est donc indispensable afin de déterminer le bon enchaînement des combinaisons.

L'algorithme 10 présente le principe d'application des combinaisons. L'idée consiste à combiner tous les prédécesseurs immédiats d'un nœud courant (nœuds pour lesquels il existe un arc aboutissant au nœud courant) de façon parallèle, puis à combiner séquentiellement le résultat avec le nœud courant, puis de réitérer jusqu'à ce que tous les nœuds aient été traités.

Toute fois certaines précautions doivent être prises lorsqu'on construit la liste des nœuds courants. Dans le cas de la figure 4.12 par exemple, les nœuds successeurs des nœuds HCDFG5, HCDFG6 sont les nœuds HCDFG8 et HCDFG7. Or il existe une dépendance entre ces deux nœuds qui introduit une erreur car il faut d'abord effectuer la combinaison séquentielle de HCDFG2346 et de HCDFG7. Le problème est résolu grâce à la fonction *Successeurs* qui renvoie la liste des nœuds constituant les successeurs immédiats de la liste des nœuds courants (nœuds pour lesquels il existe un arc issu de chaque nœud courant). S'il existe un arc reliant deux nœuds de la liste des successeurs, le nœud auquel aboutit cet arc est supprimé de la liste des successeurs.

4.5 Exploration hiérarchique

L'exploration de la fonction complète est réalisée de manière hiérarchique descendante et ascendante. En effet le parcours du graphe de la fonction complète

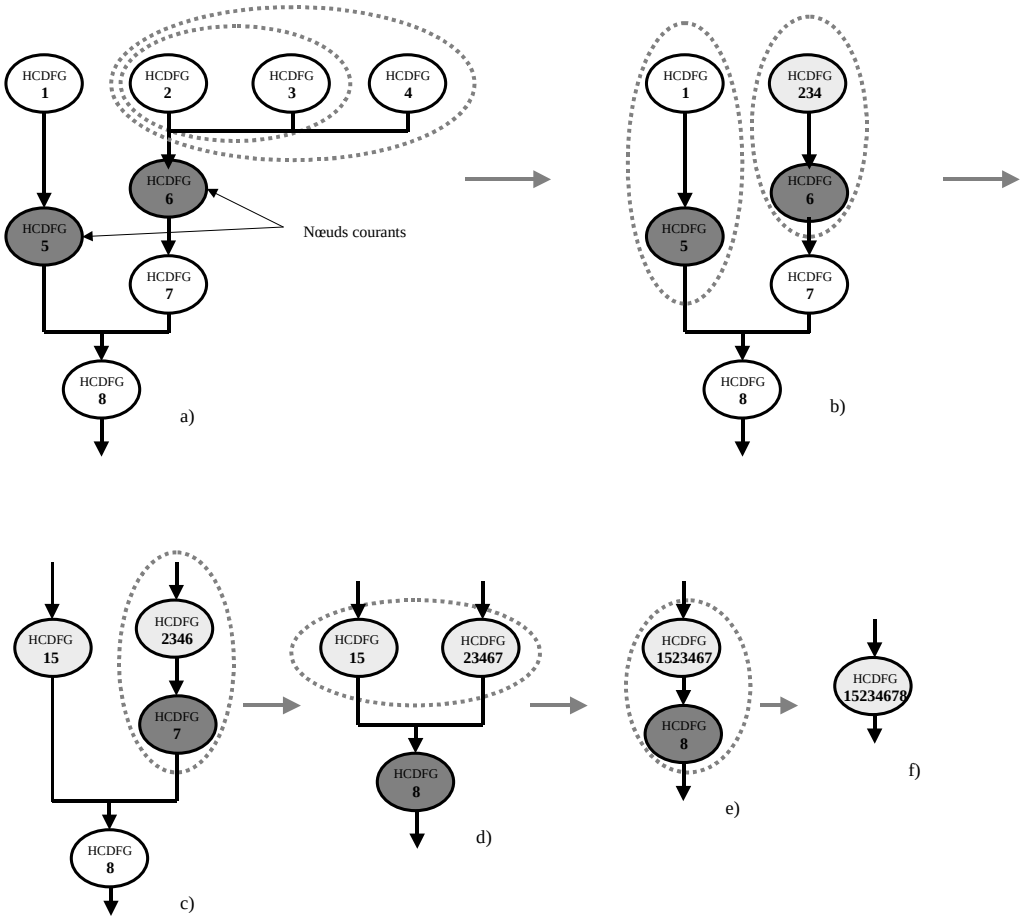

Fig. 4.12: *Application des combinaisons.*

commence par les nœuds racines de celui-ci, qui sont à priori des nœuds composites (c.a.d hiérarchiques). L'algorithme explore et estime chacun des sous-graphes correspondants. Le processus sera ensuite réitéré pour chacun des sous-graphes jusqu'à ce que le niveau feuille (DFG) soit atteint. Les algorithmes d'estimation utilisés sont alors le *list-scheduling* pour les DFGs (décrit en 4.2.2), et les méthodes décrites en 4.3 pour les CDFGs (structures de contrôles). Une fois que les graphes de plus bas niveaux ont été estimés, l'algorithme "remonte" les résultats d'estimations au niveau supérieur dans un nœud de stockage (*resultNode*). Lors de la phase ascendante du parcours, les graphes s'exécutant séquentiellement ou parallèlement sont combinés grâce aux méthodes décrites en 4.4.

L'algorithme 11 décrit la manière dont un graphe est exploré et estimé.

Si un nœud composite n'est composé que de nœuds résultats, on peut alors réaliser l'estimation du nœud composite à partir des algorithmes de combinaison définis

```
combinaisons(graph Gcourant)
DEBUT
Initialiser la liste des nœuds courants (LNC) avec les nœuds racines
tant que LNC ≠ ∅ faire
  LNC=Successeurs(LNC)
  pour chaque nœud ∈ LNC faire
    combinaison Parallèle des prédécesseurs 2 à 2
    combinaison Séquentielle du nœud courant et du nœud précédent
  fin pour
fin tant que
```

Algorithme 10: *Algorithme de combinaison hiérarchique.*

précédemment. La condition d'arrêt de parcours du graphe est validée lorsque chaque nœud composite possède un nœud résultat. Il existe alors un résultat d'estimation pour chaque nœud hiérarchique, y compris celui qui contient toute l'application.

4.6 Vers une estimation logicielle

Afin de procéder au prépartitionnement une étape d'estimation logicielle à été imaginée et commencée à être développée, notamment dans le cadre d'un séjour de recherche à l'Université d'Aalborg au Danemark. L'estimation logicielle a pour but d'évaluer plus précisément le temps d'exécution de fonctions de l'application que l'on pense devoir s'exécuter de manière logicielle. Il existe diverses méthodes pour l'estimation de performances sur processeurs programmables. Beaucoup de ces méthodes se situent dans le contexte des compilateurs reciblables. Mimola [110] est un environnement permettant le développement et l'évaluation de processeurs. Il utilise le langage de description Mimola. La description se fait à un niveau d'abstraction faible et conduit donc à des descriptions complexes et des temps de simulation très longs. Chess/checkers [111] est issu de travaux menés à l'IMEC. Il comprend entre autre un compilateur reciblable (Chess) et un simulateur du jeu d'instructions du processeur (checkers). La description du processeur se fait avec le langage nML [112]. nML permet la description structurelle et comportementale au niveau RTL. Dans

```
Explorer(graph Gcourant)
DEBUT
NbComposite=0
pour chaque nœud de Gcourant faire
  si le nœud courant est un nœud hiérarchique (composite) alors
    NbComposite=NbComposite+1
    si le composite courant n'est pas un nœudRésultat alors
      Gtemp= le sous-graphe du composite
      Explorer(Gtemp) //appel récursif
    fin si
  fin si
fin pour

TypePère = le type du père de Gcourant
si TypePère = structure de contrôle (CDFG) alors
  Appliquer méthode adéquate de combinaison des éléments de la structure de
  contrôle
sinon si NbComposite > 0 alors
  Appliquer combinaison séquentielle/parallèle
sinon
  Appliquer ordonnancement de la structure de contrôle (CDFG) ou du graphe
  feuille(DFG)
fin si
```

Algorithme 11: *Algorithme d'exploration hiérarchique.*

[113] une méthode pour l'estimation de performances basée sur le langage ISDL est présentée. La méthode consiste en un compilateur reciblable et un simulateur de jeu d'instructions généré automatiquement par l'outil. Dans [114] et [115] une méthode d'estimation reciblable est également proposée. Nous pouvons également citer l'environnement CALIFE [116] qui utilise le langage Armor. Ce langage nous apparait à priori bien adapté à notre problème, vu d'une part ses caractéristiques techniques (niveau d'abstraction, ...) et d'autre part sa disponibilité aisée.

4.6.1 Le langage Armor

Le processeur est décrit en utilisant le langage "Armor" [42]. Ce langage permet de décrire un processeur via son jeu d'instructions. Dans une description en langage Armor, des "composants" assimilables à des instructions sont définis et associés en groupes parallèles afin d'exprimer le parallélisme au niveau instruction. La règle "racine" appelée *InstructionSet* définit toutes les alternatives possibles dans le jeu d'instructions. La hiérarchie est exprimée au moyen des règles *gp* (group). De plus, les contraintes relatives au parallélisme peuvent être définies grâce à la règle *restriction*. Le comportement de chaque instruction est défini avec les règles *df* (data-flow) et *ctr* (control-flow). Celles-ci représentent les transferts registres et les opérations de traitement. Les opérations de traitement utilisent les opérateurs qui sont définis avec les règles *op* (operator), *class* (class of registers), et *mode* (operands). Les ressources sont décrites avec les règles *reg*, *regFile*, et *mem* pour les aspects mémoires et la règle *fu* pour les unités fonctionnelles. Les "composants" (instructions) sont définis individuellement en termes de sémantique opérationnelle, utilisation des ressources et comportement temporel.

4.6.2 Compilateur ArmorC

Le compilateur "ArmorC" fourni avec le langage Armor permet de compiler la description d'un processeur. La compilation génère un certain nombre d'informations qui sont regroupées dans les fichiers suivants :

1. Instruction Set file (IS) : la liste des instructions supportées par le processeur.

2. Resource Usage Information file (RUI) : décrit l'utilisation des ressources pour chaque instruction. Cette description inclut les dates de départ et de fin d'utilisation des ressources.
3. Compaction MatriX file (CMX) : décrit le parallélisme intrinsèque du processeur, c.a.d quelles sont les instructions qui peuvent êtres exécutées en parallèle.
4. Delay MatriX file (DMX) : décrit le délai qu'une instruction doit attendre avant de pouvoir utiliser le résultat issu d'une autre instruction.

4.6.3 Estimations

À partir du langage Armor et des informations générées par son compilateur nous avons défini le flot d'estimation logicielle présenté sur la figure 4.13. Dans ce flot les fichiers RUI, CMX et DMX sont utilisés afin de contraindre le fichier UAR et de guider un ordonnancement à ressources contraintes (dérivées des algorithmes présentés en 4.2). Il est aussi possible de raffiner "directement" le modèle UAR pour y intégrer des informations relatives à un modèle de processeur. Ainsi l'utilisation d'un langage tel que Armor et les informations fournies par son compilateur nous permettent d'obtenir une modélisation plus précise de la partie logicielle.

La première étape consiste à associer les nœuds du HCDFG aux instructions du processeur (fichier IS). Lorsqu'une opération du HCDFG ne peut être associée à une instruction du processeur celle-ci est décomposée en instructions plus simples (un MAC se transformant en une multiplication plus une addition).

La deuxième étape consiste à ordonnancer les instructions sur le processeur. Pour ce faire, les informations issues des fichiers RUI, CMX et DMX sont intégrées au fichier UAR afin de définir les contraintes que l'ordonnanceur va devoir respecter. L'algorithme qui a été utilisé lors de la production des courbes de compromis temps/ressources de l'étape d'estimation étant du type à contrainte de temps, une nouvelle version à contrainte de ressources de celui-ci à été développée comme résumé par l'algorithme 12.

Le résulat de la phase d'ordonnancement est représenté par un diagramme de Gantt qui exprime pour chaque cycle les instructions qui y sont ordonnancées et les ressources utilisées. Il est ainsi possible d'estimer et de comparer les perfor-

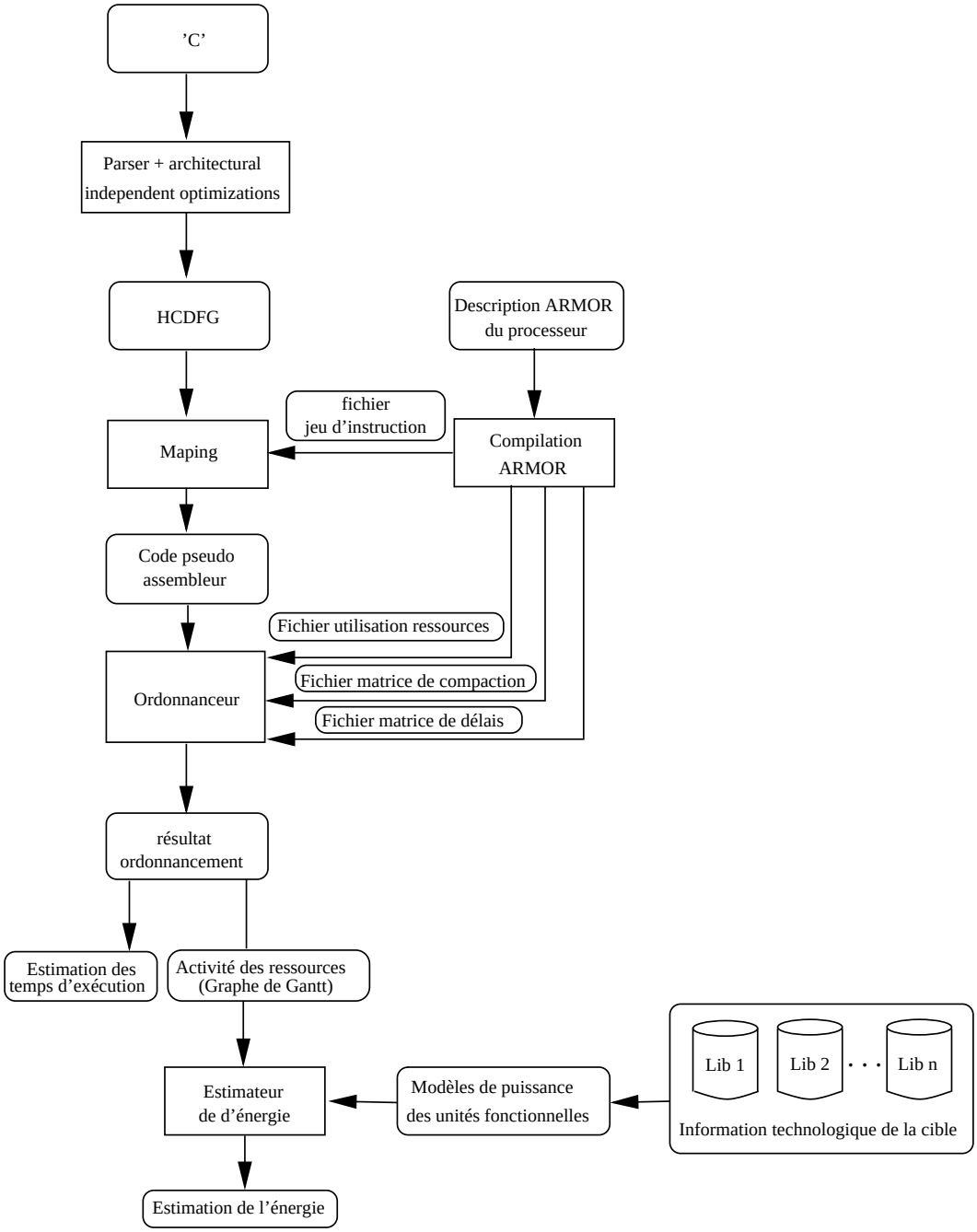

Fig. 4.13: *Le flot d'estimation logicielle qui a pour entrées i) la spécification de la fonction à estimer (langage C traduit en HCDFG), ii) la description du processeur en langage ARMOR, ainsi que iii) des informations relatives à la technologie d'implantation du processeur.*

```
pour t = nombre de cycles minimum à nombre de cycles maximum, par pas utilisateur faire
  Trier les nœuds de la liste en fonction de leurs dates ASAP et de leurs mobilités
  pour slot = 0 à t, pas + 1 faire
    Faire liste des nœuds ordonnançables
    pour tous les nœuds ordonnançables, quel que soit leur type (mémoire ou traitement) faire
      Chercher ressource disponible
      si ressource disponible alors
        Ordonnancer le nœud
      sinon si mobilité du nœud égale 0 alors
        Passer à la contrainte de temps suivante
      fin si
      Mettre à jour la liste des nœuds
    fin pour
  fin pour
fin pour
```

Algorithme 12: *Algorithme simplifié pour l'ordonnancement à ressources contraintes*

mances temporelles d'applications sur plusieurs types de micro-processeur. De plus, une phase d'estimation de la consommation a également été intégrée au flot de conception. Celle-ci est réalisée en deux étapes. La première consiste à caractériser la consommation moyenne des unités fonctionnelles du processeur à l'aide des informations fournies par le constructeur ou par estimation de celle-ci lorsqu'il s'agit d'implanter un cœur de processeur sur un FPGA par exemple. La deuxième étape associe la consommation des unités à leur activité (information d'activité issue de l'ordonnancement). Il est ainsi simple d'estimer la consommation moyenne d'une application sur un modèle de processeur pour une technologie donnée. Cette estimation de la consommation ne cherche pas la précision mais à pouvoir discriminer les solutions entre-elles et disposer, très rapidement, d'estimations pour la sélection d'un modèle de processeur. Les premiers résultats obtenus sont présentés en 5.8. La poursuite de ces travaux s'effectuera lors d'un séjour post-doctoral au sein du groupe CISS de l'Université d'Aalborg au Danemark.

