# Prototypage rapide d'applications de traitement des images sur systèmes embarqués

Jean François Nezan

UNIVERSITE DE RENNES 1

U.F.R. Structure et propriétés de la matière
Section CNU 61
(Génie Informatique, Automatique et Traitement du Signal)
Rapport de synthèse
présenté pour l'obtention de

**l'Habilitation à Diriger des Recherches**
par


# Jean-François NEZAN


Titre du mémoire

# Prototypage rapide d'applications de traitement des images sur systèmes embarqués

*Soutenue le 25 novembre 2009 devant le jury composé de :*

| | | |
|---|---|---|
| Lionel TORRES, | Professeur à l'Université de Montpellier 2, | Rapporteur |
| Bertrand GRANADO, | Professeur à l'ENSEA de Cergy-Pontoise, | Rapporteur |
| Dragomir MILOJEVIC, | Chargé de cours à l'Université Libre de Bruxelles, | Rapporteur |
| Jean-Luc DEKEYSER, | Professeur à Université des Sciences et Technologies de Lille, | Président |
| Olivier SENTIEYS, | Professeur à l'Université de Rennes 1 (ENSSAT), | Examinateur |
| Olivier DEFORGES, | Professeur à l'INSA de Rennes, | Examinateur |

# Table des matières

# Chapitre 1

# Introduction

Mes travaux s'inscrivent dans le contexte général des méthodes et outils d'aide à la conception des systèmes embarqués, et plus particulièrement dans le domaine des méthodes dédiées à l'intégration d'applications de traitement des images sur des architectures multiprocesseurs. Nous qualifierons de **systèmes embarqués** les systèmes électroniques qui réalisent des fonctionnalités bien précises (une application de traitement des images par exemple) sous des contraintes importantes (autonomie réduite, limitation des ressources matérielles, consommation électrique restreinte). Ces systèmes sont partout dans notre vie de tous les jours : téléphones portables, assistants personnels ou encore lecteurs mp3. Les principales difficultés soulevées par la mise en œuvre de tels systèmes concernent d'une part la complexité croissante des applications à embarquer (il n'est pas rare qu'un téléphone portable ait à exécuter des applications qui n'ont plus vraiment à voir avec de la téléphonie, comme de la photo ou de la vidéo) et d'autre part les exigences du marché qui imposent des temps de développement toujours réduits. Ces systèmes requièrent également une puissance de calculs toujours plus importante et des contraintes d'embarquabilité toujours plus difficiles à satisfaire. Dans de tels systèmes, la limitation de la puissance de calcul est souvent palliée par l'utilisation de circuits spécifiques dédiés. Cependant, cette solution est difficilement compatible avec un temps de développement court. De plus, elle est rendue peu flexible de par l'impossibilité de reconfiguration du composant. Une alternative peut être apportée par l'utilisation de composants logiciels de type processeur, ou matériels de type circuit logique programmable : ils ont en effet l'avantage d'être programmables et le plus souvent reconfigurables. Un enjeu important pour la conception des systèmes actuels, et notamment les systèmes embarqués, concerne alors la prise en compte du parallélisme induit par ces architectures matérielles. Ces architectures peuvent apporter une accélération importante des temps de calcul, mais font apparaître de nouveaux problèmes dans la définition d'une implantation efficace de par leurs aspects parallèles et hétérogènes.

Le parallélisme est une caractéristique importante des plateformes de calcul actuelles, et ce depuis les processeurs multicœurs jusqu'aux circuits logiques programmables. Pour paralléliser des calculs, de nombreuses approches peuvent être utilisées. Il est possible par exemple d'associer plusieurs calculateurs en réseaux, ou bien de développer des processeurs intégrant plusieurs unités de calcul, ou encore d'utiliser des architectures dédiées de type logique programmable ou enfin de construire des solutions sur mesure (ASIC). Dans le cadre logiciel (architecture à base de processeurs), le paradigme de programmation séquentielle jusqu'à présent utilisé n'est plus adapté dans le contexte des architectures parallèles. Spécifier un programme parallèle nécessite alors la définition d'un modèle plus approprié. Dans le cadre matériel (logique programmable),

l'utilisation de langages spécifiques, comme les langages VHDL ou Verilog, apporte également des limitations : les algorithmes doivent en effet être complètement réécrits par des spécialistes de la logique programmable. Le programme est alors décrit de manière parallèle, mais ne peut en aucun cas être utilisé pour un processeur. Dans le cadre de la conception logicielle/matérielle conjointe (appelé co-design), aucun des langages actuels ne propose donc de solution appropriée.

Nos travaux se situent dans le domaine du prototypage rapide. Ce domaine de recherche se réfère à des méthodologies permettant de passer d'une description d'entrée de l'application, à une implantation exécutable [KL02] et peuvent ainsi fournir des solutions pour la conception matérielle/logicielle. Les objectifs du prototypage rapide sont :

- la réduction du cycle de développement,
- la sécurisation du développement,
- l'exploration des solutions matérielles.

Ces trois points expriment une même demande, à savoir un niveau d'abstraction le plus élevé possible pour les descriptions d'entrée des méthodes de prototypage rapide. Pour y répondre, nos travaux de recherche se sont dirigés vers la méthodologie AAA (Adéquation Algorithme Architecture) qui cherche la construction automatique et optimisée de programmes parallèles à partir de représentations de l'application et de l'architecture à un haut niveau d'abstraction. Pour exposer le parallélisme potentiel d'une application, une description sous la forme d'un graphe flux de données (***dataflow programming***) est utilisée. Le parallélisme est ensuite exploité pour générer automatiquement une implantation de l'application en utilisant le maximum de parallélisme disponible sur l'architecture cible. La méthodologie AAA vise ainsi à trouver la meilleure mise en correspondance appelée ***adéquation*** entre le graphe qui modélise l'application et le graphe qui modélise l'architecture multi-composants. Le programme parallèle résultant est garanti sans interblocage (***deadlock***) et l'ordre d'exécution des ***opérations*** (les ***nœuds du graphe***, également appelés ***acteurs***) ainsi que l'allocation de la mémoire sont déterminés à la compilation.

Avant de pouvoir étudier les performances des algorithmes de placement/ordonnancement automatique, il est donc nécessaire de réaliser la modélisation d'applications à l'aide de modèles flux de données. Le premier objectif de nos recherches est alors de modéliser des applications complexes de traitement des images et ainsi mettre en évidence les caractéristiques de ces différents modèles pour le placement/ordonnancement automatique. Les applications de traitement des images peuvent être caractérisés comme des systèmes réactifs, temps-réel, embarqués et déterministes. Les systèmes ***réactifs*** constituent une classe de systèmes informatiques qui ont pour principale caractéristique de réagir continuellement à leurs environnements extérieurs. Ils reçoivent de celui-ci des flots d'informations sur leurs ports d'entrée, effectuent les traitements requis, et produisent des flots d'informations sur leurs ports de sortie. Un système est qualifié de ***temps-réel*** lorsqu'il doit respecter certaines contraintes temporelles imposées par les fonctionnalités de l'application à mettre en œuvre. Un système temps réel doit satisfaire deux contraintes importantes. La première, non restrictive aux systèmes temps-réels, concerne l'exactitude logique du résultat. La deuxième contrainte, cette fois-ci propre aux systèmes temps-réels, concerne l'exactitude temporelle. Le système doit être capable de donner un résultat dans une limite de temps impartie. Un système ***embarqué*** est un système informatique dans lequel le calculateur est englobé dans un système plus large, directement situé à l'intérieur même de l'environnement qu'il contrôle comme par exemple dans le cas d'un téléphone mobile ou encore d'une fusée spatiale. Le caractère embarqué d'une application induit de fortes contraintes qui peuvent être liées à la limitation en puissance de calcul, à la mémoire disponible limitée ou encore aux ressources réduites en énergie. Enfin, un système est ***déterministe*** si son comportement est prédictible dès la phase

de conception. Cet aspect est particulièrement important dans le sens où il permet de mettre en œuvre des techniques de placement/ordonnancement optimisées dès la phase de conception de l'application et particulièrement dans la compilation.

Si le domaine de l'analyse et la compression vidéo correspond à tous ces critères, il en va de même pour le domaine de la communication numérique. Nous avons ainsi étendu nos recherches à ce domaine, en particulier dans le cadre du projet région PALMYRE ou dans le cadre de notre collaboration avec Texas Instruments.

Il existe ainsi un certain nombre de modèles flux de données dans la littérature. Certains sont très expressifs et bien adaptés à la description d'algorithmes. Cependant, leur analyse automatique reste complexe, les rendant difficiles à exploiter pour trouver des implantations multiprocesseurs efficaces et sûres. Certains autres modèles sont moins expressifs mais peuvent être utilisés dans des algorithmes de placement/ordonnancement et pour des exécutions sur des architectures multicœurs. La phase de placement correspond à la sélection du processeur chargé de l'exécution pour chacune des parties d'un algorithme. L'ordonnancement consiste en la sélection sur un processeur donné de l'ordre d'exécution de chacune des parties de l'algorithme ayant été placées sur ce processeur. Les algorithmes de placement/ordonnancement sont NP-complets : une solution peut être vérifiée exacte, mais la recherche de la solution optimale par le test de toutes les solutions potentielles ne peut être réalisée en un temps borné. De ce fait, les algorithmes de placement/ordonnancement consistent à explorer un sous-ensemble des solutions et à converger vers une solution optimisée en un temps borné et raisonnable.

Comme nous le verrons dans ce document, nous étions initialement des utilisateurs du logiciel SynDEx. Ce logiciel met en œuvre un algorithme de placement/ordonnancement à partir d'un modèle flux de données spécifique. Les bénéfices et les lacunes de cette approche ont été mis en évidence dans mes travaux de thèse (1999-2002) sur une étude des algorithmes de décompression Mpeg-4 Part2. Dès lors, nous avons travaillé sur l'optimisation du code généré par le logiciel SynDEx (thèse de Mickael Raulet) et sur les différentes manières de modéliser une application (thèse de Fabrice Urban). Face à des besoins toujours plus importants, nous avons été amenés à modifier les modèles flux de données au cœur même du logiciel (thèse de Ghislain Roquier). Durant la période 2002-2007, les travaux ont essentiellement visé à améliorer et appliquer à des applications complexes l'outil SynDEx, développé à l'INRIA Rocquencourt dans l'équipe AOSTE.

Depuis 2007, nous avons commencé le développement d'un nouvel outil de prototypage dénommé Preesm (Parallel and Real-time Embedded Executives Scheduling Method). Une thèse a été soutenue en 2009 par Pengcheng Mu, et d'autres sont en cours dans ce contexte (Maxime pelcat, Jonathan Piat). Le but de cet outil libre est de proposer un environnement global de prototypage permettant de confronter différents modèles de représentation d'applications flux de données, d'architectures, pour différentes techniques de placement/ordonnancement. La méthodologie peut être utilisée dans une phase de définition de l'architecture cible (virtual prototyping) grâce au placement/ordonnancement automatique d'une application sur une architecture multi-composant. Une fois l'architecture cible choisie, la génération de code réalise les synchronisations et les transferts de données de manière automatique tout en assurant à l'utilisateur une implantation optimisée de ces opérations complexes. Il en résulte une méthode de conception unifiée : un même outil est utilisé tout au long de la conception. Tous ces éléments concourent à la diminution des temps de développement pour des systèmes embarqués. Ce travail montre comment, à l'heure actuelle, est mise en place notre propre suite logicielle pour le prototypage rapide, et comment elle pourra évoluer pour répondre aux futurs défis susceptibles de se poser à l'issue de ces recherches.

Parallèlement, l'équipe s'est également investie dans le comité de normalisation MPEG, et plus

particulièrement dans le groupe MPEG RVC (Reconfigurable Video Coding). MPEG est un groupe de travail de l'ISO/CEI (*Organisation Internationale de Normalisation/Commission Electrotechnique Internationale*) chargé du développement de normes internationales pour la compression, la décompression, le traitement et le codage des images animées, de données audio et de leur combinaison. Les précédents standards vidéo MPEG étaient définis sous forme de simples descriptions textuelles, puis ces textes ont été associés à des logiciels de référence. Ces documents constituent les bases pour la conception d'applications fondées sur ces technologies. De nombreuses limitations apparaissent cependant dans cette méthode de conception et le groupe RVC doit définir un nouveau processus chargé de les combler. Pour cela, RVC offre des méthodes de représentation haut niveau pour des applications de codage vidéo devant faciliter à la fois la description, la réutilisation de fonctions de base, ou encore la génération automatique de code pour des cibles matérielles et logicielles. Ces objectifs rejoignent parfaitement les études menées dans l'équipe, et nous permettent de contribuer sur les aspects descriptions des décodeurs vidéo et outils associés.

Le savoir-faire du laboratoire dans le cadre des standards de compression et de décompression vidéo a été motivé notre implication au sein du groupe de travail MPEG RVC en 2007. De plus, le formalisme choisi dans MPEG RVC pour modéliser des solutions de décodage vidéo et faisant lui aussi appel à un modèle flux de données, est proche de ceux que nous avons pu étudier depuis 1999. La modélisation d'applications sous la forme de graphes flux de données fait elle aussi partie de notre savoir-faire depuis 1999. La vision à plus long terme est l'utilisation de la norme RVC comme entrée dans le processus de prototypage rapide. Dans le cadre RVC, trois thèses sont en cours au laboratoire IETR (Matthieu Wipliez et Nicolas Siret) dont une en collaboration avec l'INT (Jérôme Gorin, encadré par Mickael Raulet pour l'IETR).

Suite à cette introduction constituant le chapitre 1, le présent rapport est constitué de sept autres chapitres. Le chapitre 2 dresse une analyse précise de la situation actuelle en termes de programmation au sens général du terme, et précise comment les approches flux de données peuvent constituer une alternative pour la conception des futurs systèmes embarqués. Le chapitre 3 détaille les modèles utilisés dans le contexte AAA (et l'outil SynDEx) dans lequel nos recherches se situent. Puis dans le chapitre 4 sont présentés les travaux menés sur le logiciel SynDEx, travaux permettant d'utiliser ce logiciel de placement/ordonnancement pour la conception de systèmes embarqués dédiés et optimisés. Le chapitre 5 notre mise en place du logiciel Preesm zfin de dépasser les limitations observées lors de l'utilisation du logiciel SynDEx. Nous présenterons ensuite, dans le chapitre 6, notre contribution à la mise en œuvre du nouveau standard MPEG RVC. La conclusion montrera dans le chapitre 7 comment ces travaux devront converger. Enfin, pour clore ce document, le chapitre 8 référence les diverses sources bibliographiques utilisées dans ce rapport ainsi que la liste de mes publications.

# Chapitre 2

# Problématique

Ce chapitre a pour but de présenter le contexte de nos travaux. Notre objectif est de passer le plus rapidement possible de la définition d'une nouvelle technique de traitement des images à une implantation optimisée sur un système électronique embarqué. L'augmentation des résolutions des images, ainsi que la nécessité de mettre en œuvre des techniques de plus en plus complexes, impliquent une utilisation optimisée des ressources disponibles (ressources en calculs et en mémoire) sur le système embarqué. Une telle optimisation nécessite généralement plusieurs mois de travail pour une équipe de plusieurs ingénieurs, ce temps comme la taille de l'équipe étant très variables en fonction conjointement de la complexité de l'application, de l'expérience de l'équipe dans le domaine et du niveau de performance souhaité pour le système final. Notre objectif est donc d'automatiser cette phase d'implantation afin d'en diminuer au maximum la complexité et donc le temps nécessaire à sa réalisation.

Ce chapitre présente ainsi l'état de l'art des solutions liées à la conception de systèmes embarqués et met en évidence le besoin d'une méthodologie de conception appropriée, sur laquelle s'appuient nos travaux. Pour cela, ce chapitre rappelle le contexte technologique de nos travaux, puis les solutions actuelles pour la programmation de composants : les processeurs standards, les processeurs graphiques (GPU), les processeurs de traitement du signal (DSP) et les systèmes de logique programmable (FPGA) seront abordés. Serront présentées ensuite les méthodes et outils permettant de concevoir des systèmes complexes constitués de plusieurs composants. Enfin, nous verrons comment nos travaux de recherche se positionnent dans ce contexte.

## 2.1 Contexte technologique

Pour saisir le contexte technologique, il est nécessaire d'appréhender quels sont les leviers technologiques qui permettent d'accroître la puissance de calcul dans un processeur. L'élément de base dans un processeur est la porte élémentaire. Plusieurs transistors sont associés avec notamment une capacité pour former cette porte élémentaire. Sans entrer dans tous les détails, la puissance consommée par un processeur peut être modélisée par l'équation définie dans [Mud01] par

$$P = A.C.V^2.\mathrm{f} + t.A.V.I_{cc}\mathrm{f} + V.I_{fuite}, \tag{2.1}$$

où :
- A : activité du processeur (nombre de portes élémentaires modifiées lors d'un coup d'horloge),
- C : capacité des portes élementaires,
- V : tension d'alimentation du processeur,
- f : fréquence de cadencement du processeur,
- t : temps d'utilisation du processeur,
- $I_{cc}$ : courant de court-circuit (lors du basculement d'une porte élémentaire),
- $I_{fuite}$ : courant de fuite (consommation du composant lorsque l'horloge est désactivée).

Le premier terme de l'équation 2.1 est appelé la ***puissance dynamique*** et dépend directement de la fréquence d'utilisation **f** et de la tension d'alimentation V du processeur. Le second terme est appelé ***puissance statique*** et le dernier ***puissance de fuite***. Cette équation reste valable pour toutes les technologies CMOS utilisées pour la fabrication des processeurs. Les technologies CMOS sont actuellement encore les plus intéressantes en termes d'intégration.

Tous les paramètres de cette équation évoluent cependant en fonction de la technologie CMOS utilisée. Ainsi, pour diminuer la consommation d'un processeur, il est possible de jouer sur la technologie des composants en améliorant notamment les paramètres C, $I_{cc}$ et $I_{fuite}$, ou encore de diminuer le nombre de portes élémentaires et leurs transitions (paramètre A). Il est également possible de faire varier la tension d'alimentation V. Ceci implique cependant des améliorations sur la technologie des composants car il est impossible de diminuer V sans diminuer la fréquence maximale d'utilisation, les deux étant liés selon l'équation

$$\mathrm{f}_{max} = \frac{(V - V_{threshold})^2}{V}, \tag{2.2}$$

où $V_{threshold}$ est la tension de seuil entre le niveau haut (un logique) et le niveau logique bas (zéro logique).

L'équation 2.2 montre ainsi, que les deux paramètres $\mathrm{f}_{max}$ et V influent sur la définition des tensions qui caractérisent les niveaux bas et haut. Ces deux paramètres sont liés car il faut pouvoir passer d'un état vers l'autre entre deux coups d'horloge (paramètre $\mathrm{f}_{max}$) tout en concervant suffisamment d'écart en tension pour distinguer les deux niveaux logiques (paramètre $(V - V_{threshold})$).

Dans le domaine de l'embarqué qui nous intéresse particulièrement, la consommation maximale est généralement fixée à quelques Watts. La puissance consommée par un processeur sera transformée en chaleur, chaleur qu'il est nécessaire de dissiper afin de ne pas détériorer le composant. On utilise le plus souvent à cet effet un dissipateur thermique car celui ci ne nécessite pas d'alimentation contrairement à un ventilateur qui consomme de l'énergie pour la dissipation.

Depuis les années 1970 et jusqu'en 2005 environ, la ***loi de Moore*** a souvent été vérifiée.

Cette ***loi*** est en fait une prévision empirique selon laquelle les puissances de calcul pouvaient être doublées tous les six mois. Depuis cette époque, le principal levier utilisé a été de miniaturiser la technologie CMOS. En plus de proposer des composants de plus en plus réduits en termes de surface, cette miniaturisation a conduit à la diminution des temps de propagation entre portes élémentaires et ainsi à augmenter la fréquence de fonctionnement des processeurs sans pénaliser la consommation globale. Pour une technologie donnée et pour le domaine de l'embarqué, la solution pour augmenter la puissance de calcul passe par l'augmentation de la fréquence des horloges dans la limite de la consommation des 1 Watt. Ceci permet d'accélérer le séquencement des actions par l'unité de contrôle.

L'ITRS (International Technology Roadmap for Semiconductors) [ITR] est un organisme international ayant pour objectif de prévoir les évolutions technologiques. Les prévisions de l'ITRS ont globalement toujours été vérifiées, donnant beaucoup de poids à cet organisme. Toutefois, les contraintes technologiques et financières font qu'à l'heure actuelle, la course à la miniaturisation est ralentie et ne permet plus de suivre la loi de Moore. Les performances en termes de miniaturisation atteignent des limites non seulement physiques, mais également économiques. Changer une technologie coûte de plus en plus cher, et nécessite de réaliser les composants dans des quantités toujours plus importantes pour atteindre des coûts unitaires intéressants. Selon l'ITRS [ITR07], cette tendance va perdurer pendant les 15 prochaines années.

Afin de continuer à augmenter les capacités de calcul malgré ce ralentissement dans l'évolution technologique, la solution est de réaliser plusieurs calculs en parallèle. Pour s'en convaincre, il faut considérer que l'augmentation de la fréquence est proportionnelle à la tension d'alimentation ($V \propto f_{max}$), comme le montre l'équation 2.2. Plus la tension d'alimentation est forte, plus il est possible d'augmenter la fréquence maximale du processeur. La ***puissance dynamique*** évolue alors en $f^3$ ($P \propto V^2xf \propto f^3$) et devient le terme le plus important dans le calcul de la puissance consommée par le processeur. La puissance de calcul d'un processeur fonctionnant à la fréquence f peut être comparée à celle de deux processeurs fonctionnant à la fréquence $\frac{f}{2}$. Pour schématiser, dans le cas des deux processeurs, la consommation de chacun d'eux évolue en $\frac{f^3}{2^3}$, soit une division par 8 de la consommation. Etant donné que nous avons deux processeurs, l'activité A de l'équation 2.1 est doublée. On peut donc conclure qu'utiliser deux processeurs en utilisant une fréquence $\frac{f}{2}$ apporte la même puissance de calcul qu'un processeur fonctionnant à la fréquence f, mais permet de gagner un facteur d'ordre 4 en consommation. Même si cette explication est simpliste, elle met en évidence l'intérêt de systèmes multicœurs pour le domaine de l'embarqué : **pour respecter les limitations en consommation, il est préférable d'utiliser plusieurs cœurs cadencés à des fréquences moindres qu'un unique cœur cadencé à une forte fréquence.**

Cette solution logique trouve sa limite dans la conception même d'un système multicœur : il est beaucoup plus complexe de programmer plusieurs cœurs qu'un seul. Plus généralement, la technologie des composants évolue plus rapidement que les méthodes de conception, ce qui correspond à l'intervalle de productivité (***productivity gap***) évoqué par l'ITRS [ITR07]. Ceci a d'ailleurs été la principale critique faite lors du lancement des processeurs TMS320C8x à la fin des années 1990. A cette époque, le fait de programmer les 4 cœurs de ce composant restait complexe et seuls des spécialistes étaient capables de tirer parti des ressources de ce composant. Depuis 2005, les processeurs standard pour ordinateurs sont également multicœurs. La différence avec les TMS320C8x se situe dans le fait qu'une méthode de conception basée sur les tâches (Threads) permet aux programmeurs de faire abstraction des aspects multicœurs dans la conception de leurs applications. L'augmentation du parallélisme de calcul est également à la base des architectures DSP, des systèmes sur puce et des architectures GPU mais régis selon des principes différents.

Nous aborderons donc dans ce chapitre les principes de ces différents composants ainsi que les solutions de programmation associées. Lors de la conception d'un système, le choix de la cible matérielle est en effet crucial pour respecter le cahier des charges sur les plans suivants : coût, consommation, encombrement et performances. L'analyse de ces contraintes aboutit à la sélection d'un ou de plusieurs composants, ce qui implique la sélection d'un langage de programmation approprié. Ce chapitre vise à préciser l'origine de ces langages ainsi que leurs limitations.

Assez classiquement, un processeur sera programmé à l'aide d'un langage de programmation logiciel, en utilisant éventuellement les services d'un système d'exploitation temps réel. Nous nous intéresserons plus précisément aux processeurs spécifiques. Cette catégorie comprend deux types de composants que sont les processeurs de traitement du signal (DSP) et les processeurs graphiques (GPU). Ils sont tous conçus pour répondre à des exigences particulières liées à des domaines applicatifs exigeants que sont le traitement du signal et les applications graphiques. D'une part, chacun de ces deux domaines possède des calculs typiques que les processeurs spécifiques optimisent. D'autre part, et cela constitue certainement la plus grande différence entre ces deux domaines, les applications de traitement du signal font partie du domaine de l'embarqué à l'inverse des applications graphiques (les GPU ont été créés pour les cartes graphiques utilisées dans les ordinateurs, notamment pour les jeux vidéo 3D). Même si le domaine applicatif que nous traitons est le traitement des images, le caractère embarqué que nous visons implique un intérêt pour les DSP plus que pour les GPU.

A la suite cette partie sur les processeurs, ce chapitre présentera les composants de logique programmable (FPGA) et les techniques de programmation associées. Nous verrons enfin que ces composants peuvent être associés dans des systèmes complets appelés MPSoC. La conception et la programmation de ces systèmes nécessitent une méthodologie de conception complète qui à l'heure actuelle trouve encore ses limites.

## 2.2 Les processeurs standards

### 2.2.1 Architecture interne

Qu'il s'agisse de GPP (General Purpose Processor) qui équipent la plupart des ordinateurs ou de processeurs spécifiques que nous aborderons plus tard, les processeurs sont caractérisés par :

- l'Unité Arithmétique et Logique (UAL), élément permettant de réaliser des calculs arithmétiques et logiques élémentaires (additions, soustractions, décalages logiques ...),
- d'une mémoire contenant un programme à exécuter ainsi que des données,
- l'unité de contrôle, également appelée séquenceur, qui contrôle l'ensemble du processeur pour réaliser l'application. Son rôle consiste à lire une instruction dans la mémoire programme, de configurer l'UAL pour la réalisation de l'instruction, d'alimenter l'UAL avec les données nécessaires à la réalisation de l'instruction, et enfin de sauvegarder en mémoire de données le résultat du calcul. Cette unité de contrôle est une machine d'état associée à des registres (classiquement : compteur de programme, accumulateur, registre d'adresse et registre d'état, pointeur de pile),
- d'unités d'entrées/sorties permettant l'échange de données entre le processeur et le monde extérieur.

L'unité de contrôle réalise des calculs les uns à la suite des autres : on parle des processeurs comme des architectures de calcul séquentielles. L'augmentation des fréquences était la principale

évolution des GPP depuis la création des processeurs jusqu'à 2005 (premiers bi-cœurs d'Intel et d'AMD). L'idée du parallélisme a été quant à elle à l'origine des DSP : plutôt que d'augmenter la fréquence des horloges, synonyme de forte consommation, les concepteurs de DSP ont travaillé sur l'architecture interne du composant pour réaliser plus de calculs par cycle horloge. Les techniques utilisées à cet effet sont le mécanisme de pipeline et la mise en œuvre d'UAL capables de réaliser plusieurs instructions simultanément (architectures VLIW pour Very Long Instruction Word). Ces dernières architectures restent cependant basées sur l'utilisation d'une unique unité de contrôle et l'on parle toujours d'architectures séquentielles.

### 2.2.2 Langages de programmation

Pour utiliser ces composants, il faut créer le programme qui sera chargé en mémoire du composant et appelé microprogramme. Pour cela, plusieurs langages peuvent être utilisés : assembleur, langage C, C++, Java, OCaml ... Certains sont très proches des microprogrammes (assembleur), d'autres s'en éloignent (langages dits de haut niveau) pour faciliter la création d'applications et augmenter leur généricité (un microprogramme étant spécifique à un processeur). Des langages comme C++, Java ou OCaml sont des langages orientés objet qui permettent par exemple d'utiliser des modèles de programmation référencés (***Design pattern***) [GHJV94].

La phase de compilation va créer le microprogramme à partir des programmes écrits dans ces langages. Les environnements de programmation sont conçus pour créer et éditer les programmes, pour les compiler et bien souvent pour les exécuter et/ou pour les simuler. Les ***debuggers*** permettent d'exécuter les programmes en mode pas à pas pour trouver d'éventuelles erreurs de programmation. Tous ces langages sont utilisés pour spécifier la séquence des calculs qui sera exécutée par le processeur. On parle de langages séquentiels. Dans le cas d'un DSP, le compilateur devra de plus analyser le programme pour faire fonctionner ses UAL en parallèle si cela est possible. Le parallélisme est analysé dès la phase de compilation, lors de la création du microprogramme. Néanmoins, l'exécution de l'application reste là encore séquentielle. Ce type de programmation est bien adapté dans le cas d'une exécution monoprocesseur. En revanche, les langages orientés objet sont peu utilisés dans le domaine de l'embarqué car les compilateurs associés n'offrent pas des possibilités d'optimisation assez importantes.

### 2.2.3 Utilisation de systèmes d'exploitation

Une autre approche pour la programmation sur processeurs consiste à utiliser un système d'exploitation (OS pour Operating System). Cette solution vise à élever le niveau de représentation de l'application. L'application est alors composée de tâches (en anglais Thread ou Task), chaque tâche étant constituée d'un programme séquentiel. Plusieurs tâches peuvent être décrites pour constituer une application et donc potentiellement exécutées en même temps. Il n'est alors plus possible de réaliser l'exécution de ces tâches directement par le séquenceur d'un processeur. L'OS est alors une surcouche logicielle entre l'application et le matériel comme illustré sur la figure 2.1.

Plus généralement, l'OS a pour rôle de masquer à l'application les particularités du matériel, et de présenter une interface à l'application. L'OS fournit donc des services à l'application en lui permettant de créer des tâches, de les faire communiquer et de les synchroniser. L'OS propose également des outils de gestion du temps, de la mémoire et des périphériques.

L'idée principale des OS est donc de pouvoir gérer des tâches concurrentes. Pour cela, chaque tâche possède une structure de données comprenant un code à exécuter, des données et les valeurs

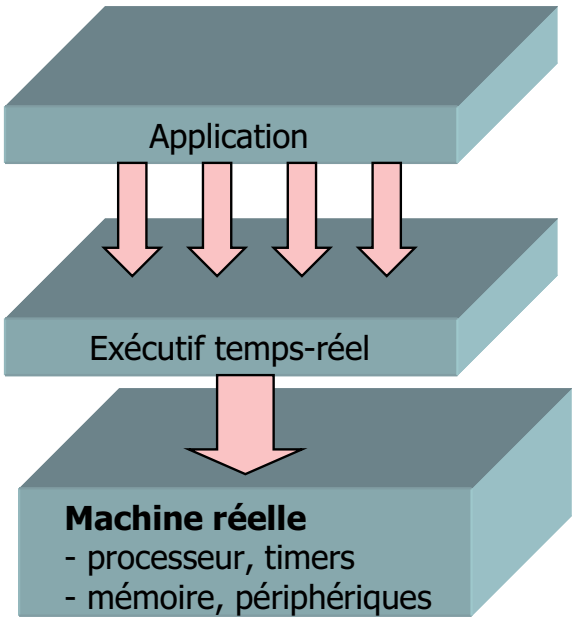



FIGURE 2.1: Position d'un RTOS

des registres du processeur. L'OS peut alors sélectionner une tâche pour la faire exécuter par le processeur. L'OS doit donc sélectionner une tâche devant être exécutée parmi celles pouvant potentiellement l'être : on parle d'ordonnancement (***scheduling***). Etant donné que cet ordonnancement est réalisé par le processeur lors de l'exécution du programme, on parle d'ordonnancement dynamique, un ordonnancement statique étant fixé dès la phase de compilation du programme. Les ressources de calcul disponibles sont alors partagées entre les tâches et l'OS.

L'avantage dans l'utilisation d'un OS est de pouvoir décrire une application complexe avec un haut niveau d'abstraction, puisqu'il s'agit simplement de manipuler des tâches partageant des données. La description d'une application par un ensemble de tâches communicantes est extrêmement pratique et il est facile d'appréhender le parallélisme potentiel d'une application avec une telle représentation. C'est d'ailleurs cette particularité qui permet d'utiliser une description multitâche dans un contexte multicœur. L'OS va connaître les tâches pouvant être exécutées à chaque instant et va ainsi pouvoir utiliser plusieurs cœurs si plusieurs tâches doivent être exécutées simultanément. Le système réalise un placement (mapping) dynamique des tâches constituant les applications à réaliser.

Par rapport aux OS, les systèmes d'exploitation temps-réel (RTOS) offrent des services supplémentaires afin de garantir des contraintes de temps précises. Les techniques mises en œuvres dans les RTOS s'avèrent plus complexes (préemption, ordonnancement optimisé, ...). Ces techniques apportent une réactivité lors de l'exécution du programme qui reste très fortement liée au contexte d'exécution. D'autre part, les RTOS sont des versions plus optimisées en termes de ressources utilisées. Il faut noter que cette exigence de contrôle des temps fait qu'il est rare d'utiliser plusieurs applications avec un RTOS comme cela peut être le cas avec un OS. Pour cette même raison, les RTOS proposés à l'heure actuelle ne peuvent généralement pas fonctionner en multicœur. Les RTOS sont le plus souvent préférés aux OS dans le cadre de la conception de systèmes embarqués et donc dans le cadre de nos recherches.

Malgré ces avantages incontestables, l'utilisation d'un OS ou d'un RTOS ne règle pas tous les

problèmes, comme le met en évidence Edward A. Lee dans [Lee06]. Voici les principales limitations que nous pouvons citer :

- l'exécution de l'application à l'intérieur d'une tâche reste séquentielle, effectuée par un unique séquenceur de calculs,
- en plus des calculs associés à chaque tâche, le processeur doit prendre en charge ceux spécifiques à l'OS,
- pour créer une solution efficace, le nombre de tâches doit rester limité et chacune d'entre elles doit réaliser un nombre conséquent de calculs (forte ou moyenne granularité),
- l'optimisation d'une application est complexe puisque de nombreux services sont proposés et que l'on peut potentiellement les imbriquer,
- le risque d'interblocages (i.e. plusieurs tâches s'attendant mutuellement) est important,
- la recherche d'erreurs de programmation (***debug***) est complexe puisque l'exécution des tâches varie suivant le contexte d'exécution,
- il n'existe pas à l'heure actuelle d'OS ou de RTOS gérant de manière satisfaisante plusieurs processeurs et/ou des architectures matérielles.

Une grande partie des recherches liées aux OS et RTOS se situe dans l'analyse d'ordonnançabilité. Derrière ce terme se posent deux questions : comment vérifier qu'un ordre d'exécution des tâches sur un processeur respecte des contraintes temporelles fixées, et comment déterminer un ordre optimal. Assez régulièrement, la simulation est utilisée pour vérifier un ordonnancement donné sans réaliser d'analyse mathématique. La simulation est réalisée avec des chronogrammes et permettent de vérifier si chaque tâche du système respecte ses échéances. Cependant, il est difficile de connaître la validité de la simulation lorsque la complexité du système augmente.

Les tests d'ordonnançabilité peuvent alors être utilisés. Ces formules mathématiques vont donner à partir des caractéristiques des tâches (période, temps de calcul, échéance) et de l'ordonnancement utilisé un résultat binaire sur la validité du système. On peut citer les travaux de Liu et Layland [LL73] sur les ordonnancements RMA (*Rate Monotonic Analysis*) et EDF (*Earliest Deadline First*), de Leung et Whitehead sur les ordonnancements DM (*Deadline Monotonic*) [LW82], de Mok et Dertouzos sur les ordonnancements LLF (*Least Laxity First*). Ces derniers ont notamment prouvé qu'il n'existait pas a priori d'ordonnancement optimal dans un contexte multiprocesseur avec préemptions [DM89]. Les travaux les plus aboutis semblent ceux effectués sur l'analyse des temps de réponse (*Response Time Analysis*, RTA) [JP86, Aud91].

Là encore, les tests d'ordonnançabilité trouvent leurs limites : ils ne couvrent pas tous les modèles de systèmes potentiellement décrits à partir de tâches. Généralement les systèmes pris en charge dans les analyses d'ordonnançabilité sont constitués de tâches périodiques (avec échéance sur requête ou non), avec préemptions, dans un cadre monoprocesseur et sans synchronisation entre tâches. Seuls quelques travaux comme l'héritage de priorité [SRL90] traitent de dépendances de données entre tâches. Les systèmes multiprocesseurs, avec dépendances de données et synchronisations impliquent une phase de placement et aboutissent à un problème d'ordonnançabilité trop complexe. D'autre part, en cas de non-faisabilité, les tests d'ordonnançabilité ne donnent pas d'indication sur les tâches qui posent problèmes.

### 2.2.4 OpenMP

OpenMP [opec] est un ensemble de directives de compilation pour paralléliser un code sur une architecture multiprocesseur. Cette initiative, lancée en 2008 par des grands groupes du domaine de l'informatique (Intel, IBM, AMD, SUN, Microsoft), est similaire aux travaux menés autours

de CILK [BJK+95] ou par l'équipe CAPS [BM01] de l'IRISA. Aujourd'hui, le jeu de directives OpenMP est amplanté pour trois langages (C, C++ et Fortran). Il est reconnu par les compilateurs de processeurs standards comme GNU GCC, le compilateur Intel, IBM ou Sun. Le principal atout d'OpenMP est sa simplicité de mise en œuvre : il suffit d'intégrer dans son code source des directives aux endroits où l'on souhaite utiliser du parallélisme. Toute la gestion des tâches et de la mémoire permettant de mettre en œuvre le parallélisme est transparente pour le programmeur. Le compilateur interprète ensuite les directives s'il en est capable, sinon le programme reste séquentiel.

Le modèle d'exécution est dit ***fork and join***, que l'on peut traduire par ***éclatement et regroupement***. Il est illustré sur la figure 2.2. Pour OpenMP, toute application compte au moins un Thread (Tâche maître, Master Thread) qui est le programme dans son exécution séquentielle. Quand le code arrive sur une directive OpenMP, d'autres tâches sont créées. La tâche maître a le contrôle sur les tâches créées, et les détruit à la fin de la section parallèle. Nous sommes donc en présence de tâches qui doivent être gérées par un OS ou un RTOS capable d'exécuter ces tâches créées dynamiquement et cela sur plusieurs processeurs. OpenMP s'utilise comme une surcouche logicielle aux OS (Windows et Linux actuellement).