Chapitre 5

Applications

Ce chapitre a deux objectifs : premièrement il présente le module **estimation système** *de l'outil* **Design Trotter**. *Cette partie de l'outil, qui a été programmée dans le cadre de cette thèse, correspond à l'implantation des différentes parties de la méthodologie présentée dans les chapitres précédents. Deuxièmement il présente les expériences qui ont été effectuées grâce à l'outil* **Design Trotter**. *Après une description des applications tests, les résultats des expériences menées sont présentés et analysés.*

5.1 Présentation de l'outil Design Trotter

5.1.1 Présentation générale

L'outil Design Trotter est composé d'un ensemble de modules qui ont été développés dans le cadre des différentes thèses conduites au sein du groupe AAS du LESTER. Cet outil, programmé en langage Java, dispose d'une interface graphique permettant d'accéder et d'utiliser les différents modules. L'outil Design Trotter comprend environ 110000 lignes de code JAVA. Les modules d'estimations s'appuient sur le modèle SPF/HCDFG présenté dans le chapitre 2. L'outil Design Trotter est constitué des modules suivants :

- L'éditeur 'C' permettant la spécification de l'application à traiter. Cet éditeur est associé à un *Parser* chargé d'effectuer la conversion du langage 'C' vers la représentation interne HCDFG.
- L'éditeur 'HCDFG' qui permet soit la spécification directe de code HCDFG soit de visualiser le code HCDFG produit par le *Parser*. Cet éditeur est associé au compilateur HCDFG dont le rôle est de transformer le code HCDFG en structure de données exploitable par les modules d'estimations.
- Le module de visualisation des graphes correspondants au code HCDFG.
- Le module d'estimation architecturale permettant d'estimer une application sur une cible FPGA. Ce module correspond à l'implantation de la méthode développée par Sébastien Bilavarn dans sa thèse [93].
- Le module d'estimation système, détaillé par la suite.
- Le module d'estimation relative. Celui-ci implantera la méthodologie développée par Lilian Bossuet [39] dans le cadre de sa thèse.
- L'éditeur de librairies. Celui-ci permet de spécifier plusieurs types de librairie en fonction du module d'estimation choisi. Pour l'estimation architecturale la librairie correspond à une description des unités de traitement et de mémorisation d'un modèle de FPGA. Pour l'estimation système, la librairie comprend d'une part une description des ressources abstraites utilisées pour l'ordonnancement et d'autre part les directives définies par l'utilisateur (ces deux points constituant les *User Abstract Rules* citées dans les chapitres précédents).

La figure 5.1 présente une capture d'écran de l'outil Design Trotter.

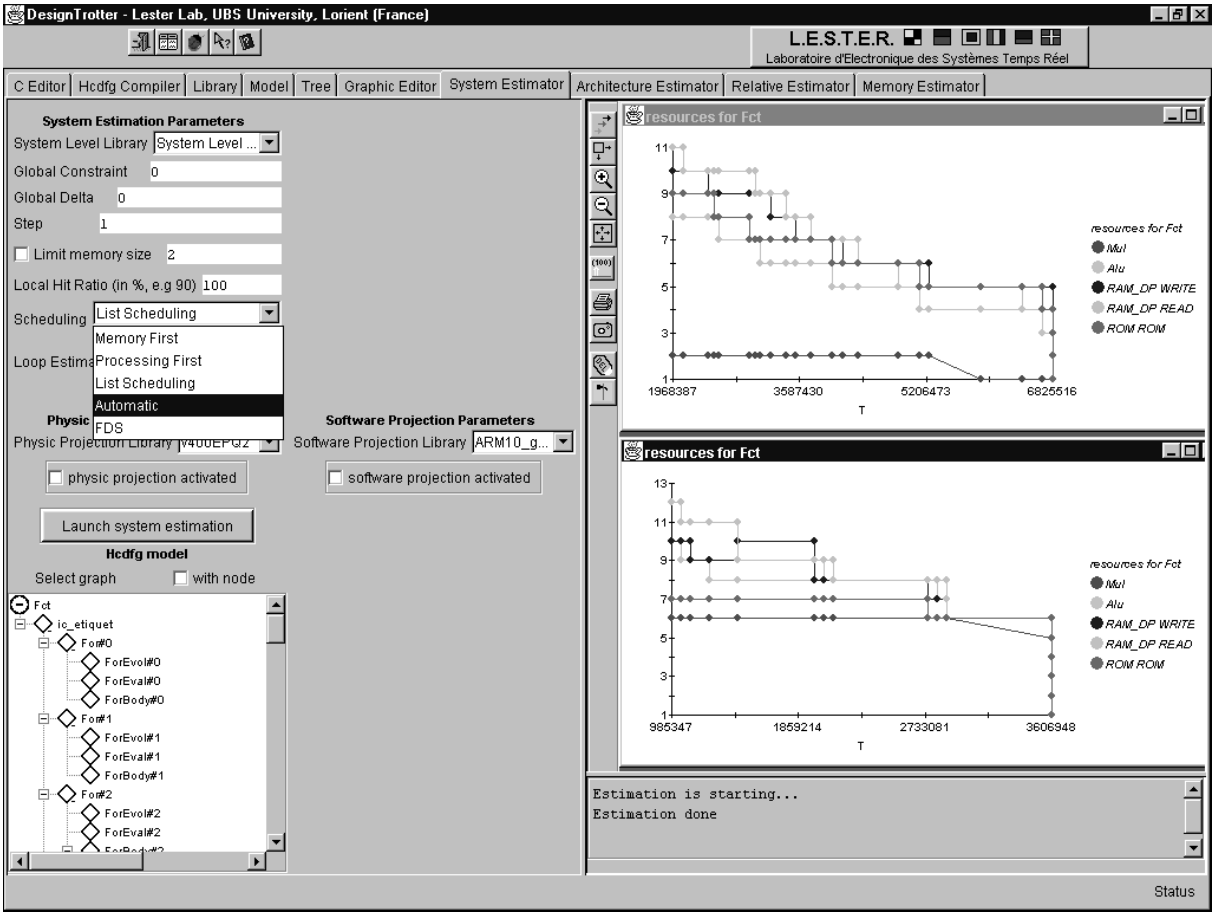

Fig. 5.1: *Capture d'écran de l'outil Design Trotter.*

5.1.2 Le module "estimations système"

Ce module a été développé dans le cadre de cette thèse. Ainsi, la méthodologie présentée dans les chapitres précédents a pu être testée et validée. De plus, les éléments permettant de réaliser la projection physique (estimation architecturale, [93]) ont également été intégrés dans le module estimation système. Ce module comprend environ 16000 lignes de code JAVA.

Les différents champs accessibles au concepteur sont :

1. Library : permet de spécifier quelle librairie utiliser pour l'estimation. La librairie est spécifiée grâce au module "éditeur de librairie".

2. Global constraint : permet de spécifier la contrainte (en cycles) pour la fonction à estimer.
3. Global Delta : permet de spécifier l'espace de recherche autour de la contrainte.
4. Step : indique le "pas" d'exploration, c.a.d la finesse d'exploration dans la gamme définie par le champ Global Delta.
5. Memory size et limit memory size : permet de spécifier la taille mémoire locale et éventuellement d'indiquer à l'outil de ne pas augmenter la taille de celle-ci pendant l'ordonnancement.
6. Scheduling : permet de choisir le type d'ordonnancement. Deux options sont possibles : le concepteur peut décider, après analyse des métriques, du type d'ordonnancement à appliquer à la fonction (à tous les niveaux de granularité de celle-ci) ou alors il peut choisir l'option automatique. Celle-ci utilise la métrique DRM pour choisir automatiquement le type d'ordonnancement à appliquer. La métrique DRM étant calculée pour tous les graphes dans la hiérarchie et pour chaque contrainte de temps, l'option automatique permet une sélection plus fine de l'ordonnancement à appliquer, et doit ainsi permettre d'obtenir de meilleurs compromis ressources / nombre de cycles).
7. Loop Scheduling : permet de décider d'activer ou non le déroulage de boucle. Lorsqu'elle est activée, cette option calcule les facteurs de déroulage automatiquement, comme décrit dans la section 4.3.2.
8. Enfin, deux options permettent d'activer soit la projection matérielle soit la projection logicielle et de spécifier les librairies à utiliser.

Les résultats sont fournis au concepteur de deux façons. La première est un affichage graphique des courbes de compromis sur la droite du module estimation système. La deuxième est une sauvegarde dans des fichiers texte de l'ensemble des estimations intra-fonction pour chaque niveau de hiérarchie de la fonction. Dans ce fichier on trouve les métriques et les courbes de compromis. Les informations relatives à ces dernières sont, pour chaque point de la courbe, le nombre de cycles, le nombre de ressources de traitement, le nombre d'accès mémoire simultanés, les tailles mémoires locales et globales (en nombre de données) et, le cas échéant, le facteur de déroulage.

5.2 Applications tests

Nous avons tout d'abord menés un certain nombre d'expériences avec plusieurs applications tests. Celles-ci sont représentatives des fonctions types que l'on retrouve dans les domaines d'applications visés par l'outil Design Trotter.

Filtrage adaptatif

Il s'agit de l'algorithme classique de filtrage de type LMS (*Least Mean Square*) qui comprend trois parties : filtrage, calcul de l'erreur et mise à jour des coefficients.

Filtrage F22

Il s'agit d'une évolution du filtre adaptatif basée sur la technique des "restes Chinois" [117]. Cet exemple permet de constater entre autre comment des transformations algorithmiques (ici du LMS vers le F22) permettent de faire ressortir le parallélisme et ainsi d'exhiber des solutions permettant d'accélérer l'application.

Filtrage de Volterra

Le filtre Volterra repose sur le principe des séries du même nom, et appartient à la famille des filtres non-linéaires. Ces derniers sont utilisés pour modéliser des phénomènes non linéaires ne pouvant pas être approximés par un filtrage linéaire classique. On observe l'emploi de filtres de Volterra dans le domaine des télécommunications pour des problèmes d'égalisation en téléphonie, en annulation d'écho accoustique, en traitement d'images. Ceux-ci sont également employés pour modéliser des phénomènes physiques en mécanique des fluides et en biologie.

Cet exemple est intéressant car son cœur est composé d'un nombre important de type flot de données avec de nombreuses données produites en interne et ré-utilisées. Nous avons choisi un filtre d'ordre 3.

Décodage de Huffman

Le principe de l'algorithme de Huffman consiste à recoder les octets rencontrés dans un ensemble de données source avec des valeurs de longueurs binaires variables.

L'unité de traitement est ramené au bit. Huffman propose de recoder les données qui ont une occurence très faible sur une longueur binaire supérieure à la moyenne, et de recoder les données très fréquentes sur une longueur binaire très courte.

Le décodage de Huffman consiste en une suite de tests qui permettent de décoder l'information.

Estimation de mouvement MPEG

Cet algorithme est utilisé dans différentes normes MPEG (MPEG 1,2,..) pour estimer les parties en mouvement dans des séquences vidéo. Il est constitué de 6 boucles *for* imbriquées. Les traitements se situent dans la boucle la plus profonde et celle juste au-dessus.

Protocole TCP

Il s'agit d'un protocole utilisé pour les communications informatiques, notamment dans le contexte TCP/IP. Cette application implique un grand nombre de tests non déterministes (c.a.d dépendants des données).

5.3 L'application détection de mouvements d'objets "ICAM"

5.3.1 Introduction

Les applications précédemment présentées sont toutes relativement simples et composées d'une unique fonction. Afin de valider notre approche sur un exemple plus complexe nous avons opté pour l'application "ICAM" dont la description fait appel à un certain nombre de fonctions. L'application de détection de mouvements ICAM nous a été fournie par le laboratoire LIST du CEA dans le cadre du projet "Epicure". L'architecture cible type pour laquelle l'application a été développée est présentée dans la figure 5.2. Il s'agit d'une caméra video "intelligente" qui est composée d'une part d'un capteur CMOS et d'autre part d'un processeur associé à de la logique reconfigurable. Les domaines d'application de ce type de caméra sont

la vidéo-surveillance pour le contrôle qualité dans l'industrie et pour la surveillance des personnes et des biens. En plus du code 'C' de l'application, des séries d'images tests issues de plusieurs séquences vidéo nous on été fournies. Ces séries d'images nous on notamment permis de réaliser des *profilings* de l'application. La figure 5.3 montre un exemple de détection de mouvement d'objets issu d'une application de surveillance.

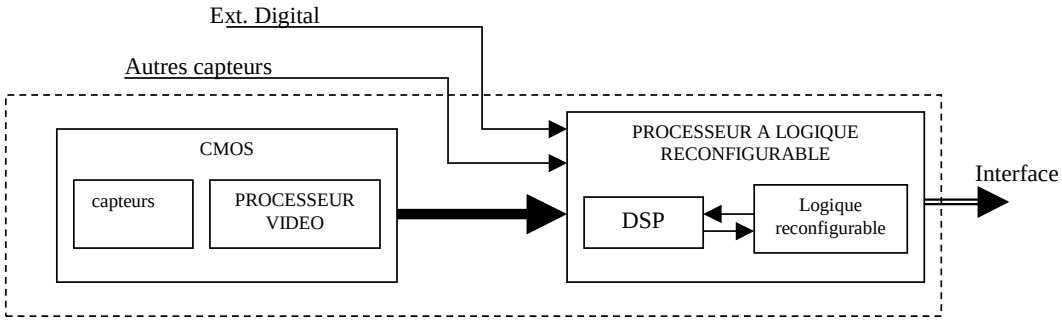

Fig. 5.2: *Architecture cible de l'application détection de mouvements. (Copyrights CEA).*

Fig. 5.3: *Exemple de détection de mouvements d'objets avec l'application ICAM.*

5.3.2 Traitements composant l'application

Le but principal de l'application est de réaliser la détection et le suivi d'objets en mouvement sur une image de référence. Pour ce faire un ensemble de traitements sont appliqués aux images vidéo issues du capteur CMOS. Ces traitements sont représentés sur la figure 5.4. Dans ce qui suit les différents traitements sont rapidement présentés.

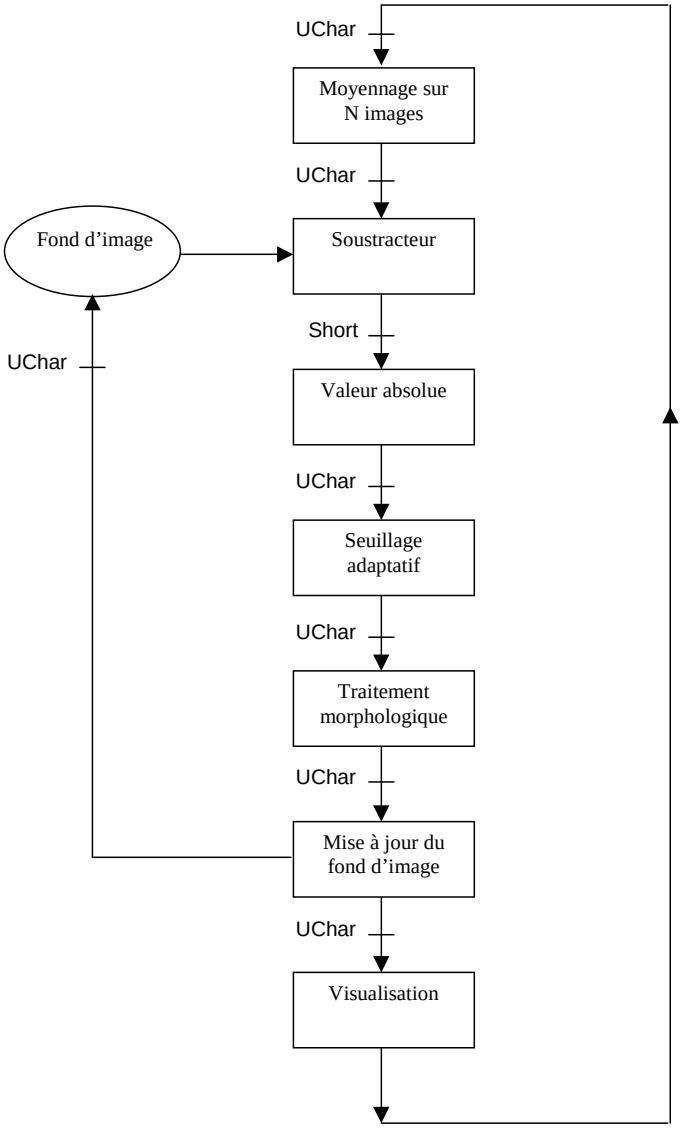

Fig. 5.4: *Ensemble des traitements nécessaires à la détection de mouvements. (Copyrights CEA).*

1. Moyennage paramétrable : Cette fonction a pour but de réduire la sensibilité au bruit du traitement. Pour ce faire la fonction effectue la moyenne de N images, le nombre d'images N étant fonction du niveau de bruit (plus le bruit est élevé plus N est grand) et de la vitesse de déplacement des objets (plus les objets se déplacent rapidement plus N est petit). Le calcul de la moyenne s'effectue sur les N images qui sont stockées dans un buffer de type FIFO.