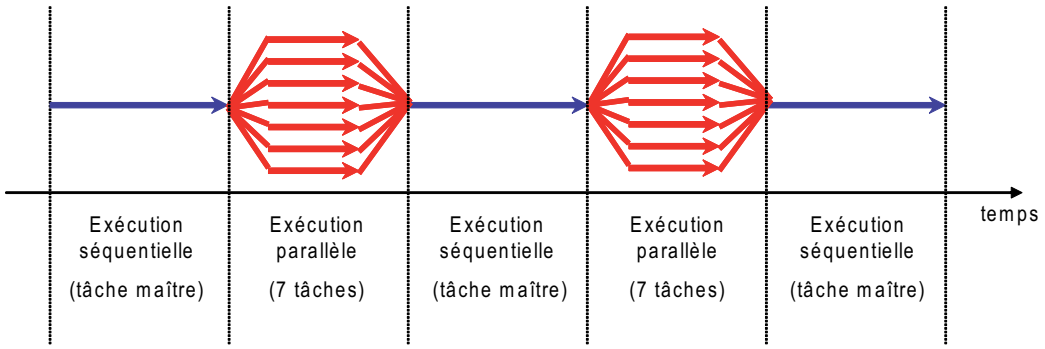



FIGURE 2.2: Modèle d'exécution d'un programme OpenMP

Chaque directive placée dans le code source signale la présence d'une section parallèlisable. La plupart des directives OpenMP supportent des clauses qui permettent de spécifier des options supplémentaires, comme les variables privées ou partagées entre tâches. Un exemple est donné dans la figure 2.3. Ce code se décrypte de la façon suivante :

- le ***#pragma omp*** signale une directive OpenMP,
- ***for*** est le nom de la directive OpenMP,
- ***private(j)*** est une clause permettant de spécifier que chaque tâche OpenMP doit avoir sa propre instance de la variable ***j***,
- et ***lastprivate (Array)*** est une clause permettant de spécifier que chaque tâche OpenMP doit avoir sa propre instance (dite privée) de la variable ***Array*** et que la variable ***Array*** est remise à jour par les instances privées à la fin de chaque itération de la boucle ***for*** et en fin de la section parallélisable.

Le compilateur va créer des tâches dynamiques qui prendront en charge une partie des itérations de la boucle, ainsi que des variables temporaires et des recopies entre variables si nécessaire. Les clauses sont primordiales pour que le compilateur connaisse les dépendances de données entre itérations de la boucle, ce qui va influer directement sur les données à transférer entre processeurs si ceux-ci exécutent des instances distinctes de la boucle. Le placement/ordonnancement des tâches créées dynamiquement et donc l'efficacité de l'exécution sont délégués à un système d'exploitation.

```
int j;
#pragma omp for private(j) lastprivate(Array)
for(j=0; j<size; j++)
{
   Array[j] = j * j + 2;
}
```

FIGURE 2.3: Exemple de code intégrant une primitive OpenMP

Cette approche de programmation offre comme principal avantage une grande simplicité d'utilisation. Cette technique d'insertion de primitives est classique dans le domaine de la programmation des processeurs et ne demande pas de remise en question de la méthode de conception. De plus, il est possible de réutiliser l'ensemble des développements déjà réalisés en langage C.

Cependant, si OpenMP permet l'optimisation de boucles en multiprocesseur, il faut tout de même remarquer qu'il n'est pas en mesure de gérer des boucles récursives qui induisent des transferts de données entre deux itérations de la boucle et donc entre tâches placées éventuellement sur des processeurs différents. De plus, il ne semble pas possible d'exécuter une nouvelle directive si la précédente n'est pas terminée. Un code comprenant plusieurs boucles successives ne sera pas forcément optimisé. D'autre part, cette approche demande une étude fine pour extraire le parallélisme d'un algorithme qui peut être très parallèle par nature, mais qui a été rendu séquentiel dans sa phase de programmation en langage C. OpenMP est une approche récente et de nombreux tests doivent être réalisés sur des applications réelles pour juger de son efficacité réelle, notamment en ce qui concerne les temps de gestion des tâches et les transferts de données qui ne doivent pas être trop longs par rapport aux temps de calculs. Enfin, pour le multicœur embarqué, il n'existe pas encore de RTOS capable de supporter les primitives OpenMP.

## 2.3 Les processeurs graphiques

Les processeurs graphiques (GPU : ***Graphic Processing Unit***) actuels offrent une très grande puissance de calcul, à un coût relativement réduit. En effet, l'évolution de ces processeurs est telle que leur puissance dépasse largement celle des CPU. De plus, ils sont associés à des mémoires très rapides, offrant une grande bande passante. Depuis quelques années, il est possible de programmer les unités de calcul des GPU et de détourner leur finalité première pour faire du calcul numérique. Ce concept est connu sous le nom de GPGPU (***General Purpose computation on GPU***). Un état de l'art des développements de calcul numérique sur GPU est disponible dans [OLG+05], et un exemple d'application d'estimation de mouvement issue de l'analyse de vidéo dans [KK04].

### 2.3.1 Architecture interne

L'architecture des GPU est très spécialisée et la plupart des opérations sont figées. Les données sont traitées massivement en parallèle en appliquant la même opération, dans une unité dédiée, à toutes les données (modèle flot/noyau ou stream processing). C'est cette spécificité qui permet au GPU d'avoir une puissance de calcul très élevée. L'architecture d'un GPU intègre traditionnellement un pipeline de calcul (Fig. 2.4).

Le processeur de nœuds (Vertex processor) calcule des transformations sur des points (position

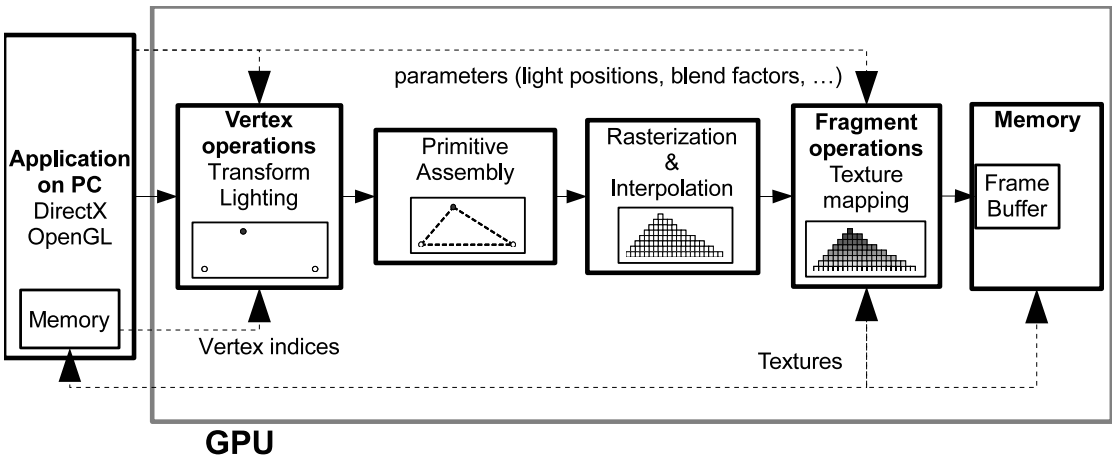



FIGURE 2.4: Architecture classique d'un GPU

3D, éclairage, ...) qui sont ensuite assemblés sous forme de triangles, puis transformés en objets affichables (Rasterization). Le processeur de fragments applique la texture sur les objets et calcule le rendu final. La puissance de calcul vient de la spécificité des traitements effectués. Les opérations de contrôle sont très réduites et chaque type d'opération est réalisé dans une unité dédiée.

Dans ce modèle d'architecture, le processeur de nœuds et le processeur de fragments sont programmables. Initialement introduits pour créer des effets de rendu personnalisés, les possibilités de programmation étendent l'utilisation des processeurs graphiques à d'autres types de calcul. Les programmes, appelés ***shaders***, sont donc appliqués sur les nœuds et les pixels.

La dernière génération de processeurs graphiques de chez NVIDIA (GeForce 8X [NVi06] et Quadro 5600) est basée sur une nouvelle architecture appelée ***Unified shader architecture*** (Fig. 2.5). Le but est de réduire les étages de pipeline, d'augmenter la flexibilité et les performances. Il n'y a donc plus qu'un seul type de processeur : le ***unified stream processor*** privilégie les boucles plutôt que la séquentialité. Le flot des données passe donc plusieurs fois à travers les processeurs suivant les opérations à effectuer, ce qui a pour effet d'augmenter l'efficacité globale : suivant les calculs à réaliser, la charge de calcul n'est plus déséquilibrée sur les différentes unités.

Chaque processeur est capable d'effectuer des opérations scalaires et vectorielles. Un contrôleur (***Thread Processor***) distribue les opérations sur les différents processeurs pour les garder actifs.

### 2.3.2 Langages de programmation

La programmation est limitée par la spécialisation des processeurs graphiques : par exemple, les opérations logiques bit à bit et de décalage n'existent pas et les applications de calcul numérique doivent être encapsulées dans du calcul graphique pour pouvoir être exécutées sur GPU. Les outils de programmation haut niveau pour GPU tels que CG (C for Graphics) ou GLSL (OpenGL shading Language) nécessitent également une maîtrise de l'interface graphique (DirectX ou OpenGL). Les opérations réellement effectuées sont masquées par l'interface graphique, ce qui ne donne qu'un contrôle limité de l'implantation. De plus, d'après quelques tests réalisés lors de la thèse de Fabrice Urban que j'ai encadrée, la même application ne donnait pas les mêmes résultats sur deux processeurs graphiques différents.

Nvidia fournit dorénavant un kit de développement basé sur le langage C appelé CUDA (Com-

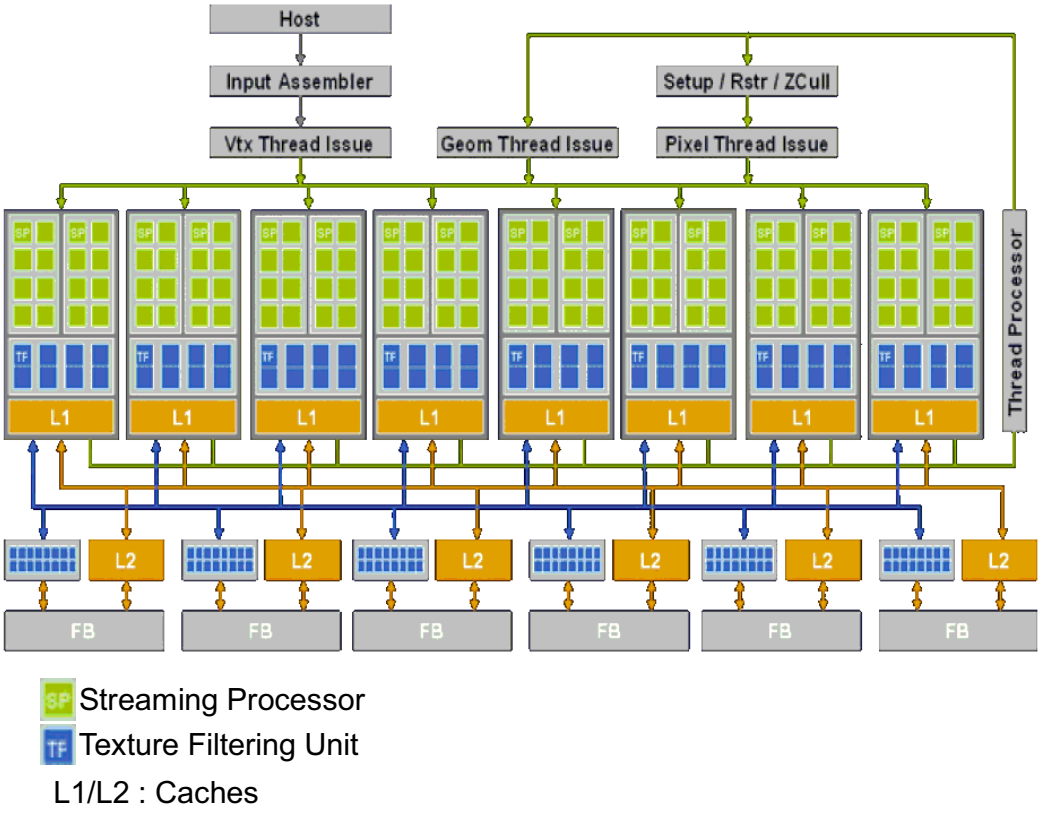



FIGURE 2.5: Diagramme blocs du GeForce 8800 GTX

pute Unified Device Architecture). La première version de CUDA a été proposée en février 2007 et cette technologie évolue encore énormément. Alors que DirectX et OpenGL étaient dédiés aux applications graphiques, l'objectif de CUDA est de rendre possible l'utilisation des cartes graphiques pour des applications non graphiques (GPGPU).

CUDA et son interface de programmation (***Application Programming Interface*** ou ***API***) permettent d'utiliser la carte graphique comme un coprocesseur pour le processeur d'un ordinateur. La carte graphique est alors capable d'exécuter un grand nombre de tâches en parallèle. L'architecture de la carte graphique est ainsi vue comme plusieurs multiprocesseurs (Fig. 2.6(a)) se partageant trois types de mémoires :

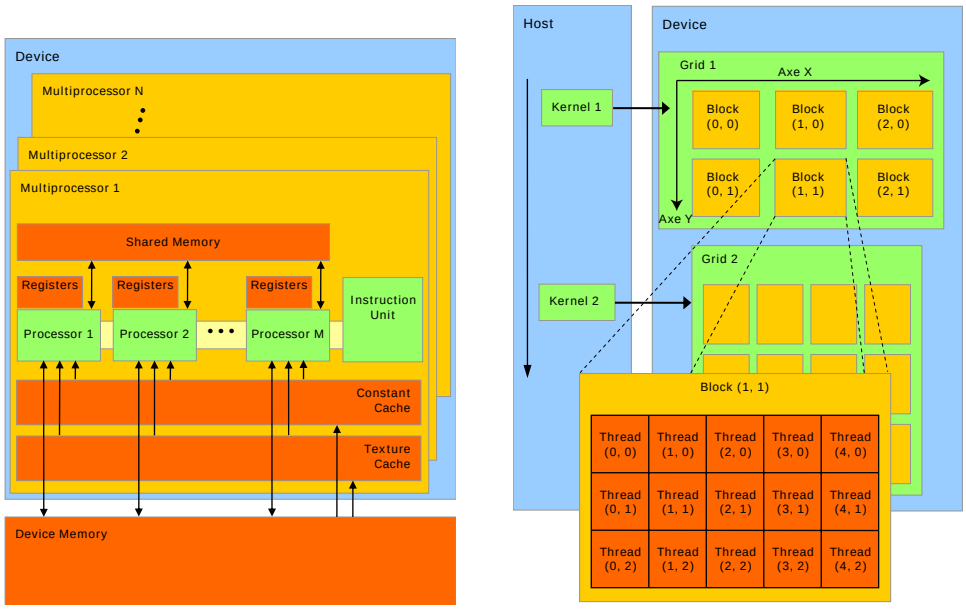
- la mémoire globale (accessible en lecture et en écriture),
- la mémoire de constantes (accessible en lecture),
- la mémoire de texture (accessible en lecture).

Chaque processeur est constitué :

- de registres (dont le nombre dépend de la carte graphique),
- d'une mémoire partagée permettant de partager des données avec les autres processeurs d'un même multiprocesseur,
- d'un cache de constantes,
- d'un cache de texture.
- d'un accès à la mémoire globale, sans mécanisme de cache.

Les multiprocesseurs sont indépendants les uns des autres. Un multiprocesseur ne peut utiliser

ni les caches, ni les registres, ni la mémoire partagée d'un autre multiprocesseur. L'utilisateur doit alors utiliser l'API CUDA similaire au langage C pour organiser ses calculs et organiser les différents types de mémoires.




(a) (b)

FIGURE 2.6: (a) Modèle d'architecture en CUDA ; (b) Modèle d'exécution en CUDA

L'architecture de chaque multiprocesseur est un SIMD (Single Instruction, Multiple Data). Le principe de la programmation est alors d'exécuter plusieurs fois un même calcul appelé noyau (***kernel***) sur des données distinctes. Un noyau est constitué de plusieurs grilles (***grid***) comme illustré figure 2.6(b). Chaque grille est elle-même constituée d'une matrice 2D de blocs et chaque bloc est constitué de tâches. Chaque bloc peut contenir jusqu'à 512 tâches divisées en plusieurs dimensions (jusqu'à 3).

Un bloc de tâches est exécuté sur un multiprocesseur. Chaque bloc dispose donc des ressources matérielles d'un multiprocesseur et chaque tâche celles d'un des processeurs. Il est ainsi possible de découper de plusieurs façons une même application.

La programmation CUDA peut être illustrée sur des exemples simples. La figure 2.7 montre deux boucles. La première présente une dépendance de données entre itérations et sera difficile à paralléliser. La seconde traite des données distinctes dans une même itération, sans dépendance d'une itération à la suivante et pourra donc être portée sur le GPU. Pour exécuter cette seconde boucle, il est possible de la diviser les *size* itérations. Par exemple, si l'on sait que $size = 10\,000$, on pourra choisir de diviser cette boucle en 20 blocs de 500 tâches. Si l'on utilise une grille de 5 blocs sur 4 et des blocs de 20 tâches sur 25, il faudra modifier le code initial pour aboutir au code de la figure 2.8. Il faut noter que ce code est moins générique (la valeur de *size* est fixée), que les choix dans la division de la boucle en blocs et en tâches sont fixés à la compilation et qu'ils doivent être validés par des tests de performance sur une carte donnée.

Lorsqu'un noyau est exécuté sur le GPU, il y a auparavant une phase de recopie des données dans la mémoire du GPU. La mémorisation du résultat implique également une recopie des données calculées par le GPU dans la mémoire adressée par le CPU. Ces transferts de données passent par

```
float *tab1, *tab2, *tab3 ;
int i, j;

for(i=0; i<size; i++)
      tab3[i+1] = fabs( tab3[i] + tab2[i] );

for(j=0; j<size; j++)
      tab1[i] = tab3[i] + tab2[i];
```

FIGURE 2.7: Exemple de boucles

le bus PCI qui permet de connecter le CPU au GPU. Il est impératif de diminuer au maximum ces transferts de données sous peine de perdre toute accélération. Il est également important de traiter des données dites ***coalescées***, c'est à dire qui sont alignées sur 64 octets dans la mémoire programme et qui seront traitées par des tâches distinctes dans un ordre croissant.

Les applications que nous voulons traiter sont principalement des algorithmes de compression / décompression vidéo. Ces applications combinent des calculs intensifs sur un ensemble de données réduit (bloc, macrobloc), des opérations de décision (tests et conditionnement) et des dépendances de données (prédiction). La parallélisation des traitements sur des architectures massivement parallèles peut s'avérer difficile. Les opérations de contrôle et les dépendances de données peuvent rendre les architectures des GPU inefficaces. En effet, ces opérations de contrôle et les dépendances de données vont générer de nombreux transferts de données sur le bus PCI entre le CPU et le GPU et créer un véritable goulot d'étranglement.

Les processeurs graphiques se présentent comme une alternative intéressante pour le calcul intensif. La puissance de calcul évolue beaucoup plus rapidement que celle des GPP, et offre une puissance de calcul importante à un coût relativement réduit. Cependant, l'approche GPU manque de généricité et est complexe à mettre en œuvre. Les modèles de programmation sont récents et évoluent rapidement. Les outils de programmation haut niveau sont peu nombreux et requièrent une maîtrise du développement graphique. L'architecture de pipeline graphique fixé est très contraignante pour le calcul numérique et sa complexité a pour conséquence une consommation élevée (entre 200 et 400 Watts selon le modèle utilisé). Cette approche ne peut alors être retenue dans un contexte embarqué.

## 2.4 Les processeurs de traitement du signal

Pour satisfaire à des contraintes de consommation importante tout en conservant une puissance de calcul élevée, les DSP (Digital Signal Processor) sont généralement utilisés. On retrouve ces composants dans tous les appareils nomades pour la vidéo (lectures MP4, téléphones portables). Les DSP sont des processeurs simplifiés dédiés aux applications de traitement du signal. Leur architecture interne est dimensionnée pour le calcul intensif. Suivant les modèles, ils permettent de réaliser des opérations sur des nombres soit à virgule fixe, soit à virgule flottante. En effet les unités arithmétiques et logiques ne sont pas construites de la même manière. Le calcul flottant requiert une architecture plus complexe qui conduit à une fréquence de fonctionnement généralement plus faible. Par conséquent, les flottants sont utilisés seulement lorsqu'une grande précision et une grande dynamique sont requises sur les valeurs traitées. Cependant, au besoin, une unité de calcul

```
// définition du noyau
__global__ void kernel(float *tab1, *tab2, *tab3){
      // coordonnées de la tâche sur l'axe x de la grille
      unsigned int x = blocIdx.x * blockDim.x + threadIdx.x;
      // coordonnées de la tâche sur l'axe y de la grille
      unsigned int y = blocIdx.y * blockDim.y + threadIdx.y;
      // nombre de tâches sur une ligne de la grille
      unsigned int nb_threads_per_row = blockDim.x ;
      // coordonnée d'une tâche
      unsigned int i = x + y * nb_threads_per_row;

      tab1[i] = tab3[i] + tab2[i];
}

// dans le programme appelant
{
      // initialisation de la taille des blocs
      dim3 dimbloc(20,25);
      // initialisation de la taille de la grille
      dim3 dimgrid(5,4);
      // appel du noyau
      kernel <<<dimgrid,dimbloc>>>(tab1, tab2, tab3);
}
```

FIGURE 2.8: Exemple de boucle avec l'utilisation de CUDA

à virgule fixe peut émuler des calculs à virgule flottante, mais avec de très faibles performances. Nous prendrons ici l'exemple des DSP Texas Instruments car ce sont les DSP les plus vendus, tout particulièrement dans le domaine de la téléphonie mobile et du traitement des images. De plus, et ceci explique certainement leur succès, la suite de programmation fournie avec les composants est la plus aboutie.

### 2.4.1 Architecture interne

La logique interne d'un cœur DSP est réduite et dédiée au calcul arithmétique et logique, et la logique de contrôle est réduite (par exemple on ne retrouve pas d'unité de réordonnancement comme sur le processeurs classiques). Le pipeline est très court : les opérations telles que l'addition et la multiplication sont exécutées en un cycle. La figure 2.9 montre l'architecture interne du DSP Texas Instruments TMS320TCI6486, qui est le dernier DSP proposé par Texas Instruments créé pour les infrastructures de télécommunication. Ce DSP possède six cœurs C64x+ (en vert sur la figure).

Le C64x+ (en rouge sur la figure) possède deux unités de calcul (A et B) qui ont chacun quatre unités (L, S, M et D). Huit instructions peuvent donc être exécutées en un cycle. L'architecture VLIW (Very Long Instruction Word) lit des instructions de 256 bits pour fournir à chaque cycle jusqu'à huit instructions 32 bits aux huit unités fonctionnelles. Un ordonnanceur permet d'affecter aux huit unités de calcul (sous forme non condensée, on parle d'***execute packet***) les instructions

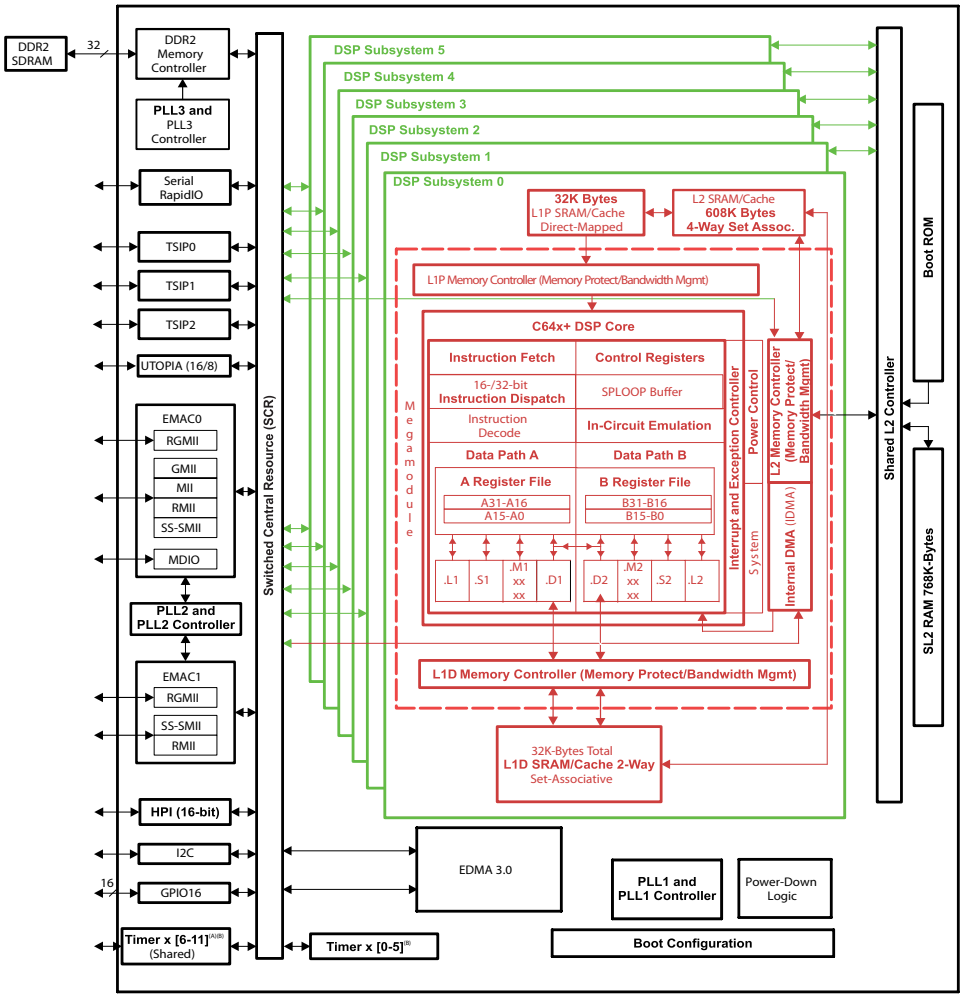



FIGURE 2.9: Architecture du C6486

contenues dans la mémoire programme sous une forme condensée (appelé ***fetch packet***). L'ordonnanceur est statique dans le sens où il effectue des affectations choisies lors de la phase de compilation. Le C64x+ permet également de condenser en mémoire programme la représentation des boucles.

En plus des opérations standard (addition, multiplication), des instructions spécialisées (comptage de bits, rotations, ...) sont implantées. Les unités de calcul 32 bits du C64x+ permettent d'effectuer une opération 32 bits, deux opérations 16 bits ou quatre opérations 8 bits, grâce aux instructions SIMD. Les unités fonctionnelles peuvent donc être exploitées de manière intensive en utilisant toute la largeur disponible.

Le TMS320TCI6486 embarque des périphériques dédiés au domaine du traitement du signal comme le bus TSIP. Des coprocesseurs de décodage Viterbi et Turbo Décodeurs ont également été intégrés dans d'autres DSP Texas Instruments mais n'ont pas été retenus dans cette version. Les transferts entre les périphériques et la mémoire, et entre la mémoire interne et la mémoire externe, sont réalisés par le DMA (Direct Memory Access), qui permet de décharger le cœur de calcul, et de paralléliser accès mémoire et traitements. Des interfaces séries, PCI et mémoires sont

également intégrées au DSP.

Pour pouvoir traiter le domaine de la vidéo embarqué, les DSP Texas Instruments intègrent également plusieurs cœurs : en plus du cœur DSP C64x+ qui a été adapté à la vidéo par rapport au C64x (optimisation des algorithmes de compression/décompression), les composants peuvent posséder également des cœurs ARM permettant l'utilisation d'un Linux embarqué pour les applications de contrôle, ainsi que des périphériques vidéo. Les cœurs ARM sont développés par la société ARM [ARM] comme des IP (*Intellectual Property*). La société ne vend pas de composants mais fournit ses IP de processeurs à des sociétés créant des composants spécifiques pour des applications données comme peut le faire Texas Instruments avec ses OMAP par exemple (Fig. 2.10). Un composant intégrant un cœurs ARM peut bénéficier de l'expérience de la société ARM dans l'architecture de processeurs, d'un compilateur spécifique, d'une suite de programmation (basée sur le compilateur GCC) connue et adaptée, d'une interface de programmation pour les périphériques ainsi que d'un noyau Linux ou WindowsCE (système Windows pour l'embarqué). De nombreux ingénieurs maîtrisent les systèmes embarqués basés sur des cœurs ARM, ce qui facilite le développement d'une application sur ces systèmes. Initialement appelés DaVinci, cette série de DSP Texas Instruments pour la vidéo évolue sous la dénomination OMAP. Les prochains OMAP (OMAP4) annoncés seront équipés de 2 cœurs ARM comme le montre la figure 2.10.

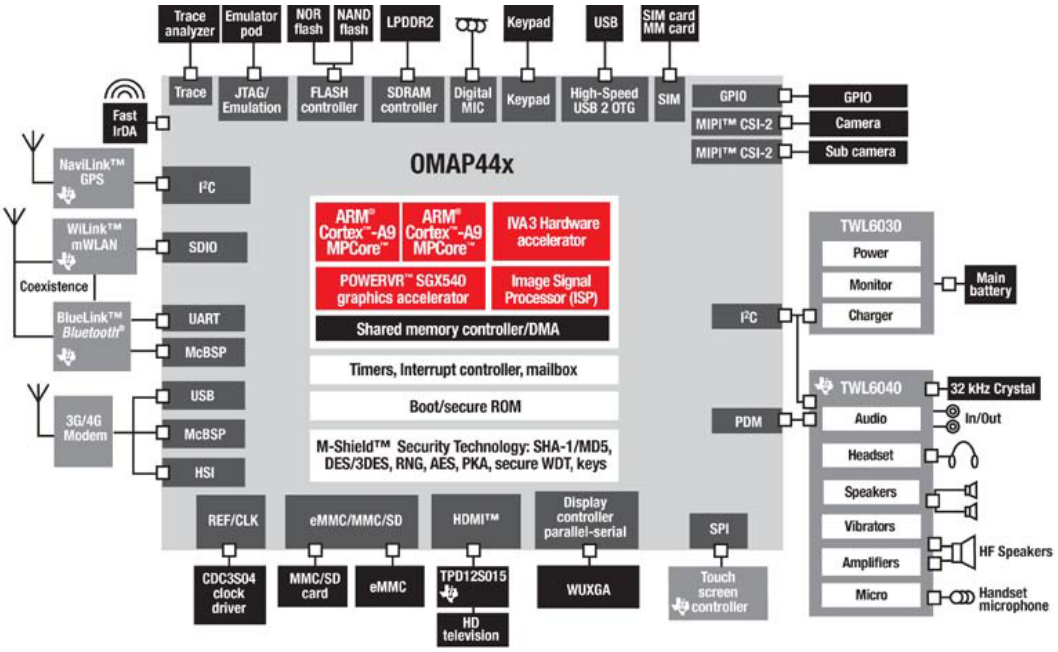



FIGURE 2.10: Architecture d'un OMAP4

L'architecture d'un DSP est dédiée au calcul intensif. Une application avec beaucoup d'opérations de contrôle sera difficilement optimisée, alors qu'un traitement intensif pourra tirer parti du VLIW et du SIMD. De plus, les performances sont déterministes, en contraste avec un GPP qui réalise des exécutions en désordre et avec des prédictions de branchement.

### 2.4.2 Programmation

Les DSP sont accompagnés d'outils de développement performants. La distribution et l'ordonnancement des instructions sur les différentes unités de calcul d'un cœur sont effectués au moment de la compilation. La création du programme condensé en mémoire est également réalisée à la compilation. Pour l'utilisateur, il est possible de programmer le composant en assembleur parallèle, en assembleur linéaire, en langage C et en langage C++, c'est à dire comme un processeur classique. Les débogueurs produisent des informations de bas niveau pour valider rapidement une application.

Les fabricants fournissent en général des compilateurs capables d'optimiser fortement un programme et d'informer le développeur sur le résultat en ajoutant des commentaires dans le programme (boucle optimisée, nombre de cycles, instructions utilisées). Ce retour permet de revenir sur l'écriture du code pour ***aider*** le compilateur en modifiant l'écriture de son programme en langage C ou par l'ajout de commandes en langage assembleur (intrinsèques). Des simulateurs permettent d'obtenir des informations supplémentaires sur l'exécution d'une application (charge de calcul, comportement des mémoires caches, temps d'exécution). Il est également possible d'utiliser des RTOS spécifiques (DSP-BIOS pour les composants Texas Instruments) ou génériques (Linux, RT Linux, MicroC-OSII) si ceux-ci ont été adaptés au DSP.

En revanche, lorsque plusieurs cœurs C64x+ ou ARM sont présents dans une même puce, la programmation se fait sur chacun des cœurs de manière séparée. Les phases de distribution et d'ordonnancement ne sont pas prises en charge ni de manière statique (à la compilation) dans les outils de développement (CCS pour les cœurs C64x, Gcc pour les cœurs ARM), ni de manière dynamique lors de l'exécution (par le RTOS DSP-BIOS). De manière à diminuer les temps de développement, des bibliothèques de fonctions sont fournies gratuitement aux utilisateurs. Ces bibliothèques comprennent des fonctions classiques optimisées pour les C64x+ et des fonctions pour mettre en œuvre de communications entre cœurs ou avec des périphériques. Lorsqu'un processeur ARM est présent, il est possible d'installer une distribution Linux et d'utiliser le cœur DSP comme un coprocesseur. L'ARM est alors utilisé pour gérer les parties de contrôle comme les périphériques et l'interface utilisateur. Pour utiliser le cœur DSP, il faut uniquement remplacer dans un programme fonctionnant sur l'ARM les parties correspondant aux calculs intensifs par des fonctions fournies (par Texas Intruments ou par un de ses partenaires). Dans ces fonctions, les transferts de données de l'ARM vers le DSP et les calculs sur le DSP puis le retour des résultats sur l'ARM sont effectués. L'utilisation de la partie DSP du composant est alors transparente pour le concepteur et les temps de conception sont très rapides. En revanche, il n'est pas simple de faire fonctionner de manière intensive les deux parties du composant (ARM et cœur DSP) puisque le programme sur l'ARM doit attendre les résultats du cœur DSP pour continuer ses calculs dans une application donnée. Le parallélisme peut être bien utilisé si plusieurs applications distinctes fonctionnent simultanément sur l'ARM mais beaucoup moins si une seule application doit utiliser l'ensemble des ressources du composant. Ce problème devient critique lorsqu'une unique application nécessite plusieurs dizaines de cœurs DSP tout en satisfaisant des contraintes de temps et de consommation très strictes, comme c'est le cas pour les stations de base dans le domaine de la téléphonie mobile, cas que nous étudions actuellement avec Texas Instruments avec les algorithmes LTE (cf. chapitre 5).

## 2.5 Les systèmes logiques

Les langages de programmation matériels ont été développés pour décrire des systèmes numériques à base de logique combinatoire, séquentielle et de mémoires. L'objectif est de créer, de simuler, mais également de fournir des descriptions de ces systèmes avant de les fondre (dans le cas de circuits spécifiques dits ASIC) ou de les programmer (dans le cas des composants de logique programmable dits FPGA). Un système logique, souvent appelé IP (***Intellectual Property***), peut être réutilisé dans plusieurs contextes et donc être vendu en tant que tel pour être intégré dans des systèmes plus complets. Il faut donc remarquer que l'ensemble des composants précédemment cités (CPU, DSP, GPU) est développé grâce à des langages de programmation matériels. Ces langages peuvent être spécifiques à un fournisseur de composants, comme l'AHDL de chez Altera, mais généralement, ils s'avèrent plus polyvalents comme le VHDL ou le Verilog.

Les systèmes numériques matériels sont caractérisés par le fait de ne pas être basés sur un séquenceur de calculs, et peuvent ainsi réaliser de nombreux calculs simultanément. Il est alors possible, avec de faibles fréquences, d'exécuter plus de calculs qu'avec un processeur. L'efficacité énergétique (définie comme le rapport entre la puissance de calcul et la consommation) d'un système numérique est alors fortement améliorée par rapport à celle d'un processeur. Pour autant, il faut que l'application visée ne soit pas intrinsèquement séquentielle pour pouvoir bénéficier d'un gain. La généricité d'un processeur est alors plus importante que celle d'un système numérique.

Ces langages matériels permettent de décrire des systèmes parallèles grâce à des descriptions structurelles. Il s'agit de relier entre elles des entités à l'aide de signaux. Chacune des entités décrites correspond à des calculs pouvant être réalisés séquentiellement (dans un ***process***, description comportementale) ou en parallèle. De plus, les langages matériels permettent de manipuler des données au bit près (et non des données dont les tailles sont des multiples d'octet comme sur un processeur), de décrire des structures hiérarchiques, des machines d'état ainsi que de préciser des contraintes de temps (comme pour un RTOS). L'utilisation de structures hiérarchique est très importante car elle permet de mettre en œuvre la réutilisation de blocs (IP).

L'étape de synthèse vise à réaliser les entités sur les ressources logiques du FPGA (à base de petites mémoires appelées des *Look Up Tables* et de bascules pour la mémorisation des résultats), et de définir la connexion entre ces ressources logiques par des ressources de routage. On parle alors de placement/routage. Il existe ainsi un lien assez direct entre les langages de programmation matériels et leur synthèse sur les composants dédiés.

Le parallélisme potentiel d'une application peut être complètement décrit à l'aide d'un langage de programmation matériel. Cependant, ces langages restent complexes à prendre en main, tout comme la validation des fonctionnalités (étapes de simulation à l'aide de chronogrammes). De plus, une solution validée fonctionnellement n'est pas forcément synthétisable, c'est à dire exécutable sur un FPGA. La technique de programmation classique revient à créer d'une part des entités capables d'exécuter des calculs, et d'autre part des entités de contrôle. Ces entités de contrôle génèrent des chronogrammes très précis qui vont permettre d'activer les entités de calcul dans un ordre correct. Très souvent, les temps de propagation des signaux à l'intérieur du composant doivent être pris en compte dans la description des entités de contrôle. Comme ces temps de propagation varient d'un composant à l'autre, il n'est pas rare d'avoir une solution fonctionnant sur un FPGA donné, mais pas sur un autre. Cette approche de conception est réservée à des spécialistes connaissant à la fois les détails des calculs d'une application ainsi que les processus de développement pour composants matériels.

## 2.6 Conception de systèmes

Jusqu'à présent, ce chapitre a montré comment programmer une application sur des composants distincts à l'aide de langages adaptés. Cependant, ces langages ne permettent pas de concevoir des systèmes complets à base de plusieurs processeurs associés à de la logique (programmable ou non). Nous allons maintenant nous intéresser aux outils et aux méthodes pour la conception et à la programmation de tels systèmes.