2. Soustraction : l'objectif de cette fonction est d'identifier les zones en mouvements et les zones fixes en réalisant la soustraction entre une image de référence et l'image issue de l'étape de moyennage. La soustraction est effectuée pixel à pixel entre les deux images, ainsi les zones fixes apparaîtront en noir (niveau de gris 0) et les zones mobiles avec un niveau de gris différent de 0.

3. Seuillage : il s'agit ici de binariser l'image issue de la soustraction afin d'isoler les objets mobiles. Pour obtenir de bons résultats la fonction de seuillage n'est pas directe et le seuil est paramétrable. La valeur du seuil est issue de l'application d'un masque de gradient sur l'histogramme de l'image à traiter. Ainsi une valeur nulle du gradient de l'histogramme indique une région fixe et par conséquent la valeur du seuil.

4. Traitement morphologique : celui-ci a pour objectif de filtrer l'image afin de faire disparaître tous les points isolés. Ce traitement se décompose de plusieurs étapes successives : tout d'abord un masque de convolution est appliqué sur l'image, ensuite l'image subit une érosion (élimination des pixels blanc non connexes) suivie d'une dilatation (détection partielle de forme d'objets). Enfin une étape de reconstruction permet de retrouver la forme complète des objets.

5. Mise à jour de l'image de référence : cette fonction consiste à mettre à jour l'image de référence par analyse des centres de gravités des objets. Si au bout d'un certain délai, les coordonnées du centre de gravité d'un objet changent, alors cet objet est considéré en mouvement. La mise à jour de l'image de référence utilise une étape d'étiquetage dont le rôle est d'attribuer à tous les pixels connexes une même étiquette (c.a.d un même niveau de gris). Le principe de l'algorithme d'étiquetage est de détecter les adjacences entre pixels et de définir l'étiquette du pixel courant en fonction de celles de ses voisins. Plus

précisément, deux voisins déjà traités sont considérés : $pixel_{n-1}$ et $pixel_{n-nc}$ avec nc le nombre de colonnes de l'image. Ensuite grâce aux coordonnées Xmin, Ymin ; Xmax, Ymax de début et de fin des objets les enveloppes englobantes et les centres de gravité de ceux-ci sont calculés. Enfin, une étape de test sur le déplacement décide de remettre ou non à jour l'image de référence. Dans le cas d'une mise à jour de l'image celle-ci doit être soustraite à l'image courante. Pour déterminer si les coordonnées des centres de gravité des objets ont changées, une comparaison entre l'image courante et l'image précédente est effectuée. Un compteur est alors incrémenté si les coordonnées ne changent pas. Si le compteur atteint une certaine valeur n alors l'objet est recopié dans l'image de référence.

5.4 L'application Matching Pursuit

L'application Matching Pursuit nous a été fournie par S. Bilavarn [93] dans le cadre de son séjour post-doctoral à l'EPFL en collaboration avec INTEL. Il s'agit d'un nouveau type de compression n'opérant non pas sur des pixels mais sur des «atomes» représentant des motifs de base pour constituer une image. Cette application est intéressante à deux titres : d'une part la spécification de celle-ci est encore en cours et mouvante, il est donc intéressant pour le concepteur de disposer d'un outil lui permettant de caractériser et d'évaluer très rapidement ses choix durant la spécification. D'autre part cet exemple est représentatif des applications multimédias futures : le gain en compression requis se traduit par un accroissement de la complexité algorithmique notamment arithmétique.

La figure 5.5 montre les éléments constituant la chaîne de traitement. La partie codage est réalisée à l'aide d'un algorithme génétique et est exécutée sur un serveur, nous avons donc décidé de ne pas nous intéressé à cette partie. Par contre, la partie décodage peut être implantée sur différent types de systèmes, dont les systèmes embarquée. Nous avons donc estimé la partie décodage. Celle-ci est composée de 4 fonctions principales comme illustré sur la figure 5.5.

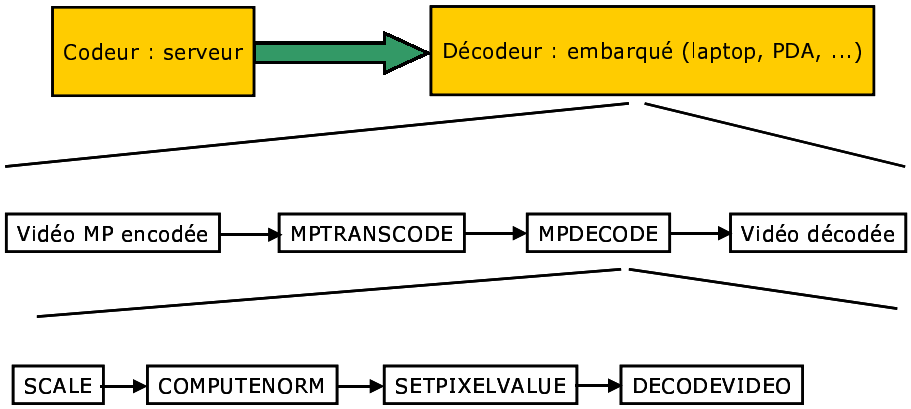

Fig. 5.5: *Chaîne de traitement "Matching Pursuit"*

5.5 Expériences sur les applications tests

Cette partie a pour objectifs d'expliquer les expériences que nous avons menées sur les différentes applications et de commenter les résultats obtenus. Ces commentaires ont deux buts : d'une part montrer le niveau de complexité que l'outil peut gérer et d'autre part montrer comment les informations fournies par l'estimation système peuvent être utilisées par le concepteur. Nous commençons par présenter l'analyse des métriques des différentes applications tests. Ensuite nous analysons quelques résultats de l'estimation système. Dans le paragraphe suivant nous montrons le flot complet d'estimation (depuis l'analyse des métriques à la projection physique) sur l'application "ICAM". Ensuite nous présentons les résultats concernant l'application Matching Pursuit. Enfin, nous terminons par quelques résultats préliminaires sur la projection logicielle.

5.5.1 Caractérisation

Dans cette partie nous commentons les résultats de la partie caractérisation du flot de conception. Pour des questions de lisibilité nous présentons les résultats pour le niveau de hiérarchie le plus haut pour chaque fonction. Le concepteur a lui accès aux résultats pour tous les niveaux via les fichiers textes générés par l'outil Design Trotter (cf. 5.1.2).

La figure 5.6 et le tableau 5.1 présentent les résultats de la caractérisation des applications tests.

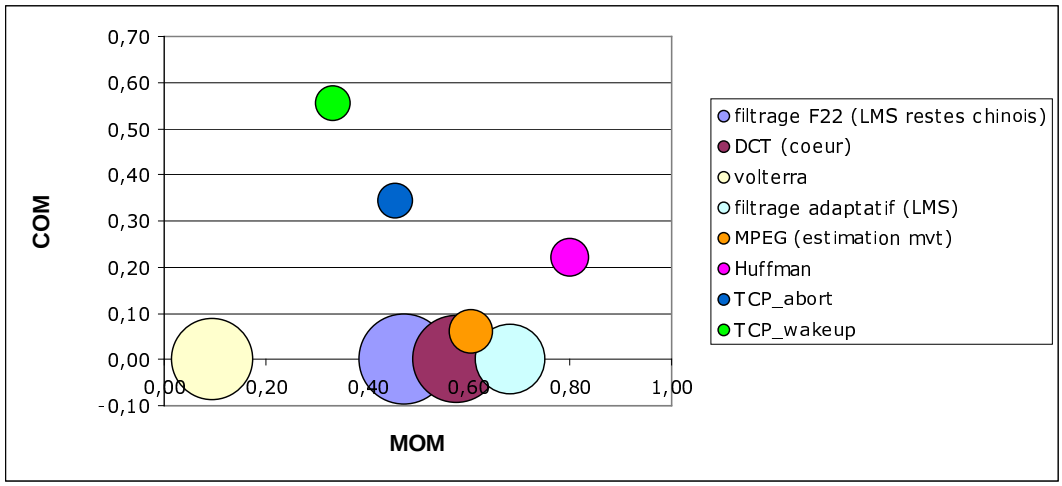

Fig. 5.6: *Récapitulatif de la caractérisation des applications tests. La métrique γ est proportionnelle au rayon du point.*

Nom de la fonction	**Chemin critique (Nb cycles)**	**Gamma**	**MOM [0,1]**	**COM [0,1]**
Filtrage F22 (LMS)	1150	6,19	0,47	0,00
DCT (coeur)	7	5,71	0,58	0,00
Filtrage de Volterra	11,00	4,82	0,09	0,00
Filtrage adaptatif	3075	3,57	0,68	0,00
MPEG (estim mvt)	136105152	1,55	0,60	0,06
Decodage de Huffman	7	1,14	0,80	0,22
TCP_abort	N/A	1	0,457	0,343
TCP_wakeup	N/A	1	0,333	0,556

TAB. 5.1 – *Récapitulatif de la caractérisation des applications tests.*

Métrique MOM

La première observation que nous pouvons faire en se référant au tableau 5.1 est que beaucoup des applications testées ont une métrique MOM $> 0{,}5$ ce qui signifie que en moyenne chaque traitement occasionne plus d'un transfert mémoire global. Cela s'explique aussi par le découpage en CDFG des applications. En effet, les CDFG (sous-graphes de test et de boucle) sont généralement de taille réduite (dans le sens où la hiérarchisation est forte, donc l'imbrication des graphes est importante). Dès qu'une structure de contrôle apparaît un nouveau graphe est créé. Les entrées/sorties de ce graphe sont alors de type globale. L'utilisation de ces données donne lieu à des échanges entre la mémoire locale et la mémoire globale d'où des valeurs de MOM relativement élevées. Nous observons par contre que pour le filtre de Volterra la métrique MOM est relativement faible (0.09). En effet, cette application est composée d'une suite de traitements sur des données locales et ce sans structure de contrôle. Les accès à la mémoire globale sont ainsi minimes (ou du moins comparables aux accès locaux) et la faible valeur de la métrique MOM indique que cette application est orientée traitement.

La métrique MOM indique donc à l'utilisateur la nécessité ou non d'utiliser des architectures (quelles soient logicielles ou matérielles) disposant de capacités importantes en ce qui concernent les entrées/sorties vers les mémoires, les bus mémoires, les buffers ou bien encore d'accès rapide à la mémoire globale. Cependant la métrique MOM reste dépendante du style d'écriture, aussi le concepteur doit en être conscient lors de la spécification de l'application.

Métrique COM

Pour rappel cette métrique reflète la proportion de tests non déterministes (c.a.d le contrôle que l'on ne peut pas supprimer car dépendant de la valeur des données). Nous observons dans le tableau 5.1 que les applications de filtrage ont des valeurs de COM nulles puisque tous les tests peuvent êtres éliminés (bornes des boucles connues par *profiling*). Ceci traduit une bonne aptitude à une implantation sur DSP ou bloc DSP au sein de FPGA (par exemple sur le modèle Stratix de chez Altera).

Pour l'application décodage de Huffman la métrique COM vaut 0,2. Au premier

abord cette valeur peut sembler faible pour une application de type décodage. Néanmoins elle indique qu'une opération sur cinq est de type contrôle non déterministe ou encore quatre opérations de transferts et/ou de traitements entre deux opérations de contrôle non déterministe. Une telle valeur de COM indique donc un faible potentiel de parallélisme et donc de possibilités d'optimisation en performance temporelle.

Enfin, pour les deux fonctions retenues du protocole TCP nous observons des valeurs de COM très élevées ($> 0{,}5$ pour tcp_wakeup) comme nous pouvions nous y attendre. Ceci traduit bien la nécessité d'implanter ce type d'application de contrôle non déterministe sur des architectures dédiées, comme les processeurs de réseaux.

Métrique γ

La métrique γ traduit le parallélisme moyen disponible dans une fonction. Par exemple pour l'application filtrage adaptatif la métrique γ est de 5,7 alors que pour le filtre F22 elle passe à 17. Ceci montre bien l'intérêt de la transformation algorithmique (basée sur les restes Chinois) qui permet ainsi d'exhiber du parallélisme au niveau traitement. Ce parallélisme peut ensuite être exploité pour augmenter les performances temporelles de l'application mais aussi réduire la consommation en énergie (le chemin critique diminuant il est ainsi possible de réduire la tension d'alimentation et la fréquence d'horologe ou d'envisager un mode veille).

5.5.2 Courbes de compromis et déroulage

Ce chapitre a pour objectif de présenter les possibilités d'exploration des boucles telles qu'elles ont été implantées dans l'outil Design Trotter (cf.4.3.2). À cet effet nous présentons deux exemples : un premier, très simple, qui permet "de bien comprendre ce qui se passe" et un second, plus complexe, qui montre les possibilités offertes par l'outil Design Trotter.

Exemple simple

Le code en langage C de cet exemple est le suivant :

```
int a[64]; int b[64]; int i;
```

```
for (i=2;i<66;i++)
{
    b[i] = x;
    a[i] = b[i-1];
}
```

Les résultats pour cet exemple sont présentés sur la figure 5.7 et dans le tableau 5.2.

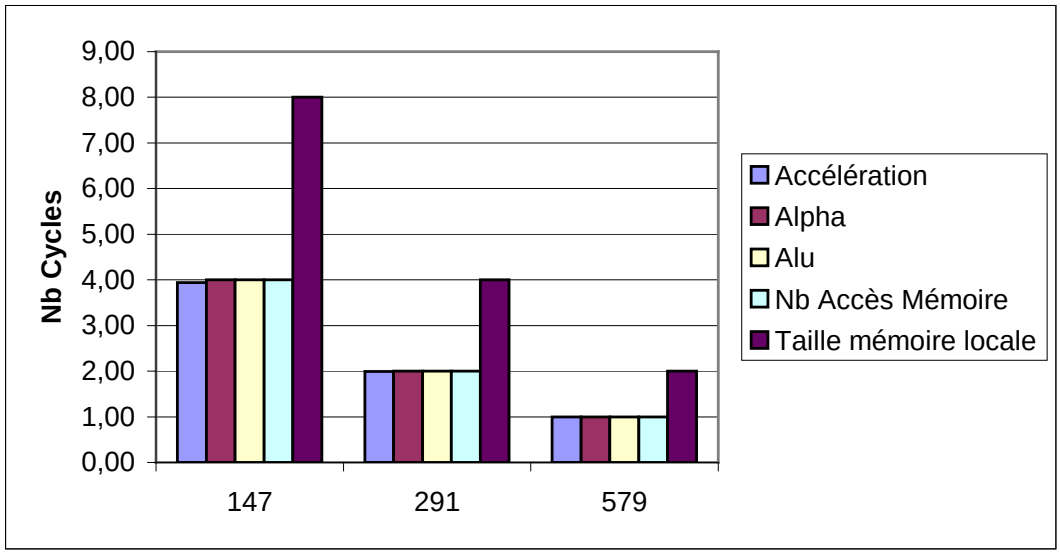

Fig. 5.7: *Exemple simple de déroulage.*

Nb cycles	Accélération	Alpha	Alu	Nb accès mémoire	Taille mémoire locale
147	3,94	4	4	4	8
291	1,99	2	2	2	4
579	1,00	1	1	1	2

TAB. 5.2 – *Courbe de compromis de la boucle de l'exemple simple.*

Le facteur maximum de déroulage trouvé est égal à 4 et dans ce cas l'accélération est de 3,94. Cette différence s'explique par le fait que c'est le cœur de boucle qui est déroulé et non la boucle complète. En effet notre approche d'estimation comptabilise dans le temps d'exécution d'une boucle le test "i $<$ borne max", le cœur de la boucle, l'évolution "i++" et à nouveau le test "i $<$ borne max" (dont le résultat retourné est l'état faux). L'exécution avec un déroulage de 4 est représenté sur la figure 5.8. La première partie de cette figure montre comment la boucle peut être déroulée :

le cœur s'exécute toujours en deux temps (b[i] = x puis a[i]=b[i-1]) mais quatre itérations peuvent être exécutées en parallèle puisque les dépendances de données le permettent, comme illustré sur la deuxième partie de la figure.

Ainsi, grâce à la gestion du déroulage par la méthode et l'outil Design Trotter, le concepteur dispose de possibilités d'exploration du parallélisme très intéressantes. En effet, le déroulage de boucle est un élément essentiel qui permet d'exhiber le parallélisme intrinsèque des applications du type multimédia ou télécommunication.

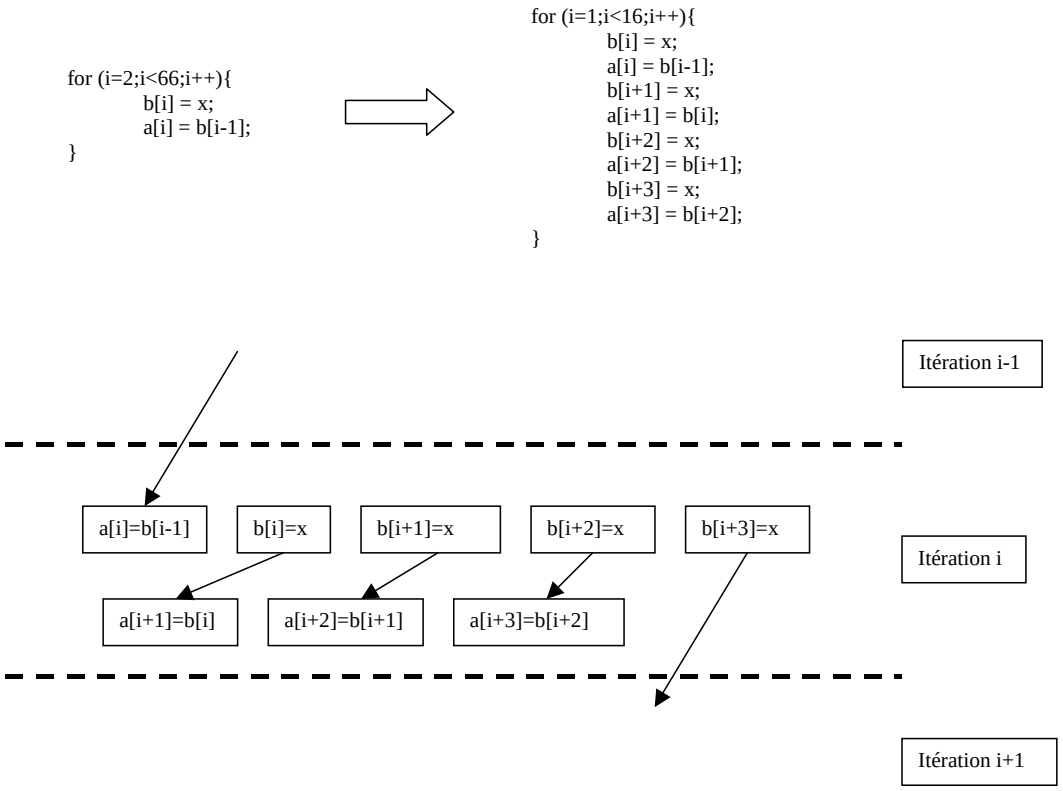

Fig. 5.8: *Exécution de la boucle avec un déroulage de 4.*

Exemple plus complexe

Le deuxième exemple est un peu plus complexe que le premier et permet de montrer comment l'outil Design Trotter arrive à gérer ce genre de situation. Le code C de cet exemple est le suivant :

```
int i; int z; int imageNew[864]; int imageOld[864];
for (i=0;i<432;i++)
{
```

```
    x=x*imagenew[i]+1;
    y=y*imageold[i]-1;
    z=x+y;
    imageNew[2*i]=imagenew[2*i-1] + imageNew[2*i+1];
    imageOld[3*i]=imagenew[2*1-2]*z;
    imageNew[i]=imageold[i];
}
```

Les résultats pour cet exemple sont présentés sur les figures 5.9,5.10 et dans les tableaux 5.3,5.4.

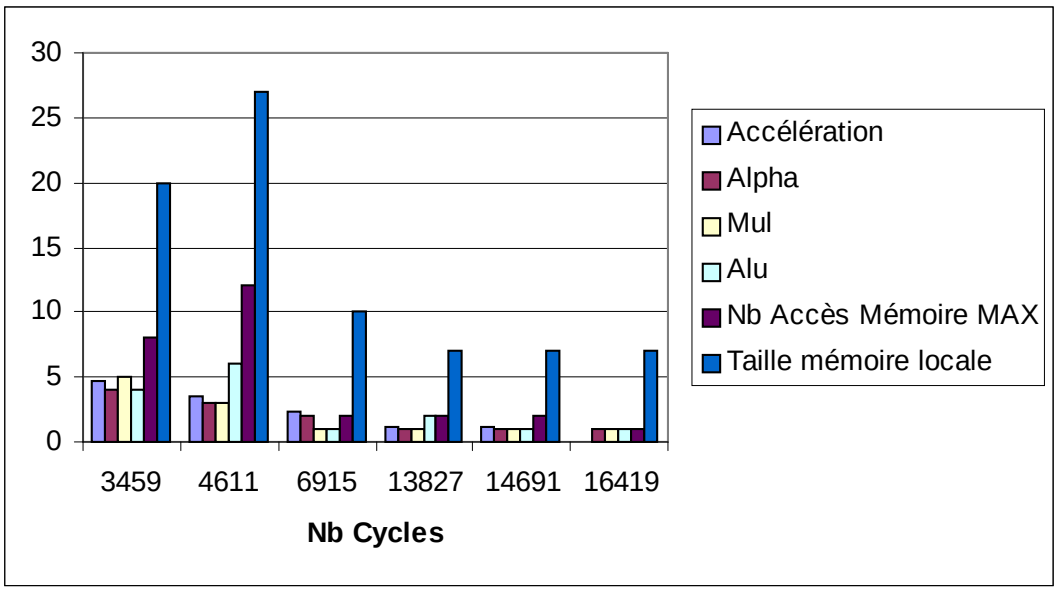

Fig. 5.9: *Exemple plus complexe de déroulage (sans lissage).*

Nb cycles	Accélération	Alpha	Mul	Alu	Nb accès mémoire	Taille mémoire locale
3459	4,74674761	4	5	4	8	20
4611	3,56083279	3	3	6	12	27
6915	2,37440347	2	1	1	2	10
13827	1,18745932	1	1	2	2	7
14691	1,11762303	1	1	1	2	7
16419		1	1	1	1	7

TAB. 5.3 – *Courbe de compromis de la boucle de l'exemple plus complexe (avant lissage).*

Nous pouvons remarquer que selon que l'on regarde les résultats d'estimations au niveau de la fonction (c.a.d le code C précédent), ou au niveau du "main()"

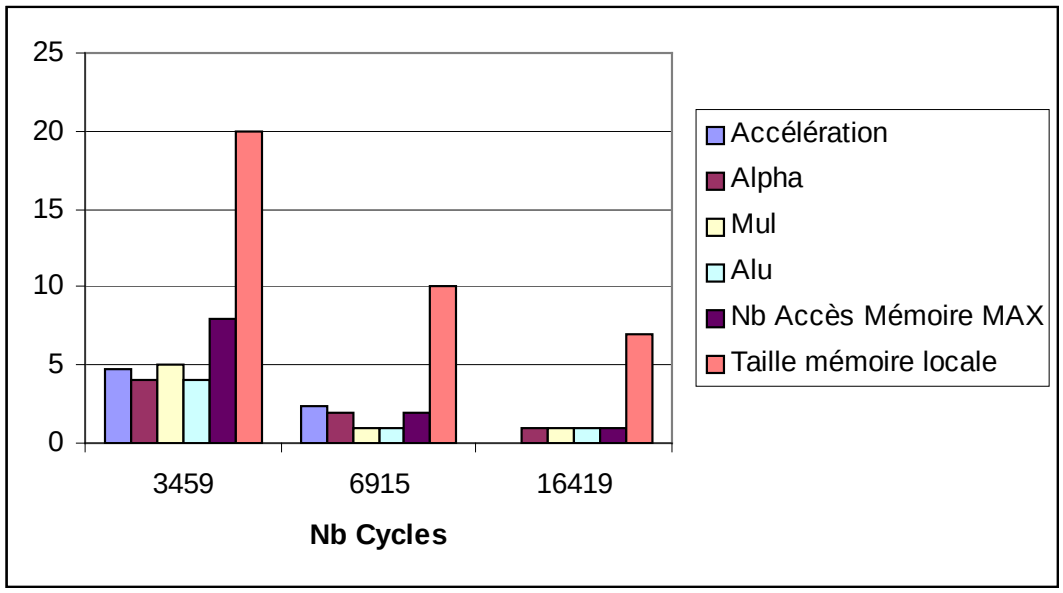

Fig. 5.10: *Exemple plus complexe de déroulage (après lissage).*

Nb cycles	Accélération	Alpha	Mul	Alu	Nb accès mémoire	Taille mémoire locale
3459	4,74674761	4	5	4	8	20
6915	2,37440347	2	1	1	2	10
16419		1	1	1	1	7

TAB. 5.4 – *Courbe de compromis de la boucle de l'exemple plus complexe (après lissage).*

(c.a.d la partie qui appelle la fonction), le nombre de solutions n'est pas le même. Au niveau de la fonction il existe des solutions supplémentaires dont le coût global (chaque type de ressource a un coût unitaire) est supérieur aux points précédents et suivants (solutions pour Nb cycles = 4611, 13827 et 14691 sur la figure 5.9). Lorsque les résultats de l'estimation sont "remontés" au niveau de hiérarchie supérieur (via des combinaisons) seuls les points particuliers (cf.4.4) sont conservés. Ainsi, ici les solutions dont le coût global est supérieur au coût d'une solution plus rapide (à sa gauche) sont éliminées (cf. fig.5.10).

5.6 Flot complet d'estimation sur l'application ICAM

5.6.1 Stratégie d'estimation

Cette partie a pour objectif de présenter la stratégie qui a été adoptée pour effectuer l'estimation de l'application ICAM. Celle-ci a été programmée en C++ pour la partie interface graphique et en C pour la partie traitement des données. La partie traitement de données comprend 31 fonctions, qui représentent 1740 lignes de code C. La première étape de l'estimation système a consisté à effectuer un *profiling* de l'application avec les images de test fournies. Cette étape nous a ainsi permis de connaître le nombre d'appels à chaque fonction et le temps passé dans chacune des fonctions. L'étape de *profiling* nous a également renseignés sur les bornes des boucles.

Suite à cette étape de *profiling* nous avons effectué une première sélection pour ne garder que les fonctions qui pouvaient présenter de réelles possibilités d'exploration (16 fonctions ont été sélectionnées, cf.tab.5.5). En effet, les fonctions qui ne disposent pas de parallélisme intrinsèque, que ce soit au niveau de leurs DFG (enchaînement d'opérations dépendantes) ou au niveau de leurs possibilités de déroulage, ne conduiront qu'à un nombre limité de solutions. Bien évidemment notre méthode et l'outil Design Trotter peuvent générer au moins une solution pour ces fonctions. La deuxième étape a ensuite consisté à estimer ces fonctions au niveau système. Cette étape finie nous avons pu procéder à une sélection de fonctions potentiellement implantables sous forme matérielle. Enfin, l'étape de projection physique (estimation

plus fine sur un modèle de FPGA) a été appliquée aux fonctions sélectionnées.

5.6.2 Analyse des métriques

Les résultats de la caractérisation des fonctions de l'application ICAM sont présentés sur la figure 5.11 et le tableau 5.5

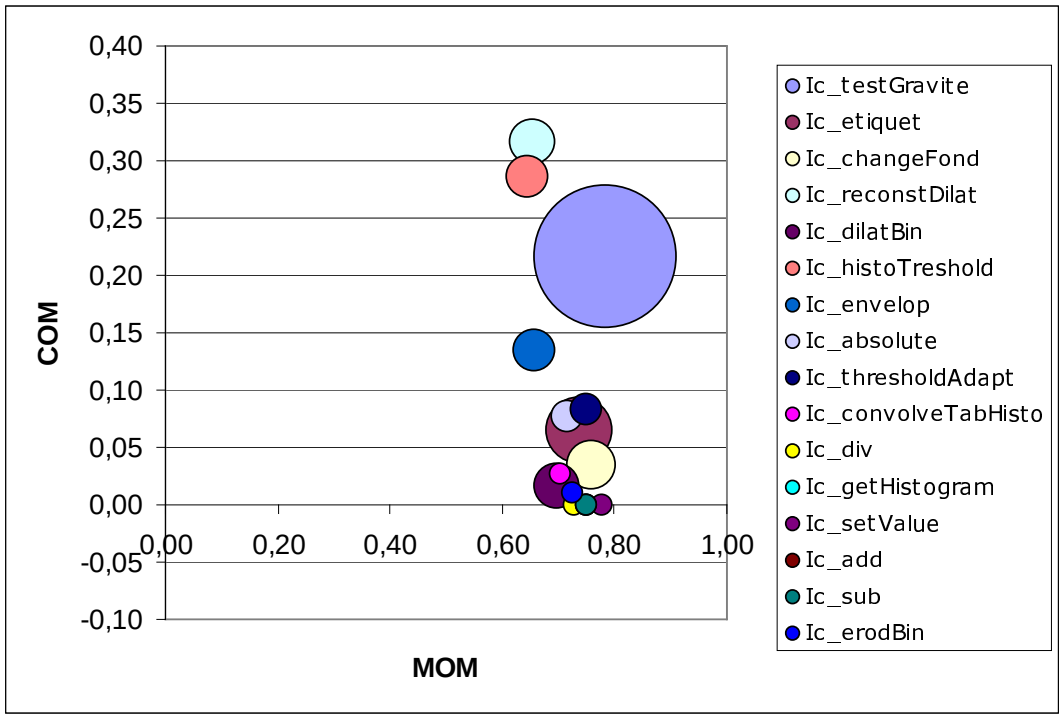

Fig. 5.11: *Récapitulatif de la caractérisation des fonctions de l'application ICAM. La métrique γ est proportionnelle au rayon du point.*

Les différentes fonctions présentent des valeurs de MOM élevées. Ceci est dû à une forte hiérarchisation des fonctions (imbrication de structures de contrôle par exemple) et de DFG (graphes au niveau de hiérarchie le plus bas) relativement courts. Ceci indique donc que l'application ICAM nécessite une architecture disposant soit d'une taille mémoire locale importante (réutilisation des données) ou alors de mécanismes d'entrées/sorties performants (lecture de données en parallèle).

5.6.3 Estimations intra-fonction

Le premier point de l'estimation système concerne le niveau de granularité à partir duquel le concepteur veut estimer les fonctions. Tous les niveaux de hiérarchie

N°	Nom de la fonction	Chemin critique (Nb cycles)	Gamma	MOM [0,1]	COM [0,1]
1	Ic_testGravite	2102,00	43,88	0,78	0,22
2	Ic_etiquet	395269,00	10,31	0,74	0,07
3	Ic_changeFond	73,00	5,62	0,76	0,03
4	Ic_reconstDilat	848144,00	4,75	0,65	0,32
5	Ic_dilatBin	49,00	4,69	0,70	0,02
6	Ic_histoTreshold	3,00	4,00	0,64	0,29
7	Ic_envelop	6098184,00	3,91	0,66	0,13
8	Ic_absolute	327683,00	2,60	0,71	0,08
9	Ic_thresholdAdapt	327683,00	2,20	0,75	0,08
10	Ic_convolveTabHisto	15879,00	1,27	0,70	0,03
11	Ic_div	524291,00	1,25	0,73	0,00
12	Ic_getHistogram	591622,00	1,22	0,75	0,00
13	Ic_setValue	1795,00	1,14	0,78	0,00
14	Ic_add	294919,00	1,11	0,75	0,00
15	Ic_sub	294919,00	1,11	0,75	0,00
16	Ic_erodBin	4776219,00	1,10	0,73	0,01

TAB. 5.5 – *Récapitulatif de la caractérisation des fonctions de l'application ICAM.*

inférieurs à celui défini par le concepteur sont estimés, et les fichiers résultats correspondants sont sauvegardés. Ceci permet soit une exploration générale des fonctions, soit une exploration plus fine permettant par exemple de repérer des parties de fonctions candidates à une implantation matérielle car disposant d'un parallélisme potentiel important (par exemple une boucle déroulée).

Dans le cadre du projet EPICURE (l'application ICAM nous a été fournie dans ce contexte) la granularité est fixée par le concepteur à la séparation Esterel (états) / fonctions C. Ensuite l'étape de partitionnement (outil du laboratoire I3S de l'Université de Nice) considère état par état la meilleure configuration en prenant comme granularité une fonction.

Afin d'illustrer l'étape d'estimation intra-fonction nous avons sélectionné deux fonctions permettant de démontrer les capacités d'exploration de notre méthode et de l'outil Design Trotter : ic_testGravite et ic_convolveTabHisto. Ces deux fonctions sont en effet assez complexes (hiérarchisation, traitements, ...) pour générer un nombre de solutions intéressants.

Fonction 1 : ic_testGravite Cette fonction comporte 378 lignes de code C ce qui se traduit par 2408 lignes au format HCDFG. Le graphe correspondant est

composé de 200 sous-graphes (comprenant la hiérarchie de la fonction et toutes les structures de contrôles, y compris graphes d'évolutions, d'évaluation, ...).

Les résultats obtenus pour cette fonction (au niveau de hiérarchie le plus haut) sont présentés dans le tableau 5.6 et la figure 5.12.

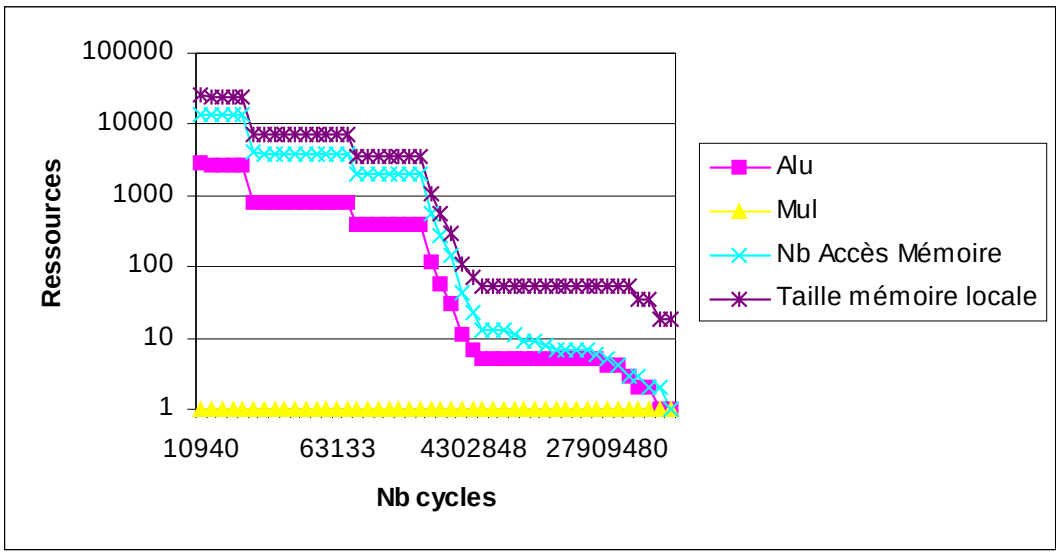

Fig. 5.12: *Courbe de compromis ressources / nb de cycles pour la fonction ic_testGravite. L'axe des ressources utilise une échelle logarithmique.*

Comme le laissait présager la métrique γ, la fonction dispose d'un fort potentiel d'accélération : jusqu'à 2614 ; les ressources nécessaires pour atteindre cette accélération sont considérables (par exemple 2775 ALU). Néanmoins, des architectures comme celles présentées dans [118] permettent d'envisager l'implantation de solutions plus raisonnables offrant quand même un bon niveau de parallélisme.