### 2.6.1 Problématique

La tendance actuelle dans le monde de la logique programmable est d'associer aux ressources logiques des mémoires. Il est alors envisageable d'utiliser ces ressources pour créer des processeurs (Microblazes chez Xilinx, Nios chez Altera). Certains composants comportent même à la fois logique programmable et processeurs (PowerPC chez Xilinx par exemple). Cette combinaison de solutions matérielles est appelée systèmes sur puce (***System-on-Chip***, SoC). Lorsque plusieurs processeurs sont utilisés, on parle de MPSoC (***MultiProcessor System-on-Chip***). Les MPSoC offrent un maximum de flexibilité et d'efficacité pour l'utilisation des ressources disponibles sur un composant. Il est alors possible de créer un composant intégrant un ensemble de ressources de calculs totalement adaptées aux contraintes de l'application. Les possibilités de programmation sont pratiquement infinies alors que l'exploration des solutions pour rechercher la solution optimale demeure extrêmement complexe. Les SoC ou les MPSoC peuvent également être fondus dans un ASIC pour plus de performances (augmentation des fréquences internes, diminution des ressources nécessaires et donc de la taille du composant et de sa consommation) si les quantités de production sont suffisantes. La conception des derniers DSP multicœurs est donc un cas particulier de MPSoC. De manière à rentabiliser la création d'un ASIC, les DSP multicœurs ne visent pas une application spécifique mais plutôt une classe d'applications afin de pouvoir être utilisés dans plusieurs contextes, augmenter le nombre de composants vendus et ainsi atteindre un prix compétitif.

La problématique pour la conception des MPSoC est de connaître et maîtriser l'ensemble des composants disponibles (IP et cœurs de processeurs), de les choisir et les associer à des périphériques pour dédier le MPSoC à une application spécifique. Enfin, les cœurs de processeurs du MPSoC doivent être programmés. Cette problématique est relativement ancienne : elle se posait déjà lorsqu'il n'y avait que des processeurs monocœurs qui devaient être associés à des ASIC sur des cartes électroniques dédiées. Il n'en reste pas moins que la conception de ces systèmes demeure extrêmement complexe.

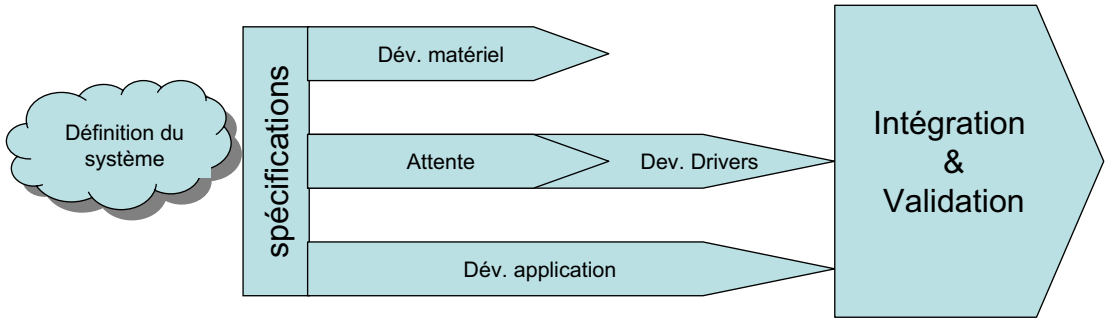



FIGURE 2.11: Flot de conception d'un MPSoC

Le flot de conception classique de ces systèmes est schématisé sur la figure 2.11. Les phases de spécifications initiales sont suivies de trois développements : celui de la plateforme, celui de l'application, et celui des drivers qui sont les bibliothèques logicielles utilisées par le logiciel pour manipuler le matériel. Alors que les développements de l'application et de la plateforme peuvent débuter au même moment, il est nécessaire d'attendre un premier prototype de la plateforme pour développer les drivers. Une fois que ceux-ci sont disponibles, il faut réaliser la phase d'intégration et valider le produit. Ce n'est qu'à cette étape qu'il est possible de vérifier que le cahier des charges a été respecté. Dans le cas contraire, il sera nécessaire de repasser par l'ensemble des étapes de conception pour corriger les erreurs. Ce cycle de développement étant étalé sur plusieurs mois et à raison de plusieurs équipes spécialisées dans leurs domaines respectifs, on comprend dès lors l'importance d'une méthodologie de conception permettant de réduire chacune des phases, de détecter des erreurs de conception le plus tôt possible dans le processus et de les corriger sans devoir reprendre le processus de développement dès le départ.

Une manière de gagner du temps dans le développement de l'application est d'utiliser un langage de programmation unique. Les langages de haut niveau comme Java, CamL ou Matlab sont assez répandus dans le monde logiciel et peuvent avoir les faveurs de leurs spécialistes. Ces langages permettent effectivement de gagner du temps lors de la conception de l'application. Cependant, ces algorithmes devront souvent être repris dans la phase d'intégration pour être adaptés à un processeur embarqué (programmation en assembleur ou langage C). Cette traduction, en plus de prendre du temps, peut aboutir à des erreurs difficiles à détecter. On voit par cet exemple que le problème d'optimisation du processus de conception doit être traité dans sa globalité.

En ce qui concerne le flot de conception classique, le principal inconvénient reste l'étape de spécification initiale suivie des trois phases distinctes effectuées en parallèle. On considère généralement que 80% des choix concernant l'architecture sont réalisés dans les premiers 20% de la durée du projet. La validité des choix effectués ne pourra pourtant être confirmée que dans la phase d'intégration réalisée en fin de projet. Il est donc impératif de pouvoir modéliser et simuler un système complet dès le début du projet et ainsi faire les meilleurs choix possibles dans la phase de spécification. Il n'est cependant pas possible d'exécuter un programme décrit dans un langage de description matériel sur un processeur. Il s'avère alors impossible d'utiliser ces langages pour simuler un système complet décrivant un processeur et le programme qu'il exécute.

Pour lever ces difficultés, il est nécessaire d'utiliser des outils comme MCSE/Cofluent [Cof] ou encore un langage comme SystemC [Sys]. Certains outils comme Poly-Mapper de PolyCore Software [pol] encore PDesigner [Pde] offrent des caractéristiques intéressantes mais leurs fonctionnalités sont plus limitées dans le sens où ils sont spécifiques aux architectures multicœurs. Il faut noter tout de même que Poly-Mapper propose une génération de code qui le différencie de ses concurrents.

Le placement/ordonnancement reste là encore manuel. PDesigner est un outil de simulation et ne peut générer de code, contrairement à Poly-Mapper. Contrairement à un outil comme Cofluent, ces outils visent principalement des architectures multicœurs.

### 2.6.2 MCSE/Cofluent

La méthodologie MCSE, utilisée dans le logiciel Cofluent Studio, a pour objectif d'aider la conception de systèmes complets comme les MPSoC. La méthodologie guide la conception à l'aide d'étapes bien définies comme le montre la figure 2.12.

Les étapes de spécification permettent de spécifier l'architecture cible ainsi que l'application.

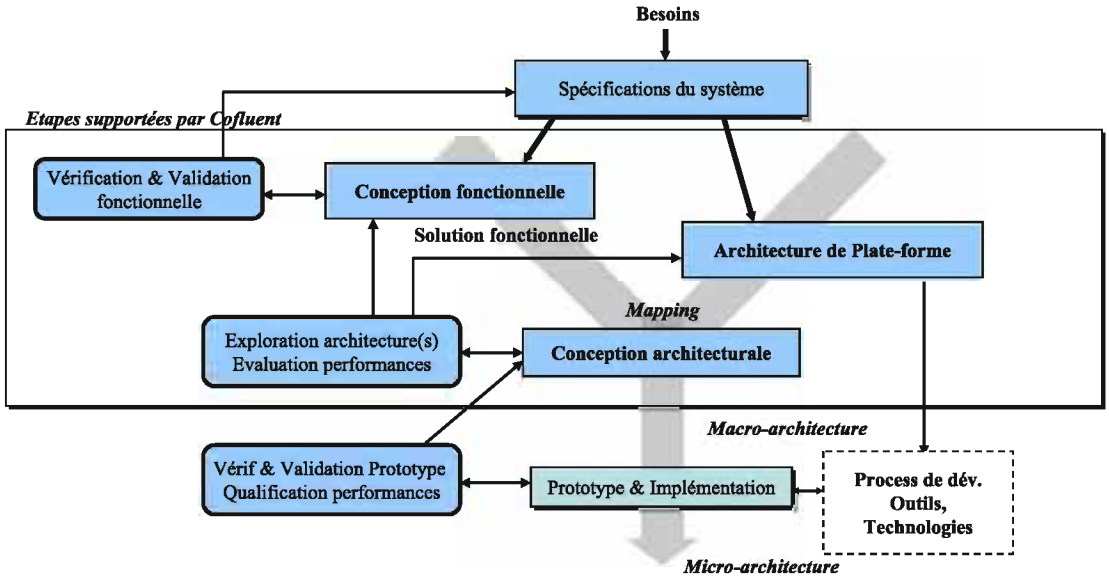



FIGURE 2.12: Flot de conception MCSE

L'application est tout d'abord validée de manière fonctionnelle puis utilisée lors de l'exploration d'architectures. Dans cette phase, l'utilisateur va devoir placer (phase de ***mapping***) les éléments de son application sur les ressources disponibles de son architecture. L'outil Cofluent Studio va ensuite fournir une simulation du système complet. Cette simulation offre diverses représentations (capture d'image, chronogrammes, graphes) afin de valider les choix réalisés dans la spécification et dans la phase de placement. L'outil peut donc être utilisé dès la phase de définition du système pour prédire les performances avant de débuter les développements. Le premier avantage est de réduire les erreurs de développement aussi bien pour le matériel que pour le logiciel, le second avantage est d'accélérer ces phases puisque les comportements logiciels et matériels sont définis très précisément, et qu'il est possible de se référer aux simulations à tout moment dans les développements futurs.

### 2.6.3 SystemC/TLM

Une alternative à la méthodologie MCSE et son outil Cofluent est maintenant possible avec SystemC/TLM. SystemC/TLM font partie des langages SLDL (System Level Design Language) tout comme HandleC ou SpecC. Ils permettent la description des applications matérielles à l'aide du langage C ou C++. Parmi ces langages, HandleC peut de plus être directement utilisé pour la phase de synthèse, mais a le principal inconvénient d'être une technologie propriétaire.

Implanté comme une extension du langage C++, SystemC est quant à lui un standard IEEE (numéro 1666) datant de décembre 2005. SystemC fournit une bibliothèque de classes pour la modélisation, dont le modèle d'exécution est adapté à la simulation de System-On-Chip (SoC) et, potentiellement, de tout système matériel numérique complexe, avec ou sans processeurs et logiciel. La modélisation SystemC couvre les concepts de comportements concurrents, de temps du système simulé et de types de données adaptés à la description du hardware (au bit près), ainsi que la modélisation de structures hiérarchiques. Les modèles en SystemC sont simulables sur station de travail de type PC ou autre, avec des simulateurs *open source* ou commerciaux qui augmentent encore la productivité.

TLM (Transaction Level Modelling), bâti sur SystemC, standardise la modélisation à un niveau d'abstraction plus élevé que le RTL, en fournissant des modèles de communication de structures de données complètes (***transactions***) entre blocs, IPs ou circuits complexes, au lieu des habituels signaux. Le standard TLM de l'Open SystemC Initiative (OSCI) permet ainsi à des concepteurs matériels ou développeurs logiciels de modéliser facilement, et simuler des blocs, IPs, circuits SOC, ou cartes, à l'aide de modules SystemC communiquant par transactions. SystemC/TLM va jusqu'à simuler l'exécution d'un programme particulier sur un processeur ainsi que les transferts de données entre processeurs.

Si SystemC/TLM possède les mêmes avantages que MCSE/Cofluent en termes de simulation de systèmes complexes, MCSE/Cofluent accélère la démarche de conception en définissant des étapes de conception précises définies dans la méthodologie MCSE, et guidées par l'interface graphique de Cofluent. L'interface graphique de Cofluent accélère également la phase de simulation du système.

### 2.6.4 UML, langage de modélisation unifié

Une technique pour développer les programmes complexes est d'utiliser le langage UML (***Unified Modeling Language***). UML est un langage graphique de modélisation des données et des traitements, standardisé par l'Object Management Group (OMG). C'est une formalisation de la modélisation objet utilisée en génie logiciel.

UML se décompose en plusieurs éléments :

- les vues : elles décrivent le système d'un point de vue donné (organisationnel, temporel, architectural, logique, etc). Les vues correspondent aux spécifications du système vues par plusieurs acteurs différents.
- les diagrammes : ce sont les éléments graphiques qui permettent de décrire le contenu des vues. Un diagramme peut faire partie de plusieurs vues. Il existe 13 types de diagrammes différents en UML 2.2.
- les modèles d'éléments : ce sont des briques élémentaires qui permettent de créer des diagrammes (classes, associations, cas d'utilisation (***use case***), etc). Il existe 13 modèles d'éléments disponibles.

Le langage UML donne donc une représentation graphique de l'ensemble des classes qui composent l'application ainsi que leurs interactions par l'intermédiaire de diagrammes de classes. Le principal avantage apporté par l'UML est de permettre la visualisation de la structure du logiciel, en mettant en évidence tout particulièrement les motifs de programmation (***design pattern***) utilisés. Cette visualisation permet de concevoir le logiciel rapidement tout en apportant facilement des modifications. Cette méthode de conception du logiciel évite aux développeurs de commencer un codage puis de devoir réaliser des modifications de code : il est plus facile de modifier le diagramme de classe en utilisant la vision donnée par UML de sa structure. L'utilisation d'UML ne dispense pas d'une conception utilisant un cycle en V. Une fois le diagramme de classe mis au point de manière grossière puis détaillée (descente du cycle en V), il existe une phase de codage pour chacune des classes et fonctions définies dans la phase de conception. Des tests unitaires (remontée du cycle en V) peuvent ensuite être réalisés sur chacune des classes de l'application. Puis la phase d'intégration permet de regrouper les différents modules réalisés avant la phase finale de validation qui consiste à vérifier les fonctionnalités du logiciel par rapport aux exigences initiales.

UML a été étendu pour le domaine du logiciel embarqué temps réel dans ce que l'on appelle le profil UML MARTE (***Modeling and Analysis of Real Time and Embedded systems***) [TGDT07]. Il est alors possible de définir des infrastructures. Ces infrastructures peuvent être

modélisées en définissant des plateformes matérielles et logicielles, ainsi que les caractéristiques des systèmes de calcul et des communications. Les infrastructures peuvent être analysées sous l'angle de performances ou de l'ordonnaçabilité.

Il existe de nombreux outils logiciels de modélisation UML. Ces outils génèrent généralement un squelette de code dans des langages orientés objets comme Java. Ils se prêtent donc bien à la programmation de processeurs. Pour une utilisation en multiprocesseur, un utilisateur pourra définir ses spécifications en UML mais devra réaliser l'interprétation de ses diagrammes dans une phase d'implantation. Rien en UML ne dit en effet comment passer d'un modèle à cette implantation. Il est possible de définir une méthode de conception à partir d'UML en créant des graphes de spécification (diagrammes de cas d'utilisation, de séquence, et d'activité) et pour la conception architecturale (diagrammes de classe, d'objet, de communication, de déploiement et de composant) et ainsi guider la création d'un système, mais les choix devront toujours être réalisés par l'utilisateur en se basant sur ses connaissances et son expérience.

### 2.6.5 Limitations

Si les approches UML, SystemC/TLM et MCSE/Cofluent apportent des solutions à la conception de systèmes complets (sur puce ou non) en diminuant les temps de développement, elles gardent une limitation importante dans le sens où **la phase de placement reste complètement manuelle**. L'utilisateur ne peut donc pas explorer l'ensemble des solutions possibles lorsque l'application devient trop complexe. En effet, les fonctionnalités des modèles supportés par ces outils pour la description des applications sont nombreuses et identiques à celles fournies par un OS temps réel monoprocesseur (tâches, boîtes aux lettres, événements, variables partagées) et affichent donc les mêmes limitations [Lee06], résumées dans la partie 2.2.3. La description d'un système est d'autant plus difficile que l'architecture peut également être décrite de manière très précise. L'espace des solutions à explorer devient trop important et la validation même d'une solution devient problématique. En effet, cette validation passe par l'analyse de chronogrammes avec pour objectif de trouver des interblocages, interblocages rendus possibles par les modèles de descriptions utilisés trop permissifs. Le problème de placement/ordonnancement étant NP complexe, une recherche exhaustive de la solution optimale ne peut être proposée par ces approches. L'utilisateur doit alors être un spécialiste de la conception de systèmes de manière à utiliser son expérience pour réduire l'exploration de l'espace des solutions, en acceptant cependant la phase d'analyse des chronogrammes à chacune de ses modifications de l'application ou de l'architecture. L'utilisation d'une heuristique réduisant le champ d'exploration pourrait être envisagée dans la mesure où les modèles de description de l'application et de l'architecture seraient moins détaillés.

D'autre part, les approches UML, SystemC/TLM et MCSE/Cofluent ne proposent pas de **génération automatique de code multiprocesseurs**, fonctionnalité dont nous reparlerons plus tard dans ce document et qui permet d'accélérer les temps d'intégration une fois les parties logicielles, matérielles et les drivers développés. La phase d'intégration nécessite donc la reprise du code écrit dans le développement logiciel si nécessaire, de programmer chacun des processeurs de manière distincte en respectant les spécifications initiales, puis enfin de réaliser les transferts de données entre processeurs et avec les coprocesseurs dédiés à l'aide des drivers développés pour l'architecture. Ces diverses tâches sont loin d'être triviales car la synchronisation des divers éléments doit être scrupuleusement respectée pour assurer un résultat conforme aux spécifications fonctionnelles initiales.

Les approches UML, SystemC/TLM et MCSE/Cofluent ne remettent pas en question le pro-

cessus de développement global de l'application. Pourtant, certaines données lors de la phase d'intégration peuvent remettre en question les choix réalisés lors de la spécification. Par exemple, des temps de communication ou des temps de calcul peuvent se révéler éloignés de la spécification initiale, ce qui risque d'engendrer des attentes inutiles. D'autre part, la prise en compte de temps chronométrés sur cible auraient éventuellement permis un placement/ordonnancement plus optimisé. La modification d'une partie de l'application ou la prise en compte des temps chronométrés va remettre en cause l'ensemble du processus de développement. Par conséquent, il est extrêmement difficile de tirer parti de l'ensemble des ressources mises à disposition sur les composants les plus récents. Il faut tout de même tempérer ces propos dans le sens où MCSE/Cofluent spécifie des étapes dans la conception. La définition de ces étapes limite les risques d'erreurs ainsi que le temps nécessaire à les corriger. Une utilisation méthodique d'UML permet d'arriver à un résultat comparable.

## 2.7 Conclusion et positionnement

### 2.7.1 Conclusion

Ce chapitre a eu pour objectif de faire le bilan des nouvelles architectures de traitement numérique et de leurs techniques de programmation. Nous avons pu voir que pour augmenter continuellement les puissances de calcul, des composants toujours plus complexes ont été imaginés. Pour exploiter le parallélisme de ces nouveaux composants, de nouvelles techniques de programmation voient le jour mais beaucoup de chemin reste à parcourir. D'ailleurs, l'ITRS fait également ce constat pour fixer une feuille de route illustrée par la figure 2.13, issue du document [ITR07]. La zone bleue correspond aux coûts liés à la conception du matériel et la zone rouge à la conception du logiciel associé. Un compilateur efficace pour architectures embarquées multiprocesseurs n'est prévue que pour 2013, la prise en compte d'architectures hétérogènes (IP, co-processeurs, processeurs ayant des fréquences différentes) en 2015, un processus automatisé en 2019 et enfin une spécification exécutable en 2021. Une spécification exécutable reste un objectif à long terme dans laquelle le document de spécification peut être utilisé durant toute la phase de conception de manière automatique. Le travail de conception est alors réduit à son strict minimum, c'est-à-dire à la rédaction de la spécification. Sans progrès dans la phase de conception, les coûts de conception augmentent puisque la complexité des applications et des architectures augmentent. En revanche, à chacune des étapes fixées par l'ITRS, un gain de productivité a été chiffré. Plus les coûts de conception sont importants, plus il sera difficile de rentabiliser le développement d'un nouveau système et seuls quelques produits de grande consommation pourront en bénéficier. La feuille de route de l'ITRS ne pourra être respectée que si un effort de recherche et de développement considérable dans les techniques de conception est mené.

La principale innovation dans les composants depuis 2005 a été de fournir plus de parallélisme avec les processeurs multicœurs, les cartes graphiques, les DSP ou les MPSoC. Que cela soit OpenMP ou CUDA, l'idée principale pour réduire la complexité de la conception logicielle est d'ajouter des primitives spécifiques dans le langage de programmation séquentiel qu'est le langage C. Alors que OpenMP repose sur l'utilisation d'un système d'exploitation pour processeurs multicœurs, CUDA a été conçu pour les cartes graphiques.

Il faut d'ailleurs noter qu'une tentative de standardisation a vue le jour en 2008 avec OpenCL (Open Computing Language) [opea]. OpenCL est donc une API comme peuvent l'être OpenMP et CUDA. Cependant, la portabilité d'un code OpenCL n'est pas encore à l'ordre du jour puis-

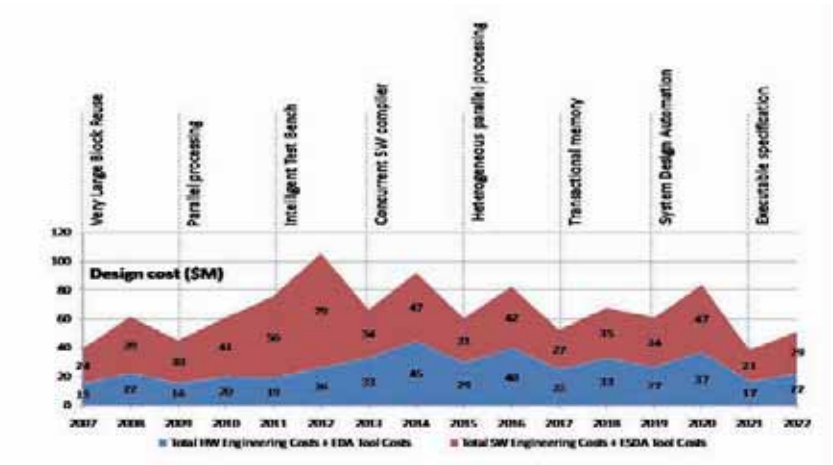



FIGURE 2.13: Impact de la technologie de conception sur le coût d'un produit électronique embarqué

qu'il doit être écrit spécifiquement pour une architecture. D'autre part, OpenCL est basé sur un niveau d'abstraction plus faible que CUDA ou OpenMP. Pour preuve, CUDA intègre les directives OpenCL dans son API. Ainsi OpenCL ne permet pas de s'abstraire de l'architecture et ne semble pas pouvoir être utilisé indifféremment sur des processeurs multicœurs et des cartes graphiques.

La première limitation des approches présentées dans ce chapitre réside dans le fait qu'elles nécessitent des architectures purement logicielles, constituées de plusieurs processeurs homogènes. Pourtant, des architectures dites hétérogènes pourront à terme posséder plusieurs processeurs cadencés à différentes fréquences ainsi que des blocs matériels optimisés pour certains calculs intensifs. Ces architectures hétérogènes se retrouvent pour l'embarqué (MPSoC et DSP) puisque ce contexte nécessite de nombreuses optimisations. Dans cette optique, on voit mal comment ces architectures pourraient être utilisées de manière optimale en partant d'une spécification séquentielle comme cela peut être le cas avec les API dont nous venons de parler. Il n'existe pas d'ailleurs pas de compilateur et d'environnement de conception pour le domaine de l'embarqué à partir de ces langages.

Les langages de programmation matériels, quant à eux, permettent d'exprimer un parallélisme à l'aide de descriptions structurelles, et constituent dès lors un meilleur point de départ pour la spécification d'un système parallèle. Cependant, ils ne peuvent être utilisés pour la simulation d'un système complet. Les approches pour la conception système constituent une avancée significative dans ce domaine. La limitation vient alors dans la phase d'exploration de l'architecture et de placement/ordonnancement qui reste une étape critique car réalisée manuellement. Les modèles de description utilisés sont très expressifs mais peuvent engendrer des interblocages et ne permettent pas de fournir une heuristique de placement/ordonnancement automatique.

Ce chapitre a eu pour objectif de présenter les méthodes et outils pour la conception de systèmes embarqués. Le besoin d'une méthodologie pour la conception de systèmes embarqués dédiés à l'analyse et la compression vidéo apparaît donc complexe et primordial pour la conception des produits dans les années à venir.

### 2.7.2 Positionnement

Il s'agit maintenant de montrer comment nos travaux se situent dans le contexte global de la recherche sur les méthodologies de prototypage rapide.

Nos travaux se basent sur le constat qu'une spécification de l'application dans le processus de prototypage rapide doit être indépendante de l'architecture cible. Il est alors nécessaire de définir un modèle de spécification qui ne soit ni trop permissif (rendant tout processus automatique impossible) ni trop restrictif. Comme nous le verrons par la suite, l'utilisation de modèles flux de données pour la spécification des applications (dataflow programming) nous semblent répondre à ce besoin.

La représentation des flux de données est depuis longtemps utilisée à l'intérieur des compilateurs ou pour la phase de synthèse matérielle. Elle permet dans ce cadre de représenter le comportement d'une application à grain fin : additions, soustractions et opérations logiques sur des mots de quelques bits à quelques octets. La programmation d'un système complet à l'aide de graphes flux de données est une nouvelle orientation qui vise à réduire l'écart qui se creuse entre les capacités offertes par les nouveaux composants et les possibilités de programmation. Ainsi, le graphe orienté est utilisé pour représenter l'application complète, du grain le plus important au plus fin. Un nœud peut alors représenter une transformée DCT sur un bloc de 8*8 pixels ou sur une image. Les arcs du graphe représentent les flux de données entre les nœuds et permettent de chaîner les opérations entre elles pour former une application complète.

La référence dans le domaine de la programmation flux de données reste encore l'ensemble des travaux réalisés à Berkekey dans l'équipe d'Edward A. Lee. L'outil issu de ces recherches est Ptolemy II [Lee01]. Cet outil a principalement pour objectif de simuler le comportement de plusieurs modèles flux de données en fonction de leur sémantique. De plus, il est possible de réaliser des analyses d'ordonnaçabilité évitant a priori les interblocages, et remplaçant les analyses de chronogrammes a posteriori. L'expressivité d'un modèle va dépendre de sa sémantique : plus le modèle sera expressif, plus son ordonnançabilité sera problématique. Ptolemy II propose donc des simulations de graphes en fonction de leur sémantique. L'outil réalise pour cela un séquencement automatique et une génération de code pour le domaine de l'embarqué. Cependant, les systèmes visés sont principalement monocœurs et l'approche choisie pour la génération automatique de code est de séquencer les calculs de sorte à limiter la consommation mémoire du système.

L'outil SynDEx sur lequel nous avons travaillé et l'outil Pressm que nous proposons maintenant seront présentés respectivement dans les chapitres 4 et 5. Ces deux outils sont basés sur la même méthodologie appelée AAA [GS03a] présentée dans le chapitre 3, mais avec des fonctionnalités différentes. Dans la méthodologie AAA, une description sous la forme flux de données permet de spécifier l'application. D'autre part, un modèle d'architecture est utilisé pour décrire la plateforme cible. Avec ces deux graphes, il est possible de mettre en œuvre une heuristique de placement/ordonnancement automatique aboutissant à la simulation du système complet (utilisation dans une phase de spécification) ainsi qu'à une génération de code automatique réduisant la phase d'intégration au minimum. La principale différence entre ces deux outils est que Preesm est OpenSource et permet à l'IETR comme à d'autres laboratoires d'améliorer les fonctionnalités de l'outil. De plus, Preesm est basé sur des modèles répandus sur lesquels il est possible de réaliser des analyses d'ordonnançabilité. Enfin, nous verrons que le modèle d'architecture utilisé dans Preesm est plus bas niveau pour prendre en compte dans le placement/ordonnancement les congestions de bus et les DMA prévus dans les composants récents (MPSoC et DSP multicœurs).

La bibliothèque SDF for Free (SDF3 [Stu07]) est une bibliothèque logicielle pour l'analyse, la

transformation, la visualisation et le placement manuel de graphes flux de données. Les classes de cette bibliothèque peuvent être utilisés par des utilisateurs développant en C++, avec des phases d'analyse de résultats. Cette approche pseudo-automatique est basée sur un modèle spécifique appelé le SADF (***Scenario Aware Data Flow*** [The07]) et fournit des fichiers de configuration pour une architecture MPSoC. SDF3 étant OpenSource, il est possible de l'utiliser, de la modifier et de l'enrichir si nécessaire. Cependant, comme nous le verrons plus tard, nous avons orienté nos développements en Java afin de faciliter l'intégration de nos travaux dans l'environnement de développement Eclipse.

Deux autres approches basées flux de données sont issues de travaux initialement effectués dans l'équipe de Edward A. Lee : CAL et DIF. En ce qui concerne le Dataflow Interchange Format (DIF [HKK+04]), il s'agit d'un ensemble d'outils sous la forme de bibliothèques JAVA pour l'analyse, la représentation, la transformation et l'ordonnancement de modèles flux de données. DIF propose une approche similaire à SDF3 avec une implantation en java. Cette bibliothèque est particulièrement intéressante et pourrait être utilisée dans nos travaux s'il nous était possible de l'enrichir au fur et à mesure des besoins. Le langage CAL (pour Caltrop Actor Language) permet quant à lui de spécifier des algorithmes flux de données quelles que soient leurs sémantiques. CAL est associé à un simulateur appelé OpenDF. Les modèles les plus dynamiques ne peuvent à l'heure actuelle pas être pris en compte dans une phase de placement/ordonnancement automatique dans un contexte MPSoC. Nous reparlerons du langage CAL en détail dans le chapitre 6 puisque nous travaillons avec ce langage dans le contexte MPEG RVC.

# Chapitre 3

# La méthodologie AAA

Au sein du laboratoire IETR Groupe Image, le thème de recherche sur le prototypage rapide a débuté en 1996. L'objectif est de pouvoir porter des algorithmes de traitement des images développés en interne sur des architectures parallèles existantes. Il s'agit ainsi d'ajouter à ces applications la validation de leur exécution temps réel. Dès le début, ces travaux ont été placés dans le cadre de la méthodologie AAA (*Adéquation Algorithme Architecture*) car cette appellation regroupe un ensemble de recherches menées au niveau national (académiques et industrielles) au sein du thème C du GDR ISIS.

La méthodologie AAA est une méthodologie pour le prototypage rapide et l'implantation optimisée d'applications distribuées temps réel embarquées sur des architectures multicomposants hétérogènes. Cette méthodologie trouve son fondement dans la théorie des graphes pour la spécification algorithmique de l'application et la spécification matérielle. L'objectif de la méthodologie AAA est de trouver la meilleure distribution et le meilleur ordonnancement de l'algorithme sur l'architecture multicomposants. Cette méthodologie est adaptée à notre problématique, à savoir des applications d'image (systèmes orientés données), et une cible matérielle essentiellement composée de plusieurs processeurs (DSP).

La méthodologie AAA est schématisée sur la figure 3.1. Elle est conçue sur un modèle en Y (Y-chart model) via les spécifications de l'application et de l'architecture matérielle par deux graphes distincts en début de la chaîne. Ensuite vient la phase de mise en correspondance appelée adéquation qui permet de faire la correspondance entre l'application et l'architecture sous la forme d'une suite de transformations effectuées sur ces deux graphes, et qui aboutit à un graphe d'implantation optimisée de l'algorithme sur l'architecture. Enfin vient la phase de génération de code destinée à la génération de codes distribués à partir de ce graphe.

A l'origine, SynDEx[1] était le seul outil de CAO niveau système supportant la méthodologie AAA, d'où notre intérêt pour cet outil. SynDEx permet le prototypage rapide et l'implantation optimisée d'applications temps réel embarquées, de la phase de spécification jusqu'à la phase de génération de code. Le résultat obtenu est tout d'abord une prédiction temporelle de l'exécution de l'algorithme sur cette architecture. Il génère enfin automatiquement pour chaque processeur un exécutif temps réel dédié. Nous allons notamment présenter dans ce chapitre les modèles AAA utilisés dans l'outil SynDEx. Une présentation plus formelle est donnée dans [Vic99] et [Gra00].

1. SynDEx Synchronized Distributed Executive

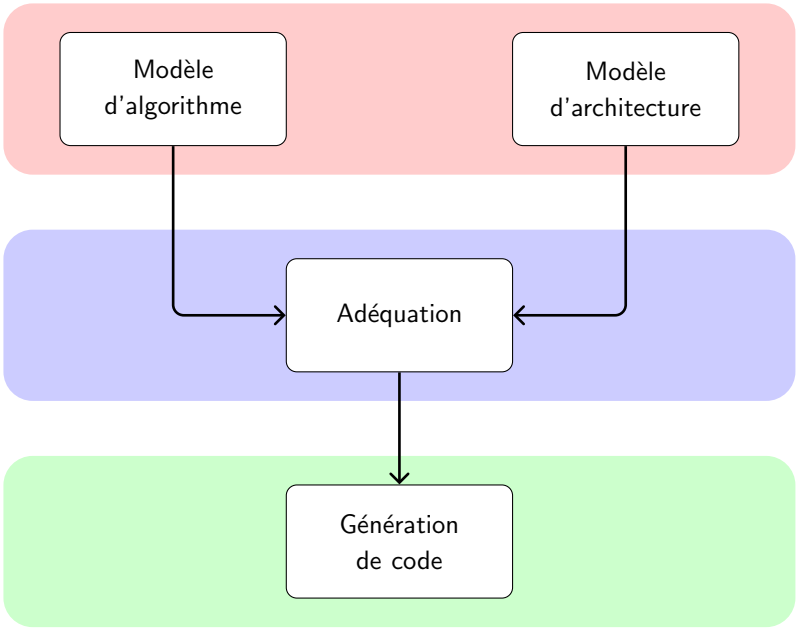



FIGURE 3.1: Méthodologie AAA

## 3.1 Modèle d'algorithme

La méthodologie AAA utilise les graphes flux de données pour spécifier l'algorithme. La notion de flux de données a été initialement introduite par Dennis [Den74] dans le cadre des architectures d'ordinateurs. Dans ce modèle, les nœuds du graphe représentent des opérations tandis que les arcs représentent des transferts de données entre opérations. Une opération dans le modèle flux de données est asynchrone et fonctionnelle [LH93]. Premièrement, cela signifie qu'une opération peut être exécutée seulement si tous les opérandes requis sont disponibles. Ce modèle ne nécessite donc pas l'utilisation d'un compteur de programme afin d'exécuter séquentiellement les opérations (instructions). Deuxièmement, une opération est sans effet de bord. Ainsi, deux opérations exécutables (qui ont leurs opérandes disponibles) peuvent être exécutées concurremment quel que soit l'ordre d'exécution, sans conduire à une situation de ***compétition*** (*race condition*).

Durant l'exécution d'une opération, des données (les opérandes) sont consommées sur les arcs entrants du nœud et des données sont produites sur les arcs sortants. Les nœuds sont des opérations partiellement ordonnées par leurs dépendances de données. Le modèle d'algorithme dans la méthodologie AAA est une extension du modèle de graphe flux de données. Tout d'abord, le modèle permet la spécification sous forme factorisée des répétitions infinies (pour prendre en compte l'aspect réactif de l'application) et finies (pour rendre plus commode la spécification de répétitions d'une même opération). Le modèle de graphe autorise aussi la prise en compte du conditionnement pour accroître l'expressivité des applications modélisables. De plus, le modèle de graphe supporte la hiérarchie. Ainsi un nœud peut contenir un ensemble de nœuds atomiques ou hiérarchiques. Un nœud atomique se définit comme une opération élémentaire ne contenant que des ports d'entrée, de sortie, ou d'entrée-sortie. Le graphe résultant est donc qualifié de graphe flux de données hiérarchique, conditionné et factorisé. Le modèle est expliqué en détail dans [Gra00, GS03b], ainsi que dans la suite du document.

### 3.1.1 Terminologie

Les nœuds du graphe représentent des opérations. On distingue trois types de nœuds primitifs qui représentent les opérations de calcul, d'entrée et de sortie. A ces nœuds s'ajoutent deux autres nœuds spéciaux qui représentent des retards et des constantes.

**Définition 3.1.1 (Opération de calcul)** *Une opération de calcul est modélisée par un nœud* fonction. *Il s'agit d'une opération qui possède au minimum un successeur et un prédécesseur. Cette opération de calcul consomme des données sur les ports d'entrée et produit des données sur les ports de sortie. Un tel nœud peut être hiérarchique en contenant des opérations de calcul, de conditionnement et de répétition.*

**Définition 3.1.2 (Opération d'entrée/sortie)** *Deux opérations sont capables d'intéragir avec le monde extérieur. L'opération d'entrée modélisée par un nœud* capteur *est une opération qui possède au minimum un successeur et qui n'a aucun prédécesseur. C'est une opération qui agit avec l'environnement extérieur en capturant les informations du monde réel et en les transmettant sous forme de données aux opérations qui la succèdent. A contrario, l'opération de sortie modélisée par le nœud* actionneur *est une opération qui possède au minimum un prédécesseur et qui n'a aucun successeur. C'est une opération qui agit avec l'environnement extérieur en transformant les données des opérations qui la précèdent en grandeurs physiques transmises vers le monde extérieur.*

**Définition 3.1.3 (Opération Constante)** *Une opération qui produit des données identiques lors de chaque itération de l'algorithme est définie comme étant une opération constante. Cette opération présente donc la particularité de n'avoir à être exécutée qu'une seule fois.*

**Définition 3.1.4 (Opération Retard)** *L'opération modélise des dépendances de données inter-itérations du graphe d'algorithme. Il s'agit d'une dépendance de données entre différentes itérations du graphe flux de données.*

Dans l'exemple de la figure 3.3, le nœud $A$ est un capteur, l'opération $D$ est un actionneur, les nœuds $B$ et $C$ modélisent des opérations de calcul et le nœud $\$Z$ est un retard ($ symbolisant le retard).

### 3.1.2 Factorisation infinie

De par l'aspect réactif des applications à embarquer, l'algorithme dans AAA/SynDEx est modélisé par un graphe flux de données infini dans lequel un motif infiniment répété est identifié pour permettre une modélisation compacte de l'application. Le graphe est réduit par factorisation infinie à son motif répété. La figure 3.2 représente le graphe infiniment répété et la figure 3.3 représente le motif du graphe où un nœud retard $\$Z$ apparaît pour rendre compte d'une dépendance de données inter-itérations.

### 3.1.3 Factorisation finie et répétitions d'opérations

Il est très fréquent dans une application qu'une opération qui la compose soit répétée plusieurs fois. Ainsi, un sous-graphe répété un nombre fini de fois peut aussi être réduit par factorisation à

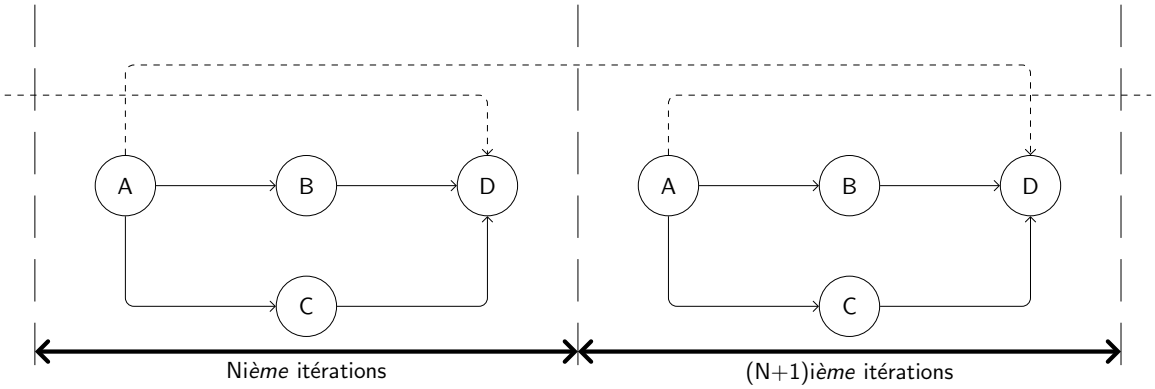



FIGURE 3.2: Graphe acyclique orienté infini

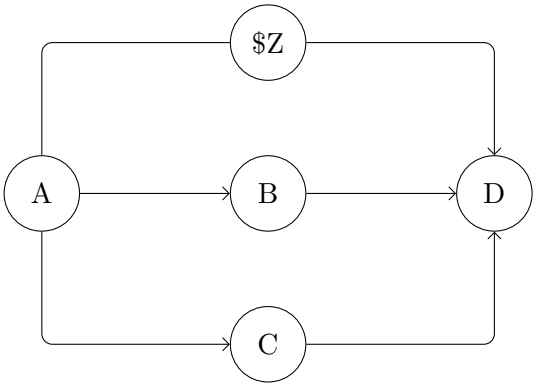


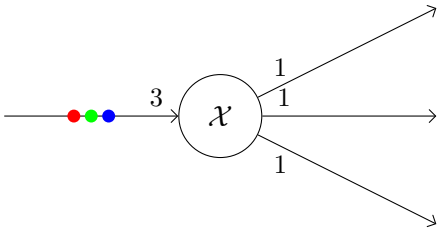

FIGURE 3.3: Factorisation finie du graphe acyclique orienté infini

son motif répété. La méthodologie AAA fournit la possibilité de spécifier les opérations répétées sous une forme factorisée. Les parties du graphe d'algorithme, appelées motifs, délimitées par ces frontières sont les parties répétées du graphe.

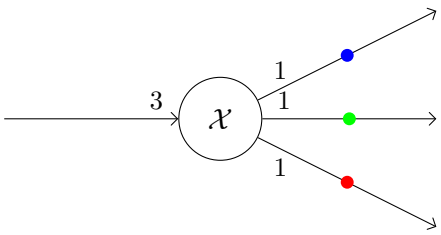
On définit les nœuds $\mathcal{X}$ et $\mathcal{M}$ qui seront utilisés pour mettre en lumière les nœuds de factorisation.

**Définition 3.1.5 (nœud $\mathcal{X}$)** *Un nœud $\mathcal{X}$ (pour eXplode) est un nœud tel que son degré (correspondant à un nombre de données) entrant est égal à 1 et son degré sortant est un entier strictement supérieur à 1. Ce nœud réalise le partitionnement uniforme des données qu'il consomme pour les produire en sortie. Le nombre de données produites par arcs sortants doit être un sous-multiple du nombre de données consommées sur l'arc entrant.*

La figure 3.4 illustre le nœud $\mathcal{X}$. On note respectivement à la fin d'un arc et au début d'un arc le nombre de données qu'il consomme et qu'il produit. Ici, le nœud $\mathcal{X}$ consomme 3 données sur l'arc entrant et produit 1 donnée sur les trois arcs sortants. Il y a trois données sur l'arc entrant de la figure 3.4(a). Après exécution, les données sont uniformément partitionnées sont les trois arcs sortants de la figure 3.4(b). Il s'agit d'une partition 3 vers 1.




(a) Sommet $\mathcal{X}$ avant exécution



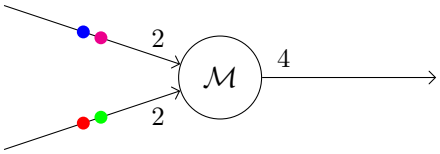

(b) Sommet $\mathcal{X}$ après exécution

FIGURE 3.4: Sommet *Explode*

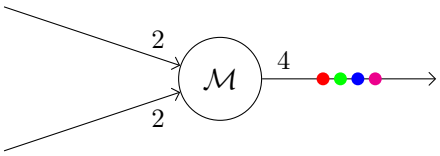
**Définition 3.1.6 (nœud $\mathcal{M}$)** *Un nœud $\mathcal{M}$ (pour iMplode) est un nœud tel que son degré entrant est un entier strictement supérieur à 1 et son degré sortant est égal à 1. Ce nœud réalise la concaténation uniforme des données qu'il consomme pour les produire vers l'unique sortie. Le nombre de données consommées par arcs entrants doit être un sous-multiple du nombre de données produites sur l'arc sortant.*

La figure 3.5 illustre le nœud $\mathcal{M}$. Ici, le nœud $\mathcal{M}$ consomme 2 données sur chacun de ses arcs entrants et produit 4 données sur son arc sortant. Il s'agit d'une concaténation 2 vers 4.

Il existe en outre 4 nœuds dits de factorisation nommés *fork* ($\mathcal{F}$), *join* ($\mathcal{J}$), *diffuse* ($\mathcal{D}$) et *iterate* ($\mathcal{I}$). Ils sont définis ainsi :




(a) Sommet $\mathcal{M}$ avant exécution




(b) Sommet $\mathcal{M}$ après exécution

FIGURE 3.5: Sommet *implode*

- **nœud $\mathcal{F}$ :** Partitionnement de la donnée en entrée aux différents motifs factorisés en sortie,
- **nœud $\mathcal{J}$ :** Regroupement des données partitionnées en entrée par les motifs factorisés en un vecteur en sortie,
- **nœud $\mathcal{D}$ :** Diffusion de la donnée en entrée aux motifs factorisés en sortie,
- **nœud $\mathcal{I}$ :** Itération du motif factorisé récursivement sur lui-même.

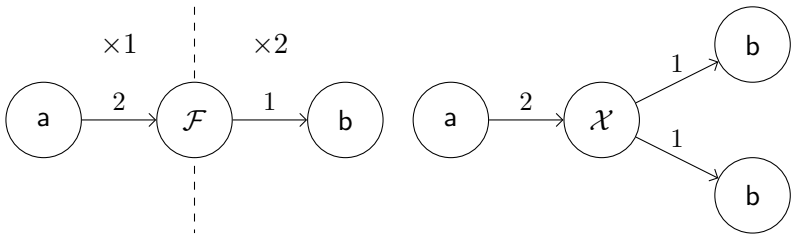



FIGURE 3.6: graphe d'une factorisation *Fork* et son graphe développé équivalent

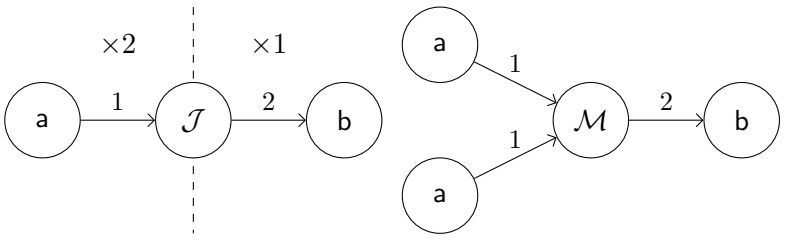



FIGURE 3.7: graphe d'une factorisation *Join* et son graphe développé équivalent

Pour chacun des graphes d'algorithme utilisant un de ces nœuds de factorisation, il existe un graphe équivalent développé utilisation des nœuds $\mathcal{X}$ et $\mathcal{M}$ comme l'illustrent les figures 3.6, 3.7,3.8 et 3.9. La pondération sur les arcs représente la quantité de données traitées par les opérations.

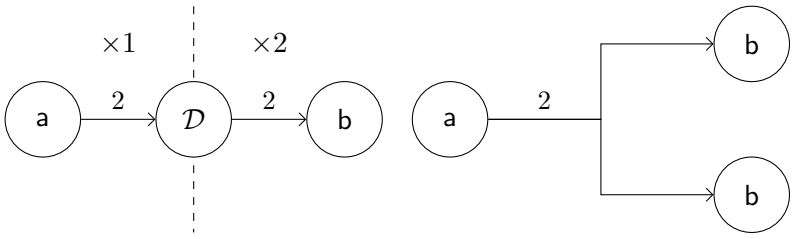



FIGURE 3.8: graphe d'une factorisation *Diffuse* et son graphe développé équivalent

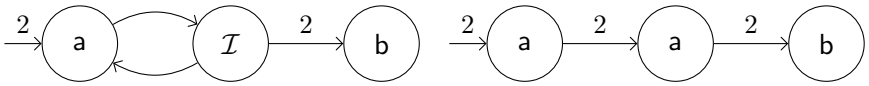



FIGURE 3.9: graphe d'une factorisation *Iterate* et son graphe développé équivalent

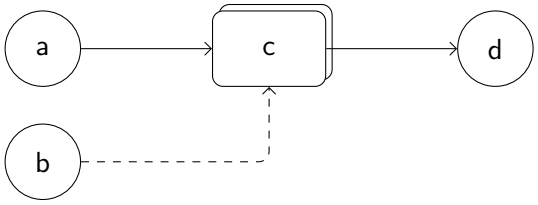
### 3.1.4 Conditionnement

Pour spécifier le contrôle dans le modèle algorithmique, le graphe flux de données est étendu pour permettre l'expression du conditionnement. Il faut noter que cette notion existait déjà dans le modèle de Dennis [Den74]. Une opération est conditionnée si son exécution dépend de la valeur d'une des données qu'il consomme en entrée. Il s'agit de la donnée de contrôle et l'arc qui l'achemine est appelé arc de conditionnement. La valeur de contrôle portée par un arc de conditionnement permet de faire le choix parmi un ensemble de sous-graphes alternatifs. L'imbrication des conditionnements est traduit par une représentation hiérarchique du graphe d'algorithme. À chaque interaction avec l'environnement, concrétisée par un ensemble d'événements d'entrée, les valeurs des arcs de conditionnement déterminent à partir des valeurs d'entrée l'ensemble des opérations à exécuter pour obtenir les événements de sortie. Chaque nœud non conditionné produit ses événements sur ses sorties dès que tous les événements sur les entrées sont arrivés. Les nœuds dénommés `CondI` et `CondO` dans AAA permettent d'encapsuler les sous-graphes alternatifs.

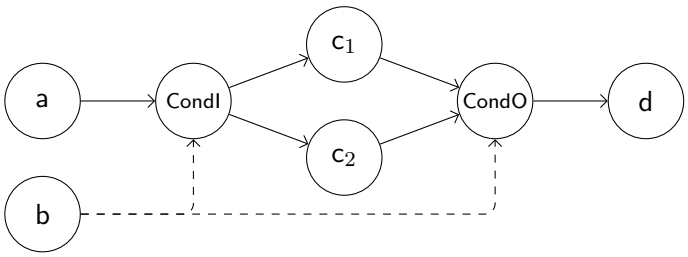
**Définition 3.1.7 (nœud CondI)** *Le nœud* `CondI` *est le nœud de conditionnement entrant qui agit comme un multiplexeur. Il est utilisé durant la phase d'initialisation du conditionnement afin d'aiguiller la sortie d'une opération vers le sous-graphe qui doit être exécuté.*

**Définition 3.1.8 (nœud CondO)** *Le nœud* `CondO` *est le nœud de conditionnement sortant qui agit comme un démultiplexeur. Il est utilisé durant la phase de finalisation du conditionnement afin de stocker les sorties des sous-graphes conditionnés vers une donnée unique consommée par une opération.*

L'exemple de la figure 3.10 illustre le conditionnement dans AAA/SynDEx. Le nœud $c$ est un nœud hiérarchique que l'on représente par un rectangle. Ce nœud est conditionné par la donnée de contrôle produite par $b$ sur l'arc de conditionnement $(b, c)$ représenté en pointillé. Selon la valeur qu'il produit (un booléen ici), l'opération $c_1$ ou $c_2$ est exécutée. Le développement du graphe à l'aide des nœuds de conditionnement est donné sur la figure 3.10(b).




(a) Graphe factorisé




(b) Graphe développé

FIGURE 3.10: Conditionnement

## 3.2 Modèle d'architecture

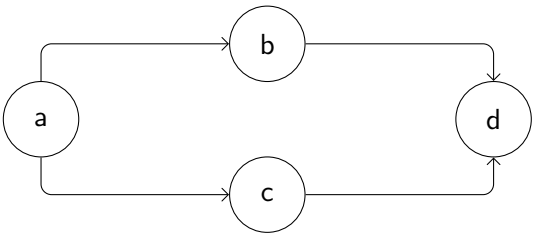
Le modèle d'architecture hétérogène multi-composants [GLS99a, Gra00] choisi dans AAA/SynDEx est un hypergraphe orienté, dont chaque nœud est un automate fini (machine séquentielle) et chaque arc une connexion physique entre deux automates finis. Une architecture à plusieurs composants est alors représentée par un réseau d'automates finis interconnectés à l'aide de media de communication (bus, mémoires partagés ...). Le choix de l'automate comme composant atomique apporte une bonne précision dans le modèle, tout en n'étant pas trop compliqué lorsqu'il s'agira de l'exploiter durant la phase d'adéquation.

Il existe deux types de nœuds : les séquenceurs et les mémoires. Les séquenceurs se divisent en deux catégories. Les *opérateurs* ou séquenceurs d'instructions qui permettent de séquencer les opérations de calcul et les *communicateurs* ou séquenceurs de communication pour séquencer les opérations de communication. De même les mémoires sont de deux catégories. La mémoire RAM à accès aléatoire stocke les données ou les programmes, la mémoire SAM à accès séquentiel stocke les données avec un ordre d'arrivée préservé (de type *first-in first-out*). Un médium de communication est alors vu comme une mémoire reliée par des fils à d'autres séquenceurs.

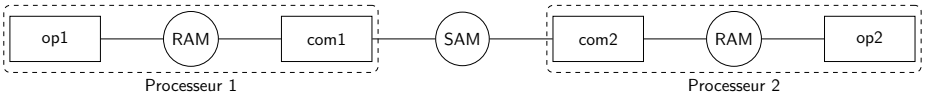
Dans ce modèle, un processeur est formé d'un unique opérateur et de plusieurs communicateurs (un communicateur par media connecté à l'opérateur). Un opérateur exécute une partie de l'algorithme et un communicateur exécute des opérations de communication lorsqu'un transfert de données est requis. L'opérateur et les différents communicateurs sont reliés entre eux à travers une mémoire partagée de type RAM du processeur. La figure 3.11(b) représente un graphe d'architecture composé de deux processeurs connectés via un medium de communication de type SAM (vue comme une FIFO). Chaque processeur est constitué d'un opérateur et d'un communicateur qui communiquent au travers d'une mémoire partagée.

Ce qui distingue le modèle d'architecture adopté dans AAA par rapport à d'autres modèles existants, est qu'il n'est ni trop abstrait pour pouvoir par exemple tenir compte des délais de communication, ni trop précis pour pouvoir être suffisamment générique. L'architecture interne globale de n'importe quel processeur peut ainsi être décrite. Il faut remarquer que des extensions

de la méthodologie à la synthèse architecturale ont aussi été réalisées [Kao04].




(a) Graphe d'algorithme




(b) Graphe d'architecture

FIGURE 3.11: Graphe d'algorithme et graphe d'architecture

## 3.3 Transformations de graphes et adéquation

### 3.3.1 Transformations du graphe d'algorithme

Avant d'effectuer l'adéquation, le graphe d'algorithme doit être transformé sous une forme développée pour rendre compte du parallélisme potentiel de données. La première transformation du graphe a pour objectif de développer les opérations factorisées ou conditionnées et de substituer les nœuds hiérarchiques par les sous-graphes qu'ils contiennent. La récursion traite successivement les différents niveaux hiérarchiques.

On appelle cette transformation la ***mise à plat***, où le graphe résultant est un graphe flux de données où tous les nœuds du graphe sont atomiques (la hiérarchie est mise à plat) et auquel les nœuds factorisés et conditionnés sont développés. Pour rendre compte du parallélisme de données, la construction du graphe transformé nécessite le développement des différents nœuds factorisés pour mettre en évidence des répétitions d'opérations. Des nœuds $\mathcal{M}$ et $\mathcal{X}$ sont donc insérés lorsque des nœuds $\mathcal{J}$ et $\mathcal{F}$ apparaissent tandis que des hyperarcs sont créés pour prendre en compte les situations de diffusion. Les opérations de conditionnement nécessitent également l'insertion de nœuds `CondI` et `Cond0` pendant la phase de mise à plat.

Effectuer cette transformation avant l'adéquation permet de se ramener pour les traitements de l'adéquation à un formalisme de graphe plus simple, plus proche des graphes flux de données habituels. On définit ainsi une implantation comme une solution particulière de placement et d'ordonnancement des opérations sur l'architecture. Le critère d'optimisation retenu dans le logiciel SynDEx pour le choix de la meilleure implantation, dans l'espace des implantations possibles, est l'implantation donnant une latence minimale (minimisant la durée globale de l'application en tenant compte des coûts de communication inter-processeurs).

### 3.3.2 Adéquation

L'adéquation consiste à réaliser une implantation optimisée d'un algorithme sur une architecture donnée. Il s'agit alors d'exploiter le parallélisme potentiel de l'algorithme pour utiliser au

mieux le parallélisme physique de l'architecture. Le graphe d'implantation est obtenu par transformation du graphe d'algorithme et du graphe d'architecture. Cela consiste à réaliser un placement et un ordonnancement non seulement des opérations de l'algorithme sur les opérateurs de l'architecture, mais aussi des opérations de communication sur les communicateurs lorsqu'il existe des précédences de données entre des opérations distribuées.

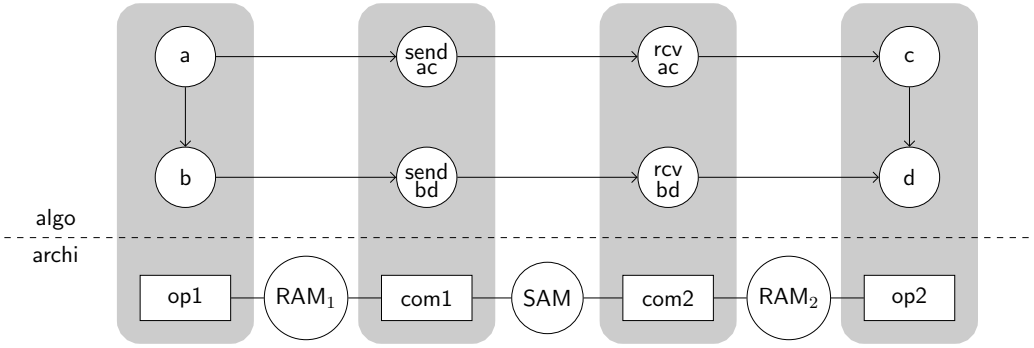



FIGURE 3.12: Graphe transformé avec opérations de communication inter-processeur et distribution sur les séquenceurs de calcul et de communication

**Placement**

Le placement consiste à affecter chaque opération de l'algorithme à un opérateur capable de l'exécuter. Le graphe d'algorithme est décomposé en autant (ou moins) de sous-graphes disjoints qu'il y a de nœuds opérateurs dans le graphe d'architecture. La figure 3.12 représente une telle partition issue des graphes de la figure 3.11 où les opérations $a$ et $b$ sont exécutées sur le processeur op1 et les opérations $c$ et $d$ sur le processeur op2. Ensuite pour chacune de ces opérations, il faut ajouter des nœuds d'allocation mémoire locale et affecter ces nœuds à une RAM connectée à l'opérateur qui exécute les opérations. Enfin, il faut affecter chaque dépendance de données d'opérations appartenant à des sous-graphes distincts, à une route reliant les deux opérateurs (chemin dans le graphe de l'architecture). Il est ainsi nécessaire de créer et insérer, entre les deux opérations de l'algorithme, autant d'opérations de communication que de communicateurs (com1 et com2) et autant de nœuds d'allocation de mémoires pour les données transférées que de nœuds mémoire SAM et RAM sur la route.

Dans l'exemple de la figure 3.12, les opérations $a$ et $c$ sont dépendantes et sur des processeurs distincts : on ajoute donc des nœuds d'envoi ($sendac$) et de réception ($rcvac$) sur les communicateurs respectifs des processeurs. L'allocation de la mémoire (données et programme) est quant à elle représentée sur la figure 3.13.

Enfin, il faut affecter ces éléments aux nœuds correspondants du graphe de l'architecture. Ceci conduit à une partition de l'ensemble des nœuds de communication et des nœuds d'allocation respectivement en autant de sous-graphes que de communicateurs et de mémoires. Pour la suite, il est important de noter que les nœuds d'allocation sont ceux qui permettent de déterminer la taille des mémoires nécessaires pour l'application.

**Ordonnancement**

L'ordonnancement consiste à rendre total l'ordre des opérations par une extension linéaire de l'ordre partiel associé à chaque sous-graphe de l'algorithme formé d'opérations de calcul

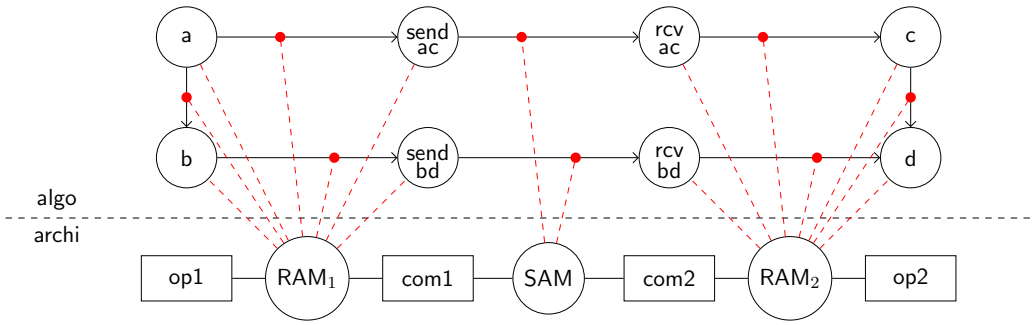



FIGURE 3.13: Allocation mémoire des dépendances de données et de la mémoire programme

et d'opérations de communication. Cette phase est nécessaire car l'opérateur est un automate séquentiel.

Une implantation est donc le résultat d'une transformation sur le graphe d'algorithme (ajout de nouveaux nœuds et de nouveaux arcs) en fonction du graphe de l'architecture, lui même transformé (détermination de toutes les routes possibles) [Gra00]. L'ensemble de toutes les implantations possibles, étant donnés un algorithme et une architecture, est formalisé comme une composition de deux relations : le placement et l'ordonnancement, chacune d'elles mettant en correspondance deux couples de graphes (algorithme, architecture) [Vic99]. Un graphe d'algorithme est combiné avec un graphe d'architecture pour conduire à un graphe d'algorithme transformé (distribué et ordonnancé). Chacune de ces implantations possibles a des performances – au sens de la latence – différentes. Ces performances sont obtenues par le calcul de chemins critiques sur le graphe de l'implantation étiquetés par les durées d'exécution caractéristiques des opérateurs, des communicateurs et des mémoires de l'architecture.

Comme nous venons de le voir, le critère d'optimisation retenu est la latence, il est donc nécessaire de connaître les temps d'exécution au pire cas (Worst Case Execution Time, WCET) associés à chaque opération atomique, temps pouvant être différent d'un processeur à un autre. Les temps de communication se déduisent à partir d'un temps unitaire en fonction du type de données transmises, multiplié par le nombre de données à transmettre. L'ordonnancement des opérations sur chaque processeur est déterminé *hors ligne* (fixé a priori, on le qualifie aussi d'ordonnancement statique ou d'ordonnancement déterminé à la compilation).

Le résultat de l'implantation optimisée se modélise sous la forme d'un nouveau graphe, obtenu par la transformation du graphe d'architecture et du graphe d'algorithme. Le placement et l'ordonnancement dans le cas multicomposants est un problème d'optimisation NP-Complet. Cela signifie qu'il n'existe aucun algorithme connu pour résoudre ce problème en un temps polynomial. Il est d'ailleurs très improbable qu'un tel algorithme existe. Par conséquent, des heuristiques sont utilisées afin de déterminer une approximation de la solution optimale en un temps très nettement réduit. Or une application décrite notamment à une faible granularité, et utilisant la répétition de sous-graphes, peut très vite déboucher sur un graphe à plusieurs milliers de nœuds. Ceci justifie la recherche de techniques rapides pour réaliser l'adéquation, notamment des heuristiques dites ***gloutonnes*** (sans remise en cause des décisions précédentes) capables de s'exécuter très rapidement. Parmi toutes les transformations possibles, l'heuristique d'optimisation fondée sur la prédiction de performances conserve celle qui minimise la durée d'exécution de l'algorithme (latence). Une description détaillée de l'heuristique mise en œuvre dans AAA/SynDEx est donnée dans [GLS99a].

### 3.3.3 Synchronisations entre séquenceurs

Les répétitions infinies de l'application ainsi que les synchronisations entre opérateurs doivent être ajoutées dans le graphe d'implantation optimisée. En effet, les applications réactives à implanter sont répétitives par nature alors que le graphe initial a été infiniment factorisé. Des boucles doivent être insérées dans chaque séquence d'opérations et de communications. L'ordonnancement ne fait pas non plus apparaître les synchronisations entre les différents séquenceurs d'un processeur. La synchronisation consiste à rendre cohérente l'exécution de l'ensemble de l'application lors de l'accès en exclusion mutuelle aux données partagées par les opérations de ces séquences. Cela est assuré par la synchronisation des différentes machines séquentielles. La génération automatique de code distribué se fait suivant des règles décrivant la transformation d'un graphe d'implantation optimisé en un graphe d'exécution. Pour chaque opérateur et communicateur un programme séquentiel est construit, formé respectivement de la séquence des opérations de calcul et de la séquence des opérations de communication qu'il doit exécuter.

Pour garantir les précédences d'exécution entre les opérations appartenant à des séquences de calcul et/ou de communication différentes, et pour garantir l'accès en exclusion mutuelle aux données partagées par certaines opérations de ces séquences, on ajoute donc des opérations de synchronisation. On distingue deux situations particulières : la situation *tampon plein* qui permet de garantir qu'une opération consommatrice ne peut être exécutée que lorsque les données ont été produites par l'opération consommatrice et la situation *tampon vide* qui permet de garantir qu'une opération productrice ne puisse être exécutée avant que l'opération consommatrice ne soit terminée.

Le code généré pour chaque processeur est constitué d'une séquence de calcul principale et d'autant de séquences de communication que de média auxquels cet opérateur est connecté. Les réseaux de Petri sont un bon moyen de représentation des programmes séquentiels où les transitions représentent les opérations et où les jetons dans les places correspondent à l'état courant de l'exécution du programme. Il est alors relativement aisé de représenter les séquenceurs sous cette forme où des opérations de synchronisation sont ajoutées afin de garantir un accès cohérent aux ressources. Ces synchronisations permettent ainsi au programme complet (tous les séquenceurs) de respecter l'ordre partiel du graphe d'algorithme initial, n'introduisant ainsi pas d'interblocage dans une itération infinie entre la séquence de calcul et celles de communication, ou entre deux itérations infinies. Les principes majeurs de la génération de code de SynDEx sont détaillés dans le document [Gra00].

Chaque séquence est exécutée par un séquenceur différent. Ces séquenceurs sont synchronisés au niveau des dépendances de données qui les relient. Des synchronisations sont générées pour chaque dépendance reliant deux opérations exécutées par deux séquenceurs différents (communicateur-communicateur, opérateur-communicateur, communicateur-opérateur).

#### *3.3.3.1 Synchronisations opérateur-communicateur*

Comme nous l'avons vu auparavant, un processeur est composé d'un opérateur et d'autant de communicateurs que de media connectés à lui. La mémoire (de type RAM) est commune à tous les séquenceurs, ce qui nécessite la mise en place de synchronisations afin de garantir un cheminement consistant des données. Les synchronisations sont réalisées à l'aide de deux opérations nommées `Suc` et `Pre`. En utilisant le vocable des sémaphores [Dij68], ces opérations sont l'équivalent des opérations $P$ et $V$ qui manipulent des sémaphores. Un sémaphore $s$ est une variable protégée uniquement manipulable pour les opérations $P(s)$ qui permet l'attente du sémaphore $s$, et $V(s)$ qui

permet la libération du sémaphore $s$. Les sémaphores assurent qu'une donnée n'est pas consommée avant d'être produite et qu'une donnée produite par une opération à l'itération $n$ n'est pas écrasée par la même opération à l'itération $n+1$ avant que toutes les opérations consommant la donnée à l'itération $n$ n'aient pas été exécutées.

On utilise les qualificatifs *full* et *empty* sur les opérations `Suc` et `Pre` afin de différencier les situations de synchronisation. Dans la situation *tampon plein*, on utilise la paire `Suc full` et `Pre full` et dans la situation *tampon vide*, on utilise la paire `Suc empty` et `Pre empty`.

L'exemple de la figure 3.14 illustre les synchronisations nécessaires à mettre en place entre l'opération de calcul $a$ et l'opération de communication *sendac* à l'aide d'un réseau de Petri partiel. Les opérations de synchronisation `Suc` et `Pre` sont insérées pour accéder en exclusion mutuelle aux données qu'elles partagent dans la mémoire $RAM1$. Les mots *full* et *empty* signifient que le segment de mémoire alloué sur $RAM1$ pour la dépendance de données entre $a$ et *sendac* est respectivement plein ou vide.

Il faut remarquer que la synchronisation communicateur-opérateur en réception de la donnée n'est pas présentée. Il s'agit en fait du dual de la synchronisation opérateur-communicateur. Ainsi, dans l'exemple de la figure 3.14, les qualificatifs pour les sémaphores *full* et *empty* doivent simplement être inversés, l'opération de $a$ est remplacée par l'opération *rcvac* et l'opération *sendac* est remplacée par l'opération $c$.

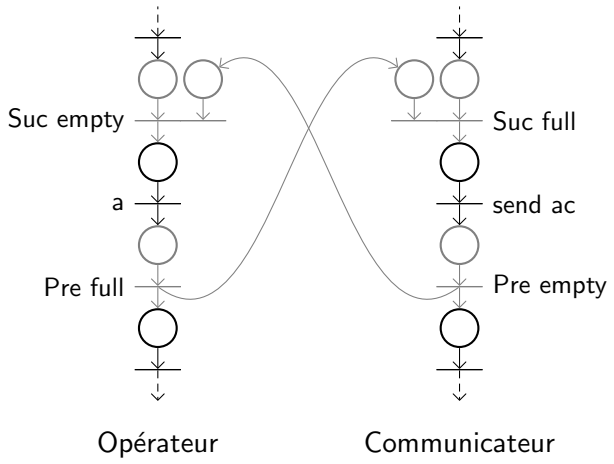



FIGURE 3.14: Synchronisation opérateur-communicateur par RAM

#### *3.3.3.2 Synchronisations communicateur-communicateur*

**Synchronisation par SAM**

Un médium de communication de type SAM a un fonctionnement de type FIFO bidirectionnelle. Les données produites sont envoyées vers la mémoire SAM (via l'utilisation de l'opération `send`) tandis que le séquenceur de communication du consommateur vient les lire (via l'utilisation de l'opération `rcv`). Les synchronisations entre ces deux opérations sont transparentes et gérées le plus souvent par le médium au niveau matériel : une écriture est bloquante si la mémoire est pleine tandis qu'une lecture est bloquante lorsque la mémoire est vide. Les synchronisations sont implicites et ne nécessitent pas la mise en place de mécanisme pour contrôler l'accès à la mémoire. L'envoi et la réception sont exécutés de manière asynchrone entre l'émetteur et le récepteur.

La figure 3.15 illustre les synchronisations implicites placées entre les opérations de communication *sendac* et *rcvac*. Le graphe du processeur produisant la donnée est à gauche (graphes du séquenceur du calcul et du séquenceur de communication), celui du processeur consommateur est à droite. Il n'y a pas d'ajout de transitions dans les deux séquences (ce qui indique que AAA/-SynDEx n'ajoute aucune opération supplémentaire) et les deux places ajoutées en amont des transitions *sendac* et *rcvac* représentent les drapeaux *empty flag* et *full flag* gérés par le medium au niveau matériel.

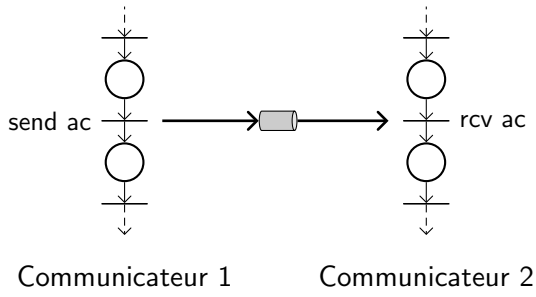



FIGURE 3.15: Synchronisation communicateur-communicateur par SAM

**Synchronisation par RAM**

Le médium de communication de type RAM utilise une mémoire à accès aléatoire. On peut donc venir lire les données dans un ordre différent de celui où elles ont été écrites. Afin de garantir une écriture et une lecture cohérentes dans la mémoire ainsi que l'exclusion mutuelle à la mémoire, on ajoute des synchronisations dans les séquenceurs de communications. A l'inverse d'un modèle SAM, les synchronisations sont maintenant gérées explicitement par le modèle AAA. Elles sont assurées par un mécanisme de sémaphores comme dans le cas des synchronisations opérateurs-communicateurs.

Imaginons que la communication par le medium SAM entre les deux processeurs de l'exemple précédent soit remplacée par un medium de type RAM. La figure 3.16 représente un réseau de Petri partiel correspondant, ne laissant apparaître que les opérations d'écriture (write) et de lecture (read) sur les deux communicateurs. Il faut rendre l'accès exclusif à la mémoire. Les opérations de synchronisation `SucR` (équivalent à $P$) et `PreR` (équivalent à $V$) sont insérées pour permettre l'accès en exclusion mutuelle des données qu'elles partagent dans la mémoire RAM.

Le résultat de ces transformations aboutit à un ensemble de machines séquentielles synchronisées qui fonctionnent de manière répétitive jusqu'à l'infini. La figure 3.17 représente les deux séquenceurs du processeur 1 synchronisés et dans lesquels des boucles sont insérées pour rendre compte du caractère répétitif de l'application.

## 3.4 La génération de macro-code

Une fois le graphe d'implantation optimisée déterminé, un fichier contenant du macro-code peut être automatiquement généré pour chaque processeur de l'architecture. AAA/SynDEx génère un ***macro-code*** fondé sur le formalisme précédent, indépendant des langages utilisés par les processeurs. L'exécutif créé pour chaque processeur contient les allocations de mémoire, les initialisations, et les séquenceurs de calcul et de communication sous la forme d'une boucle infinie. Le macro-code

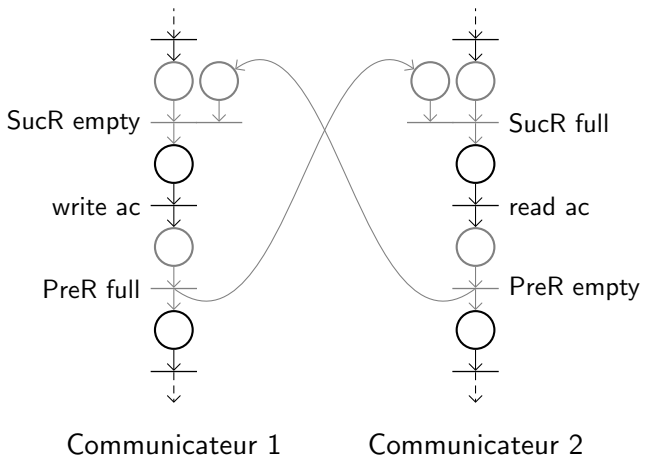



FIGURE 3.16: Synchronisation communicateur-communicateur par RAM

est composé de macro-instructions. On distingue deux types de macro-instructions : les macros applicatives et les macros systèmes. D'un coté, les macros applicatives représentent les opérations de l'algorithme. Chaque macro est alors vue comme un appel de fonction, qui devra être traduit selon la nature du langage cible utilisé (ASM, C, Java,...). D'un autre coté, les macros systèmes sont insérées pour rendre compte des allocations de la mémoire (les arcs du graphe transformé, les sémaphores, ...), des séquenceurs, des boucles et des opérations de communication et de synchronisation.

### 3.4.1 Transformation du macro code

Chaque macro-code doit être traduit vers le langage cible des processeurs. Pour permettre la génération d'un code compilable, un macro-processeur est utilisé pour la transformation. Le macro-processeur transforme la liste de macro-instructions en un code source propre à la cible matérielle envisagée. Cela consiste à remplacer chaque macro-instruction par une définition. Chaque définition est spécifiée à travers des bibliothèques (aussi appelée noyaux) dépendantes du processeur cible ou encore du media de communication reliant deux processeurs. Plusieurs types de bibliothèques existent. Elles permettent la transformation des macros systèmes vers du code spécifique aux différents opérateurs de l'architecture, comme les allocations mémoire, les synchronisations entre séquences ou encore les opérations de communication. D'autres bibliothèques sont plus proches de l'algorithme et traduisent par exemple les prototypes ou les appels de fonctions à partir des macros applicatives. Puisque le code distribué généré par SynDEx est générique, il existe un grand nombre de traductions possibles amenant à un code source compilable. Cette phase est réservée à l'utilisateur qui doit choisir la traduction la plus appropriée à la cible envisagée. Le logiciel libre GNU-M4 est le macro-processeur utilisé pour la phase de transformation du macro-code. Les bibliothèques sont quant à elles écrites en langage m4 (cf. partie 4.2).