Afin d'illustrer les possibilités concernant l'exploration au sein de la hiérarchie des fonctions nous avons sélectionné un sous-graphe de la fonction ic_testGravite. Les résultats obtenus pour ce sous-graphe sont présentés dans le tableau 5.7 et la figure 5.13.

Les résultats obtenus pour ce sous-graphe (une boucle FOR) de la fonction ic_testGravite montrent les possibilités d'exploration multi-granularité de notre méthode et de l'outil Design Trotter. On observe dans le tableau 5.7 que ce sous-graphe dispose d'un potentiel intéressant d'accélération grâce au déroulage de la boucle : par exemple une accélération de 15 est possible grâce à un facteur de déroulage égal à 2

N°	Nb cycles	Accélération	Alu	Mul	Nb accès mémoire	Taille mémoire locale	Cible potentielle
1	10940	2614,85	2775	1	14020	25236	HW
2	12634	2264,24	2755	1	13764	24788	HW
3	13265	2156,53	2755	1	13764	24788	HW
4	13461	2125,13	2755	1	13763	24788	HW
5	22267	1284,70	2751	1	13743	24752	HW
6	37979	753,22	791	1	3943	7112	HW
7	41181	694,65	789	1	3933	7094	HW
8	43925	651,26	789	1	3933	7094	HW
9	49413	578,92	789	1	3933	7094	HW
10	52157	548,47	789	1	3931	7094	HW
11	54901	521,05	789	1	3929	7094	HW
12	57645	496,25	789	1	3929	7094	HW
13	60389	473,70	789	1	3928	7094	HW
14	63133	453,11	789	1	3927	7094	HW
15	71365	400,85	789	1	3927	7094	HW
16	75828	377,25	397	1	1967	3566	HW
17	85085	336,21	397	1	1967	3566	HW
18	137221	208,47	397	1	1967	3566	HW
19	139965	204,38	397	1	1966	3566	HW
20	142709	200,45	396	1	1965	3566	HW
21	145467	196,65	396	1	1964	3566	HW
22	145582	196,50	395	1	1963	3564	HW
23	264758	108,05	115	1	563	1044	HW
24	529260	54,05	59	1	283	540	HW
25	1055520	27,10	31	1	143	288	HW
26	2414976	11,85	11	1	43	108	FPGA
27	4302848	6,65	7	1	23	72	FPGA
28	8009992	3,57	5	1	13	54	FPGA
29	8547816	3,35	5	1	13	54	FPGA
30	9623464	2,97	5	1	13	54	FPGA
31	10161288	2,82	5	1	11	54	FPGA
32	10699112	2,67	5	1	9	54	FPGA
33	11236936	2,55	5	1	9	54	FPGA
34	11774760	2,43	5	1	8	54	FPGA
35	12312584	2,32	5	1	7	54	FPGA
36	13926056	2,05	5	1	7	54	FPGA
37	16615176	1,72	5	1	7	54	FPGA
38	26833832	1,07	5	1	7	54	FPGA
39	27371656	1,05	5	1	6	54	DSP embarquable
40	27909480	1,02	4	1	5	54	DSP embarquable
41	28447500	1,01	4	1	4	54	DSP embarquable
42	28447503	1,01	3	1	3	52	DSP embarquable
43	28592958	1,00	2	1	3	34	DSP embarquable
44	28592960	1,00	2	1	2	34	DSP embarquable
45	28606419	1,00	1	1	2	18	DSP embarquable
46	28606421		1	1	1	18	Proc. embarquable

TAB. 5.6 – *Résumé de l'estimation système de la fonction ic_testGravite. Les cibles potentielles sont données dans le cadre d'un système embarquable.*

Nb cycles	Accélération	Alpha	Alu	Nb accès mémoire	Taille mémoire locale
1351	105,01	14	56	280	504
2699	52,56	7	28	140	252
9439	15,03	2	8	40	72
18875	7,52	1	4	20	36
37593	3,77	1	2	10	18
40337	3,52	1	2	10	18
45825	3,10	1	2	10	18
48569	2,92	1	2	8	18
51313	2,76	1	2	6	18
54057	2,62	1	2	6	18
56801	2,50	1	2	5	18
59545	2,38	1	2	4	18
67777	2,09	1	2	4	18
81497	1,74	1	2	4	18
86985	1,63	1	1	2	18
133633	1,06	1	2	4	18
136377	1,04	1	2	3	18
139121	1,02	1	1	2	18
141865		1	1	1	18

TAB. 5.7 – *Résumé de l'estimation système d'un sous-graphe de la fonction ic_testGravite.*

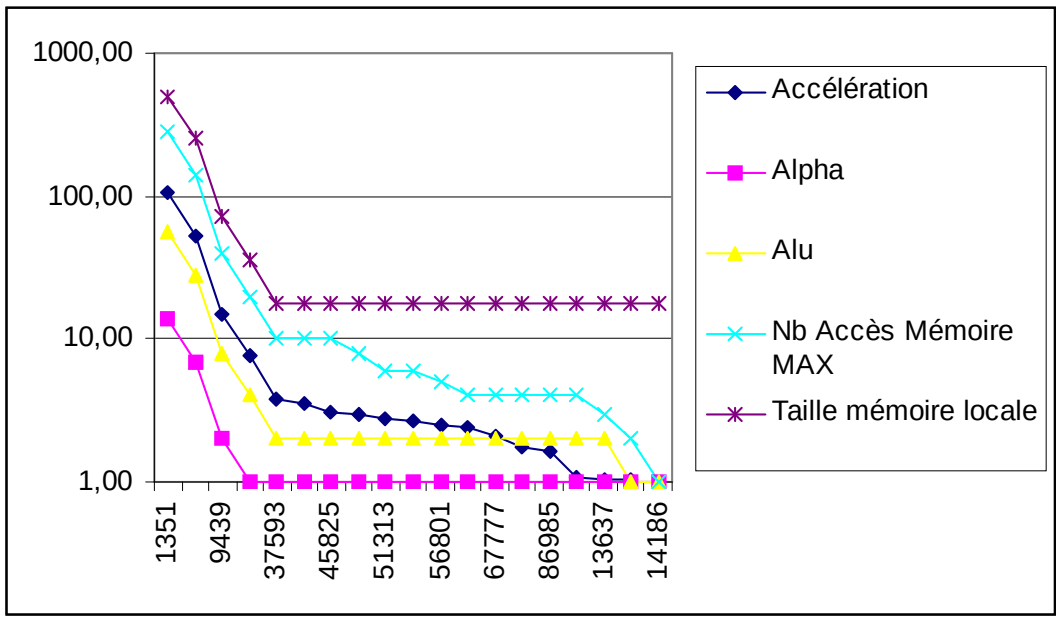

Fig. 5.13: *Courbe de compromis ressources / nb de cycles pour un sous-graphe de la fonction ic_testGravite. L'axe des ressources utilise une échelle logarithmique.*

(nécessitant 8 ALU, 40 accès mémoire simultanés et une taille mémoire locale égale à 72). Ainsi, grâce à l'exploration de l'espace de conception à tous les niveaux de hiérarchie, le concepteur peut effectuer une analyse plus ou moins fine des fonctions composant l'application à traiter. Il peut par exemple détecter certains sous-graphes (typiquement des boucles déroulables offrant de bonnes possibilités d'accélération), et décider de les implanter sur la cible la plus adaptée. Il faut cependant garder à l'esprit que les choix effectués localement (c.a.d pour des sous-graphes) peuvent avoir un impact sur le graphe complet.

À ce point du flot de conception le concepteur dispose d'un certains nombres de solutions. C'est maintenant à lui de sélectionner certaines de ces solutions en vue de la phase d'implantation. Cette sélection peut s'opérer selon divers critères tels la priorité donnée au temps d'exécution, à la surface disponible, la favorisation de tel ou tel type de ressources (traitement, accès mémoire), ...

À titre d'exemple nous avons effectué une sélection en considérant l'accélération et le nombre total de ressources correspondant (c.a.d que nous n'avons pas privilégié un type particulier de ressources, chaque type de ressources ayant un coût égal à 1). De plus nous avons écarté les solutions nécessitant beaucoup de ressources. (En effet, l'application ICAM étant du type embarquée et mobile les cibles architecturales potentielles ne peuvent être que relativement simples et de puissances limitées (processeur du type ARM7, FPGA Virtex V400EPQ2, ...)). Nous avons ainsi sélectionné les solutions numéro 25-26-27-28-31-36 (du tableau 5.6) en vue d'implantations matérielles et la solution 46 en vue d'une implantation logicielle sur un processeur embarquable de type Arm par exemple (type de processeur visé dans le contexte EPICURE). Il serait également possible d'envisager "d'accélérer" la solution à ressources unitaires (solution 46) (en fait exécuter cette solution le plus rapidement possible sans ajouter de ressources) en utilisant une architecture dont le temps de cycle est court et qui dispose d'accès mémoire optimisés. Enfin, notre phase de sélection n'a pas fait apparaître de solution implantable sur un modèle de DSP embarquable (c.a.d dont le nombre de ressources reste relativement limité par rapport aux "GROS" DSP). En effet, les solutions qui auraient pu être implantées sur ce genre de DSP (solutions 39 à 45) ne présentent pas d'accélération notable ($<$

1,1) par rapport à la solution 46. La dernière colonne du tableau 5.6 indique, dans le cadre d'un système embarqué, les cibles potentielles pour chaque solution.

Bien entendu, en fonction du type d'implantation visée pour l'application (embarquée ou non, mobile ou fixe, ...) il est possible de sélectionner plus ou moins de solutions et d'envisager leur implantation sur tout type de cibles. Cette selection est à la charge du concepteur, en fonction des contraintes qu'il doit respecter. Ainsi, dans un contexte autre que celui de EPICURE il aurait été tout à fait envisageable d'implanter par exemple les solutions 34-35 sur un "gros" DSP du type TMSC6x.

Les solutions sélectionnées pour implantation matérielle ont été "soumises" à la phase de projection physique. Ce point est décrit un peu plus loin, cf.5.6.4.

Fonction 2 : ic_convolveTabHisto Cette fonction comporte 200 lignes de code C ce qui se traduit par 1233 lignes au format HCDFG. Le graphe correspondant est composé de 96 sous-graphes.

Les résultats obtenus pour cette fonction (au niveau de hiérarchie le plus haut) sont présentés dans le tableau 5.8 et la figure 5.14.

N°	Nb cycles	Accélération	Alu	Mul	Nb accès mémoire	Taille mémoire locale	Cible potentielle
1	1893	10,75	20	18	66	188	FPGA
2	2523	8,07	16	14	31	71	FPGA
3	3783	5,38	12	10	22	53	FPGA
4	7563	2,69	8	6	13	35	FPGA
5	17391	1,17	5	3	7	23	FPGA
6	18903	1,08	5	3	6	23	FPGA
7	19407	1,05	5	3	5	21	FPGA
8	19410	1,05	4	3	4	19	FPGA
9	19738	1,03	4	2	4	15	FPGA
10	19741	1,03	3	2	3	15	FPGA
11	20077	1,01	3	1	3	11	FPGA
12	20084	1,01	2	1	2	10	FPGA
13	20355		1	1	1	6	Proc. embarquable

TAB. 5.8 – *Résumé de l'estimation système de la fonction ic_ convolveTabHisto. Les cibles potentielles sont données dans le cadre d'un système embarquable.*

Cette fonction étant moins complexe que la précédente et incluant plus de dépendances de données (et donc moins de parallélisme), le nombre de solutions trouvées

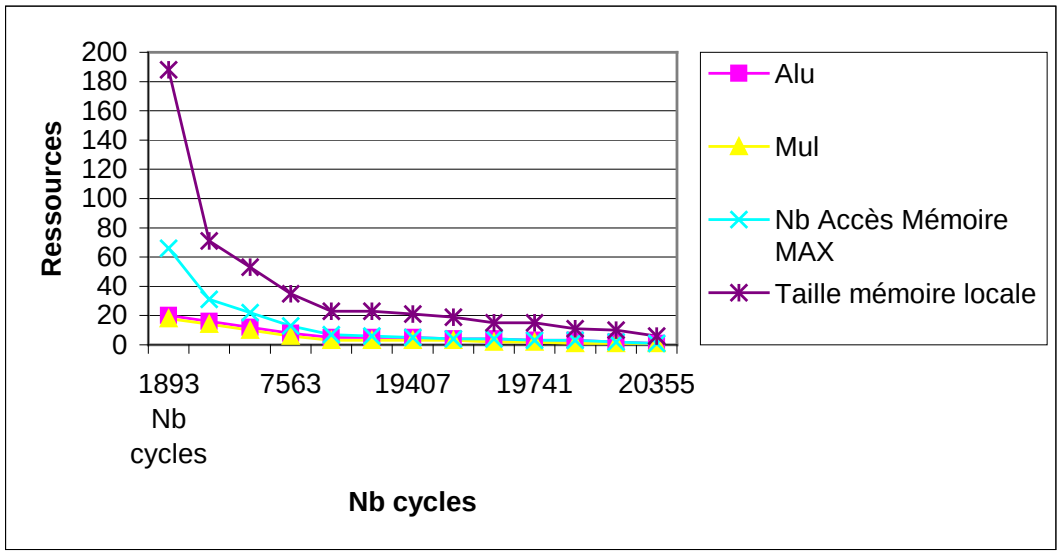

Fig. 5.14: *Courbe de compromis ressources / nb de cycles pour la fonction ic_convolveTabHisto.*

est plus faible. Cependant, les résultats incluent un certain nombre de solutions offrant un bon rapport accélération / nombres de ressources (solution 1-2-3-4). Ces solutions ont donc été sélectionnées pour la phase de projection matérielle. La solution 13 semble appropriée à une implantation logicielle sur un processeur simple du type ARM7. Les autres solutions n'offrent pas assez d'accélération au regard du sur-coût en ressources nécessaires.

Une fois que les différentes fonctions ont été estimées de la même manière que les deux fonctions présentées précédemment, leurs résultats d'estimation sont passés, soit aux étapes de partitionnnement système (estimation inter-fonctions et approche temps réel) en vue de l'estimation de l'application complète (cf.2.1), soit à l'étape de projection physique en vue du partitionnement réalisé dans le projet EPICURE.

5.6.4 Projection physique

Nous présentons ici les résultats de l'étape de projection physique pour les deux fonctions précédemment estimées au niveau système. L'étape de projection physique sur cible FPGA a été initialement développée par Sébastien Bilavarn dans le cadre de sa thèse au LESTER [93]. Nous avons donc adapté et modifié cette méthode pour qu'elle s'intègre aux travaux développés dans cette thèse (passage d'information

du module estimation système au module projection physique, ...). Il s'agit donc ici d'illustrer la méthode globale (spécification - estimation système - projection physique). Ainsi l'analyse des résultats de la projection physique reste limitée, le lecteur intéressé pourra se référer à [93] pour de plus amples informations.

Notons également qu'il est possible d'adapter ce type de projection à tout type de cible. En effet la partie estimation système est complètement séparée de la phase de projection (passage d'information d'une partie à l'autre).

Les informations qui sont fournies au concepteur par la phase de projection physique sont : le nombre d'états, le temps d'exécution, le nombre de cellules logiques (utilisées par l'unité de traitement, de contrôle et de mémoire) et le nombre de cellules dédiées (utilisées par l'unité de traitement, de contrôle et de mémoire). Durant la phase de projection physique nous avons désactivé l'option qui élimine les solutions qui requièrent plus de ressources que n'en dispose le modèle de FPGA considéré (un Virtex V400EPQ2). Il est ainsi possible de disposer de plus de solutions potentielles, charge ensuite au concepteur de trouver dans la même famille (la caractérisation des opérateurs reste valide) le modèle disposant de suffisamment de ressources.

Projection physique de la fonction 1 (ic_testGravite) Les résultats obtenus pour cette fonction (au niveau de hiérarchie le plus haut) et pour les solutions sélectionnées suite à l'estimation système (solutions 25-26-27-28-31-36) sont présentés dans le tableau 5.9.

N°	Temps (ns)	Nb états	Nb CL	Nb CD
25	20191042,08	4384	868	357
26	46196075,90	4415	428	191
27	82309179,39	857	340	56
28	153223136,97	614	296	36
31	194375278,15	618	296	32
36	266391525,22	625	296	28

TAB. 5.9 – *Résumé de la projection physique (estimation architecturale) de la fonction ic_testGravite.*

Projection physique de la fonction 2 (ic_convolveTabHisto) Les résultats obtenus pour cette fonction (au niveau de hiérarchie le plus haut) sont présentés dans le tableau 5.10.

N°	Temps (ns)	Nb états	Nb CL	Nb CD
1	313390,407	194	976	32
2	347133,963	227	976	35
3	356774,979	229	976	35
4	366415,995	229	976	33

TAB. 5.10 – *Résumé de la projection physique (estimation architecturale) de la fonction ic_convolveTabHisto.*

Les résultats ainsi obtenus permettent d'estimer les performances temporelles et le coût en surface des solutions sélectionnées. Dans le cadre du projet EPICURE ce sont ces valeurs qui sont passées de l'estimateur système à l'outil de partitionnement du laboratoire I3S de l'Université de Nice.

5.7 Estimation de l'application Matching Pursuit

La figure 5.15 montre les métriques obtenues pour les quatres fonctions principales du décodeur. Il apparaît clairement que seule la fonction DecodeVideo comprend quelques tests non-déterministes, limités cependant à 3%.

On observe également que les accès mémoires globaux sont relativement fréquents (lectures des données issue du flot vidéo). Cet exemple montre à nouveau comment un concepteur peut rapidement caractériser une application et se faire une première idée alors que la spécification est encore en cours. On peut cependant noter que dans le cadre des travaux de S. Bilavarn à l'EPFL ce n'est pas la fonction DecodeVideo qui a été choisie pour être optimisée malgré une valeur de γ relativement élevée. C'est un autre critère, le temps passé dans les fonctions, qui a été retenu. Cela nous amène à réfléchir sur la possibilité de définir d'autres métriques et de les intégrer à l'outil Design Trotter.

Les figures 5.16, 5.17, 5.18 et 5.19 montrent quant à elles les résultats d'estimations pour les quatres fonctions principales.

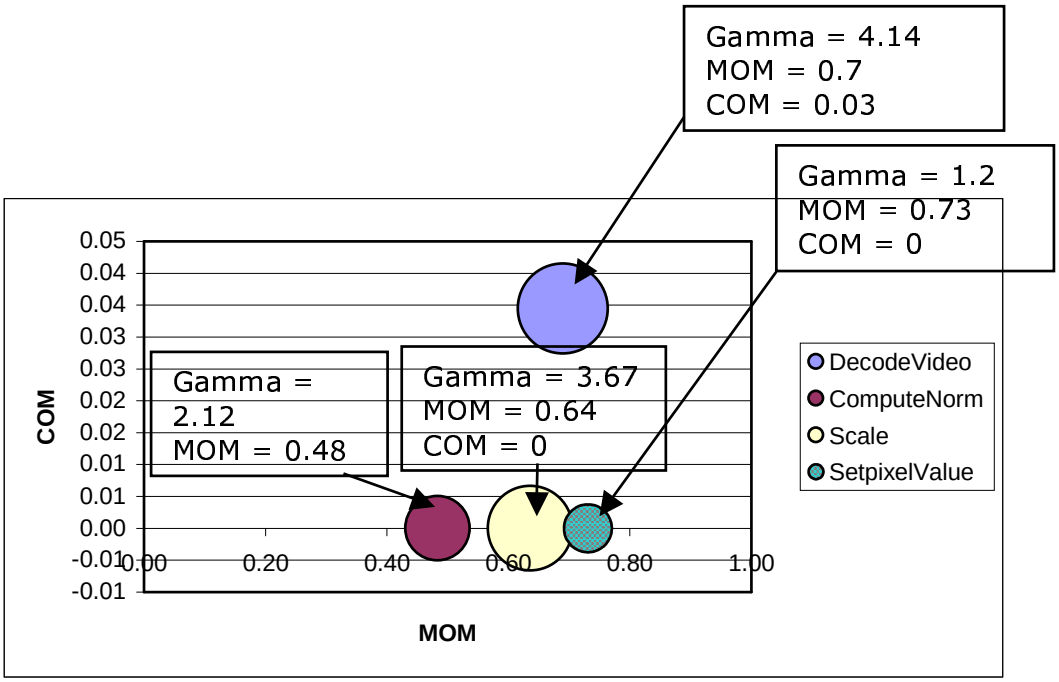

Fig. 5.15: *Récapitulatif de la caractérisation de l'application Matching Pursuit. La métrique γ est proportionnelle au rayon du point.*

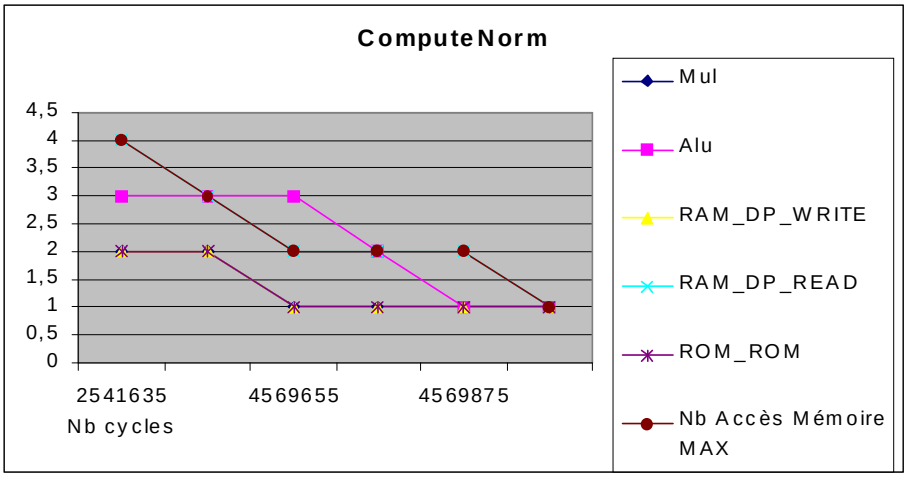

Fig. 5.16: *Courbe de compromis ressources / nb de cycles pour la fonctionCompute-Norm.*

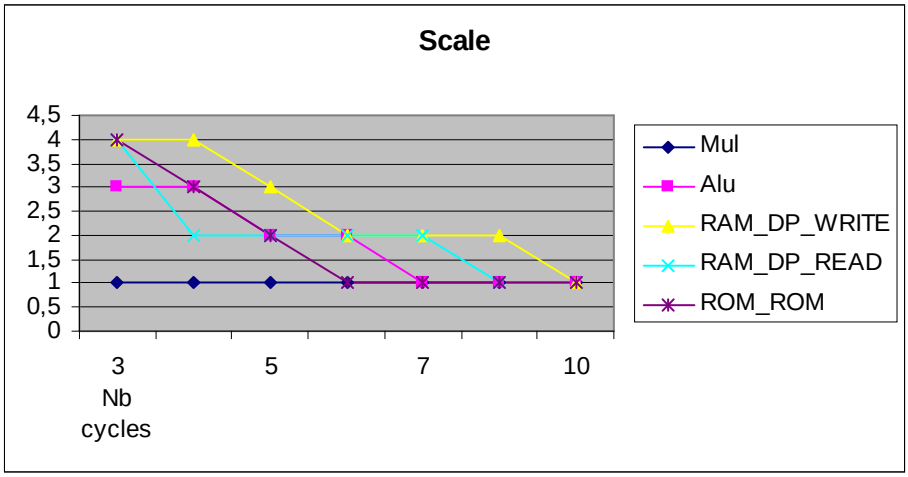

Fig. 5.17: *Courbe de compromis ressources / nb de cycles pour la fonction Scale.*

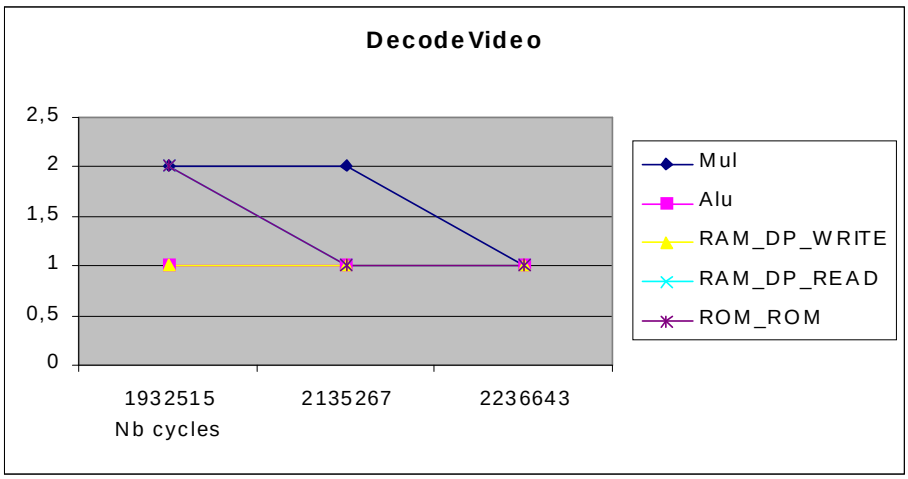

Fig. 5.18: *Courbe de compromis ressources / nb de cycles pour la fonction Decode-Video.*

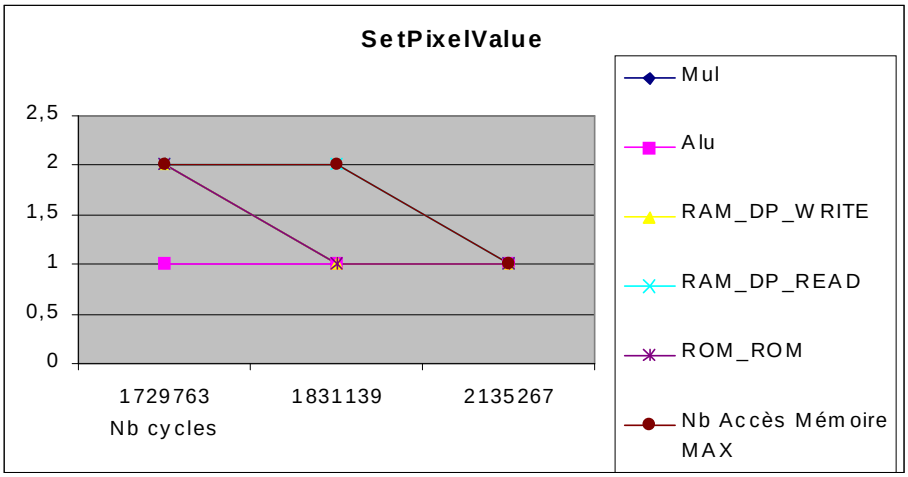

Fig. 5.19: *Courbe de compromis ressources / nb de cycles pour la fonction SetPixelValue.*

On observe que globalement le parallélisme potentiel est relativement faible, avec tout de fois la possibilité de diviser respectivement par 1,8 et 3.3 les temps de traitements pour computeNorm et Scale sans pour autant augmenter de manière dramatique les ressources requises.

Ainsi, grâce à notre méthode et à l'outil Design Trotter il est très aisé et très rapide pour un concepteur d'effectuer les premières analyses au niveau système de son application. De plus, il également possible d'analyser pendant la phase de spécification les répercussions des modifications algorithmiques.

5.8 Résultats préliminaires de la projection logicielle

La méthodologie présentée en 4.6 est actuellement en cours d'intégration à l'outil Design Trotter. Dans cette partie nous présentons quelques résultats préliminaires permettant d'illustrer la méthode proposée.

La cible choisie est le MIPS-lite, une version simplifiée du MIPS III+. Le code en VHDL synthétisable de ce processeur est disponible gratuitement [119]. Les unités fonctionnelles de ce processeur ont été synthétisées sur un FPGA Xilinx XCV300e en utilisant l'outil "ISE WebPACK". D'après les analyses temporelles de l'outil, le processeur peut tourner à une fréquence de 20Mhz. La consommation des unités fonctionnelles quant à elle a été estimée grâce à l'outil "Power Worksheet" de Xilinx.

La puissance et l'énergie moyenne estimées par opération sont résumées dans le tableau 5.11. On note qu'aucun accès mémoire n'apparaît, ceci est dû au fait que les traitements sont de type local et ne font pas accès à la mémoire externe du processeur.

Les informations fournies grâce à la compilation du modèle Armor du processeur ont ensuite été utilisées pour raffiner le fichier UAR de Design Trotter.

Tout d'abord les unités fonctionnelles du processeur ont été ajoutées dans le fichier UAR, ainsi que la liste des instructions qu'elles peuvent effectuer. Ceci a été réalisé notamment à l'aide du fichier IS (*Instruction Set*) généré par le compilateur Armor.

Ensuite l'ordonnanceur à ressources contraintes (alg.12) a été utilisé conjointement avec les informations issues des fichiers RUI, CMX et DMX (précisant respectivement l'utilisation des ressources par instructions, les instructions exécutables en parallèle et les délais inter-instructions, cf.4.6.2). Le résultat de l'ordonnancement donne lieu à une seule solution (puisqu'à ressources contraintes) dont le détail est représenté par un diagramme de Gantt.

Enfin, l'activité des unités fonctionnelles trouvée sur le diagramme de Gantt a été associé à leurs consommation moyenne afin d'en déduire l'énergie consommée.

Unité	Nb CLB	Flip-flop	Shift reg.	Puissance moyenne (mW)	Énergie/opération (pJ)
ALU	203	0	398	13	650
Mux_Bus	96	0	190	6	300
Ctrl	82	0	162	5	250
Mem_ctrl	132	65	227	9	450
Mult	723	174	1383	48	2400
Pc_next	122	30	237	8	400
Reg_bank	1660	1025	2287	97	4850
Shifter	177	0	1348	11	550

TAB. 5.11 – *Caractérisation des unités fonctionnelles du processeur Mips-lite.*

L'exemple utilisé pour les expériences préliminaires est l'algorithme pour l'approximation de la racine carrée de la somme de deux entiers signés au carré. Il est défini tel que : $\sqrt{a^2+b^2} \approx ((0.875x + 0.5y), x)$ avec $x = max(|a|, |b|)$ et $y = min(|a|, |b|)$. La consommation en énergie de l'algorithme a été estimée avec deux configurations architecturales différentes. Dans la première les deux multiplications

$0.875x$ et $0.5y$ sont calculées en utilisant des décalages et des soustractions, à savoir : x est décalé de 3 vers la droite pour effectuer $0,125x$, et y est décalé de 1 vers la droite pour effectuer $0,5y$. Ensuite $0,125x$ est soustrait de x pour obtenir $0,875x$. Dans la seconde version nous avons activé le multiplieur du processeur. Dans ce cas les multiplications sont réalisées directement par le multiplieur. Les temps d'exécution et les consommations en énergie des deux implantations sont présentées dans le tableau 5.12.

Algorithme	**Architecture**	**Temps d'exécution estimé (ns)**	**Énergie estimée (nJ)**
Sans multiplication	Sans multiplieur	1190	190
Avec multiplications	Avec multiplieur	1140	206

TAB. 5.12 – *Estimations du temps d'exécution et de la consommation sur le processeur Mips-lite pour deux configurations algorithme/architecture.*

Tout d'abord nous pouvons constater que l'ajout du multiplieur dans le processeur permet de réduire le temps d'exécution du programme d'environ 4% (puisque les décalages, soustractions et multiplications sont toutes exécutées en un temps de cycle). Cependant, la consommation en énergie ne s'en trouve pas réduite mais augmentée d'environ 5%. Ceci peut s'expliquer par le fait que l'unité multiplieur occupe plus de surface que les unités ALU et shift, ceci augmente donc la consommation du point de vue FPGA.

Cette petite expérience, malgré sa simplicité, permet d'illustrer comment il est possible d'étendre l'outil Design Trotter pour estimer les temps d'exécution et la consommation d'applications sur un modèle de processeur. L'implantation de la méthode proposée en 4.6 sera finalisée lors d'un séjour postdoctoral au sein du groupe CISS de l'Université d'Aalborg au Danemark.

5.9 Discussion

Les différentes expériences menées avec l'outil Design Trotter nous ont permis de tester et de valider la méthode que nous avons proposée. Les résultats obtenus sont conformes avec nos attentes. Il reste cependant des points à améliorer, ceux-ci sont détaillés dans le chapitre 5.10.

De plus, nous pouvons évoquer le temps de calcul pour les estimations. Celui-ci est généralement court, c.a.d inférieur à 5 minutes pour les fonctions les plus complexes et pour tous les niveaux de granularité d'une fonction (par exemple une fonction comprenant 200 graphes donne lieu à 200 courbes de compromis stockées dans 200 fichiers). N'oublions pas qu'un nombre important de solutions sont évalués et que seules les plus intéressantes sont conservées (grâce à l'utilisation des points particuliers). Néanmoins les temps de calculs peuvent augmenter sensiblement lors de l'estimation avec déroulage de boucle (jusqu'à 30 minutes). En effet deux facteurs ralentissent alors les estimations. D'une part le calcul des facteurs de déroulage compatibles avec les boucles (décomposition en facteurs premiers puis simplification des configurations possibles) est pour le moment assez long, il semble possible de l'optimiser de manière significative. D'autre part le ré-ordonnancement de graphes complexes avec des facteurs de déroulage importants participe également à ce ralentissement. Cependant il est possible de régler le compromis vitesse / nombre de solutions explorées en réglant les "pas" d'exploration cf.4.2.2 et 4.3.3.

Enfin il est assez difficile de comparer les résultats obtenus avec d'autres méthodes car la méthode d'estimation système telle que nous l'avons définie et implantée n'a, à notre connaissance, guère d'équivalence. Notre approche vise à guider le concepteur très tôt dans le cycle de conception en lui permettant d'une part, d'analyser la nature des fonctions grâce aux métriques et d'autre part de discriminer les solutions d'exploitation du parallélisme au sein d'une fonction.

5.10 Bilan

Ce chapitre a tout d'abord présenté l'outil **Design Trotter** et plus particulièrement la partie **estimation système**. Les résultats expérimentaux ont été obtenus par l'application de la méthode proposée (grâce à l'outil Design Trotter) sur un ensemble d'applications tests et sur une application plus conséquente (ICAM) et sur une autre autre (Matching Pursuit) de complexité relativement importante et surtout encore en cours de spécification. Nous avons montré comment les résultats fournis par notre méthode (analyse des métriques, courbes de compromis ressources

/ nombre de cycles, projection physique, ...) peuvent être utilisés par le concepteur pour l'aider dans ses choix algorithmiques et architecturaux lors de la conception d'un système complexe.