### 3.4.2 Factorisation dans SynDEx

L'outil SynDEx est un outil de CAO niveau système qui supporte la méthodologie AAA. Dans le modèle algorithmique formel, les nœuds de factorisation font parties intégrantes du graphe. Dans l'outil SynDEx, les nœuds de factorisation ne sont pas explicitement spécifiés dans le graphe

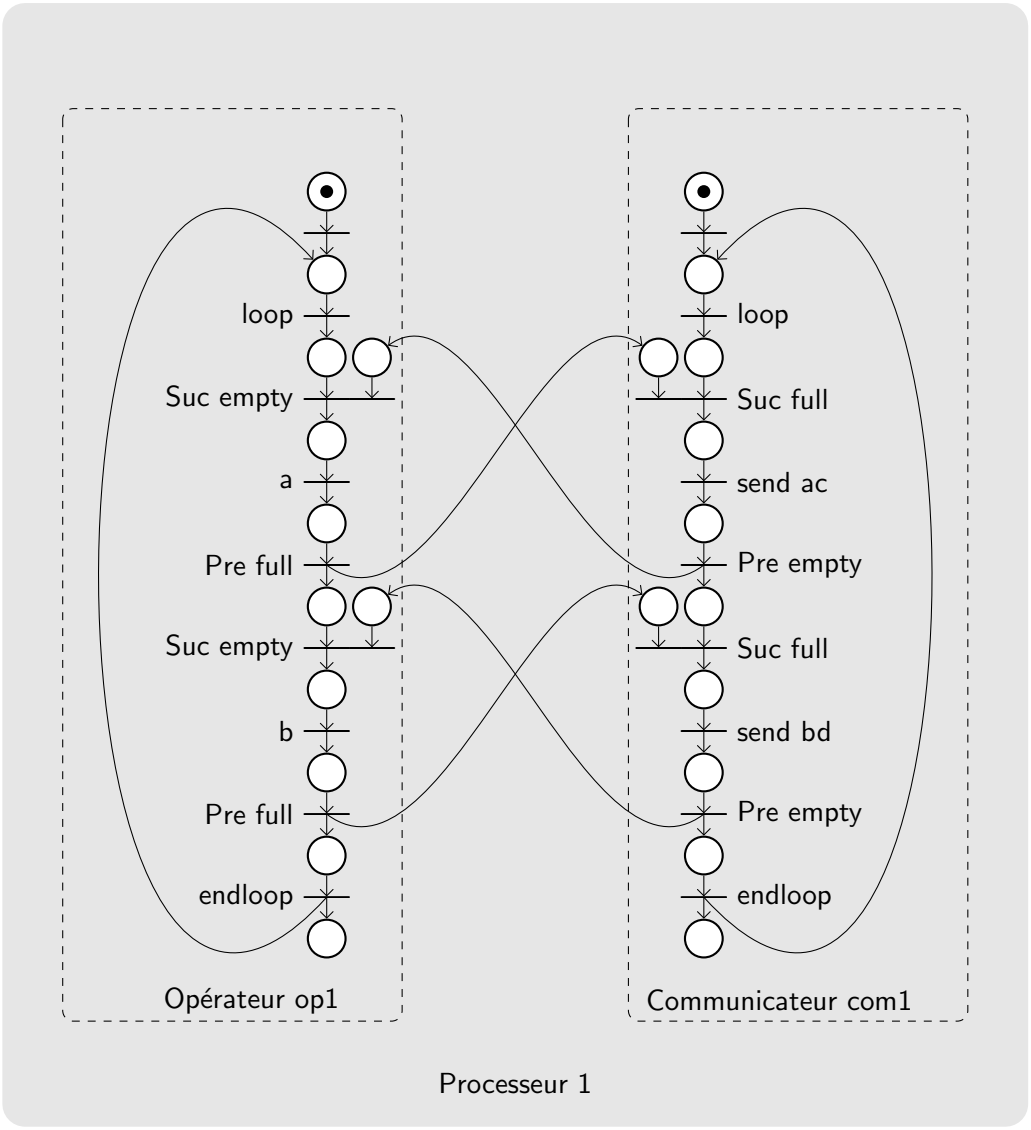


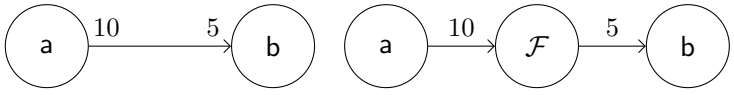

FIGURE 3.17: Réseau de Petri des séquenceurs de calcul et de communication du processeur 1

d'algorithme, mais définis implicitement par l'analyse des tailles des ports des différents nœuds du graphe.

**Description SynDEx d'un nœud $\mathcal{F}$**

Il s'agit d'une répétition d'un nœud déduit de ses arcs entrants. Si le rapport entre la taille du port de sortie sur lequel est connecté un arc entrant et de la taille du port d'entrée connecté à ce même arc est un entier strictement supérieur à un, alors il y a répétition de ce nœud. Lorsque plusieurs arcs entrent dans le nœud répété, il faut que leurs rapports soient égaux, sinon le graphe est *inconsistant*. L'inconsistance signifie que la déduction du nombre de répétitions d'une opération est contradictoire entre les rapports des différents arcs entrants. Le graphe développé laisse apparaître des nœuds $\mathcal{M}$ `implode` et $\mathcal{X}$ `explode` entre chaque arcs entrants du nœud répété pour permettre l'explosion des données produites.

La figure 3.18 illustre la répétition de l'opération $b$. La figure 3.18(a) représente un graphe avec deux opérations $a$ et $b$ reliées par un arc $(a,\ b)$. Les valeurs en début et en fin de l'arc représentent respectivement le nombre de données produites par $a$ et consommées par $b$. Le rapport entre la taille du port d'entrée de l'arc $(a,\ b)$ et la taille du port de sortie de $(a, b)$ spécifie implicitement le facteur de répétition de l'opération $b$. Dans l'exemple de la figure 3.18, le facteur est deux.



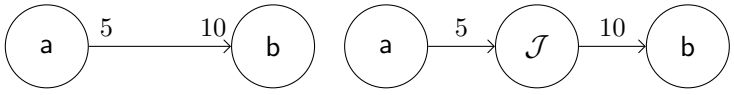

(a) Graphe SynDEx (b) Graphe factorisé

FIGURE 3.18: Répétition de l'opération $b$ avec explosion de données

**Description SynDEx d'un nœud $\mathcal{J}$**

Il s'agit d'une répétition implicite d'un nœud déduit de ses arcs sortants. Si le rapport entre la taille du port de sortie sur lequel est connecté un arc entrant et de la taille du port d'entrée connecté à ce même arc est un entier strictement supérieur à un, alors il y a répétition de ce nœud. Lorsque plusieurs arcs entrent dans le nœud répété, il faut que leurs rapports soient égaux, sinon le graphe est inconsistant. Le graphe développé laisse apparaître des nœuds $\mathcal{M}$ `implode` entre chacun des arcs sortants du nœud répété pour permettre l'explosion des données produites.

La figure 3.19 illustre l'implosion de données par l'ajout d'un nœud `implode` noté $\mathcal{M}$. L'utilisateur décrit sous SynDEx deux opérations $a$ et $b$ et un arc $(a,\ b)$. Le rapport entre la taille du port de sortie connecté à $(a,\ b)$ et la taille du port de sortie spécifie implicitement le facteur de répétition de $a$. Dans l'exemple de la figure 3.19, le facteur est deux.




(a) Graphe SynDEx (b) Graphe factorisé

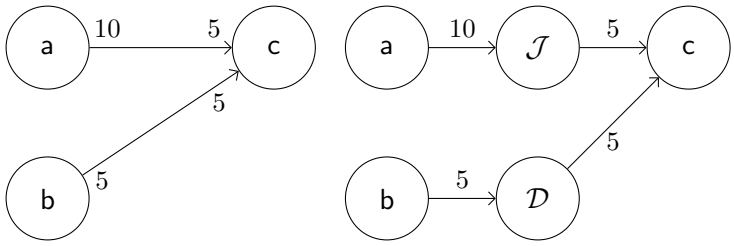
FIGURE 3.19: Répétition de l'opération $a$ avec implosion de données

**Description SynDEx d'un nœud $\mathcal{D}$**

La diffusion consiste à envoyer une même donnée sur plusieurs opérations ou sur plusieurs

répétitions d'une opération factorisée. La diffusion dans SynDEx est déduite des arcs entrants du nœud factorisé. Nous avons vu précédemment qu'il y a une répétition si le rapport est un entier strictement supérieur à 1 et que le graphe est consistant si les différents rapports sur les arcs entrants sont égaux. Cependant, certains arcs entrants peuvent avoir un rapport égal à 1. Nous sommes dans ce cas dans une situation de diffusion où l'extrémité initiale d'un tel arc est un nœud qui diffuse les données aux répétitions du nœud factorisé. Le graphe développé laisse apparaître un hyper-arc dont les extrémités finales sont les différentes répétitions du nœud factorisé.

La figure 3.20(a) illustre la diffusion de données par l'ajout d'un hyper-arc entre $b$ et les deux répétitions de $c$. Le rapport entre la taille du port d'entrée connecté à $(a, b)$ et la taille du port de sortie spécifie implicitement le facteur de répétition de $c$ qui est 2. Le rapport sur l'arc $(b, c)$ est de 1, il y a donc une diffusion de données. Le graphe développé est illustré sur la figure 3.20(b).



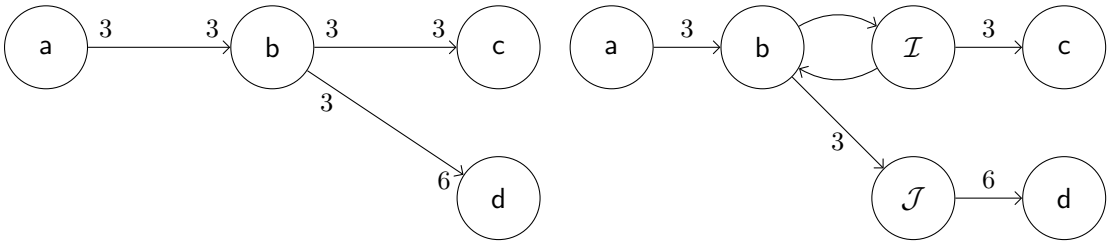

(a) Graphe SynDEx (b) Graphe factorisé

FIGURE 3.20: Diffusion

**Description SynDEx d'un nœud $\mathcal{I}$**

L'itération est la situation duale de la diffusion. Elle est déduite des arcs sortants d'un nœud factorisé. Elle apparaît lorsqu'il existe un rapport strictement supérieur à 1 sur les arcs sortants et que certains arcs sortants ont un rapport égal à 1. Nous sommes dans ce cas dans une situation d'itération. Dans un tel cas, la sortie de la i*ème* répétition du nœud factorisé devient l'entrée de la $(i+1)$*ème* répétition.

La figure 3.21 illustre le développement de l'opération *iterate*. Le rapport sur l'arc $(b, d)$ spécifie implicitement le facteur de répétition de $b$ qui est de 2. L'arc $(b, c)$, quant à lui, a un rapport de 1, il y a donc une itération du nœud factorisé. Le graphe développé est illustré par la figure 3.20(b) où la 1*ère* répétition $b_1$ est connectée à $b_2$. Il faut remarquer qu'une itération ne peut être effectuée qu'à partir d'un nœud fonction puisqu'il nécessite des ports d'entrée et de sortie pour le passage de la $i$*ème* à la $(i+1)$*ème* itération.




(a) Graphe SynDEx (b) Graphe factorisé

FIGURE 3.21: Itération

## 3.5 Conclusion sur la méthodologie AAA

Dans ce chapitre, nous avons détaillé les modèles de la méthodologie AAA, modèles notamment utilisés dans le logiciel SynDEx. L'idée majeure de cette méthodologie est de distinguer dès la phase de spécification les algorithmes des architectures. De plus, afin de permettre des traitements automatiques, des représentations sous la forme de graphes sont utilisées. Une phase de placement/ordonnancement est alors mise en œuvre pour essayer de trouver la meilleure distribution et le meilleur ordonnancement de l'algorithme sur l'architecture multicomposants. Le résultat est alors un graphe d'implantation précisant le séquencement des opérations sur chacune des unités de calcul ainsi que les transferts de données et les synchronisations nécessaires entre ces unités de calcul. Ce graphe peut alors être traité dans une phase de génération automatique de code utilisé par un compilateur (pour des processeurs) ou par un outil de synthèse (pour des FPGA). Ces aspects théoriques ont été à la base des travaux sur l'outil SynDEx à l'INRIA Rocquencourt (pour plateformes multiprocesseurs) et de l'outil SynDEx-IC à l'ESIEE (pour des FPGA).

# Chapitre 4

# Travaux relatifs à l'outil SynDEx

De 1999 jusqu'à 2007, nos travaux étaient basés sur l'utilisation du logiciel SynDEx. Lors de ma thèse (1999-2002), il s'agissait de modéliser des applications complexes de traitement vidéo (décompression MPEG-4 Part2) à l'aide de cet outil pour aboutir à des implantations multiprocesseurs optimisées. Nous verrons dans ce chapitre qu'un grand nombre d'applications ont été développées dans notre équipe en utilisant ce logiciel SynDEx. Ces applications permettent non seulement d'illustrer des travaux de recherches comme nous le verrons par la suite, mais également de rationnaliser les travaux au sein du laboratoire. Notre investissement dans des projets collaboratifs étant souvent liés à un contexte applicatif, nous nous attachons à modéliser ces applications dans l'outil pour faciliter leur réutilisation dans les divers travaux de thèse.

Cette expérience dans la modélisation d'applications a montré l'intérêt de l'approche AAA ainsi que certaines limitations. Trois thèses [Rau06, Urb07, Roq08] ont eu pour objectif de lever certaines de ces restrictions. Ces travaux se situent en amont de SynDEx dans l'étape de modélisation, mais nécessitent aussi le développement en aval des bibliothèques de traduction du macro-code générique en un macro-code exécutable, puisque le code généré par l'outil SynDEx n'est pas un code compilable. Cette partie va permettre d'illustrer comment les travaux menés au sein de l'équipe ont permis d'améliorer l'utilisation du logiciel SynDEx et d'utiliser cet outil tout au long du processus de développement d'une application, de sa description fonctionnelle jusqu'à son optimisation sur une plateforme multiprocesseur embarquée.

## 4.1 Modélisation d'applications

Nous avons principalement développé des modélisations pour des applications de compression d'image et de vidéo basées sur les travaux internes au laboratoire (Codec LAR), et sur des standards MPEG-4 Part2, MPEG-4 AVC et SVC comme nous le détaillons dans cette partie. Il faut noter que la principale caractéristique de ces applications est qu'elles sont déterministes et, ainsi, bien adaptées au modèle d'application SynDEx. Les travaux en collaboration avec Mitsubishi ITE sur l'UMTS, avec le groupe CPR de l'IETR sur le MC-CDMA ont montré que la démarche pouvait être identique pour des domaines d'application connexes tel que les télécommunications.

### 4.1.1 Codage LAR

La méthode de codage LAR (***Locally Adaptive Resolution***) est une technique de compression d'images adaptative, avec ou sans perte, et développée à l'origine par Olivier Déforges [Déf04] au sein du laboratoire IETR. Initialement introduite pour le codage des images multi-niveaux de gris, la méthode a été étendue aux images couleurs. Le codage LAR utilise une résolution variable, localement adaptée à l'activité locale. Dans ce schéma de codage, le découpage en blocs utilisé n'est pas uniforme contrairement au codage JPEG, qui travaille sur des blocs de pixels $8 \times 8$ ou $16 \times 16$, limitant ainsi les effets de blocs.

Le codeur est composé de deux couches : un codeur de type spatial permettant les forts taux de compression, et un codeur de type spectral pour coder l'image d'erreur. La méthode offre ainsi un schéma de codage ***scalable*** où l'image est transmise progressivement par raffinement local de la résolution. La figure 4.1 représente le schéma général du codage et du décodage d'image LAR. Le codeur spatial est basé sur une représentation en blocs de l'image en fonction de l'activité (estimée par un gradient morphologique). Typiquement, les tailles de blocs vont de $2 \times 2$ à $16 \times 16$ par croissance dyadique.

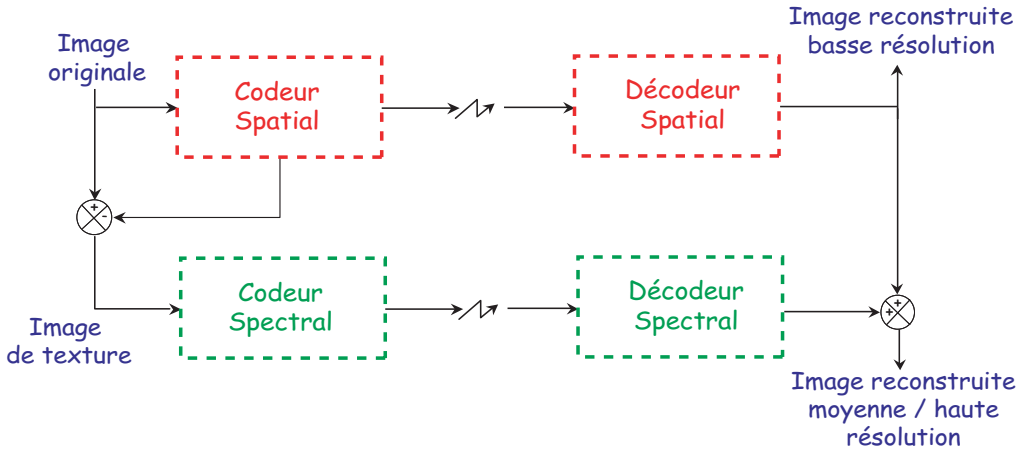



FIGURE 4.1: Structure générale du codage LAR

**Codeur spatial**

Le principe repose sur la notion de résolution locale (taille de pixel) variable. L'image est découpée en macroblocs de taille $16 \times 16$, eux-mêmes partitionnés selon une structure de type ***quad-tree*** adapté à l'activité locale de l'image. En choisissant comme critère une mesure de gradient dans le bloc dans un espace YUV, une image basse résolution est obtenue en remplissant chaque bloc par sa valeur moyenne. Elle est ensuite codée en utilisant un schéma prédictif de type MICD.

Une quantification psychovisuelle simple peut ainsi être mise en œuvre en appliquant une quantification forte pour les petits blocs situés sur les contours, et une plus fine pour les grands blocs localisés dans les zones uniformes. Un post-traitement simple est appliqué pour lisser les zones homogènes, puis un filtre d'interpolation est utilisé pour étendre l'image à la pleine résolution. Cette méthode de codage permet d'obtenir de forts taux de compression tout en supportant une représentation fidèle des contours. Ainsi, le rapport de compression entre l'image originale et l'image bas débit sortant du codeur spatial est de 30 à 40, avec une qualité visuelle meilleure que

celles obtenues par les schémas de compression standardisés (JPEG, JPEG 2000).

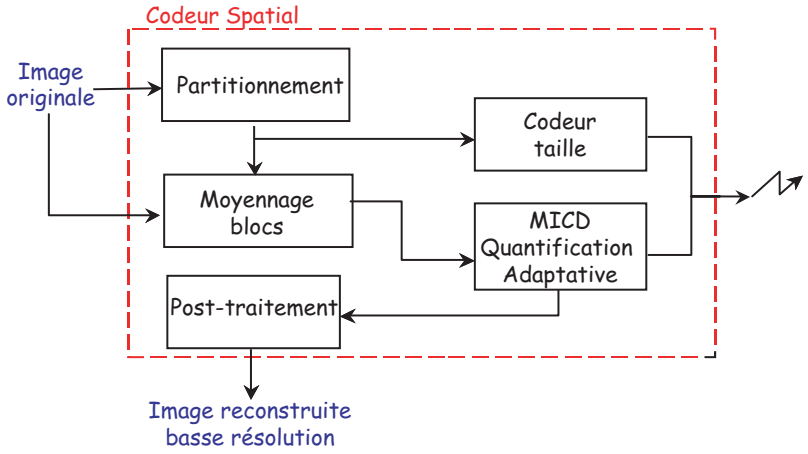



FIGURE 4.2: Codeur spatial du LAR

**Codeur spectral**

La texture des blocs (image d'erreur du codeur spatial) est alors codée avec le codeur spectral pour raffiner l'image. Une approche de type transformée en cosinus discret (***Discrete Cosinus Transform*** ou DCT) ou transformée de Hadamard à taille de bloc variable est appliquée à cette texture des blocs, où à la fois taille et composante continu (DC) sont déjà fournies par le codeur spatial. La nature des blocs permet ici un codage progressif dépendant du contenu : restauration de la texture des zones uniformes via les grands blocs et/ou amélioration des contours à travers les blocs $2 \times 2$, les blocs $4 \times 4$, les blocs $8 \times 8$ et les blocs $16 \times 16$. La figure 4.3 représente le schéma bloc du codage de l'image d'erreur avec une transformée de type DCT.

Dans les méthodes classiques, le passage au codage des images couleur consiste essentiellement à dupliquer pour les trois composantes le schéma général. Dans l'approche LAR, les images sont traitées dans un espace YUV, et les tailles de bloc sont estimées à partir des deux composantes de chrominance pour donner une grille unique pour les couleurs. Le codeur spatial seul, appliqué aux composantes UV, permet d'obtenir une très bonne qualité d'image d'un point de vue couleur.

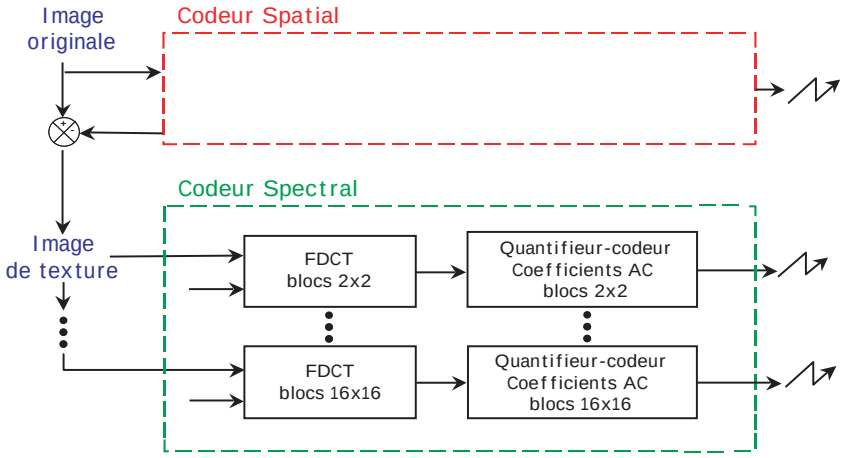



FIGURE 4.3: Codeur spectral du LAR

L'application LAR présente plusieurs avantages par rapport à d'autres applications. Tout d'abord, nous avons décrit l'ensemble du codec LAR, c'est-à-dire le codeur et le décodeur. Ainsi, il est possible d'intégrer cette application dans de nombreux contextes. Par exemple, pour illustrer une chaîne de communication numérique complète comme nous avons pu le faire dans le projet CPER PALMYRE, nous pouvons insérer le codeur avant la phase de modulation, et le décodeur à la suite de la démodulation. Un autre avantage de l'application LAR est qu'elle représente une application typique du traitement vidéo : découpage d'une image en blocs, calculs intensifs, nombre de données important et variable en fonction de la résolution traitée, caractère déterministe du traitement. En revanche, la représentation de l'application LAR sous le logiciel SynDEx n'utilise pas la fonctionnalité de conditionnement, fonctionnalité qui parfois peut amener à des résultats sous-optimaux dans la phase de placement/ordonnancement, ou qui peut poser des difficultés dans la génération de code comme nous le verrons par la suite. Enfin, les descriptions de l'application LAR étant réalisée au sein même du laboratoire, cette application donne une cohérence à l'ensemble de l'équipe : les spécialistes des algorithmes prennent ainsi conscience plus facilement des problématiques liées à la conception de systèmes embarqués, et de l'impact de leurs modifications algorithmiques dans ce contexte. Pour les spécialistes de la partie prototypage, l'application LAR fournit un cas concret d'application dans lequel il est possible de bénéficier en interne d'une grande expertise, et dont il est possible de modifier certains paramètres.

### 4.1.2 Décodeur MPEG-4 Part2

MPEG (*Moving Picture Expert Group*) est un groupe de travail de l'ISO (*Organisation Internationale de Normalisation*) chargé du développement de normes internationales pour la compression et la décompression de contenu multimédia (vidéo, audio...). Le décodeur MPEG-4 développé permet de décoder des vidéos rectangulaires (partie 2 de la norme), les plus couramment utilisées. Le standard MPEG-4 est divisé en ***profils***, limitant l'ensemble des outils à mettre en œuvre pour implémenter un codeur ou un décodeur. Pour chacun de ces profils un ou plusieurs ***niveaux*** ont été mis en place pour restreindre la complexité des calculs. Les travaux sur le décodeur MPEG-4 ont débuté dans le cadre de ma thèse [Nez02] et ont été complétés par Mickael Raulet [Rau06]. L'objectif était de permettre la description AAA/SynDEx du décodeur MPEG-4 Part2 et notamment les profils *Simple Profile* et *Advanced Simple Profile*. Ce décodeur est décrit sous la forme d'un graphe flux de données AAA/SynDEx et réalise le décodage des images *Intra* notées I, *Prédites* notées P, correspondant au profile simple (*Simple Profile* ou SP), ainsi qu'aux images *Bidirectionnelles* notées B correspondant au profile simple avancé (*Advanced Simple Profile* ou ASP).

Pour descrire le décodeur, différentes descriptions AAA/SynDEx ont été réalisées au sein du laboratoire. Elles diffèrent par le choix de la granularité des opérations. On distingue trois niveaux de description distincts du décodeur. Tout d'abord, un décodeur à gros grain, au niveau image (haut-niveau illustré dans la figure 4.5), a été réalisé. L'opération atomique qui réalise le décodage de l'image encapsule toutes les fonctionnalités à mettre en œuvre (VLC, scan inverse,...). Cependant, le choix de la granularité au niveau de l'image n'est pas satisfaisant pour une parallélisation efficace des opérations. En effet, une seule opération réalise le décodage complet d'une image qui ne peut donc être implantée que sur un unique processeur. Les composantes de luminance et de chrominance YUV peuvent donc être traitées indépendamment. De plus, l'opération de décodage d'une image contient un grand nombre de sous-algorithmes (prédiction AC/DC, DCT inverse, ...) qui réalisent des opérations distinctes. Prendre en compte ces différentes opérations dans la description peut alors augmenter le parallélisme potentiel de l'application.

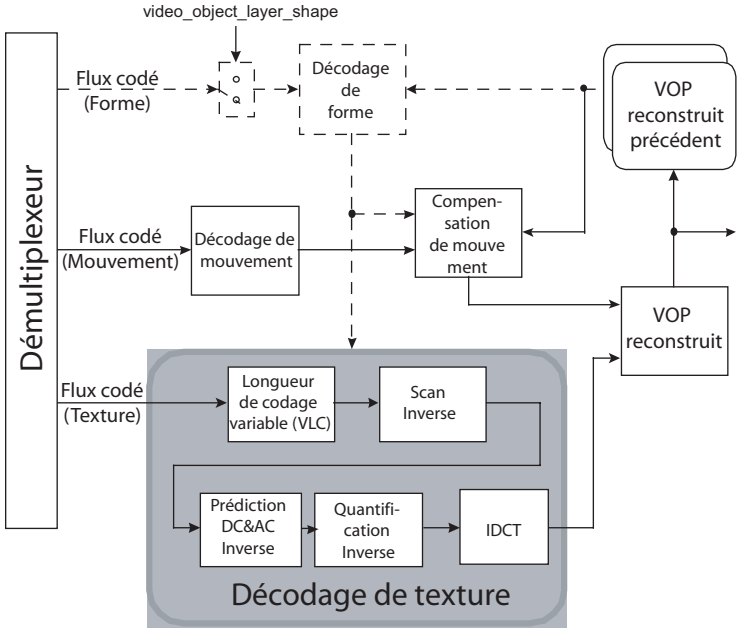



FIGURE 4.4: Schéma général du décodeur MPEG-4 Part2

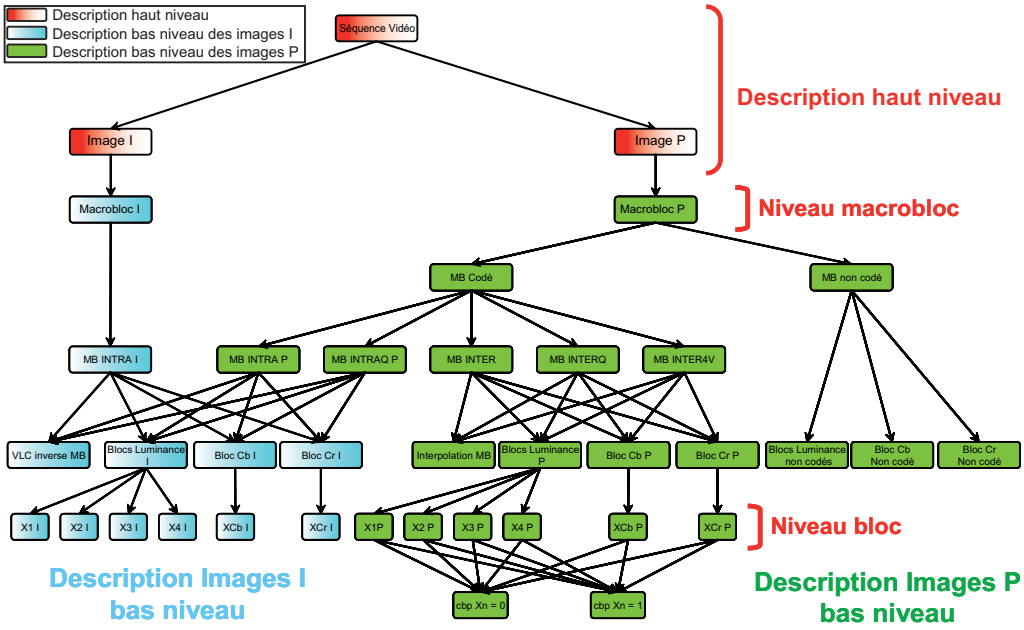



FIGURE 4.5: Schéma bloc MPEG-4 Part2 images I et P bas niveau

Un décodeur à grain fin (niveau bloc dans la figure 4.5) a alors été réalisé afin de mettre à jour les opérations de plus bas-niveau contenu dans le décodage telles que la VLC inverse, la DCT inverse ou encore la prédiction AC/DC, qui traitent les données au niveau bloc ou macrobloc (un macrobloc étant constitué de 6 blocs $8 \times 8$ dans un format 4 :2 :0, 4 pour la composante de luminance et 1 pour chaque composante de chrominance). Dans cette description, les 6 blocs qui composent un macrobloc couleur sont modélisés par des opérations hiérarchiques distinctes où les opérations atomiques telles que la DCT inverse ou la prédiction inverse AC/DC apparaissent comme des nœuds du graphe. Cette description traitant les blocs $8 \times 8$ a permis une meilleure exposition du parallélisme. Cependant, cette description n'a pas apporté de bons résultats (pour des chronométrages sur cible, se référer directement aux thèses [Nez02, Rau06, Urb07]) lors de nos implantations puisque les décodeurs générés s'avéraient plus lents. deplus, la quantité de données allouée sur chacun des composants était accrue et l'exécution même de l'adéquation par le logiciel SynDEx devenait problématique.

Cette perte de performance peut s'expliquer par plusieurs éléments. Tout d'abord, les modèles utilisés dans SynDEx ne prennent pas en compte le temps d'initialisation de transferts (celui-ci étant considéré comme nul) dans le processus de placement/ordonnancement, temps qui devient non négligeable à grain fin par rapport à celui d'une opération élémentaire. De plus, le logiciel SynDEx réalise une mise à plat du graphe avant le placement/ordonnancement et ne réalise pas de regroupement d'opérations (factorisation) pour limiter la taille des graphes à traiter dans le placement/ordonnancement. Dans notre cas, le nombre de nœuds du graphe après mise à plat est proportionnel au nombre de macroblocs de l'image. Ainsi, le décodeur à grain fin pour une solution d'image QCIF (99 macroblocs) est un graphe constitué de 32 273 nœuds. D'une part, avec une complexité de plus de 10 000 nombre de nœuds (ce qui correspond à 40 macroblocs) l'algorithme ne termine pas son exécution. D'autre part, l'algorithme glouton mis en œuvre pour le placement/ordonnancement utilise une fonction de coût qui ne voit pas plus loin que le voisinage directement connecté à l'opération déjà ordonnancée : ceci implique un résultat de placement/ordonnancement assez écarté de la solution optimale. En outre, la taille du code généré étant directement liée au nombre de nœuds dans le graphe d'algorithme, les fichiers générés étaient trop importants pour être compilables par certains compilateurs comme CCS pour les DSP Texas Instruments. Enfin, la taille des données allouées dans la génération de code était trop importante pour placer ces données en mémoire interne et pour pouvoir bénéficier des accélérations rendues possibles par les architectures DSP.

Toutes ces raisons nous ont poussé à utiliser une description à grain intermédiaire (niveau macrobloc). Cette description est basée sur une opération atomique traitant un macrobloc couleur entier ($6 \times 8 \times 8$).

La description est faite de manière hiérarchique afin de réaliser le conditionnement de l'exécution des opérations. Le conditionnement permet en effet de distinguer le décodage des images de type I, P et B. Il faut ajouter à cela une condition pour prendre en compte les images notées S (*Skipped*) qui sont des images non traitées par le décodeur et tout simplement ignorées.

Le décodeur MPEG-4 Part2 a fait l'objet de nombreux portages sur les cartes multiprocesseurs du laboratoire et a été intégré dans les produits de la société Innes [inn], jeune entreprise rennaise spécialisée dans les technologies d'affichage dynamique. Enfin, cette expérience à été la base des sujets de recherche relatifs au logiciel SynDEx détaillés dans ce chapitre (notamment la minimisation mémoire ou la gestion de la mémoire cache) comme pour ceux menés actuellement sur le logiciel Preesm (chapitre 5).

### 4.1.3 MPEG-4 AVC et SVC

Le standard SVC (Scalable Video Coding) est le nom donné à une extension du standard H264/MPEG-4 AVC. H.264/MPEG-4 AVC a été développé conjointement par l'ITU-T et l'ISO/IEC JTC 1. Ces deux groupes collaborent dans le JVT (Joint Video Team). Les travaux sur ces standards ont été réalisés lors des projets du pôle de compétitivité Images et Réseaux Mobim@ges (AVC), Scalim@ges (SVC) principalement par Médéric Blestel sur un contrat d'ingénieur de recherche (CDD) et se poursuivent à l'heure actuelle dans le projet SVC4QoE (projet FUI8 débuté en octobre 2009). Le décodeur SVC a été appelé *Open SVC Decoder*, et il est disponible en *open source* [SVC].

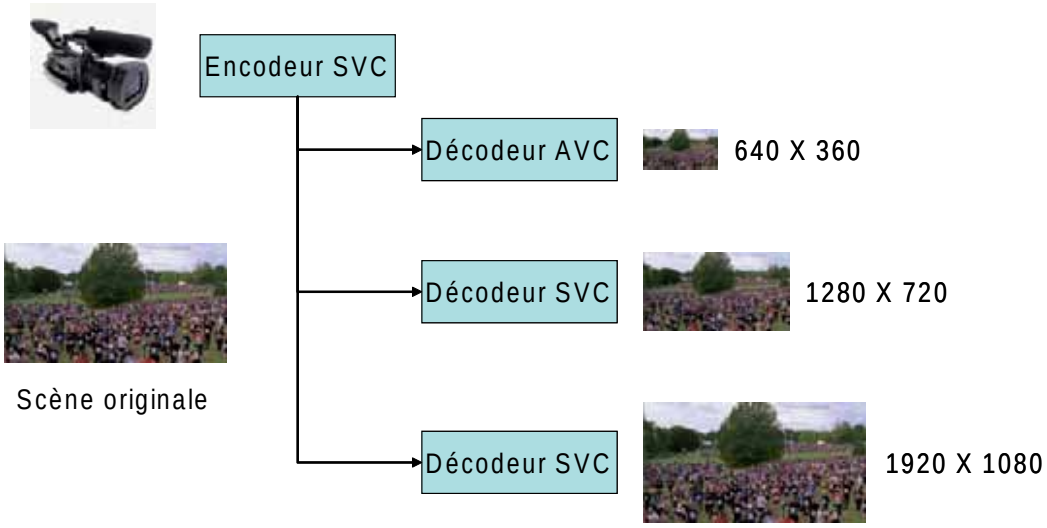



FIGURE 4.6: Schéma de principe de la scalabilité spatiale

Le décodeur AVC est compatible avec le profile ***baseline***. De plus, certaines fonctionnalités du profile ***main*** ont été ajoutées comme les images B. Notre solution intègre aussi certaines fonctionnalités du profile ***high*** comme l'utilisation de transformations $8 \times 8$ au lieu d'une IDCT $4 \times 4$, l'utilisation de tables de quantifications propriétaires et la gestion du pas de quantification différent entre les images de luminances et celles de chrominances. Seules quelques fonctionnalités du profile ***main*** n'ont pas été implantées dans notre solution comme le codage entrelacé (PicAFF, MBAFF) et la prédiction pondérée : elles demeurent peu utilisées et trop coûteuses pour être envisagées dans un contexte embarqué. De plus, ces fonctionnalités ne sont pas reprises dans la spécification du codage SVC, ce qui limite leur intérêt.

SVC étend les fonctionnalités d'AVC en ajoutant de la scalabilité. Le principe de la scalabilité est de pouvoir, à partir d'un même flux codé, décoder un même contenu dans plusieurs formats de qualités variables en fonction des débits de transmission disponibles et/ou de la puissance de calcul du terminal. Cette notion de scalabilité est illustrée par la figure 4.6. La scalabilité peut être temporelle (ajout d'images), spatiale (agrandissement de la taille des images) et/ou qualitative (augmentation du débit par image). Ainsi, les flux codés SVC s'adaptent plus facilement aux contraintes liées aux réseaux de distribution (optimisation de la bande passante) et aux terminaux (adaptation du flux aux puissances de calcul disponibles). De plus, il est possible avec SVC d'utiliser plusieurs réseaux de distributions simultanément pour un même contenu et améliorer ainsi le service rendu à l'utilisateur final.

Les fonctionnalités de notre décodeur SVC sont conformes au profile ***scalable baseline***. Le décodeur SVC implante les images B, le codage entropique CABAC, les transformées 8x8 sur la luminance dans les étages d'amélioration. La scalabilité spatiale est restreinte aux ratios 1.5 et 2 entre deux étages de scalabilité. Les scalabilités en qualité et temporelle sont supportées sans restriction.

TABLE 4.1: Comparatif entre le JSVM et *Open SVC Decoder*

| | JSVM | | *Open SVC Decoder* | | | |
|---|---|---|---|---|---|---|
| Séquence | tps (ms) | fréq (img/sec) | tps (ms) | fréq (img/sec) | Résolutions | Accélération |
| SVCBST-1 | 31.2 | 3.2 | 0.873 | 114.5 | 360x240 et 720x480 | 36 |
| SVCBST-2 | 23.3 | 4.3 | 0.873 | 114.5 | 360x240 et 720x480 | 26 |
| SVCBST-14 | 137 | 0.14 | 2.692 | 7.4 | 640x480, 1280x720 et 1920x1080 | 52 |
| SVCBMST-3 | 20.8 | 2.88 | 1.76 | 34.09 | 360x240, 360x240 et 720x480 | 16 |

A l'heure actuelle, il n'existe que deux autres solutions de décodage SVC : le JSVM (*Joint Scalable Video Model*) qui est le logiciel de référence fournit par le JVT [JSV], et un décodeur développé par IMEC [IME]. L'IMEC a optimisé le code du JSVM et propose un décodeur jusqu'à 20 fois plus rapide que le logiciel de référence. Les tests réalisés sur notre solution SVC donnent des résultats jusqu'à 52 fois plus rapides que le JSVM. Ces tests (Tab. 4.1) ont été réalisés en utilisant la version 9.16 du JSVM à l'aide d'un PC équipé d'un processeur Intel Core 2 duo cadencé à 2.4 GHz. Le projet a été développé en utilisant SynDEx et a été testé sur diverses plateformes comme un PDA (processeur ARM ou xscale) et sur des DSP TI TMS320C64x.

### 4.1.4 Estimation de mouvement pour l'encodage vidéo AVC/SVC

Cette étude a été menée dans le cadre de la thèse CIFRE de Fabrice Urban, réalisée en collaboration avec Thomson Corporate Research. Cette application a été choisie car elle représente l'opération la plus coûteuse en calculs dans un algorithme de codage vidéo. De plus, c'est une opération dont le potentiel de parallélisme est fort comparativement à des algorithmes de décodage : la lecture d'un flux est en effet séquentielle et les algorithmes de codages actuels traitent les données en macroblocs dans un ordre ***raster*** avec une prédiction spatiale. L'estimation de mouvement a été spécialement étudiée dans le contexte du standard H264/AVC sur des résolutions HD, apportant quelques spécificités comme le codage de blocs de taille variable ou comme la précision au quart de pixel.

L'estimation de mouvement est une opération complexe dans le sens où il existe un grand nombre de paramètres (résolution, taille de la zone de recherche) qui influent sur la qualité des prédictions de mouvement et donc directement sur le débit obtenu, sur la complexité des calculs à réaliser ainsi que sur la taille mémoire requise. Les travaux ont permis d'optimiser l'estimation de mouvement sur une plateforme DSP/FPGA en proposant notamment un nouvel algorithme appelé HDS (*Hierarchical Diamond Search*). Cet algorithme est basé sur la notion de décomposition hiérarchique présente dans l'algorithme HME (*Hierarchical Motion Estimation*). Au lieu de faire

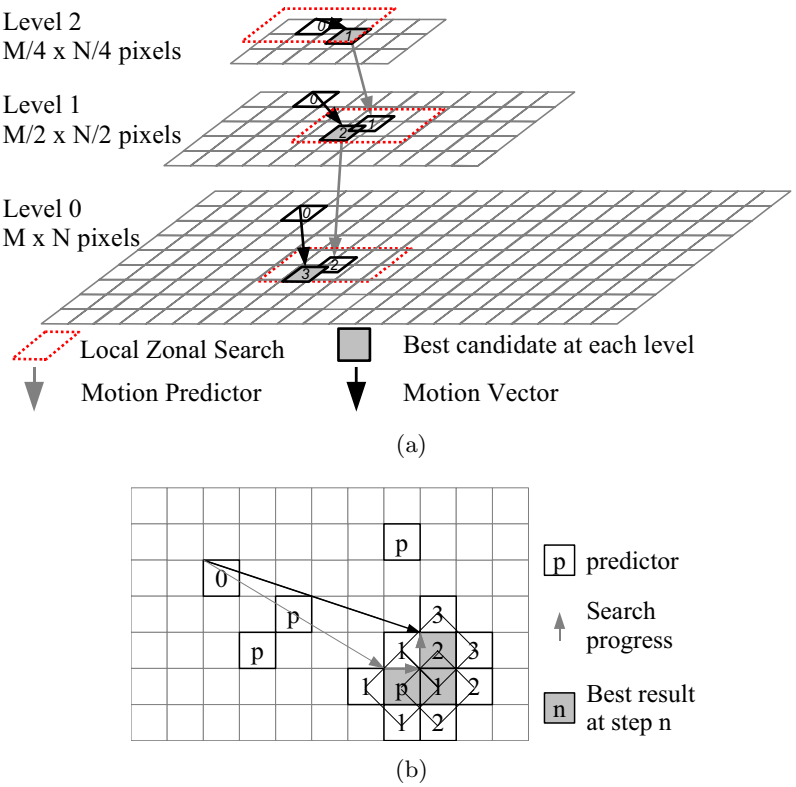
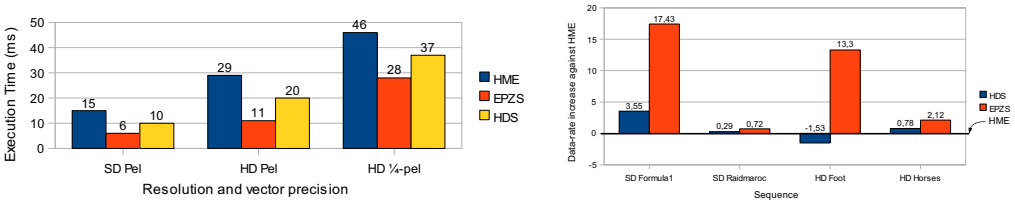



FIGURE 4.7: (a) Décomposition hiérarchique ; (b) Raffinement local en diamant

une recherche exhaustive sur chacun des niveaux hiérarchiques comme c'est le cas dans l'algorithme HME, HDS utilise un raffinement local en diamant, solution utilisée dans l'algorithme EPZS et qui constitue une référence dans le domaine. La technique est illustrée par la figure 4.7.




(a) complexité des calculs (b) qualité des vecteurs de mouvement

FIGURE 4.8: Comparaison des algorithmes HME, EPZS et HDS

L'avantage de l'algorithme HDS est d'améliorer considérablement l'encodage des vecteurs de mouvement par rapport à EPZS en utilisant les prédicteurs hiérarchiques de HME, tout en réduisant considérablement la charge de calcul de l'algorithme HME. La figure 4.8 illustre ces gains en termes de temps de compression (HDS se situe entre HME et EPZS) et en termes de qualité des vecteurs de mouvement. La qualité des vecteurs de mouvement est illustrée en comparant le taux de compression d'un encodeur H264 utilisant les algorithmes HDS et EPZS par rapport aux taux obtenus en utilisant l'algorithme HME, et ceci pour une qualité visuelle constante. Ces résultats ont été valorisés par deux parutions dans deux revues internationales [UPND08, UNR09].

## 4.2 Optimisations du processus de génération de code

### 4.2.1 Principe de fonctionnement

Dans la méthodologie AAA, la phase de placement/ordonnancement est suivie d'une phase de génération de code (Fig. 3.1). A la fin du processus de génération par SynDEx, il existe autant d'exécutifs générés que d'opérateurs dans l'architecture, chacun d'eux correspondant à un fichier distinct. Chaque fichier d'exécutif est un code intermédiaire indépendant de l'opérateur, c'est-à-dire du processeur. Il est composé d'une liste d'appels de macros qui seront traduites par un macro-processeur dans le langage source choisi pour l'opérateur correspondant (C, ou assembleur par exemple) comme illustré dans la figure 4.9. Chacun de ces programmes sources sera ensuite compilé et chargé sur les opérateurs par des outils spécifiques (par exemple *Visual C++* pour PC ou *Code Composer Studio* sur DSP Texas Instruments). Les traductions spécifiques pour les différentes cibles sont contenues dans des dictionnaires appelés *noyaux d'exécutifs SynDEx*.

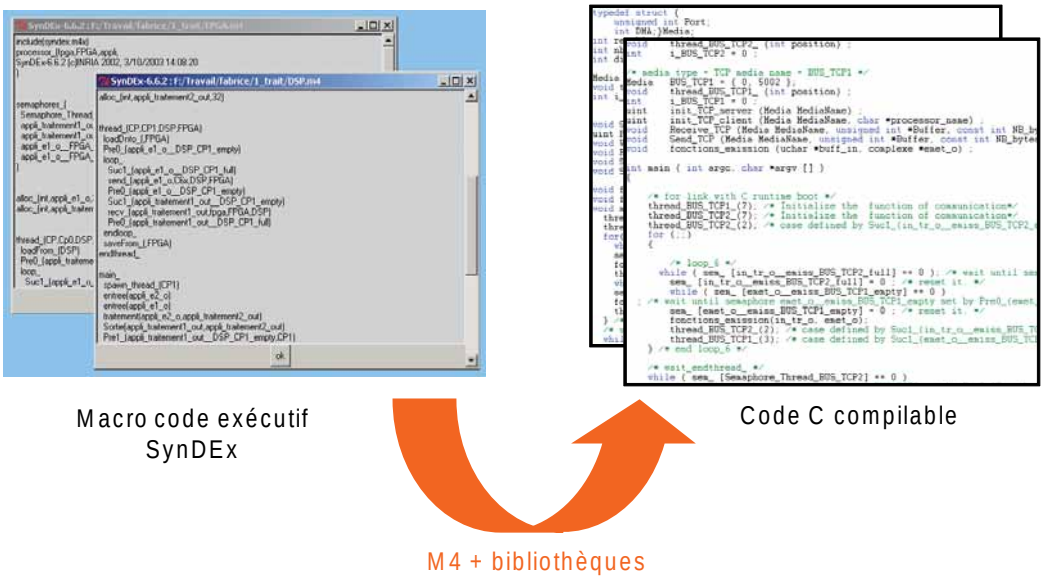



FIGURE 4.9: Etapes de la génération de code

Le développement des noyaux d'exécutifs SynDEx a débuté au laboratoire en 2001 lors d'un stage réalisé par Mickael Raulet et Yann Le Mener. Il s'agissait de créer un noyau dédié au DSP de la famille C6x de TEXAS INSTRUMENTS. Fabrice Urban et Ghislain Roquier ont continué à améliorer les fonctionnalités de ces noyaux.

Les définitions de macros qui sont dépendantes de l'opérateur (du processeur) peuvent être classées en deux ensembles. Le premier ensemble est un jeu extensible de macros applicatives réalisant les opérations de l'algorithme. Le second ensemble, que nous appelons *noyau d'exécutif*, est un jeu de de macros système qui supportent :

- le chargement initial des mémoires programmes,
- la gestion mémoire (allocation statique, copies et fenêtres glissantes de macro-registres),
- le séquencement (sauts conditionnels et itérations finies et infinies),
- les transferts de données inter-opérateurs (macro-opérations de communication transférant le contenu de macro-registres),

– les synchronisations inter-séquences (assurant l'alternance entre écritures et lectures de chaque macro-registre partagé entre la séquence de calcul et les séquences de communication),
– le chronométrage (pour permettre la mesure des caractéristiques des opérations de l'algorithme et des performances de l'implantation).

Nous verrons dans la suite qu'un noyau d'exécutif regroupe un type de processeur et les types de média connectés à ce processeur. Nous appellerons bibliothèques les définitions des macros associées à chaque type de processeur et à chaque médium de communication.

### 4.2.2 Organisation des noyaux d'exécutifs en bibliothèques

Comme évoqué précédemment, les bibliothèques SynDEx sont nécessaires pour la génération automatique de code. Elles contiennent la traduction des macros du macro-code généré par SynDEx en des fonctions compilables. Pour rendre ces bibliothèques les plus génériques possibles et les réutiliser facilement, nous avons classé ces bibliothèques de façon hiérarchique (Fig. 4.10). L'approche a été validée par l'utilisation des nombreuses plateformes en notre pocession et appartenant à diverses constructeurs (Sundance, Vitec, Pentek, Spectrum Digital). Le changement de plateforme de prototypage ou l'ajout de nouveaux composants nécessite bien sûr le développement de nouveaux noyaux.

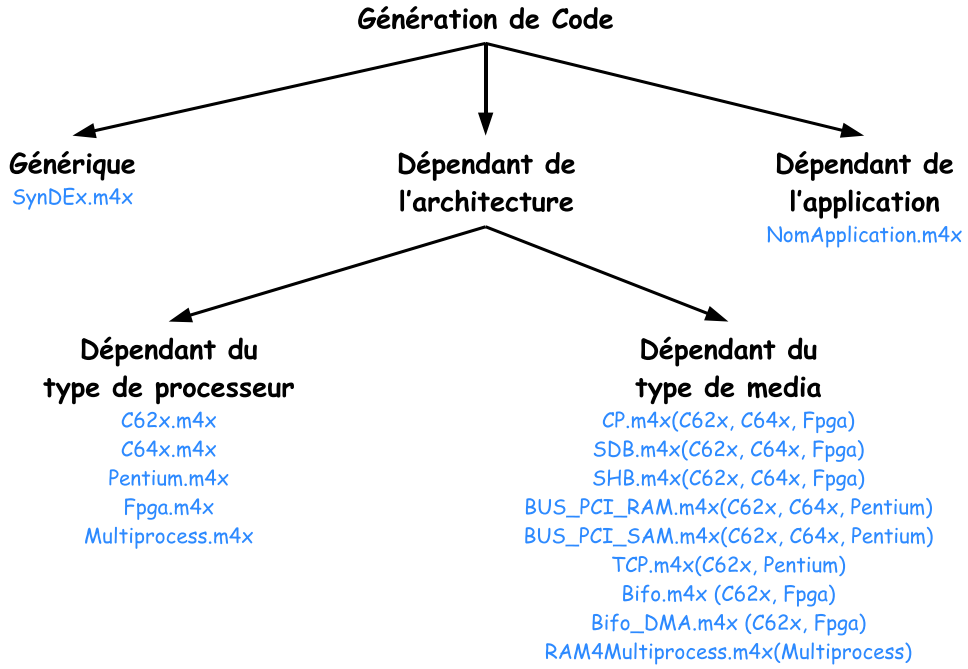



FIGURE 4.10: Arborescence des bibliothèques SynDEx

Ces bibliothèques constituent le cœur de notre chaîne de prototypage, du portage de nos applications dans le monde embarqué (architecture multi-DSP).

**Noyau générique.** Cette bibliothèque contient les traductions génériques (valables pour toutes les cibles). Cela concerne seulement les macros indépendantes de l'architecture. De ce fait elle

n'évolue que très rarement.

**Bibliothèque de l'application.** Cette bibliothèque contient les traductions des appels des fonctions de l'application. Cette bibliothèque est simple et est écrite par l'utilisateur. Les propriétés de l'application sont également contenues dans ce fichier, telles que des options de chronométrage ou d'optimisations spécifiques.

**Bibliothèques pour les processeurs** Ces bibliothèques prennent en charge les opérations au niveau système, c'est-à-dire les allocations mémoires, la création et l'exécution des tâches, ainsi qu'une couche de base pour la gestion des interruptions lors des communications. Celles qui vont nous intéresser plus particulièrement permettent de générer l'exécutif pour :

- un PC sous *Windows,* monoprocesseur ou multiprocesseur (multitâche),
- un DSP C6x utilisant l'OS temps réel développé par Texas Instruments DSP/BIOS,
- un FPGA.

Le code est écrit en langage C pour garantir une grande généricité et un maintien (mise à jour) aisé. La génération de code pour FPGA a pour objectif de fournir automatiquement les bibliothèques de communication DSP-FPGA.

La modélisation d'un FPGA étant différente de celle d'un processeur, sa prise en compte dans le logiciel SynDEx et dans la génération de code associé s'avère réalisable. Cependant, il n'est possible d'exécuter qu'une opération du graphe d'algorithme sur ce type de composant.

**Bibliothèques de communication.** Elles sont très importantes pour distribuer l'application. Pour avoir un intérêt, les communications doivent être rapides et garantir le transfert des données de manière sûre. Les synchronisations étant générées par SynDEx, et donc s'appuyant sur un modèle théorique, elles sont valides par construction (pas d'interblocage ou d'erreur de synchronisation). La rapidité est obtenue en utilisant les périphériques disponibles sur les plateformes de manière optimale : le modèle (SAM ou RAM) le plus adapté est choisi, un DMA (Direct Memory Access) est utilisé dans la mesure du possible. L'utilisation du DMA permet de décharger l'unité de calcul, de synchroniser les média matériellement, et d'utiliser les interruptions pour détecter la fin des transferts : les calculs et les transferts mémoire concurrents deviennent alors envisageables. On se rapproche ainsi du modèle théorique utilisé par SynDEx pour calculer la latence. Les synchronisations générées (`Suc` et `Pre`) garantissent un fonctionnement correct. Les bibliothèques développées ont été :

- le bus SHB (Sundance High speed Bus) et SDB (Sundance Digital Bus) (modèles SAM) , pour réaliser des communications entre DSP sur les plateformes Sundance,
- le bus PCI (modèle RAM) pour Sundance permettant de faire communiquer un DSP avec le processeur du PC dans lequel la carte est intégrée,
- le bus Vitec (modèle RAM) dédié à la carte multi-DSP Vitec permettant d'interconnecter les DSP et le processeur de PC,
- le bus TCP (modèle SAM) permettant d'interconnecter deux PC, mais aussi d'émuler une plateforme multiprocesseur sur un seul PC en interconnectant les processus,
- le bus PIPE (modèle SAM) qui est une FIFO logicielle interconnectant des processus ou des tâches,
- le bus RAM (modèle RAM) interconnectant des tâches.

Le développement des noyaux PCI et TCP permet d'intégrer des PC aux diverses plateformes de développement. Des fonctionnalités sur PC, telles que l'affichage ou la lecture d'un fichier sur disque dur, peuvent donc être facilement utilisées sur toutes les cartes à notre disposition.

### 4.2.3 Minimisation de la mémoire

L'introduction de la minimisation de la mémoire dans la méthodologie AAA a été étudiée dans le cadre de la thèse de Mickaël Raulet [Rau06].

#### *4.2.3.1 Motivations*

Suite aux études réalisées sur les applications vidéo, nous avons pu nous rendre compte que la taille de la mémoire allouée dans le code généré par SynDEx était trop importante pour cibler des systèmes embarqués. Diminuer cette taille mémoire est donc naturellement devenu un de nos objectifs de recherche. Dès lors, plusieurs approches pouvaient être envisagées. La solution a été de travailler sur la phase d'allocation de la mémoire dans le code généré par SynDEx, c'est à dire sur le graphe développé après l'adéquation. En effet, ce graphe contient souvent un grand nombre de nœuds pour lesquels SynDEx va effectuer une allocation mémoire pour chacun de ses ports de sortie. Cela conduit à une allocation mémoire globale qui peut être conséquente dans le code généré.

#### *4.2.3.2 Principes*

Deux algorithmes de minimisation mémoire ont été mis en œuvre. Ces deux algorithmes reposent sur l'analyse de la durée de vie des données devant être allouées dans la phase de génération de code. Un graphe d'intervalles est utilisé afin de représenter la durée de vie des ces données lors d'une exécution. La technique du coloriage de graphe appliqué au graphe d'intervalles permet alors de réutiliser un même segment mémoire pour plusieurs données.

Le premier algorithme, appelé ***registre*** utilise des ensembles de données à la fois de mêmes tailles et de mêmes types, pour cela des données allouées dans le séquenceur de calcul (Fig. 3.17). Le second, appelé ***tétris***, exploite des ensembles de données de mêmes types mais pas nécessairement de mêmes tailles. De plus, l'algorithme ***tétris*** permet de prendre en compte les segments de mémoire qui contiennent des données qui vont être communiquées (utilisées dans un séquenceur de communication). Pour cela, le placement des sémaphores a été modifié lors de la génération de code tout en gardant les spécificités du graphe temporel fourni par SynDEx. Ceci modifie le graphe d'intervalle en minimisant les buffers communiqués entre processeurs.

L'ensemble de ces techniques ont été validées sur les applications vidéo du laboratoire mais également sur des algorithmes de démodulation FM et une chaîne de modulation/démodulation UMTS en collaboration avec Mitsubishi ITE, ainsi que sur une chaîne de MC-CDMA développée dans le groupe CPR de l'IETR. En fonction des caractéristiques des graphes d'application, des gains très variables ont pu être constatés : un faible gain (10%) dans le cas de la démodulation FM, un gain non négligeable dans le cas des chaines UMTS et MC-CDMA (6 fois moins de données allouées) et un gain allant jusqu'à 52 sur des applications de décodage vidéo MPEG décrites à grain intermédiaire comme le montre le tableau 4.2. Dans cet exemple, les 3 versions de description de l'application MPEG-4 Part2 sont comparées sur une séquence d'images de 80*64 pixels (20 macroblocs) et de 176*144 pixels (99 macroblocs). Dans la version de SynDEx, plus

on descend dans la hiérarchie de l'application, plus le nombre d'opérations augmente et plus la mémoire utilisée augmente. Avec la minimisation mémoire ***tétris***, il se trouve que la description intermédiaire prend moins de mémoire que la version de haut niveau. On peut donc conclure que la minimisation mémoire est très performante. D'autre part, plus on utilise une granularité fine dans la description, plus le parallélisme potentiel est important et plus la taille de la mémoire nécessaire augmente.

| | taille 80*64 | | | taille 176*144 | | |
|---|---|---|---|---|---|---|
| | *SynDEx** | *Tétris** | *Gain* | *SynDEx** | *Tétris** | *Gain* |
| haut niveau | 436 | 409 | 1,07 | 1 070 | 937 | 1,14 |
| niveau intermédiaire | 3 350 | 291 | 11 | 40 480 | 780 | 52 |
| bas niveau | 7 876 | 768 | 10 | x | x | x |

* occupation mémoire en Ko
x non compilable (trop d'opérations pour le compilateur *TEXAS INSTRUMENTS)*

TABLE 4.2: Occupation mémoire du décodeur MPEG-4 suivant la granularité de la description

### 4.2.4 Utilisation de la mémoire cache

L'utilisation de la mémoire cache dans la méthodologie AAA a été réalisée dans le cadre de la thèse de Fabrice Urban [Urb07].

#### *4.2.4.1 Motivations*

La méthodologie AAA/SynDEx ne prend pas en compte la spécificité de l'architecture mémoire des composants embarqués. Or, sur un processeur de traitement du signal, il peut exister plusieurs niveaux de mémoire :

- une mémoire très réduite fonctionnant à la vitesse du processeur contenant des données temporaires (cache de niveau 1 ou "scratchpad"),
- une mémoire interne restreinte légèrement plus lente (cache de niveau 2 ou RAM interne),
- une mémoire externe, dont l'accès est considérablement plus lent mais de grande taille (RAM externe).

Pour des applications de traitement vidéo, les mémoires internes ne sont pas assez grandes, notamment pour la haute définition. Les données sont donc allouées en mémoire externe, dont l'accès est très lent, pouvant ainsi réduire les performances d'un facteur cent. Pour accélérer les traitements, un mécanisme d'accès aux données manuel doit souvent être mis en place pour rapatrier les données utiles de manière temporaire en mémoire interne. Cependant cette solution n'est pas générique, ce qui exige une bonne maîtrise de l'architecture et de l'algorithme pour mettre en place une technique efficace. De plus, le développement d'un tel mécanisme peut être long. Afin de réduire le processus de portage sur DSP, nous voulons utiliser le mécanisme de cache offert par les DSP C64x de Texas Instruments. Dans un contexte d'applications distribuées, nous nous heurtons cependant très vite au problème de cohérence de cache. L'approche que nous proposons est complémentaire avec une solution de minimisation mémoire comme celle que nous venons de présenter. Un exemple typique qui nécessite une optimisation globale des allocations mémoire et

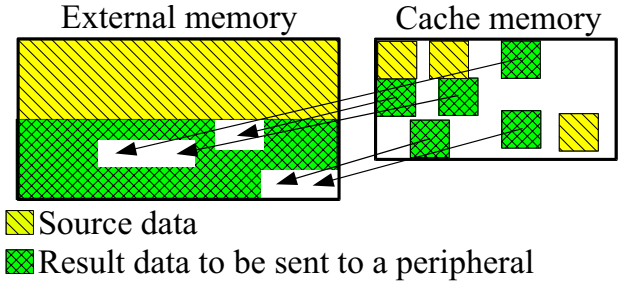
de la gestion des mémoires caches est celui du traitement des images haute résolution où la taille d'une image est supérieure à celle des mémoires internes.

La principale avancée de ces travaux réside dans la gestion de la cohérence de cache entre processeurs réalisée directement dans le code généré par SynDEx.

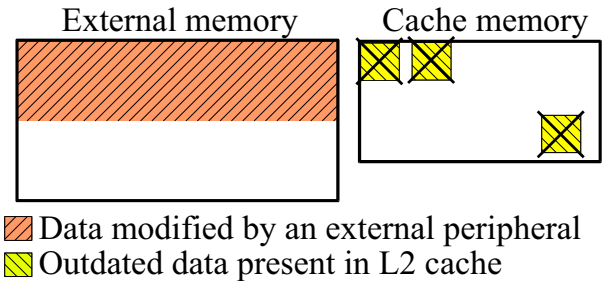
#### *4.2.4.2 Principes*

Avec le mécanisme de cache, il coexiste deux copies d'une donnée (en mémoire externe et en cache). Lorsque l'une ou l'autre est modifiée, les données ne sont plus cohérentes. Un protocole doit être mis en place de façon à s'assurer que rien n'accède aux données périmées [TM93]. Le contrôleur de cache ne sait pas gérer la cohérence dans un contexte multiprocesseur [Tex6a]. Nous avons montré dans ces travaux comment utiliser la génération automatique de code pour palier ce manque.

La configuration et la gestion de la mémoire sont généralement faites en utilisant des libraires fournies par le fabriquant de DSP. La technique usuelle de gestion du cache est l'utilisation des fonctions ***writeback()*** et ***invalidate()***. La fonction ***writeback()*** permet de recopier le contenu de la mémoire cache dans la mémoire externe cachable alors que la fonction ***invalidate()*** remet à jour les données en mémoire cache (une invalidation va provoquer une remise à jour du contenu de la mémoire cache).




(a) données modifiées par le CPU




(b) données modifiées par un périphérique

FIGURE 4.11: Gestion de la cohérence de cache

Nous avons pu constater que seules les données allouées dans le code généré pouvaient être dans des situations d'incohérence de cache. Les autre données du programme généré sont contenues dans la pile et ne sont donc pas concernées par ce problème. Les situations d'incohérence de cache apparaissent dans deux situations identifiées :

1. lorsque des données en mémoire cache sont modifiées par le CPU et sont ensuite lues par un périphérique : la donnée de sortie de la fonction est allouée en mémoire externe cachable

(Fig. 4.11-a). Le cache est utilisé pour stocker temporairement les résultats et les données source. A la fin du calcul, tous les résultats n'ont pas encore été écrits dans la mémoire externe. Par conséquent, la donnée doit être mise à jour avant qu'un périphérique y accède. Il faut alors exécuter la fonction ***writeback()***.

2. lorsque des données cachable sont modifiées par un périphérique : un périphérique a mis à jour la mémoire externe (Fig. 4.11-b). Les adresses correspondantes doivent être invalidées pour empêcher le CPU de les utiliser. Il faut exécuter la fonction ***invalidate()***.

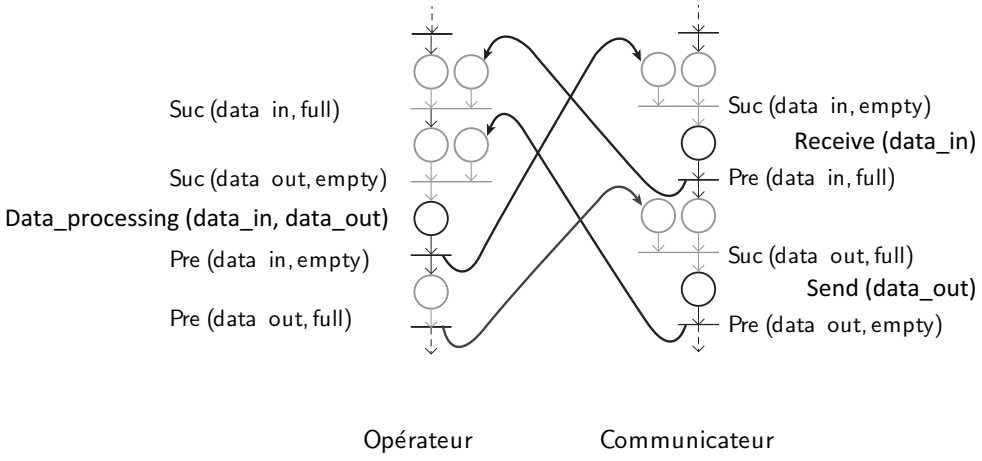



FIGURE 4.12: Synchronisation de l'exécutif

La figure 4.12 représente les deux séquenceurs d'un processeur. Les opérations ***Suc()*** et ***Pre()*** correspondent respectivement à une attente et une libération de sémaphore. Les emplacements mémoire concernés par un potentiel problème de cohérence de cache sont en mémoire externe. Si le CPU et le DMA accèdent à une même adresse, alors la cohérence de cache doit être assurée car le DMA contourne le contrôleur de cache. Dans nos bibliothèques de code, tous les transferts inter-processeurs sur DSP sont optimisés et réalisés par le DMA. La cohérence de cache doit donc avoir lieu lors du déblocage du séquenceur de communication par le séquenceur de calculs, lorsque ce dernier délivre une opération ***Pre()*** (Fig. 4.12). De façon générale, une opération Pre(mem, empty) est associée à un ***invalidate()*** et un Pre(mem, full) à un ***writeback()***. Cette association peut être réalisée dans les noyaux d'exécutifs pour que le code généré automatiquement évite les situation d'incohérences afin d'utiliser le mécanisme de cache sur chacun des processeurs de la plateforme.

#### *4.2.4.3 Conclusion*

Ces travaux ont permis d'utiliser les mécanismes de cache sur les DSP TMS320C64x. Nous avons mesuré une diminution des performances de l'ordre de 10% par rapport à des programmes où l'ensemble des données étaient en mémoire interne. En revanche, ce mécanisme de cache est devenu indispensable pour augmenter la résolution des images traitées dans nos applications jusqu'aux format HD (Haute Définition). Le caractère automatique réduit considérablement les risques d'erreurs, surtout lorsque l'application grandit avec le nombre de processeurs et le nombre de données transférées.

Ce travail a été réalisé au niveau des noyaux de génération de code. Pour davantage de généricité, il aurait été intéressant de prendre en compte la spécificité des mémoires dans l'adéquation, à savoir : pouvoir décrire plus précisément les processeurs (modèle de mémoire interne, scratchpad, mémoire externe cachable ou non) et pouvoir ajouter des contraintes sur des données. Un ordonnancement des opérations en tenant compte de l'optimisation des accès à la mémoire externe avec l'utilisation du cache serait en outre profitable.

### 4.2.5 Utilisation d'un OS multitâche

L'utilisation d'un OS multitâche dans la méthodologie AAA a été réalisée dans le cadre de la thèse de Ghislain Roquier [Roq08].

#### *4.2.5.1 Motivations*

Les recherches menées autour de la problématique de l'utilisation d'un OS (Operating System) sur les processeurs avaient pour objectif de simplifier les mécanismes de synchronisation entre séquenceurs dans le code généré [RRND06]. Il s'agit là encore d'un post-traitement qui se situe au niveau de la transformation du macro-code généré par SynDEx.

La méthodologie AAA rend l'utilisation de systèmes d'exploitation temps-réel facultative. En effet, l'ordonnancement des opérations allouées sur les différents processeurs est déterminé statiquement à la compilation. Il garantit le bon fonctionnement de l'application. Cependant, la transformation du code générique vers un code compilable des séquenceurs de calcul et de communication générés par AAA/SynDEx est une opération peu aisée. La synchronisation de ces deux séquenceurs conduit à des appels imbriqués de fonctions et des sauvegardes de contexte sous la forme de variables partagées. La protection des données en cas d'interruptions matérielles ou de l'utilisation du debogueur par exemple n'est pas assuré. De plus, le code généré automatiquement sans l'utilisation de RTOS est complexe à lire et donc à comprendre.

#### *4.2.5.2 Principe*

L'utilisation d'un RTOS a ainsi pour but de faciliter la mise en œuvre des séquenceurs sous la forme d'un programme multitâche, et parallèlement de réduire la taille du code généré par les méthodes antérieures. Grâce au RTOS, il est possible de créer une tâche (*thread*) distincte par séquenceur de calcul ou de communication : le RTOS gérera son exécution dynamiquement lors de l'exécution du programme. Le modèle AAA n'est pas préemptif, il n'y a donc pas de priorité d'exécution d'un séquenceur particulier. Nous avons par conséquent donné aux tâches des priorités égales pour que l'ordonnancement des tâches ne soit pas préemptif. Une tâche peut être dans quatre états distincts : bloqué, prêt, actif, terminé. En supposant l'utilisation d'un unique processeur avec une unique unité de calcul, seule une tâche peut être active à chaque instant durant l'exécution. Il faut donc définir une politique d'activation des tâches. Nous avons fait le choix d'un ordonnancement de type FIFO des tâches, ce qui implique que lorsqu'une tâche est exécutée, elle ne peut être bloquée que lorsqu'elle se termine ou lors d'un appel système (une attente sur un sémaphore par exemple). Inversement, une tâche bloquée passe à l'état prêt lorsqu'un sémaphore qui la bloquait est libéré. Elle peut alors être exécutée (état actif) selon la disponibilité de l'unité de calcul.

Cette approche a nécessité la remise à jour de l'ensemble des noyaux SynDEx pour réaliser la traduction des macro-instructions. Dorénavant, celles-ci permettent la configuration d'un RTOS résident capable d'exécuter et de gérer l'ordre d'exécution des différents séquenceurs. Un grand nombre de RTOS existent pour différents types de processeurs. Leurs primitives sont souvent spécifiques à une famille particulière de processeurs. Nous avons utilisé le RTOS DSP-BIOS pour les bibliothèques liées aux DSP TMS320C6x. Nous avons aussi développé pour les processeurs généralistes des noyaux à l'aide des interfaces de programmation WinAPI et POSIX afin d'interagir avec les systèmes d'exploitation de types Windows et UNIX. Il faut remarquer que l'OS de Windows ne gère pas le temps-réel, tandis que l'API POSIX fournit quant à elle des services temps-réel. La traduction permet alors la création des tâches, la création des sémaphores et la manipulation des sémaphores. Les nouvelles bibliothèques ont été validées sur l'ensemble de nos applications.

Afin de valider nos travaux, nous avons choisi de réaliser une implantation du codec LAR selon l'approche proposée sur une architecture composée de trois DSP et d'un PC hôte. Le codeur et le décodeur sont respectivement implantés sur le premier et le deuxième DSP de la plateforme. On utilise les contraintes de placement à cet effet.

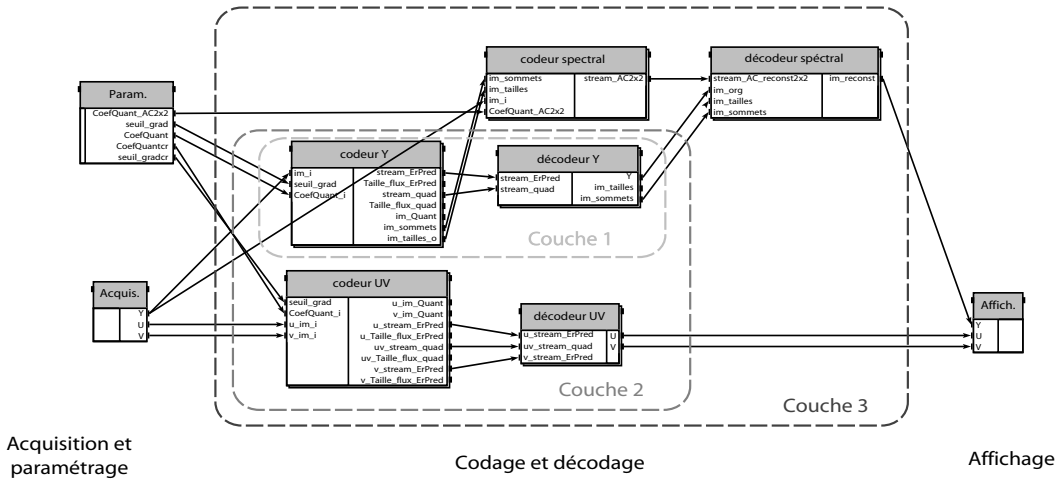



FIGURE 4.13: Schéma du codeur/décodeur LAR testé

La description du codec couleur LAR illustre la scalabilité de celui-ci en différentes couches fonctionnelles illustrées sur la figure 4.13 (la description du codec est hiérarchique) :

- **Couche 1 :** codec vidéo spatial pour la luminance (composante Y), caractérisé par des blocs $2 \times 2$, $4 \times 4$ et $8 \times 8$).
- **Couche 2 :** codec vidéo spatial pour la chrominance (composantes U et V) ajouté à la couche 1.
- **Couche 3 :** codec vidéo spectral pour les blocs $2 \times 2$, caractérisé par l'ajout de l'erreur résiduelle, ajouté à la couche 2.

Cette implantation a permis de valider les noyaux de génération de code sur une application complexe. De plus, comme nous pouvons le constater sur les tableaux 4.3 et 4.4, le surcoût lié à l'ordonnanceur est faible. Le temps d'exécution du codec est légèrement plus lent avec l'utilisation du RTOS, comme nous pouvons le remarquer dans le tableau 4.3. Le temps d'exécution du codeur est la partie de l'algorithme la plus lente (typiquement 4 fois moins rapide que le décodeur),

c'est pourquoi le comportement temps-réel du codec LAR est donné par le temps d'exécution du codeur. L'utilisation du RTOS a aussi un impact sur la mémoire des processeurs : il est d'environ 55 kilo-octets dans ce programme. Cependant, plus la mémoire est utilisée, plus l'impact devient proportionnellement petit puisque le surcoût du RTOS est constant. Par exemple pour la troisième couche du décodeur (celui dont l'impact en mémoire est le plus important), comme on peut le constater sur le tableau 4.4, le RTOS augmente la mémoire utilisée par le codeur et par le décodeur de respectivement 7% et 5% de la mémoire interne du DSP.

| Couche | Sans RTOS | Avec RTOS |
|---|---|---|
| 1 | 18.03 ms | 18.05 ms |
| 2 | 25.35 ms | 25.45 ms |
| 3 | 31.84 ms | 32.07 ms |

TABLE 4.3: Temps d'exécution du codage/décodage LAR

| Couche | Codeur | | Décodeur | |
|---|---|---|---|---|
| | sans RTOS (en ko) | avec RTOS (en ko) | sans RTOS (en ko) | avec RTOS (en ko) |
| 1 | 874 | 928 | 899 | 953 |
| 2 | 747 | 802 | 658 | 713 |
| 3 | 642 | 697 | 528 | 583 |

TABLE 4.4: Mémoire utilisée (programme et données) par le codage/décodage LAR

#### *4.2.5.3 Conclusion*

Un des points bénéfiques de l'utilisation de RTOS dans le cadre de AAA/SynDEx consiste en la réduction de la taille du code généré. La génération de code qui permet l'interaction avec le noyau DSP/BIOS est beaucoup plus proche des automates synchronisés générés par SynDEx, ce qui rend le code généré plus lisible et les noyaux SynDEx plus simples à créer. De plus, comme nous avons pu le constater sur l'exemple du LAR, l'impact du RTOS est plutôt faible, aussi bien d'un point de vue de la mémoire que du temps d'exécution de l'application. La maintenance du code est facilitée car les primitives offertes par le RTOS ont été longuement testées par le fournisseur et fonctionnent de manière identique quel que soit le processeur cible sur lequel il est implanté. Devant ces nombreux avantages, cette nouvelle méthode de génération de code a donc progressivement remplacé la méthode employée auparavant.