Conclusions et Perspectives

5.11 Conclusion

La complexité sans cesse grandissante des systèmes embarqués et les diverses contraintes auxquelles ils doivent répondre rendent nécessaire le développement de nouvelles méthodes de conception. Le succès commercial de tels systèmes dépend de leur capacité à répondre aux attentes des consommateurs : fonctionnalités originales, volume et poids faibles, consommation en énergie réduite, ergonomie d'utilisation et bien sûr coût acceptable. Les concepteurs de ces systèmes doivent donc faire face d'une part, à la pression du délai de mise sur le marché des nouvelles générations de produits, et d'autre part, à la nécessité d'en augmenter la puissance de calcul et d'en diminuer la consommation d'énergie tout en minimisant les coûts de fabrication.

Pour répondre à ces problèmes il est indispensable d'évaluer un très grand nombre de solutions potentielles afin d'y trouver celle(s) qui propose(nt) le(s) meilleur(s) compromis répondant aux critères du concepteur.

5.11.1 Réponse à la problématique et travail réalisé

Cette exploration se heurte à deux grands problèmes : d'une part les concepteurs ne disposent pas de tous les outils de développement pour toutes les cibles potentiellement utilisables, et d'autre part les niveaux d'abstraction trop faibles auxquels opèrent les outils traditionnels de co-design. Ceux-ci se situent en effet à des niveaux d'abstraction relativement bas, où les modèles utilisés sont détaillés et précis. Ces niveaux de détails conduisent à des temps d'exploration beaucoup trop lents pour trouver, en un temps raisonnable, les solutions offrant les meilleurs compromis.

La problématique posée dans cette thèse était de définir une nouvelle méthode

de conception permettant l'exploration de vastes espaces de conception au niveau système. Cette exploration doit se faire très tôt dans le cycle de conception, avant que la cible architecturale ne soit connue (ou alors que très partiellement définie). De plus, ce type d'exploration doit être rapide car à ce niveau d'abstraction l'espace de solutions est très vaste. La demande pour ce type d'exploration existe et est nécessaire, comme nous l'a confirmé la société Thales dans le cadre du projet EPICURE.

Ainsi, l'idée directrice du travail présenté dans cette thèse a été de développer un **estimateur système** dont les résultats puissent être utilisés par le concepteur très tôt dans le flot de conception.

Les ambitions de départ étaient de proposer une méthode répondant à la problématique posée précédemment et de développer un outil informatique permettant de démontrer sa faisabilité et ses capacités.

Plus précisément, nous nous étions fixé comme objectifs de proposer une méthode :

- permettant d'explorer automatiquement de vastes espaces de conception afin d'offrir au concepteur un ensemble de solutions prometteuses au plus tôt dans le cycle de développement
- capable de traiter des applications complexes décrites en langage de haut niveau, hiérarchiques, comprenant des structures de contrôle et contenant des structures de données multi-dimensionnelles (tableaux)
- offrant des étapes de caractérisation et d'estimation de l'application. La caractérisation doit permettre de décrire la nature des fonctions composant l'application afin de guider et le concepteur et les étapes suivantes du flot de conception. L'étape d'estimation doit permettre d'explorer, très rapidement, l'espace de conception par l'exhibition et l'exploitation du parallélisme intrinsèque de l'application
- s'intégrant dans le flot global d'estimation du LESTER, offrant ainsi des points d'entrées à d'autres étapes

Pour répondre à ces objectifs nous avons développé la méthode présentée dans cette thèse. Le travail effectué a porté sur les points suivants :

- Élaboration de la méthode de conception
- Participation à la définition du modèle HCDFG
- Développement de l'outil Design Trotter : mise en œuvre des solutions proposées dans la méthode

5.11.2 Résultats

Les objectifs que nous nous étions fixés ont été atteints : i) la méthode définie est cohérente et efficace, ii) l'outil Design Trotter et plus particulièrement le module estimation système permettent l'exploration rapide de fonctions complètes décrites en langage de haut de niveau (actuellement le langage C).

Ainsi, les résultats obtenus grâce à la méthode proposée permettent d'apporter une aide au concepteur lors de la phase initiale de la conception d'un système. Cette aide se matérialise d'une part, sous la forme d'informations (métriques) caractérisant l'application seule (nature des fonctions, parallélisme potentiel) et d'autre part sous la forme de courbes de compromis parallélisme (ressources) / contraintes de temps qui représentent autant d'implantations potentielles sur un modèle architectural abstrait (estimation système). Celles-ci sont générées pour tous les niveaux de la hiérarchie, en effet une implantation mixte (1 HCDFG en matériel, le reste en logiciel par exemple) peut s'avérer être la meilleure solution notamment pour l'accélération, il est donc essentiel que le concepteur puisse accéder à ces informations. Toutes ces informations permettent de renseigner le concepteur sur les besoins en ressources de l'application ce qui lui permettra ainsi de définir une architecture initiale.

Enfin, les solutions obtenues lors de l'estimation système peuvent servir de points d'entrée aux étapes de projection physique et logicielle. Durant celles-ci le modèle architectural abstrait utilisé durant l'estimation système est remplacé par celui de cibles matérielles ou logicielles précises, permettant ainsi d'affiner les estimations sur un nombre réduit d'architectures.

5.11.3 Analyses critiques

Bien que globalement satisfaisante la méthode et l'outil peuvent encore être améliorés. Parmi ces améliorations nous pouvons citer : l'intégration d'une étape

d'estimation de la taille mémoire, l'analyse des dépendances de données complexes lors du déroulage de boucle, ...

Ces éléments nous amènent tout naturellement à évoquer les perspectives possibles aux travaux présentés dans cette thèse.

5.12 Perspectives

Comme évoqué précédemment un certain nombre d'améliorations est possible. Certaines de ces améliorations sont uniquement de l'ordre du développement informatique de l'outil, d'autres supposent d'abord de préciser les méthodes à implanter. Nous présentons tout d'abord les perspectives à courts termes puis celles à plus long termes.

5.12.1 Perspectives à courts termes

Les perspectives à courts termes concernent trois points principaux et sont du type ingénierie informatique. Le premier concerne l'intégration dans l'outil Design Trotter de métriques supplémentaires : calculs d'adresses (proportion des opérations du type calcul d'adresses, histogramme des formats de données. Ces métriques permettront de fournir au concepteur et à l'outil des informations supplémentaires pour mieux guider les choix architecturaux (respectivement nécessité d'unité de génération d'adresses et dimensionnement plus précis des opérateurs).

Le second point est relatif à l'intégration de l'étape d'estimation mémoire. Il s'agit de réutiliser une méthode existante, celle de "Balasa" [120] et de l'adapter à l'outil Design Trotter et sa structure de donnée et conjointement d'implanter le calcul de la métrique HDRM.

Enfin, le troisième point consiste à "renforcer" l'outil, c.a.d à l'utiliser sur un plus grand nombre d'applications afin de corriger les erreurs forcément présentes (l'outil Design Trotter est composé d'environ 110000 lignes de code JAVA).

5.12.2 Perspectives à moyens termes

Les perspectives à plus long terme concernent essentiellement la poursuite du développement des modèles d'architecture visées, notamment celui d'une cible logicielle : le travail à réaliser consistera à affiner l'utilisation du langage Armor et à préciser l'intégration des informations fournies lors de la phase de compilation d'un modèle de processeur. Ce travail sera effectué dans le cadre d'un séjour postdoctoral au sein du groupe CISS de l'Université d'Aalborg au Danemark.

Parmi les autres modèles architecturaux à implanter on trouve notamment les architectures reconfigurables. L'intégration de ces modèles architecturaux dans l'outil Design Trotter permettra d'effectuer des projections sur différents types d'architectures dédiées et ainsi élargira son champ d'action.

5.12.3 Perspectives à longs termes

Un des aspects qu'il serait intéressant d'approfondir dans le domaine de l'exploration est celui de l'influence de la spécification sur l'estimation. La manière dont le concepteur décrit son application peut avoir une grande influence sur les résultats d'estimation. Nous pouvons distinguer au moins deux aspects : d'une part le langage et le modèle associé utilisé pour la spécification, et d'autre part, pour un même langage, les algorithmes utilisés. Le concepteur peut avoir une approche plutôt logicielle (approche "informatique", exemple : C) ou plutôt matérielle (approche "électronique", exemple : VHDL). Le choix du langage de spécification qu'il utilise est d'ailleurs souvent guidée par le type d'approche auquel il est habitué. Ensuite, il existe plusieurs algorithmes pour décrire un même traitement, quelque soit le langage choisi. C'est surtout ce deuxième point qu'il serait intéressant d'intégrer à notre méthodologie et à l'outil Design Trotter. Une proposition serait, grâce à l'utilisation de métriques (existantes ou nouvelles), de mesurer l'influence des choix algorithmiques du concepteur. Ainsi, alors que cette influence est aujourd'hui mesurable après analyse des résultats d'estimation, elle le serait plus tôt dans le flot de conception et surtout plus automatisée.

Bibliographie

[1] G. E. Moore. Cramming more components onto integrated circuits. *Electronics, Volume 38, Number 8, April 19*, 1965.

[2] J. Johansson. System design into silicon. Ericsson, 2000.

[3] S. Borkar. A vlsi system perspective for microprocessors beyond 100 nm. In *Intel keynote speech, GLSVLSI*, 2002.

[4] D-T. Marr, F. Binns, D-L. Hill, G. Hinton, D-A. Koufaty, J-A. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Tech. Jour.*, 6, February 2002.

[5] R. Schreiber, S. Aditya, B. R. Rau, V. Kathail, S. Mahlke, S. Abraham, and G. Snider. High-level synthesis of nonprogrammable hardware accelerators. *Journal of VLSI Signal Processing*, 2001.

[6] J. Plantin and E. Stoy. Aspects on system-level design. In *International Symposium on Hardware/Software Codesign (CODES)*, Rome, Italy, May 1999.

[7] Raphaël David, Daniel Chillet, Sebastien Pillement, and Olivier Sentieys. Dart : A dynamically reconfigurable architecture dealing with next generation telecommunications constraints. In *9th IEEE Reconfigurable Architecture Workshop RAW*, Fort Lauderdale, USA, April 2002.

[8] G. Sassatelli, L. Torres, J. Galy, G. Cambon, and C. Diou. The systolic ring : A dynamically reconfigurable architecture for embedded systems. In *International Conference on Field-Programmable Logic and Applications (FLP)*, Belfast, Northern Ireland, August 2001.

[9] M. Wan, H. Zhang, V. George, M. Benes, A. Abnous, V. Prabhu, and J. Rabaey. Design methodology of a low-energy reconfigurable single-chip dsp system. *Journal of VLSI Signal Processing*, 2000.

[10] R. Hartenstein, M. Herz, Th. Hoffmann, and U. Nageldinger. Kressarray xplorer : A new cad environment to optimize reconfigurable datapath array architectures. In ACM/Ieee *Asian South Pacific Design Automation Conference(ASP-DAC)*, Pacifico Yokohama, Yokohama, Japan, January 2000.

[11] E. A. Lee. What's ahead for embedded software ? *IEEE Computer Magazine*, pages 18–26, September 2000.

[12] Gérard Berry. The esterel primer. http ://www-sop.inria.fr/meije/esterel/esterel-eng.html.

[13] N. Halbwachs. A tutorial of lustre. 1993.

[14] P. Bournai, B. Cheron, B. Houssais, and P. Le Guernic. Rt-0128 - manuel signal. Technical report, INRIA, INRIA RENNES, 1991.

[15] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara. *Hardware-Software Co-Design of Embedded Systems : The Polis Approach.* Kluwer Academic Publisher, June 1997.

[16] T. Grandpierre, C. Lavarenne, and Y. Sorel. Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In *International Symposium on Hardware/Software Codesign (CODES)*, Roma, Italy, May 1999.

[17] L. Bianco, M. Auguin, G. Gogniat, and A. Pegatoquet. A path based partionning algorithm for time constrained embedded systems design. In *International Symposium on Hardware/Software Codesign (CODES)*, Seattle,USA, March 1998.

[18] T. Yen and W. Wolf. Sensitivity-driven cosynthesis of distributed embedded systems. In *IEEE International Symposium on System Synthesis (ISSS)*, Cannes, France, September 1995.

[19] B. P. Dave and N. K. Jha. Casper : Concurrent hardware-software co-synthesis of hard real-time aperiodic specification of embedded system architectures. In

Design Automation and Test in Europe Conference (DATE), Paris, France, February 1998.

[20] R. Kandem, A. Fonkua, and A. Zenatti. Hardware/software partitionning of multirate system using static scheduling theory. In *IEEE Conference on Computer Design (CCD)*, 1999.

[21] A. Österling, Th. Benner, R. Ernst, D. Herrmann, Th. Scholz, and W. Ye. *Hardware/Software Co-Design : Principles and Practice*, chapter The COSYMA System. Kluwer Academic Publisher, 1997.

[22] G. De Micheli D.C. Ku. Hardware c, a language for hardware design version 2.0. Technical report, Stanford University, 1990.

[23] D.D.Gajski, R.Dömer, and J.Zhu. *System-Level Synthesis*, chapter 10 : IP-centric Methodology and Design with the SpecC Language. Kluwer Academic Pub., 1999. A.A.Jerraya, J-P.Mermet(eds.).

[24] Systemc web page. http ://www.systemc.org/.

[25] P. Coste, F. Hessel, P. Le Marrec, Z. Sugar, M. Romdhani, R. Suescun, N. Zergainoh, and A. Jerraya. Multilanguage design of heterogenous systems. In *International Symposium on Hardware/Software Codesign (CODES)*, Roma, Italy, May 1999.

[26] Ptolemy web page. http ://ptolemy.eecs.berkeley.edu/.

[27] System level design language initiative web page. http ://www.inmet.com/SLDL/.

[28] Cadence virtual component co-design (vcc). http ://www.cadence.com/datasheets/vcc.html.

[29] Coware web page. http ://www.coware.com.

[30] Esterel studio web page. http ://www.esterel-technologies.com/.

[31] Seamless web page. http ://www.mentor.com/seamless/.

[32] J. Madsen and al. Lycos : the lyngby co-synthesis system. *Design Automation for Embedded Systems*, 2(2), March 1997.

[33] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong. System-level exploration with specsyn. In ACM/Ieee *Design Automation Conference(DAC)*, San Francisco, USA, 1998.

[34] M. Auguin, L. Capella, F. Cuesta, and E. Gresset. Codef : a system level design space exploration tool. In *International Conference on Acoustics, Speech and Signal Processing(ICASSP)*, Salt Lake City, USA, May 2001.

[35] A. D. Pimentel, L. O. Hertzberger, P. Lieverse, P. van der Wolf, and Ed F. Deprettere. Exploring embedded-systems architectures with artemis. Ieee *Computer*, 34(11) :57–63, November 2001.

[36] B. P. Dave. CRUSADE : hardware/software co-synthesis of dynamically reconfigurable heterogeneous real-time distributed embedded systems. In *Design Automation and Test in Europe Conference (DATE)*, Munich, Germany, 1999.

[37] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood. Hardware-software co-design of embbeded reconfigurable architectures. In ACM/Ieee *Design Automation Conference(DAC)*, Los Angeles, USA, June 2000.

[38] L. Shang and N. K. Jha. Hardware-software co-synthesis of low power real-time distributed embedded systems with dynamically reconfigurable fpgas. In *VLSI-design*, Bengalore, India, January 2002.

[39] L. Bossuet, G. Gogniat, and J-L. Philippe J-Ph. Diguet. A modeling method for reconfigurable architectures. In *IEEE International Workshop on System-on-Chip for Real-Time Applications*, Banff, Canada, July 2002.

[40] L. Bossuet, W. Burleson, G. Gogniat, V. Anand, A. Laffely, and J.L. Philippe. Targeting tiled architectures in design exploration. In *10th Reconfigurable Architectures Workshop (RAW)*, Nice, France, April 2003.

[41] F. Charot and V.Messé. A flexible code generation framework for the design of application specific programmable processors. In *International Symposium on Hardware/Software Codesign (CODES)*, Roma, Italy, May 1999.

[42] F. Charot. Armor description language user's manual. Technical report, IRISA-INRIA, Rennes, 2001.

[43] M. Manjunathaiah, G.M. Megson, T. Risset, and S. Rajopadhye. Uniformization of affine dependence programs for parallel embedded system design. In *Internationnal Conference on Parallel Processing*, 2001.

[44] P. V. Knudsen and J. Madsen. Pace : A dynamic programming algorithm for hardware/software partitioning. In *International Symposium on Hardware/Software Codesign (CODES)*, Pittsburgh, USA, March 1996.

[45] M. Kaul, R. Vemuri, S. Govindarajan, and I. Ouaiss. An automated temporal partitionning and loop fission approach for fpga based reconfigurable synthesis of dsp applications. In ACM/Ieee *Design Automation Conference(DAC)*, New Orleans, USA, June 1999.

[46] T. Blickle, J. Teich, and L.Thiele. System level synthesis using evolutionnary algorithms. *Design Automation for Embedded Systems*, 1998.

[47] R. Szymanek and K. Kuchcinski. Design space exploration in system level synthesis under memory constraints. In *Euromicro conference*, Milano, Italy, March 1999.