### 4.2.6 Conclusion sur la génération de code

Les noyaux de génération automatique de code concernant les plateformes Sundance, Pentek et Vitec sont maintenant tous disponibles. Ceci permet d'utiliser toutes les ressources matérielles disponibles (plusieurs PC, DSP, FPGA) en ne touchant qu'au code C ou VHDL des fonctions de l'application. Autrement dit, les synchronisations et les transferts de données nécessaires sont désormais gérés automatiquement. De plus, les différentes fonctions sont elles aussi ordonnancées automatiquement par SynDEx de manière optimisée sur les différents composants de l'architecture. Nos travaux ont permis d'optimiser l'utilisation de ces plateformes (optimisation mémoire, gestion de la cache, utilisation de RTOS résident). L'utilisateur peut donc désormais se concentrer sur l'application (débogage, optimisation).

## 4.3 Méthode de développement

L'automatisation du portage d'une application distribuée sur une plateforme multiprocesseur a permis de définir une méthode de développement allant de la vérification fonctionnelle jusqu'à l'application distribuée finale. Au fur et à mesure de la maturité des outils et notamment des bibliothèques de génération de code, cette méthode a pu évoluer. Dans les premières phases, des outils comme AVS ou Ptolemy étaient utilisés en amont de AAA/SynDEx pour faire de la vérification fonctionnelle [Fre01] [Nez02]. Il était alors nécessaire de réaliser des traductions pour ensuite utiliser SynDEx. Par la suite, la chaîne de prototypage a été simplifiée en intégrant dans SynDEx des outils de vérification fonctionnelle, évitant ainsi des traductions, sources d'erreurs et en réduisant le nombre d'outils à maîtriser et à acheter [Rau06]. Pour ce faire, la technique est relativement simple : ajouter des fonctions destinées à vérifier le bon fonctionnement de l'algorithme (lecture/écriture de fichier, comparaison, affichage,... ). Dans le graphe d'algorithme de SynDEx, cela se traduit par l'ajout de nouvelles opérations qui sont connectées aux données à vérifier. Ensuite, peu importe l'architecture utilisée et la distribution des opérations, les données sont automatiquement transférés vers un processeur (de l'architecture cible) capable d'exécuter l'opération de vérification (par exemple un PC pour l'affichage et accès disque). Afin de tenir compte des spécificités du traitement des images d'un point de vue entrée/sortie, des fonctions d'acquisition d'images (par *webcam* ou par lecture de fichier), et d'affichage (une fenêtre associée à chaque appel de fonction de visualisation) sont disponibles, sous environnement standard de développement ($Visual\ C++$, dev-cpp), sur PC [Rau06]. L'utilisateur est ainsi en mesure d'appeler directement des opérations spécifiques de visualisation dans son graphe d'algorithme. Ces visualisations peuvent être utilisées en association avec les outils de débogage classiques sur PC et sur DSP pour valider fonctionnellement les développements.

La maturité des bibliothèques de communications nous a permis d'utiliser la vérification fonctionnelle dans AAA/SynDEx de manière fiable. L'ensemble du laboratoire a pu se placer dans une optique d'utilisateur de la méthodologie et utiliser l'outil de manière intensive. La méthode pour chaque étape de développement a donc pu évoluer avec une sécurité de conception accrue. Cette approche permet notamment de proposer des stages ou des thèses dans un cadre de développement unifié. Il est alors plus facile pour les développeurs de valoriser leurs travaux portant sur une partie du processus en utilisant l'ensemble des travaux déjà réalisés. Par exemple, une personne travaillant sur une application vidéo pourra facilement réutiliser les blocs déjà développés et générer très rapidement des exécutifs sur toutes les plateformes matérielles disponibles au laboratoire.

Dans la suite de ce paragraphe nous présentons les trois étapes majeures de la méthodologie de développement (Fig. 4.14) : la vérification fonctionnelle, le portage et l'optimisation sur cible monoprocesseur, et le portage sur cible multiprocesseur.

### 4.3.1 Vérification fonctionnelle

Elle permet de valider les algorithmes et la distribution des opérations en quatre sous-étapes.

**Modélisation de l'application.** L'application est décrite sous forme d'un graphe flux de données, avec une approche descendante. L'algorithme est donc initialement constitué de macro-opérations. Cela permet d'avoir rapidement un graphe flux de données opérationnel pour débuter l'étape suivante du processus. Ensuite les opérations sont raffinées. Comme nous avons pu le voir précédemment, il convient de choisir le bon grain de description afin d'obtenir un niveau de

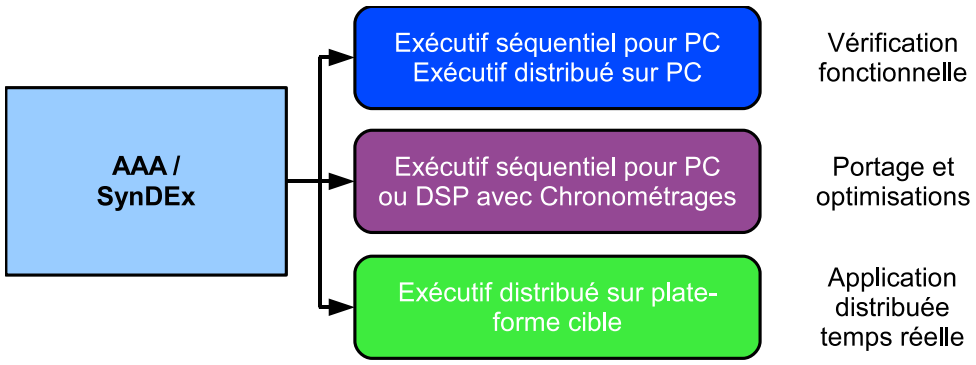



FIGURE 4.14: Etapes de développement

parallélisme acceptable, tout en maîtrisant les besoins en termes d'allocation mémoire.

**Vérification fonctionnelle monoprocesseur.** La génération automatique de code fournit un programme en C pour une première implantation sur PC (monoprocesseur). Ainsi, l'utilisateur peut créer les fonctions en langage C associées à chacune des opérations du graphe flux de données et tester les fonctionnalités de son application avec un environnement de compilation standard. Nous utilisons des primitives de visualisation, de façon à accélérer la mise au point des algorithmes de traitement d'images. Lorsque le fonctionnement de l'application est validé, l'algorithme monoprocesseur va servir de référence pour les étapes suivantes.

**Exploration architecturale.** SynDEx permet, avant même l'acquisition d'une plateforme de prototypage, de faire de l'exploration architecturale, c'est-à-dire de dimensionner au mieux le nombre d'opérateurs (processeurs) nécessaires pour le portage de l'application. Il est alors nécessaire d'estimer les performances de l'architecture en termes de temps d'exécution de chaque opération sur chaque type de processeur, et en termes de vitesse de transfert sur les média de communication. Le parallélisme potentiel de l'application et les performances attendues en fonction d'une architecture cible peuvent alors être évalués. La figure 4.15 décrit le portage d'une application d'estimation de mouvement sur une plateforme composée de cinq processeurs connectés par une mémoire distribuée. Les fonctions d'entrées-sorties sont exécutées sur le premier processeur, et l'application est distribuée sur les quatre autres processeurs. La description de l'application est hiérarchique, seule la vue du dessus est visible, pour ne pas surcharger la figure. Il est alors possible d'ajouter ou de retirer des processeurs, d'estimer les performances et le nombre de processeurs requis, afin de dimensionner au mieux les ressources nécessaires.

**Vérification fonctionnelle multiprocesseur.** Grâce aux bibliothèques SynDEx du processeur PC et des communication TCP, PIPE, et RAM, la distribution de l'application peut être validée sur une plateforme virtuelle composée de PC multitâche (un seul exécutable, un seul espace d'adressage), de PC multiprocessus (plusieurs exécutables, plusieurs espaces d'adressages) ou de plusieurs PC. Cette étape est nécessaire pour identifier le plus tôt possible des erreurs dues à la parallélisation de l'application (notamment concernant l'accès aux données), et les corriger sur une plateforme maîtrisée par les développeurs (un PC). Il est alors possible de contrôler le résultat de chaque opération pour vérifier le bon fonctionnement de l'application distribuée. Pour ce faire, le graphe d'algorithme est simplement dupliqué, une instance est exécutée sur monopro-

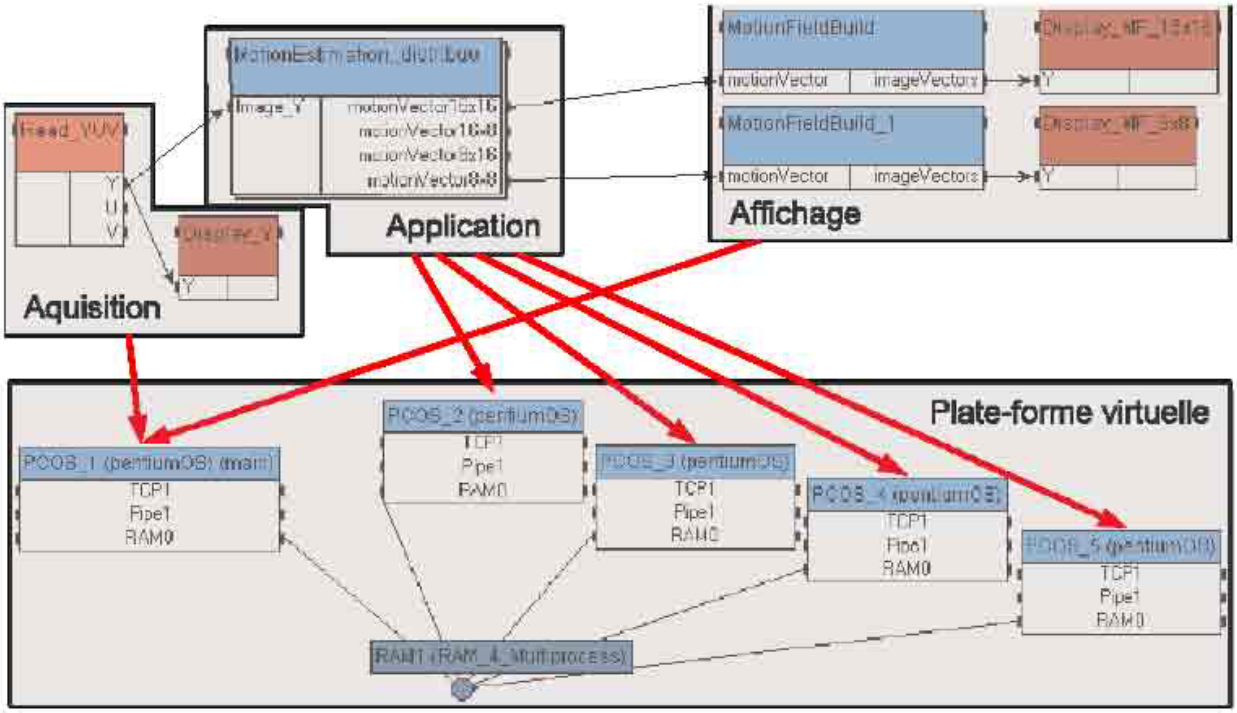



FIGURE 4.15: Exploration architecturale sur plateforme de type Vitec

cesseur séquentiellement (instance référence), et une autre sur la plateforme virtuelle distribuée, en insérant des contraintes de placement. Des opérations, ayant le rôle de sondes, sont connectées entre les deux instances sur certaines données pour vérifier leur cohérence (par exemple : comparaison bit à bit entre les deux jeux de données, les résultats étant fournis sous forme de traces). L'outil s'assure de l'acheminement des données à travers la plateforme. La figure 4.16 présente la vérification fonctionnelle de l'application distribuée de la figure 4.15. L'opération d'estimation de mouvement est dupliquée et l'instance de référence est exécutée séquentiellement sur un processeur. Des fonctions de test (comparaison des vecteurs entre les deux instances) permettent de vérifier la cohérence des données en affichant par exemple les caractéristiques des vecteurs qui différent afin d'identifier la source d'erreur. Cette technique de validation est ensuite utilisée tout au long du processus de développement.

### 4.3.2 Portage et optimisation monoprocesseur

Le graphe flux de données est ensuite directement utilisé pour porter progressivement les opérations sur processeur cible. L'acheminement des données est toujours géré par l'outil, ce qui permet à l'utilisateur de se concentrer sur le portage et l'optimisation monoprocesseur (optimisation de l'écriture du programme, utilisation du langage assembleur ou de bibliothèques optimisées livrées avec le composant), opération par opération. Pour vérifier que le changement de processeur et les optimisations réalisées n'introduisent pas d'erreur, la technique précédemment utilisée permet de valider les travaux.

Pour mesurer les performances en termes de rapidité, le code généré insère automatiquement les fonctions de chronométrage. L'utilisateur doit simplement compiler les sources et lancer l'exécution du programme avec l'outil adéquat à la cible. Les durées de chacune des fonctions (opérations

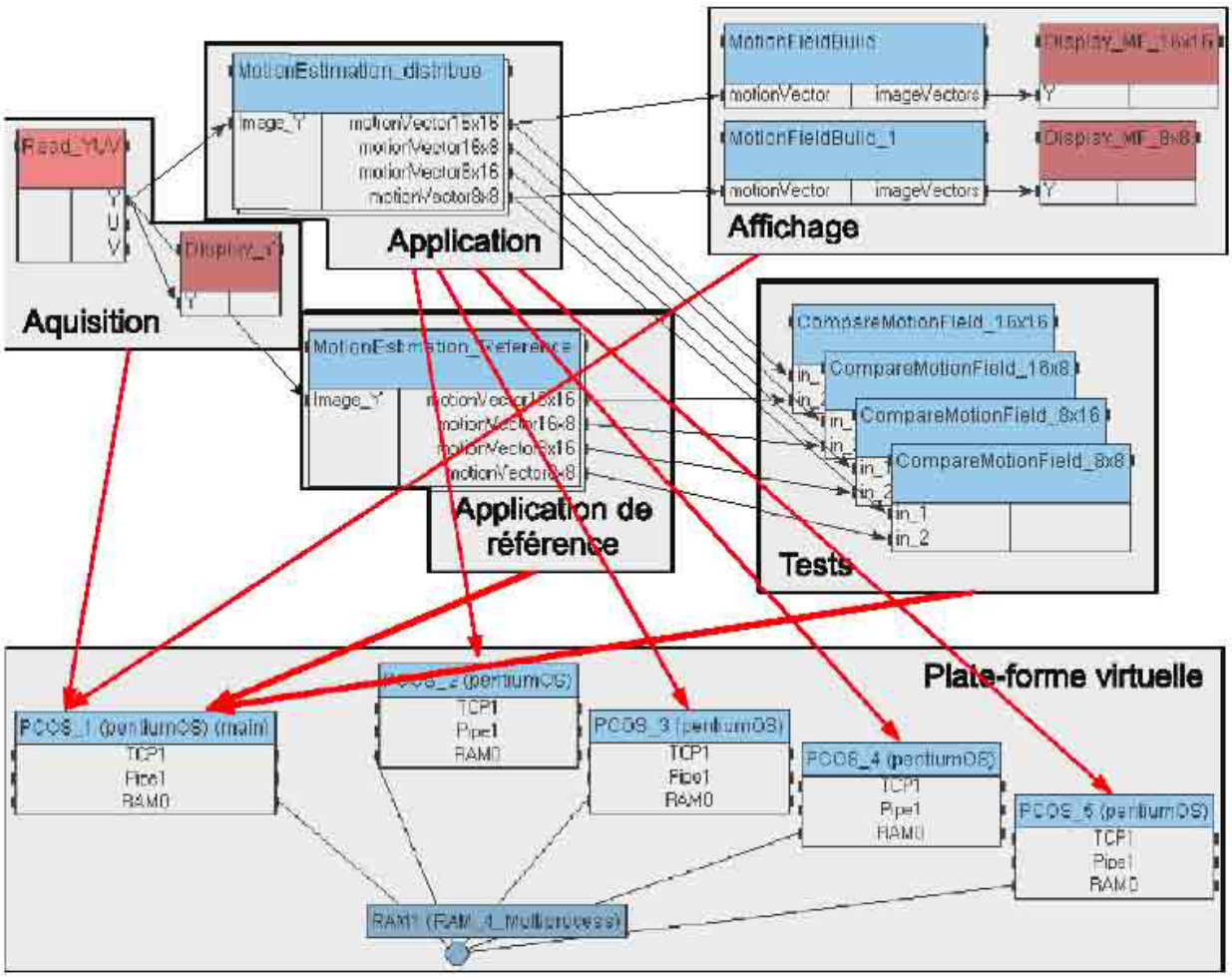



FIGURE 4.16: Vérification fonctionnelle multiprocesseur

du GFD) sont alors affichées en fin d'exécution. Cette étape doit être réalisée pour chacun des processeurs (PC ou DSP) de la cible finale.

**Remarque :** Dans certains cas, cette étape peut être conduite avec l'exploration architecturale, lorsqu'il est nécessaire de réaliser des chronométrages afin d'évaluer précisément les performances. Un premier portage avec les optimisations qui s'imposent sont donc nécessaires.

### 4.3.3 Application distribuée temps réel

SynDEx réalise l'adéquation sur l'architecture parallèle et génère un exécutif distribué temps réel. L'exécutif est optimisé en fonction de la plateforme multiprocesseur. L'utilisateur peut contraindre l'adéquation en forçant par exemple les entrées/sorties sur un processeur. Plusieurs configurations architecturales peuvent être simulées en faisant varier le nombre et le type des processeurs ou connexions. Il est également possible de retravailler la description de l'application pour faire apparaître plus de parallélisme.

L'avantage principal de cette chaîne de prototypage réside dans sa simplicité. La plupart des tâches réalisées par l'utilisateur concernent la spécification de l'application. L'utilisation de SynDEx et des compilateurs est limitée à des opérations simples. Une grande partie des tâches com-

plexes (adéquation, synchronisations, transferts de données et chronométrages) sont effectuées automatiquement.

## 4.4 Conclusions du chapitre

Dans cette partie, nous avons tenté de résumer l'ensemble des travaux réalisés avec l'outil SynDEx dans le cadre de la méthodologie AAA.

Nous avons aussi présenté dans ce chapitre les trois axes autours desquels se sont articulés ces travaux : la description d'applications complexes, l'optimisation du processus de génération de code ainsi que la définition d'un processus de développement cohérent.

L'utilisation de cette méthodologie réduit le cycle de développement et apporte un gain de temps important dans le développement d'applications multiprocesseurs. L'ordre d'exécution garanti (pas d'interblocages), l'automatisme du processus et son indépendance vis-à-vis de l'architecture cible apportent une sécurité dans la conception de ces applications. L'heuristique pour l'adéquation mise en œuvre dans SynDEx procure un processus automatique et optimisé en vue du placement et de l'ordonnancement de l'algorithme sur l'architecture, mais également en termes de minimisation spatiale (mécanismes non résidents) et temporelle (optimisation de la latence).

Les objectifs que nous nous fixons ont souvent pu être résolus à ***l'extérieur*** du logiciel en agissant sur la description de l'application ou sur la phase de génération de code. Il n'en reste pas moins que certains objectifs auraient pu être traités à l'intérieur même de la phase de placement/ordonnancement. Il faut noter que certains procédés que nous avons mis en place dans la génération de code ont été particulièrement complexes à réaliser, notamment à cause du langage M4 et du macro-processeur associé.

D'autre part, le modèle utilisé dans SynDEx pour la description de l'architecture pourrait évoluer de sorte à mieux prendre en compte les spécificités du matériel dans le placement/ordonnancement. Cela a d'ailleurs été spécifié dans la méthodologie AAA [GS03a], sans être intégré dans l'outil SynDEx : on peut donc considérer que l'outil SynDEx est une interprétation partielle de la méthodologie AAA. L'idée serait alors de prendre en compte certains mécanismes comme l'initialisation des DMA par le CPU d'un processeur ou encore la contention sur les bus de communication, deux problèmes qui n'étaient pas critiques entre plusieurs composants mais qui le deviennent dans un contexte multicœur ou MPSoC.

Certaines fonctionnalités au niveau de l'interface graphique comme le choix d'un niveau hiérarchique intermédiaire (à la place d'une mise à plat complète du graphe) pour l'adéquation apporteraient également un gain important puisqu'une même description algorithmique pourrait être utilisée afin de choisir la meilleure granularité pour le placement/ordonnancement. Le choix pourrait alors se faire manuellement ou automatiquement, statiquement dans la phase de placement/ordonnancement ou dynamiquement lors de l'exécution du programme.

La mise à plat complète du graphe d'algorithme pose d'ailleurs un problème lorsque qu'un graphe est répété un grand nombre de fois comme nous avons pu le constater à plusieurs reprises. Par exemple, dans nos descriptions MPEG-4 Part2, le fait d'augmenter la résolution de l'image traitée va augmenter dans les mêmes proportions le nombre de nœuds du graphe, ce qui peut nous amener à plusieurs dizaines de milliers de nœuds à traiter dans le placement/ordonnancement. Si les algorithmes sont sensés supporter de telles complexités, on peut se demander si une autre approche n'est pas envisageable puisqu'il s'agit tout de même d'un nombre réduit d'opérations dans le graphe avant mise à plat (une cinquantaine). De plus, le code généré à cause de cette

mise à plat est tellement volumineux qu'il n'est parfois plus compilable (comme cela a été le cas à plusieurs reprises avec CCS).

Suite à l'adéquation, dans la phase de génération du code M4 par SynDEx, la gestion des conditionnements implique que chacun des appels de fonction soit encapsulé dans une structure conditionnelle. Il est alors impossible pour un compilateur de réaliser ses optimisations, ce qui est particulièrement pénalisant dans le cas de processeurs VLIW.

Enfin, sur un plan plus théorique, les modèles utilisés dans l'outil SynDEx pour la description des applications sont des modèles flux de données, mais, contrairement à ceux utilisés dans Ptolemy II, il n'est pas possible de faire des analyses préalables d'ordonnançabilité.

Toutes ces perspectives pour SynDEx que nous venons d'énumérer se sont révélées impossibles à mettre en œuvre sans travailler sur le code source même de l'outil. Etant donné que SynDEx n'est pas *open source*, il nous a été impossible de le faire progresser plus encore. Cet état de fait a également été un frein pour les collaborations de l'équipe. Devant les problèmes techniques et juridiques rencontrés, mais toujours motivés par l'approche AAA, nous avons été amenés à repositionner nos objectifs de recherche comme nous allons le voir dans le chapitre suivant.

Il faut noter qu'une nouvelle version de SynDEx (V7) vient d'être proposée (juillet 2009). Cette nouvelle version doit permettre de spécifier des actions périodiques. La notion de période choisie est **stricte** [Ker09], c'est-à-dire que chaque opération du graphe doit débuter son exécution dès le début d'une nouvelle période. Ce type de périodicité n'est pas très utilisé dans les systèmes de traitement des images ou en télécommunication, puisque les données peuvent toujours être mémorisées en début de période en attendant leur traitement, du moment que celui-ci soit effectué avant la prochaine échéance. Nos travaux se situent donc plutôt dans des problèmes pour lesquels les tâches sont dites à **échéance sur requête**. L'ordonnancement de nos problèmes peut être plus souple que des périodes strictes. Le risque est alors de décrire des applications ordonnançables (au sens échéance sur requête) et que le logiciel SynDEx V7 le définisse comme non ordonnançable (au sens période stricte). Dans SynDEx V7, nos travaux sur l'optimisation mémoire n'ont pas été implémentés. Les applications et des bibliothèques de génération de code présentées dans ce chapitre ayant été développées sous SynDEx V6, leur réutilisation avec la nouvelle version de SynDEx devra être étudiée.

# Chapitre 5

# Projet PREESM

Suite à notre expérience sur l'outil SynDEx, nous avons donc été amenés à redéfinir notre manière de concevoir le développement d'outils pour le prototypage rapide. Pour notre équipe, il s'agit essentiellement de capitaliser nos travaux dans un cadre ouvert et évolutif. Ce cadre de développement a été appelé Preesm : ***Parallel and Real-time Embedded Executives Scheduling Method***. Preesm respecte les principes de la méthodologie AAA [GS03a]. On peut donc considérer SynDEx et Preesm comme deux implantations distinctes de la méthodologie AAA.

Trois thèses sont en cours dans le cadre du projet Preesm. La première thèse est réalisée par Jonathan Piat et vise à optimiser la gestion des répétitions dans les phases de transformation de graphe et de génération de code [Pia10], tandis que Maxime Pelcat dans la seconde thèse cherche à optimiser le processus de placement/ordonnancement ainsi que la génération de code pour plateformes à base de DSP multi-cœurs [Pel10]. Cette dernière thèse fait l'objet d'un contrat de collaboration avec Texas Instruments, ce qui nous fait bénéficier des derniers composants disponibles dans le domaine. De plus, nous devons étudier dans le cadre de cette thèse les futurs standards de communication sans fil LTE (***Long Term Evolution***). La thèse de Matthieu Wipliez [Wip10], même si elle est plus orientée vers la standardisation RVC dont il est question dans le chapitre 6, contribue également à la mise en œuvre de Preesm sur la définition de l'architecture logicielle et sur la gestion de l'interface graphique avec Graphiti.

Etant donné que nous étions tributaires des choix de modélisation (en termes de description des algorithmes, des architectures et de description des implantations) dans l'outil SynDEx, la mise en place du projet Preesm a été pour nous l'occasion d'étudier d'autres modèles de spécification autour de la méthodologie AAA, et ainsi d'essayer de choisir des modèles capables de lever les limitations trouvées dans l'utilisation du logiciel SynDEx. Ghislain Roquier a notamment étudié dans sa thèse les modèles flux de données pour la description des algorithmes dans AAA [Roq08]. Des optimisations sur les techniques d'ordonnancement statique, afin de prendre en compte les caractéristiques des MPSoC, ont déjà été proposées dans le cadre de la thèse de Pengcheng Mu [Mu09]. D'autre part, nous verrons dans ce chapitre comment l'architecture logicielle a été choisie pour faciliter les développements au sein du laboratoire, mais également pour des collaborations entre universitaires et/ou industriels.

## 5.1 Etude des modèles de spécification

### 5.1.1 Modélisation de l'application

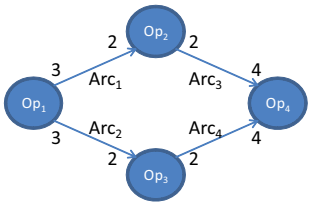
Cette étude a été menée principalement dans le cadre de la thèse de Ghislain Roquier [Roq08], mais toujours en concertation avec l'ensemble de l'équipe. Ces travaux ont permis de dresser un état de l'art sur la sémantique des modèles flux de données, et particulièrement ceux simulés avec l'outil Ptolemy II. Ces modèles sont nombreux et diffèrent les uns des autres par leur expressivité et leurs domaines d'application. Plus un modèle est expressif, plus il sera pratique à utiliser dans une phase de spécification de l'algorithme, mais plus les phases d'adéquation et de génération automatique de code seront complexes. Comme notre domaine applicatif comprend majoritairement des applications déterministes, un modèle statique est bien adapté. Ce type de modèle utilise toujours le même nombre de données en entrée et produit le même nombre de données sur ses ports de sortie. Ces modèles étant prédictibles, ils sont bien adaptés aux traitements automatiques durant une phase de compilation et à une exécution statique.

Cette étude nous a permis en outre de nous familiariser avec le modèle flux de données synchrones (***Synchronous DataFlow*** ou SDF). Ce modèle a été introduit au milieu des années 80 par Lee et Messerschmitt [LM87a, LM87b]. Il est proche des ***computation graphs*** de Karp et Miller [KM66] ou encore des graphes marqués pondérés dans la théorie des réseaux de Petri [Mur89]. Le modèle SDF est une restriction du modèle de processus à flux de données. L'intérêt majeur de ce modèle est de rendre les propriétés de vivacité du système et d'utilisation bornée de la mémoire décidables avant compilation. La vivacité (***liveness***) du système signifie que son exécution ne s'arrêtera pas prématurément. C'est une propriété critique pour les systèmes réactifs. L'utilisation bornée de la mémoire (***boundedness***) signifie que l'exécution infinie d'un tel système ne provoquera pas de fuite de mémoire. Un ordonnancement statique avant compilation des acteurs (nœuds du graphe flux de données) peut alors être déterminé sous la forme d'une séquence cyclique bornée d'invocations des acteurs. De manière plus formelle, un acteur SDF est un acteur flux de données qui ne contient qu'une seule règle d'exécution [LP95] et qui est valable pour toutes les valeurs possibles des jetons. Le nombre de jetons consommés et produits est donc constant à chaque exécution de l'acteur.

Un atout particulièrement intéressant des graphes SDF repose dans la possibilité de prouver que l'application décrite par ce graphe peut être ordonnancée de manière statique à l'aide de sa matrice de topologie (Fig. 5.1). Pour construire une telle matrice, il faut reporter les valeurs associées à chacun des arcs du graphe en utilisant la convention suivante : des valeurs positives pour une écriture sur cet arc (port de sortie d'un nœud) et une valeur négative pour une lecture sur cet arc (port d'entrée d'un nœud du graphe). Le graphe peut être ordonnancé si le rang de cette matrice est égal au nombre de nœuds du graphe moins un [LM87b]. On parle de graphe ***consistant***. Ce modèle est alors prédictible dès la spécification de l'algorithme : c'est la raison pour laquelle nos travaux se sont tournés vers ce modèle.

#### Comparaison des modèles SDF et SynDEx

Le modèle d'algorithme utilisé dans SynDEx est un graphe acyclique orienté (***Directed Acyclic Graph*** ou DAG), mais qui a été enrichi au fur et à mesure par des notions de hiérarchie, de conditionnement et de répétition, en ayant toujours à l'esprit des contraintes de placement/ordonnancement dans un contexte multiprocesseur. Les formalismes des graphes SynDEx et SDF sont donc intrinsèquement différents. Leur comparaison n'est donc pas un exercice aisé. Le modèle de graphe de SynDEx est un DAG augmenté de nœuds de factorisation pour les répétitions (*fork*,




$$\begin{bmatrix} 3 & -2 & 0 & 0 \\ 3 & 0 & -2 & 0 \\ 0 & 2 & 0 & -4 \\ 0 & 0 & 2 & -4 \end{bmatrix}$$

Matrice de topologie

| | $Op_1$ | $Op_2$ | $Op_3$ | $Op_4$ |
|---|---|---|---|---|
| $Arc_1$ | 3 | -2 | 0 | 0 |
| $Arc_2$ | 3 | 0 | -2 | 0 |
| $Arc_3$ | 0 | 2 | 0 | -4 |
| $Arc_4$ | 0 | 0 | 2 | -4 |

FIGURE 5.1: Un graphe SDF et sa matrice de topologie

*join*, *iterate*, *diffuse*) et de nœuds et d'arcs spécifiques au conditionnement [LS97, Gra00]. Le formalisme de graphe de SynDEx n'associe pas de fonction de pondération sur les arcs : les arcs sont donc considérés comme homogènes (l'intégralité des données présentes sur les arcs entrants sont consommés lors de l'invocation des nœuds), hormis lorsqu'une situation de factorisation apparaît. Seuls les nœuds de factorisation permettent la présence de répétitions. Ces répétitions peuvent être implicites, c'est à dire déduites des tailles des ports entrant et sortant connectés à un arc, ou spécifiées par l'utilisateur de manière explicite avec l'outil SynDEx.

L'interprétation des arcs est elle aussi très différente. AAA/SynDEx considère les dépendances de données comme des « *agrégats de cellules mémoire contiguës* » [Gra00] (correspondant par exemple à des tableaux ou à des structures C) dont la taille est fixée *a priori* et qui est égale à la taille du port de sortie d'une opération. Dans le modèle général des flux de données, les communications s'effectuent au travers de canaux unidirectionnels avec une unique extrémité initiale et une unique extrémité finale (il s'agit d'hypergraphes dans AAA/SynDEx). La profondeur des canaux n'est pas spécifiée *a priori* et peut être finie ou infinie. Cependant, ces profondeurs peuvent être déterminées à la compilation dans le cas du modèle SDF, en fonction de la stratégie d'ordonnancement utilisée. On remarque alors qu'à partir du moment où la profondeur des canaux est déterminée, il est alors facile de les mettre en œuvre sous la forme de tableaux. Le modèle SDF permet ainsi de découpler la politique d'ordonnancement des méthodes employées pour représenter les canaux (tableaux et adressage statique avec pointeurs de lecture et écriture sans modulo, tampon circulaire et adressage dynamique avec pointeurs de lecture et d'écriture avec modulo, listes chaînées ...).

Nous avons comparé les modèles SDF et ceux de SynDEx en utilisant les notations utilisées dans les graphes SDF. Un graphe SDF est monorythme si pour tous les arcs du graphe, la consommation et la production de jetons sont égales. Un graphe SynDEx sans factorisation et sans conditionnement peut donc être considéré comme un graphe SDF monorythme. Comme pour SynDEx, chaque acteur d'un SDF monorythme n'est invoqué qu'une seule fois dans la séquence périodique, la consistance des graphes est alors systématique.

Pour chacune des spécificités du modèle SynDEx, une comparaison avec le modèle SDF a été réalisée. SynDEx autorise par exemple les répétitions d'opérations : la spécification de telles opérations se fait alors de manière implicite ou explicite. La spécification implicite est déduite de la taille des ports. Lorsque deux opérations sont adjacentes par un arc sur lequel les rythmes de

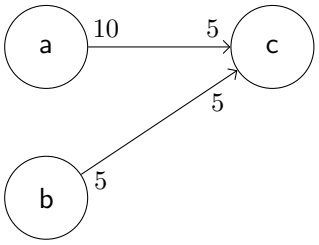
production et de consommation sont différents, il faut impérativement que l'une soit multiple de l'autre. Le modèle SDF est plus général : les rythmes de production et de consommation sont des entiers quelconques (ils peuvent notamment être premiers entre eux) et les répétitions des acteurs sont déduits de l'analyse SDF du graphe. Nos travaux ont alors montré que l'on pouvait utiliser les SDF dans la méthodologie AAA pour étendre la représentation des répétitions. Les propriétés d'analyse des SDF restent valables dans ces cas de répétitions. Ces propriétés peuvent être notamment utilisées dans la phase de placement/ordonnancement pour l'optimisation des opérations répétées, ce qui fait l'objet des travaux de thèse de Jonathan Piat.

Une autre différence entre le modèle SDF et le modèle utilisé dans SynDEx concerne les diffusions de données. La diffusion consiste à envoyer une même donnée sur plusieurs opérations ou sur plusieurs répétitions d'une opération factorisée. Dans le premier cas, cela signifie que plusieurs arcs homogènes peuvent être connectés à un même port de sortie. D'un point de vue formel, un arc SDF ne peut avoir qu'une unique extrémité initiale et une unique extrémité finale tandis que le modèle algorithmique de SynDEx utilise des hyper-arcs avec une extrémité initiale et plusieurs extrémités finales qui définissent le nombre de diffusions. Pour illustrer cette différence, prenons l'exemple de la figure 3.20. Le graphe de référence associé est le graphe 5.2(a), l'interprétation du graphe selon SynDEx conduit à une diffusion de données. En effet, l'opération $c$ a un rapport de production/consommation de 2 avec $a$ et de 1 avec $b$. L'opération $c$ est donc répétée 2 fois et les données produites par $b$ sont diffusées sur $c$. Cela conduit à la transformation représentée sur le graphe 5.2(b) à l'aide des opérations implicites `fork` et `diffuse`. L'opération `fork` sépare les données produites pour les répétitions de $c_1$ et $c_2$, tandis que l'opération `diffuse` leurs diffuse les données de $b$. L'interprétation du graphe de référence selon le modèle SDF est tout à fait différente : l'analyse du graphe donne le vecteur de répétition $\mathbf{q} = (1,\ 2,\ 2)^T$ : le vecteur de répétition est issu de la matrice de topologie et signifie dans notre exemple que l'acteur $a$ est répété 1 fois, $b$ 2 fois et $c$ 2 fois. Les données produites par $b$ ne sont donc plus diffusées vers $c$, ce qui conduit au graphe SDF monorythme équivalent 5.2(c). Autrement dit, la diffusion implicite utilisée dans SynDEx est donc incompatible avec le modèle SDF. Cependant, la méthodologie AAA n'étant pas remise en question, on peut considérer que l'utilisateur doit explicitement décrire dans son graphe les diffusions en ajoutant un nœud `fork` lorsqu'il veut diffuser une donnée vers deux consommateurs.

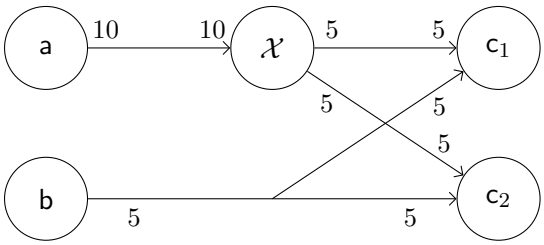
**Conclusion sur les modèles d'algorithmes**

En ce qui concerne les fonctionnalités de hiérarchie et de conditionnement que l'on peut utiliser pour construire un graphe d'algorithme avec l'outil SynDEx, les travaux de Jonathan Piat vont établir une équivalence dans le modèle SDF. La définition de la hiérarchie dans un SDF a déjà été décrite en collaboration avec l'Université du Maryland [PaMR09]. Il sera alors possible d'exprimer les applications dans Preesm avec les mêmes fonctionnalités que dans SynDEx, tout en utilisant les propriétés des SDF dans la phase de placement/ordonnancement. Le prix de cette modification est qu'il faudra forcément définir les graphes d'une manière légèrement différente que dans SynDEx et qu'une réutilisation directe des graphes SynDEx dans Preesm est et restera impossible. Le modèle SDF étant plus général, une transformation vers un modèle SynDEx peut par contre être envisagée.

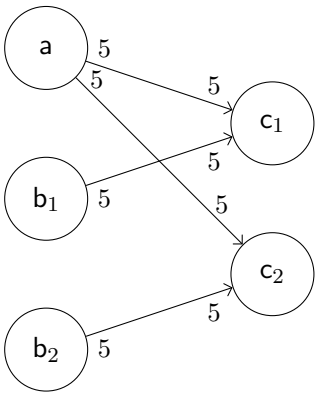
Le modèle SDF est à la base d'un grand nombre de travaux dans le cadre de la synthèse logicielle pour des architectures monoprocesseurs [PHLB95, HKB05]. Dans le cadre de la synthèse logicielle multiprocesseur, le DAG est quant à lui plus fréquemment utilisé de par le plus grand parallélisme de données qu'il permet d'exposer [KA99, Sin07]. Cependant, le passage d'un graphe SDF vers un DAG est possible en considérant le fait que le SDF est un DAG factorisé. Comme nous le verrons plus tard, nous réalisons dans Preesm une transformation SDF vers DAG pour mettre en œuvre




(a) Graphe de référence




(b) Transformation SynDEx du graphe de référence




(c) Équivalent SDF à rythme unique du graphe de référence

FIGURE 5.2: Diffusion

les techniques de placement/ordonnancement de la littérature.