[48] P. Eles, K. Kuchcinski, Z. Peng, A. Doboli, and P. Pop. Scheduling of conditional process graphs for the synthesis of embedded systems. In *Design Automation and Test in Europe Conference (DATE)*, Paris, France, February 1998.

[49] D. Chillet, J-Ph. Diguet, J-L. Philippe, and O. Sentieys. Méthodologie de conception des unités de mémorisation appliquée au traitement du signal temps réel. *Revue TSI*, 1996.

[50] W. Verhaegh, P. Lippens, E. Aarts, J. Korst, J. Van Meerbergen, and A. van der Werf. Improved force-directed scheduling in high throughput digital signal-processing. Ieee *Transaction on Computer-Aided Design*, 14 :945–960–815, August 1995.

[51] S. Wuytack, F. Catthoor, G. De Jong, B. Lin, and H. De Man. Flow graph balancing for minimizing the required memory bandwidth. In *IEEE International Symposium on System Synthesis (ISSS)*, pages 127–132, La Jolla, USA, November 1996.

[52] M. Al-Mouhamed. Lower bound on the number of processors and time for scheduling precedence graphs with communication cost. IEEE *Transaction on Parallel and Distributed Systems*, 4(8) :875–888, August 1993.

[53] P. Slock, S. Wuytack, F. Catthoor, G. de Jong, and H. De Man. Fast and extensive system-level memory exploration for ATM applications. In *IEEE International Symposium on System Synthesis (ISSS)*, Antwerp, Belgium, September 1997.

[54] P. R. Panda. Memory bank customization and assignment in behavorial synthesis. In IEEE *International Conference on Computer-Aided Design(ICCAD)*, Santa-Clara, CA, February 1999.

[55] J. Ramanujam, J. Hong, M. Kandemir, and A. Narayan. Reducing memory requirements of nested loops for embedded systems. In ACM/IEEE *Design Automation Conference(DAC)*, Las Vegas, USA, June 2001.

[56] F. Quilleré and S. Rajopadhye. Optimizing memory usage in the polyhedral model. *ACM Trans. on Programming Languages and Systems*, 22(5) :773–815, September 2000.

[57] S. Wuytack, J-Ph. Diguet, F. Catthoor, and H. De man. Formalized methodology for data reuse exploration for low-power hierarchical memory mappings. IEEE *Transaction on* VLSI *Systems*, 6(4) :529–537, December 1998.

[58] C. Kulkarni, F. Catthoor, and H. De Man. Hardware cache optimization for parallel multimedia applications. In *Proc. EuroPar Conf. "Lecture notes in computer science" series, Springer Verlag*, Southampton, UK, September 1998.

[59] J. B. Carter, W. C. Hsieh, L. B. Stoller, M. R. Swanson, L. Zhang, and S. A. McKee. Impulse : Memory system support for scientific applications. *Journal of Scientific Programming*, 7(3-4) :195–209, 1999.

[60] P. R. Panda, H. Nakamura, N. D. Dutt, and A. Nicolau. A data alignment technique for improving cache performance. In IEEE *International Conference on Computer Design*, Santa-Clara, USA, October 1997.

[61] P. Grun, A. Halambi, N. Dutt, and A. Nicolau. Rtgen : an algorithm for automatic generation of reservation tables from architectural descriptions. In *IEEE International Symposium on System Synthesis (ISSS)*, San Jose, USA, November 1999.

[62] H. Saito and .al. The design of the promis compiler. In *International Conference on Compiler Construction (CC), also in "Lecture Notes in Computer Science 1575", Springer Verlag. March 1999*, San Jose, USA, March 1999.

[63] L. Guerra, M. Potkonjak, and J. Rabaey. System-level design guidance using algorithm properties. In Ieee *Workshop on* Vlsi *Signal Processing*, San Diego, USA, October 1994.

[64] F. Vahid and D. D. Gajski. Closeness metrics for system-level functional partitioning. In *EDAC*, pages 328–333, Brighton, U.K., September 1995.

[65] J-Ph. Diguet. *Estimation de Complexité et Transformations d'Algorithmes de Traitment du Signal pour la Conception de Circuits VLSI.* PhD thesis, Université de Rennes I, October 1996.

[66] D.Sciuto, F.Salice, L.Pomante, and W.Fornaciari. Metrics for design space exploration of heterogeneous multiprocessor embedded systems. In *International Symposium on Hardware/Software Codesign (CODES)*, Estes Park, USA, May 2002.

[67] D. D. Gajski, N. Dutt, A. Wu, and S. Lin. *High Level Synthesis (introduction to chip and system design).* Kluwer Academic Publisher, 1992.

[68] Suif web page. http ://suif.stanford.edu/.

[69] J. Henkel and R. Ernst. A hardware/software partitioner using a dynamically determined granularity. In ACM/Ieee *Design Automation Conference(DAC)*, San Francisco, USA, 1997.

[70] J. Henkel and R. Ernst. High-level estimation techniques for usage in hardware/software co-design. In ACM/Ieee *Asian South Pacific Design Automation Conference(ASP-DAC)*, Yokohama, Japan, February 1998.

[71] B. G. Halde and J. Madsen. A flexible architecture representation for high-level synthesis. In *Second Asian Pacific Conference on Hardware Description Languages*, 1994.

[72] P. V. Knudsen and J. Madsen. Communication estimation for hardware/software codesign. In *International Symposium on Hardware/Software Codesign (CODES)*, Seattle,USA, March 1998.

[73] P. V. Knudsen and J. Madsen. Integrating communication protocol selection with hardware/software codesign. IEEE *Transaction on Computer-Aided Design*, 18(8) :1077–1095, March 1999.

[74] D.D. Gajski, F. Vahid, S. Narayan, and J. Gong. Specsyn : An environment supporting the specify-explore-refine paradigm for hardware/software system design. IEEE *Transaction on* VLSI *Systems*, 6(1) :84–100, March 1998.

[75] B. R. Rau and M. Schlansker. Embedded computer architecture and automation. IEEE *Computer*, pages 75–83, April 2001.

[76] Shail Aditya, Vinod Kathail, and B. Ramakrishna Rau. Elcor's machine description system :version 3.0. Technical Report HPL-98-128, Hewlett Packard Information Technology Center, October 1998.

[77] G. Gogniat, M. Auguin, L. Bianco, and A. Pegatoquet. A codesign back end approach for embedded system design. ACM *Transaction on Design Automation of Electronic Systems*, 6(3), July 2000.

[78] C. Wong, P. Marchal, P. Yang, A. Prayati, F. Catthoor, R. Lauwereins, D. Verkest, and H. De Man. Task concurrency management methodology to schedule the mpeg4 im1 player on a highly parallel processor platform. In *International Symposium on Hardware/Software Codesign (CODES)*, Copenhaguen, Denmark, April 2001.

[79] The matador project for task concurrency management.

[80] P. Lieverse, P. Van der Wolf, K. Vissers, and E. Deprettere. A methodology for architecture exploration of heterogeneous signal processing systems. *Journal of VLSI Signal Processing for Signal, Image and Video Technology*, 29(3), November 2001.

[81] P. Lieverse, T. Stefanov, P. van der Wolf, and Ed F. Deprettere. System level design with spade : an m-jpeg case study. IEEE *International Conference on Computer-Aided Design(ICCAD)*, November 2001.

[82] F. Theeuwen. Tss : Tool for system simulation at philips research. Talk at the MEDEA Workshop on System Simulation, 1998.

[83] T. Givargis, F. Vahid, and J. Henkel. System-level exploration for pareto-optimal configurations in parameterized systems-on-a-chip. In IEEE *International Conference on Computer-Aided Design(ICCAD)*, San Jose, USA, November 2001.

[84] I. Bolsens K. Van Rompaey, D. Verkest and H. De Man. Coware, a design environment for heterogeneous hardware/software systems. In ACM/IEEE *Design Automation Conference(DAC)*, Geneve, Switzerland, September 1996.

[85] A. Sangiovanni-Vincentelli and G. Martin. Platform-based design and software design methodology for embedded systems. *IEEE Design and Test of Computers*, November-December 2001.

[86] A. Sangiovanni-Vincentelli. Defining platform-based design. *EEDesign of EE-Times*, February 2002.

[87] Metropolis : Design environment for heterogeneous systems. www.gigascale.org/metropolis.

[88] The gigascale silicon research center. www.gigascale.org.

[89] D. Davis. Architectural synthesis. Xcell journal, issue 44, pp30-34, 2002.

[90] Th. Gourdeaux, J-Ph. Diguet, and J-L. Philippe. Design trotter : Interfunction cycle distribution step. In *11th Int. Conf. RTS Embedded Systems*, Paris, France, April 2003.

[91] A. Azzedine, J-Ph. Diguet, and J-L. Philippe. Large exploration for hw/sw partioning of multirate and aperiodic real-time systems. In *International Symposium on Hardware/Software Codesign (CODES)*, Estate Park, USA, May 2002.

[92] CEA-LETI, LESTER-Université Bretagne Sud, I3S-Université Nice Sophia-Antipolis, THALES, and Esterel Technologies. Projet epicure : Environnement

de partionnement et de co-développement pour utilisation sur architectures reconfigurables. http ://www.industrie.gouv.fr/rntl/FichesA/Epicure.htm.

[93] S. Bilavarn. *Exploration Architecturale au Niveau Comportemental - Application aux FPGAs.* PhD thesis, L.E.S.T.E.R, Université de Bretagne Sud, France, 2002.

[94] J-Ph. Diguet, G. Gogniat, P. Danielo, M. Auguin, and J-L. Philippe. The SPF model. In *Forum on Design Language* (FDL), Tübingen, Germany, September 2000.

[95] H. Gomaa. *Software Design Methods for Concurrent and Real-Time Systems.* Addison-Wesley Publishing Company, 1993.

[96] M. Miranda, M. Janssen, F. Catthoor, and H. De Man. ADOPT : Efficient Hardware Address Generation in Distributed Memory Architectures. In 9^{th} IEEE/ACM *Int. Symp. on System Synthesis*, La Jolla, USA, November 1996.

[97] S. Govindarajan. Scheduling algorithms for high-level synthesis. Technical report, University of Cincinnati, department of ECECS, March 1995.

[98] S. Davidson et Al. Some experiments in local microcode compaction for horizontal machines. IEEE *Transaction on Computers*, pages 460–477, July 1981.

[99] P. G. Paulin and J. P. Knight. Force directed scheduling in automatic data path synthesis. In ACM/IEEE *Design Automation Conference(DAC)*, 1987.

[100] M. Rahmouni and A.A Jerraya. Formulation and evaluation of scheduling techniques for control flow graphs. In ACM/IEEE *Design Automation Conference(DAC)*, Brighton, UK, 1995.

[101] P. D. Wasserman. *Neural Computing, Theory and Practice.* Van Nostrand Reinhold, 1989.

[102] J. Holland. *Adaption in Natural and Artificial Systems.* Ann Arbor : the University of Michigan Press, 1975.

[103] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Publishing Company Inc, 1989.

[104] E. Martin, O. Sentieys, H. Dubois, and J L. Philippe. GAUT, an architectural synthesis tool for dedicated signal processors. In ACM/IEEE *Design Automation Conference(DAC)*, Hamburg, Germany, October 1993.

[105] F. Catthoor, S. Wuytack, E. DeGreef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom Memory Management Methodology.* Kluwer Academic Publishers, 1998.

[106] P. Lippens, J. van Meerbergen, W. Verhaegh, and A. van der Werf. Allocation of multiport memories for hierarchical data streams. In IEEE *International Conference on Computer-Aided Design(ICCAD)*, Santa Clara, USA, November 1993.

[107] A. Kountouris and C. Wolinski. Combining speculative execution and conditional resource sharing to efficiently schedule conditional behaviors. In ACM/IEEE *Asian South Pacific Design Automation Conference(ASP-DAC)*, Hong Kong, January 1999.

[108] D-J. Wang and Y. H. HU. Rate optimal scheduling of recursive DSP algorithms by unfolding. *TCS*, 41 :672–675, October 1994.

[109] A. Nicolau and R. Potasman. Incremental tree height reduction for high level synthesis. In ACM/IEEE *Design Automation Conference(DAC)*, San Fransisco, USA, June 1991.

[110] P. Marwedel. The mimola design system : Tools for the design of digital processors. In ACM/IEEE *Design Automation Conference(DAC)*, 1984.

[111] Target Compiler Technologies n.v. Chess/checkers : A retargetable tool-suite for embedded processors. Technical White Paper, January 2002.

[112] A. Fauth, J. Van Praet, and M. Freericks. Describing instruction set using nml (extented version). Technical report, TUB/IMEC, Berlin, Germany ; Leuven, Belgium, 1995.

[113] G. Hadjiyiannis and S. Devadas. Techniques for accurate performance evaluation in architecture exploration. IEEE *Transaction on* VLSI *Systems*, 2002.

[114] J. M. Rabaey N. Ghazal, A. R. Newton. Predicting performance potential of modern dsps. In ACM/Ieee *Design Automation Conference(DAC)*, Los Angeles, USA, June 2000.

[115] J. M. Rabaey N. Ghazal, A. R. Newton. Retargetable estimation scheme for dsp architecture selection. In ACM/Ieee *Asian South Pacific Design Automation Conference(ASP-DAC)*, Yokohama, Japan, January 2000.

[116] F. Djéya and F. Charot. Approche d'estimation flexible de performances pour dsp. In *7ème SYMPosium en Architectures nouvelles de machines (SYM-PA'7)*, Paris, France, April 2001.

[117] J-Ph. Diguet, O. Sentieys, J-L. Philippe, and E. Martin. Probabilistic resource estimation for pipeline architecture. In Ieee *Workshop on* Vlsi *Signal Processing*, Sakai, Japan, October 1995.

[118] W. R. Davis, N. Zhang, K. Camera, F. Chen, D. Markovic, N. Chan, B. Nikolic, and R. W. Brodersen. A design environment for high throughput, low power dedicated signal processing systems. In *IEEE Custom Integrated Circuits Conference (CICC)*, San Diego, USA, May 2001.

[119] Open cores web page. http ://www.opencores.org.

[120] N. Dutt G. Grun and F. Balasa. System level memory size estimation. Technical report, University of California, 1997.

Publications personnelles :

Y. Le Moullec, J-Ph. Diguet, D. Heller, J-L. Philippe, *Estimation du parallélisme au niveau système pour l'exploration de l'espace de conception de systèmes enfouis*, Technique et Science Informatiques (RSTI-TSI), Vol. 22, n°3/2003, Lavoisier Hermes-Science publications. http://lester.univ-ubs.fr:8080/∼moullec/tsi03.pdf

Y. Le Moullec, N. Ben Amor, J-Ph. Diguet, J-L. Philippe and M. Abid, *Multi-granularity Metrics For The Era Of Strongly Personalized SOCs*, IEEE/ACM Design Automation and Test in Europe(DATE 2003), Munich, Germany, March 2003. http://lester.univ-ubs.fr:8080/∼moullec/date03.pdf

Y. Le Moullec, J-Ph. Diguet and J-L. Philippe, *Design-Trotter : a Multimedia Embedded Systems Design Space Exploration Tool*, IEEE Workshop on Multimedia Signal Processing (MMSP 2002), St. Thomas, US Virgin Islands, December 2002. http://lester.univ-ubs.fr:8080/∼moullec/mmsp02.pdf

Y. Le Moullec, J-Ph. Diguet and J-L. Philippe, *Design-Trotter : recombinaisons hiérarchiques dans l'étape d'estimation intra-fonction*, Journées Francophones sur l'Adéquation Algorithme Architecture (JFAAA 2002), Monastir, Tunisia, December 2002. http://lester.univ-ubs.fr:8080/∼moullec/jfaaa02.pdf

Y. Le Moullec, J-Ph. Diguet and J-L. Philippe, *A Power Aware System-Level Design Space Exploration Framework*, IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS 2002), Brno, Czech Republic, April 2002. http://lester.univ-ubs.fr:8080/∼moullec/ddecs02.pdf

Y. Le Moullec, J-Ph. Diguet and J-L. Philippe, *Fast and Adaptive Data-flow and Data-Transfer Scheduling for Large Design Space Exploration*, Great Lakes Symposium on VLSI(GLSVLSI 2002), New-York, USA, April 2002. http://lester.univ-ubs.fr:8080/∼moullec/glsvlsi02.pdf

Y. Le Moullec, J-Ph. Diguet and J-L. Philippe, *A Scheduling Framework for System-Level Estimations*, IEEE International Conference on Electronics, Circuits and Systems(ICECS 2000), Kaslik, Lebanon, December 2000.
http://lester.univ-ubs.fr:8080/∼moullec/icecs00.pdf

S. Bilavarn, J.P. Diguet, G. Gogniat, Y. Le Moullec and J.L. Philippe, *Méthode de Conception d'Architectures Hétérogènes pour les Applications de Traitement Numérique du Signal*, Journées Nationales du Réseau Doctoral de Microélectronique (JNRDM 2000), Montpellier, May 2000. http://lester.univ-ubs.fr:8080/∼moullec/jnrdm00.pdf

Glossaire

- ASAP : As Soon As Possible
- ASIC : Application Specific Integrated Circuit
- ASIP : Application Specific Instruction set Processor
- ASSP : Application Specific Standard Processor
- ALAP : As Late As Possible
- COM : Control Orientation Metric
- DRM : Data Reuse Metric
- DSP : Digital Signal Processor
- FPGA : Field Programmable Gate Array
- HCDFG : Hierarchical Control and Data Flot Graph
- HDRM : Hierarchical Data Reuse Metric
- IP : Intellectual Property
- MOM : Memory Orientation Metric
- OARC : Online Average Resources Computation