### 5.1.2 Modélisation de l'architecture

Cette étude a été réalisée dans le cadre de la thèse de Pengcheng Mu et se poursuit avec les travaux de Maxime Pelcat. Le modèle utilisé pour la description du graphe d'architecture dans SynDEx est très simple. Une proposition d'extension pour la méthodologie AAA a été émise dans [GS03a] sans pour autant avoir été intégrée dans l'outil SynDEx. Dans ce modèle, les notions de mémoires internes (avec une distinction entre les mémoires de données et les mémoires programmes) et de bus ont été ajoutées. Cependant, avec l'avènement du multicœur et des MPSoC, il est nécessaire d'être de plus en plus précis sur les modes de communication entre processeurs et/ou coprocesseurs. Il est par exemple essentiel de pouvoir décrire le fonctionnement de commutateurs (***switch***) souvent utilisés pour connecter des bus. C'est la raison pour laquelle nous proposons une nouvelle modélisation de l'architecture dans AAA.

Ce modèle permet de spécifier l'utilisation d'un DMA (Direct Memory Access), capable de prendre en charge des communications en même temps que les cœurs continuent d'effectuer des calculs. La principale difficulté dans le choix d'un modèle d'architecture est qu'il doit être suffisamment précis pour que le placement/ordonnancement fasse des choix cohérents, mais pas trop précis car le calcul des coûts nécessaires au placement/ordonnancement doit être assez rapide. Le modèle actuellement utilisé est amené à évoluer pour trouver le bon compromis entre ces deux écueils.

Le modèle proposé est constitué des éléments suivant :

- **Processeur.** Le processeur est un opérateur. Il peut exécuter des opérations de calcul et de communication. Un processeur exécute les opérations de manière séquentielle. Il possède une mémoire interne accessible en lecture/ecriture pour des communications. Le processeur doit configurer un communicateur pour réaliser une communication et avant d'exécuter un nouveau calcul. La communication et le calcul peuvent être réalisés en parallèle. Un processeur est un nœud dans le graphe d'architecture.
- **Coprocesseur.** Le coprocesseur est un autre type d'opérateur. Il contient généralement une opération contenant du parallélisme et du pipeline. Il est utilisé pour une opération spécifique (généralement trop longue sur un processeur). Un coprocesseur ne peut pas configurer de communicateur, cette configuration étant réalisée par le processeur (de manière optionnelle) avec lequel le coprocesseur doit réaliser une communication. Le coprocesseur est un nœud dans le graphe d'architecture.
- **Mémoire.** Une mémoire est utilisée pour mémoriser des données lors de l'execution du programme. Une mémoire est un terminal esclave auquel peut accéder un processeur ou un communicateur via un nœud de communication ou un lien de communication (Bus ou FIFO). Une mémoire peut avoir plusieurs accès simultanés si elle possède plusieurs ports. La vitesse de lecture/écriture dépend de la bande passante donnée sur le lien de communication connecté à un port de la mémoire. Une mémoire est un nœud dans le graphe d'architecture.
- **Communicateur.** Un communicateur est un composant utilisé pour la réalisation de communications, et qui ne peut pas exécuter des opérations de calcul. Ceci correspond classiquement à un DMA (Direct Memory Access). Le communicateur est configuré par un processeur avant de réaliser une communication. Les communications ne peuvent être exécutées sur un communicateur que séquentiellement. Un communicateur est un nœud dans le graphe d'architecture.

- **Nœud de communication.** Un nœud de communication permet la connexion de plusieurs liens de communication (Bus ou FIFO). Il ne peut pas exécuter des opérations de calcul. Un nœud de communication modélise un commutateur considéré idéal (il peut réaliser plusieurs communications simultanément si elles ne mettent pas en jeu les mêmes opérateurs). Un nœud de communication est un nœud dans le graphe d'architecture.
- **Bus.** Un bus est hyperarc du graphe d'architecture. Cela signifie qu'il permet de connecter plusieurs nœuds du graphe entre eux (et souvent plus de deux). Les données sur le bus sont accessibles par tous les nœuds qui y sont connectés. A un instant donnée, un bus ne permet de réaliser qu'un transfert de donnée point à point.
- **FIFO.** Une FIFO est un arc orienté permettant de connecter un nœud du graphe en émission à un nœud du graphe en réception. Une FIFO bidirectionnelle peut être modélisée à l'aide de deux FIFOs.

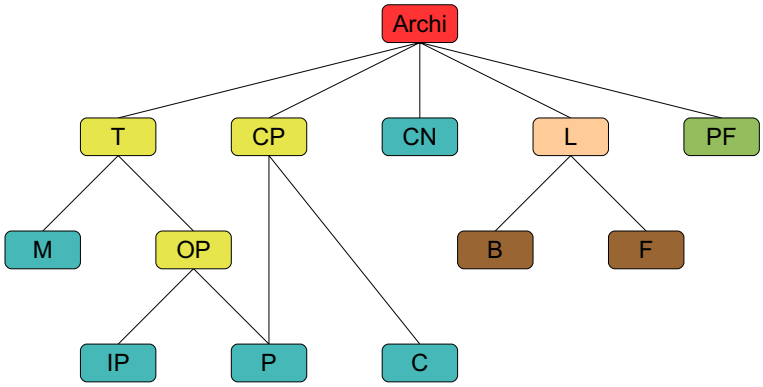



FIGURE 5.3: Organisation du modèle d'architecture

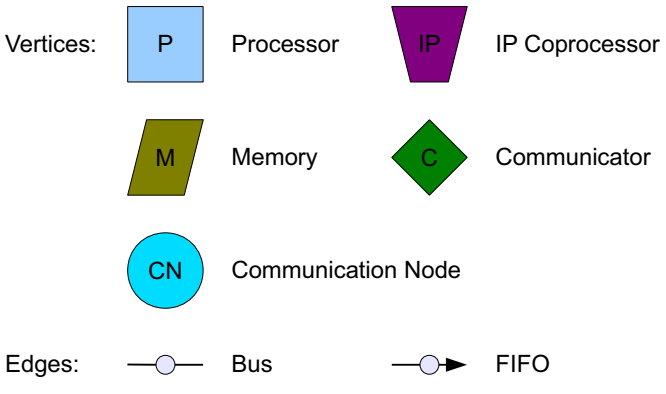
Ce modèle peut être considéré comme hiérarchique comme le montre la figure 5.3. L'union des processeurs $P$ et des coprocesseurs $IP$ est appelé l'ensemble des opérateurs $OP$, où $OP = P \cup IP$. L'union de $OP$ et des mémoires $M$ est appelé l'ensemble des terminaux $T$, tel que $T = OP \cup M$. L'union des communicateurs $C$ et de $P$ est appelé l'ensemble des opérateurs de communication $CP$, $CP = C \cup P$. Un nœud de communication est noté $CN$. Enfin l'union des bus $B$ et des FIFOs $F$ est appelé l'ensemble des liens de communication $L$, soit $L = B \cup F$.

$PF$ représente quant à lui l'ensemble des propriétés du graphe d'architecture et est constitué de 4 fonctions $c$, $s$, $a$ and $b$.

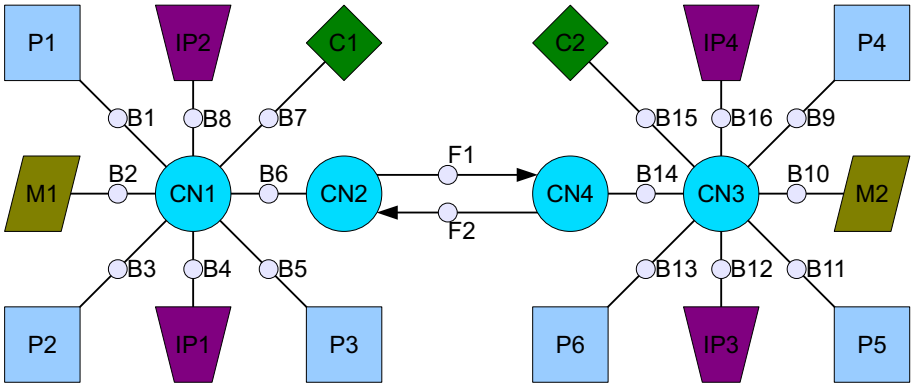
- La fonction $c : P \mapsto \{C_i | C_i \subseteq C\}$ fournit la configuration du processeur. $c(p_i)$ est l'ensemble des communicateurs pouvant être configurés par le processeur $p_i$.
- La fonction $s : P \times C \mapsto \mathbb{N}^+$ fournit le temps de configuration (set up) d'un communicateur par un processeur avant le départ d'une communication, avec $\mathbb{N}^+$ l'ensemble des entiers positifs. $s(p_i, c_j)$ est le temps passé par le processeur $p_i$ pour configurer le communicateur $c_j$ avec $c_j \in c(p_i)$.
- La fonction $a : T \mapsto \{CP_i | CP_i \subseteq CP\}$ fournit l'accessibilité d'un terminal par un opérateur de communication. $a(t_i)$ est l'ensemble des opérateurs de communication pouvant accéder à un terminal $t_i$. Lorsqu'un processeur $p_i$ est utilisé pour réaliser une communication, il peut toujours accéder à sa mémoire interne, d'où $p_i \in a(p_i)$.
- La fonction $b : L \mapsto \mathbb{N}^+$ fournit la bande passante moyenne d'un lien de communication. $b(l_i)$ est le nombre d'octets transférés par unité de temps.

Un exemple de description d'architecture est donnée dans la figure 5.4, représentant deux

multicœurs Texas Instruments TMS320C6474 (tricœurs) connectés par un bus SRIO (modèle de FIFO bidirectionnelles).




(a) Légende




(b) Deux DSP TMS320C6474 connectés par un bus SRIO

FIGURE 5.4: Exemple d'architecture multi-DSP

## 5.2 Optimisations des algorithmes de placement/ordonnancement

Les algorithmes de placement/ordonnancement ont été largement étudiés dans la littérature et particulièrement sur les applications modélisées par des DAG (Directed Acyclic Graph) [Sar89]. En revanche, décrire des algorithmes à l'aide d'un DAG se révèle difficile car ce modèle est contraignant. Nous opérons dans Preesm une transformation du graphe SDF, utilisé par l'utilisateur pour décrire son algorithme, vers un DAG, qui sera utilisé dans le placement/ordonnancement. Un exemple de DAG est illustré dans la figure 5.5.

Certains algorithmes de placement/ordonnancement de la littérature ne considèrent pas les communications entre processeurs de manière satisfaisante [Sar89, HCAL89, WG90, YG94, KA96]. Ces algorithmes utilisent comme architecture cible des topologies complètement connectées dans lesquelles il existe une connexion point à point entre tous les processeurs. Ces topologies sont cependant rarement utilisées dans des systèmes réels. Certains travaux [SL93, KA95, GLS99b, SS05] dont fait partie SynDEx permettent de décrire des architectures réelles et d'en tenir compte dans le placement/ordonnancement. Nous avons donc orientés nos recherches vers les algorithmes

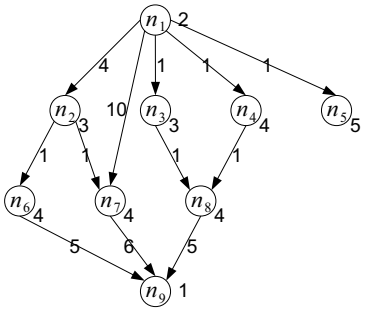



FIGURE 5.5: Exemple de DAG

dits de ***list scheduling*** qui sont les plus courants dans la littérature et qui diffèrent de l'algorithme glouton mis en œuvre dans SynDEx. Nous avons eu pour objectif d'utiliser l'algorithme le plus performant dans ce domaine [Sin07] et nous avons proposé trois techniques pour l'améliorer :

- l'utilisation de nouvelles priorités de nœuds,
- l'utilisation du fils critique,
- l'utilisation du retard de communication.

### 5.2.1 Les nouvelles priorités de nœuds

L'algorithme de référence débute par la création d'une liste à partir du DAG. Cette liste joue un rôle important dans la phase de placement. Dans l'algorithme de référence, deux valeurs appelées respectivement niveau supérieur (***top level***) et niveau inférieur (***bottom level***) sont utilisées pour la création de cette liste et sont appelées un groupe de priorités. Le niveau supérieur correspond à un coût calculé depuis le début sur le chemin critique d'un DAG et le nœud courant à traiter. Le niveau inférieur donne le coût entre le nœud courant et la fin du chemin critique.

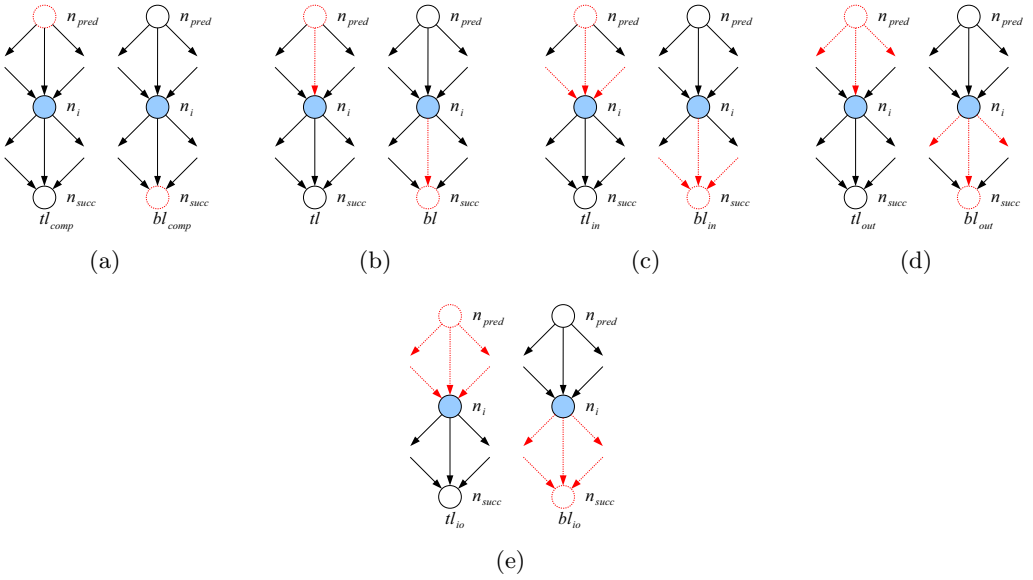



FIGURE 5.6: Les 5 groupes de priorité

Dans l'algorithme de référence, ce coût prend en compte seulement les nœuds précédent et

suivant le nœud courant comme illustré dans la figure 5.6(a) (nœuds en rouge) ou seulement un petit nombre d'arcs de communication, comme illustré dans la figure 5.6(a) (arcs en rouge). Les nouvelles priorités proposées intègrent l'ensemble des arcs entrants et sortants du nœud courant dans le coût de façon à calculer la liste de référence (Fig. 5.6(c), 5.6(d) et 5.6(e)). Les équations précises de ces calculs de coût sont données dans [Mu09].

Il faut noter que chaque groupe de priorité aboutit à la création d'une liste. Certaines listes peuvent être identiques avec des groupes de priorités différentes comme le montrent les résultats obtenus sur le graphe de la figure 5.5 du tableau 5.1. Dans notre exemple, l'algorithme de ***list scheduling*** devra traiter 3 listes différente, mais aurait pu devoir en traiter jusqu'à 5.

TABLE 5.1: Différentes listes statiques générées

| Groupe de priorités | Liste générée | No. |
|---|---|---|
| Groupe (a) | $n_1, n_4, n_3, n_2, n_8, n_7, n_6, n_5, n_9$ | (1) |
| Groupe (b) | $n_1, n_2, n_4, n_3, n_7, n_6, n_8, n_5, n_9$ | (2) |
| Groupe (c) | $n_1, n_2, n_4, n_3, n_7, n_6, n_8, n_5, n_9$ | (2) |
| Groupe (d) | $n_1, n_2, n_4, n_3, n_7, n_8, n_6, n_5, n_9$ | (3) |
| Groupe (e) | $n_1, n_2, n_4, n_3, n_7, n_8, n_6, n_5, n_9$ | (3) |

### 5.2.2 Le fils critique

Le chemin critique est le chemin dans le DAG, partant d'un nœud de départ vers un nœud d'arrivée, qui produit la valeur la plus forte lorsque l'on additionne les pondérations de chaque nœud et de chaque arc du chemin critique. Les pondérations étant des temps de calcul (pour un nœud) ou de communication (pour un arc), le chemin critique correspond à la limite inférieure de la latence de l'application. Dans l'algorithme de placement/ordonnancement, un nœud non placé est dit libre. Chaque nœud libre peut être placé à l'itération suivante du processus de placement. Il s'agit de sélectionner celui qui permettra d'optimiser la latence de l'application. Le fils critique est défini par rapport au nœud courant, comme le premier nœud libre situé sur le chemin critique. Nous proposons de traiter de manière prioritaire le fils critique dans l'algorithme de placement/ordonnancement en l'exécutant le plus tôt possible. La sélection du processeur pour le nœud courant prend en compte le fils critique, c'est-à-dire son successeur le plus important. Ceci a pour effet d'optimiser la latence sur le chemin critique et donc souvent de l'application en général.

L'exemple donné figure 5.7(a) permet de montrer le gain de l'utilisation du fils critique dans un cas simple. L'architecture utilisée pour cet exemple est une architecture à trois processeurs connectés par un bus. L'application est représentée par un DAG (Fig. 5.7(a)) à quatre nœuds. Le chronogramme de la figure 5.7(b) est le résultat de l'algorithme de référence, et le chronogramme de la figure 5.7(c) est le résultat de l'algorithme proposé avec le fils critique. Le fait d'avoir considéré le fils critique a permis de prendre en compte des communications issues des nœuds courants $n_1$, $n_2$ et $n_3$ et ainsi placer judicieusement $n_4$ sur le même processeur. La latence globale s'en trouve ainsi diminuée.

### 5.2.3 Le retard de communication

La dernière technique d'optimisation vise à modifier le positionnement de certaines communications. En effet, l'algorithme de placement/ordonnancement donne également l'ordre des communi-

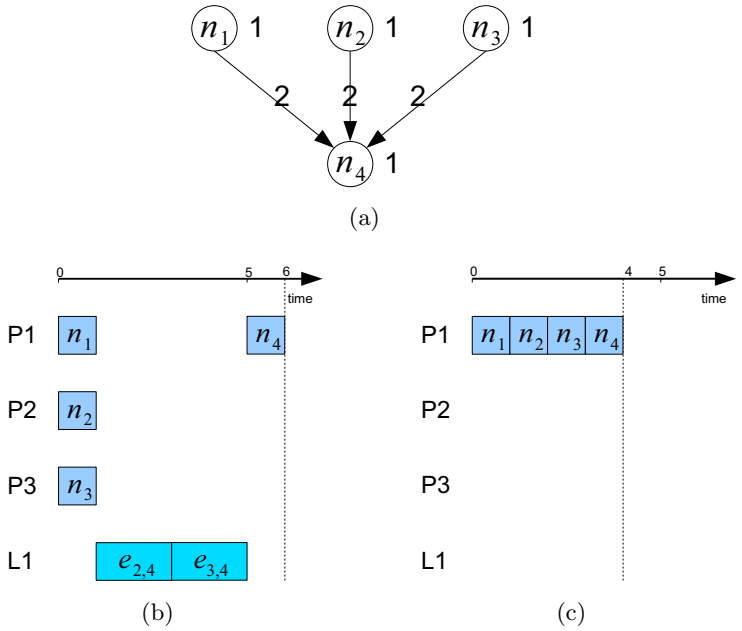



FIGURE 5.7: Un DAG et deux chronogrammes issus du placement/ordonnancement

cations sur les liens de communication de l'architecture. Dans l'algorithme de référence, lorsqu'une communication entre processeurs est nécessaire, celle-ci est placée le plus proche possible du nœud producteur des données. Ceci donne sur l'exemple de la figure 5.8 le temps $t_s(e_{ij})$.

L'intégration d'un retard de communication permet de placer cette communication plus tard dans le temps ($ALAP(e_{ij})$ sur notre exemple). Ceci a pour intérêt de pouvoir potentiellement insérer une autre communication entre $e_{ab}$ et $e_{ij}$ sur le lien de communication $L_1$ entre les processeurs $P_1$ et $P_2$ et ainsi optimiser les communications entre processeurs et diminuer la latence globale.

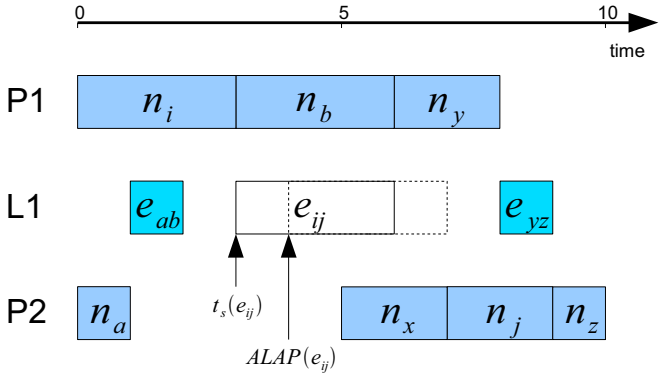



FIGURE 5.8: Principe du retard de communication

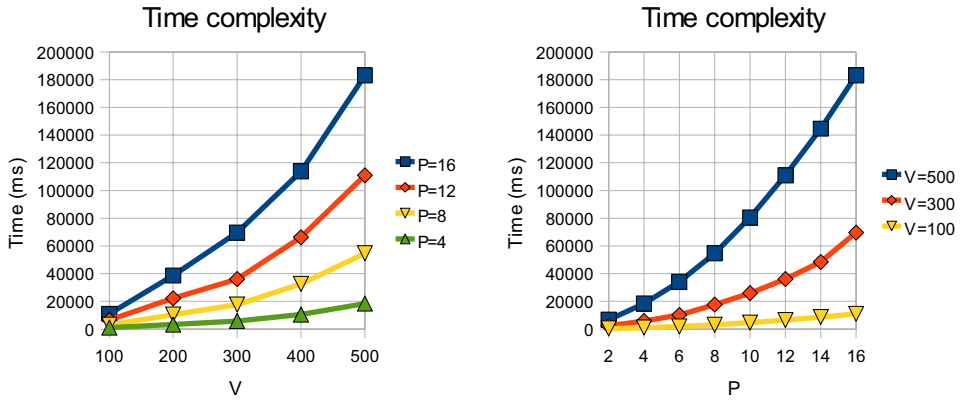
### 5.2.4 Résultats et perspectives

Les trois techniques proposées ont été testées séparément et de manière combinée. Pour réaliser les tests, nous avons utilisé les graphes donnés en exemple avec l'algorithme de référence, ainsi

que des DAG générés aléatoirement. L'étude a été menée en classant les DAG en fonction de l'importance des communications dans ces DAG. Nous avons alors pu constater qu'il était impossible de déterminer a priori les bienfaits des techniques les unes par rapport aux autres. En revanche, le fait de combiner ces techniques et de sélectionner le meilleur résultat apporte un gain non négligeable. Nous arrivons généralement à diminuer le nombre de processeurs nécessaires (ce qui n'est pas négligeable dans un contexte embarqué) et nous réalisons des gains de latence allant jusqu'à 80%. Plus les communications sont importantes dans l'application décrite, plus nos résultats sont probants. Ceci démontre que nous arrivons bien à prendre en compte les problèmes de communication dans le placement/ordonnancement.

La complexité de l'algorithme ainsi généré par rapport à l'algorithme de référence se trouve toutefois augmentée. L'algorithme de référence a une complexité en temps de $O\left(PE^2O\left(routing\right)+V^2\right)$, avec $P$, $V$ and $E$ respectivement le nombre de processeurs, le nombre de nœuds du DAG et le nombre d'arcs du DAG. $O\left(routing\right)$ représente le nombre de liens sur une route entre deux processeurs, valeur figée dans notre approche. L'algorithme proposé augmente cette complexité d'un facteur $P$.

La figure 5.9 montre les temps obtenus en faisant varier le nombre de nœuds $V$ des DAG et le nombre de processeurs $P$. Dans ce test, tous les processeurs sont connectés par un commutateur. On observe bien que le temps augmente proportionnellement aux carrés du nombre $V$ et du nombre $P$. Les temps donnés ont été mesurés sur un Pentium Dual-Core PC 2.4GHz. Il faut alors 3 minutes pour le placement/ordonnancement d'un DAG de 500 nœuds sur une architecture constituée de 16 processeurs. Ce temps reste raisonnable dans un contexte de prototypage rapide, même dans la phase de spécification lorsque l'on doit réaliser une exploration architecturale.




(a) (b)

FIGURE 5.9: Time complexity of the advanced heuristic

## 5.3 Architecture logicielle de Preesm

Afin de pouvoir collaborer de manière simple avec des partenaires universitaires ou industriels, l'environnement de Preesm est basé sur deux notions : le logiciel libre et le concept de plugin. L'architecture logicielle a été principalement proposée par Matthieu Wipliez, qu'il a mise en œuvre avec Maxime Pelcat et Jonathan Piat.

Preesm peut être vu comme le cœur d'un système communiquant avec des plugins. Chaque plugin va prendre en charge une partie du processus de prototypage. Ces notions de plugin et de logiciels libres sont déjà existantes sous la plateforme de développement logiciel Eclipse [ecl]. Les nombreux avantages proposés par Eclipse pourront être alors utilisés comme par exemple :

- l'utilisation de plusieurs langages de programmations dans les plugins,
- l'intégration de compilateurs (C, C++, Java, OCaml) dans le processus de prototypage rapide,
- la définition d'une licence par plugin (la licence libre CeCCIL B à l'heure actuelle),
- l'utilisation des bibliothèques GEF pour la mise au point d'interfaces graphiques.

Preesm est donc développé dans le langage Java comme un plugin Eclipse, communiquant avec d'autres plugins Eclipse grâce aux fonctionnalités de ce logiciel. Plusieurs outils sont actuellement développés et collaborent selon le principe exposé sur la figure 5.10.

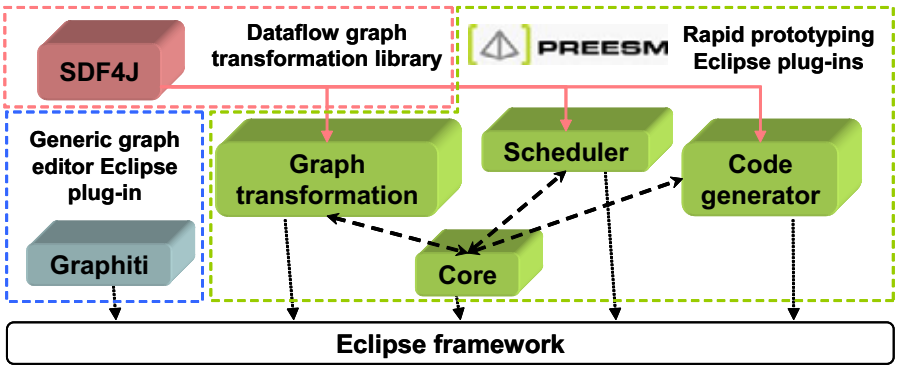



FIGURE 5.10: Structure logicielle

La première étape du processus consiste en la description de l'algorithme et de la plateforme cible à l'aide de deux graphes distincts. L'utilisation d'un éditeur graphique pour la création, la modification et la sauvegarde de ces graphes génère un gain de temps important dans le processus de développement. Le projet Graphiti [grab] a été initié pour fournir un tel éditeur graphique également dans l'environnement de programmation Eclipse. Graphiti sera utilisé dans le processus de prototypage proposé, tout comme dans les travaux concernant MPEG RVC présentés au chapitre 6. Pour cela, la principale caractéristique de Graphiti est de pouvoir être rapidement configuré et ainsi supporter tout type de format de fichier décrivant un graphe.

Le projet SDF4J [sdf] a été initié pour fournir une bibliothèque *open source* permettant de manipuler des graphes SDF dans le langage Java. Les besoins en termes de description des architectures étant plus spécifiques, nous n'avons pas jugé nécessaire de créer une bibliothèque équivalente pour la gestion des graphes d'architecture.

Le projet Preesm [pre] quant à lui concerne l'ensemble des calculs liés au prototypage rapide comme le placement/ordonnancement, la simulation ou encore la génération de code. Preesm utilise Graphiti et SDF4J pour la création et la manipulation des graphes décrivant l'algorithme et l'architecture. Preesm est constitué de plusieurs plugins Eclipse. Le premier, appelé Preesm Core, est chargé de l'exécution d'un workflow, qui est un graphe orienté représentant la liste des opérations de prototypage rapide à effectuer. Chacune de ces opérations fait l'objet d'un plugin Preesm spécifique. Chacun de ces plugins est optionnel et ajoute une fonctionnalité à l'ensemble du processus. A l'heure actuelle, il existe trois plugins Preesm en plus du Core : un pour la transformation de graphe, un second pour le placement/ordonnancement et un dernier pour la

génération de code. Chacun de ces plugins sera détaillé par la suite.

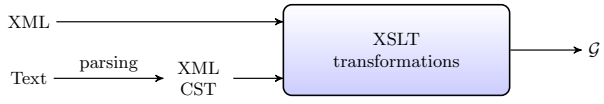
### 5.3.1 Graphiti : un éditeur de graphe générique

Graphiti est un plugin *open source* pour l'environnement de programmation Eclipse. Il s'y intègre pour permettre la visualisation de graphes plutôt que la visualisation du fichier mémorisant ce graphe. Ce logiciel est basé sur l'utilisation de la bibliothèque GEF (pour Graphical Editor Framework). Graphiti est générique dans le sens où il supporte l'édition et la représentation de tout type de graphe. Il est utilisé pour les graphes suivants : réseaux d'acteurs CAL [EJ03a, Jan07], un sous ensemble de la norme de représentation des architectures IP-XACT [spi08], les applications représentées sous le format GraphML [BEH+01] ainsi que les workflows de Preesm.

Pour qu'un éditeur de graphes puisse supporter un type de graphe particulier, il doit posséder une configuration spécifique définie dans un fichier XML et décrivant :

1. la ***syntaxe abstraite*** du graphe, c'est-à-dire les différents types de nœuds et d'arcs qui peuvent constituer le graphe, ainsi que les attributs que les objets de chaque type peuvent comporter,
2. la ***syntaxe visuelle*** du graphe, c'est-à-dire les couleurs et les formes qui devront être visualisées pour chaque élément apparent,
3. les transformations permettant de retrouver les différents éléments du graphe dans un format de fichier spécifique et de les mémoriser dans Graphiti (fichier XML), ou de créer un fichier spécifique à partir de Graphiti (Fig. 5.12).

Deux types de transformation peuvent être utilisés : transformation de XML vers XML, ou d'un format texte vers du XML (Fig. 5.12). Une transformation XML vers XML est par exemple utilisée pour transformer une description d'un algorithme au format GraphML (fichiers XML) en un arbre abstrait utilisable par Graphiti (fichier XML différent). Ces transformations sont indépendantes de la visualisation gérée par l'éditeur. Les formats XML sont transformés par des feuilles XSLT, alors que les formats textes sont pris en main et représentés sous la forme d'arbres abstraits (***Concrete Syntax Tree***, CST) représentés en XML à l'aide d'une grammaire LL(k) du parseur Grammatica [graa]. De manière symétrique, deux types de transformations peuvent être utilisées pour la sauvegarde : transformation de XML vers XML ou de XML vers texte.




(a) reading an input file to $\mathcal{G}$


(b) writing $\mathcal{G}$ to an output file

FIGURE 5.11: Input/output with Graphiti's XML format $\mathcal{G}$

A partir des fichiers de configuration, l'éditeur se charge de la visualisation du graphe dans l'environnement Eclipse, de sa modification par l'utilisateur ainsi que de sa sauvegarde si nécessaire. La création des fichiers de configuration reste simple et peut être réalisée en quelques heures par

une personne novice en programmation XSLT, alors que la prise en main des bibliothèques GEF nécessite plusieurs mois.

Graphiti est un outil complètement indépendant de Preesm. Cependant, l'éditeur de graphe permet de manipuler graphiquement les graphes utilisés dans Preesm : les workflows, les graphes d'architectures (format IP-XACT) et les graphes d'algorithmes (format GraphML). Sans Graphiti, les fichiers correspondant seront ouverts par un éditeur de texte, ce qui limite les possibilités de manipulation. La figure 5.12 montre un projet Preesm sous Eclipse dans lequel l'application est éditée avec Graphiti.

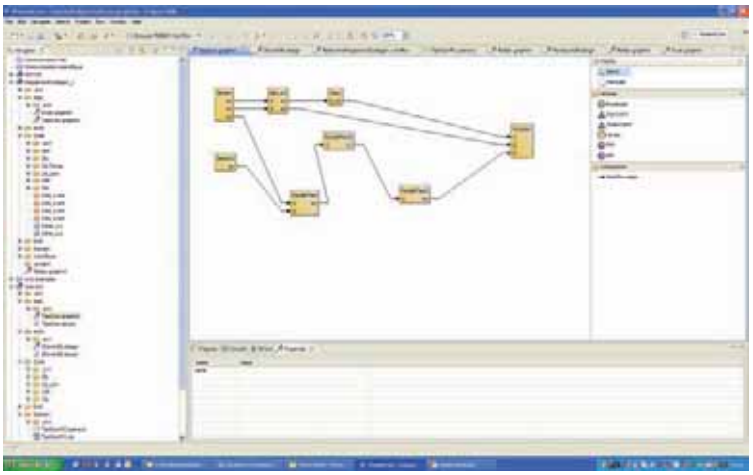


FIGURE 5.12: Projet Preesm sous Eclipse avec vue de l'application avec Graphiti

### 5.3.2 SDF4J : une bibliothèque Java pour les transformations de graphes flux de données

SDF4J signifie *Synchronous Data Flow For Java*. Cette bibliothèque offre des classes Java pour la manipulation de graphes Synchronous Data Flow Graph (SDF), Homogeneous Synchronous Data Flow Graph (HSDF) et Directed Acyclic Graph (DAG). Un exemple de manipulation peut être la mise à plat d'un graphe hiérarchique (création d'un graphe ayant un unique niveau hiérarchique). Un autre exemple peut être la transformation d'un type de graphe vers un autre (SDF vers HSDF, HSDF vers DAG ...). L'objectif est de rassembler dans une même bibliothèque l'ensemble des transformations usuelles et de fournir à l'utilisateur le plus de transformations possibles, quel que soit le but de son application. SDF4J est disponible gratuitement sous licence GPL [sdf].

Le modèle flux de données SDF [LM87b] est utilisé dans la spécification des applications sous Preesm. Le modèle SDF de base ne comporte pas de hiérarchie. Cette fonctionnalité s'avère pourtant indispensable pour une méthode de conception car cela permet la réutilisation de blocs fonctionnels. Nous avons donc été amenés à la définir cette fonctionnalité dans [PaMR09]. Une des transformations de SDF4J est la mise à plat d'un graphe hiérarchique, fonctionnalité qui permet de ne manipuler qu'un graphe simple lors d'un algorithme de placement/ordonnancement.

SDF4J fournit à ses utilisateurs des transformations de graphes SDF en DAG (Directed Acyclic Graph), car ce sont ces modèles qui sont très souvent utilisés dans les algorithmes de placement/ordonnancement de la littérature.

Une dernière transformation disponible grâce à SDF4J est la transformation de SDF vers HSDF. Dans ce dernier modèle, le nombre de données sur un arc est identique à l'émission et à la réception (caractère homogène du flux de données). Le HSDF est une version de représentation moins compacte que le SDF (moins pratique pour une spécification) mais qui permet d'augmenter l'expression du parallélisme de l'application, et qui donc présente un intérêt dans la phase de placement/ordonnancement.

D'autres méthodes sont actuellement étudiées pour fournir à l'utilisateur de cette bibliothèque de nouvelles possibilités.

### 5.3.3 Preesm : un processus de conception pour la conception de systèmes logiciels et matériels

Preesm est un outil de prototypage rapide réalisant plusieurs traitements distincts. La figure 5.13 décrit un enchaînement de traitements (appelé workflow) classique. Comme nous l'avons vu dans la section 5.3.2, le modèle flux de données choisi en entrée de Preesm est le modèle SDF, du fait de ses possibilités de vérification formelle et de son utilisation appropriée pour de l'ordonnancement statique multiprocesseur. Pour être efficace, il convient de traiter des graphes d'algorithmes pouvant aller jusqu'à quelques milliers de nœuds. L'architecture est décrite dans le modèle standardisé de description au format XML IP-XACT langage (standard IEEE du consortium SPIRIT [spi08]). Une architecture classique traitée dans Preesm contient jusqu'à quelques dizaines de cœurs. Il est possible de définir des paramètres (par exemple la taille maximale d'une image à traiter) ou des contraintes (parties d'algorithmes forcées sur certains processeurs) dans ce que nous appelons le *scénario*.

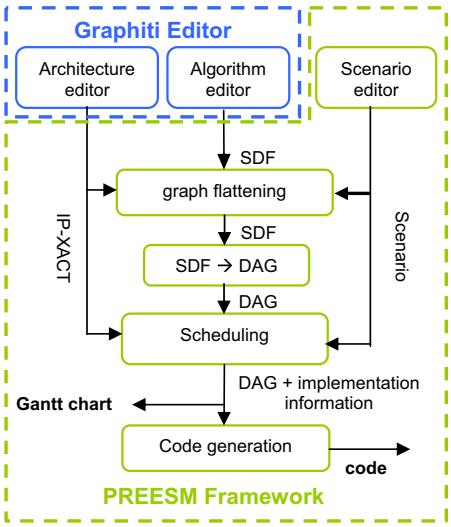



FIGURE 5.13: Exemple de workflow à partir d'un graphe SDF jusqu'à la génération de code

Le plugin dédié à la transformation de graphe a pour objectif de modifier le graphe SDF. Ce plugin génère des clusters définis dans [PBL95] visant à réduire le nombre de nœuds à traiter dans la phase de placement/ordonnancement et à faciliter la gestion des répétitions (groupes de nœuds devant être répétés lors de l'exécution).

Ensuite, le graphe SDF est converti en DAG pour la phase de placement/ordonnancement. Les différents nœuds du graphe d'algorithme sont alors placés et ordonnancés sur les éléments de

l'architecture pouvant les exécuter. Trois algorithmes de placement/ordonnancement ont actuellement été portés dans Preesm. Ils correspondent à des algorithmes proposés dans [Kwo97] et qui ont été optimisés :

- Algorithme de *list scheduling*,
- Algorithme FAST,
- Algorithme génétique.

Deux actions peuvent être retrouvées dans tous les **algorithmes de placement/ordonnancement** : sélectionner un cœur disponible pour un nœud du graphe, et évaluer le coût de la solution suite à cette sélection. La première action est indépendante de l'architecture, mais pas la seconde. Encore une fois, le plugin de placement/ordonnancement (Fig. 5.14) a été divisé en deux parties de telle sorte qu'il sera facile de faire évoluer l'une ou l'autre par des contributeurs différents.

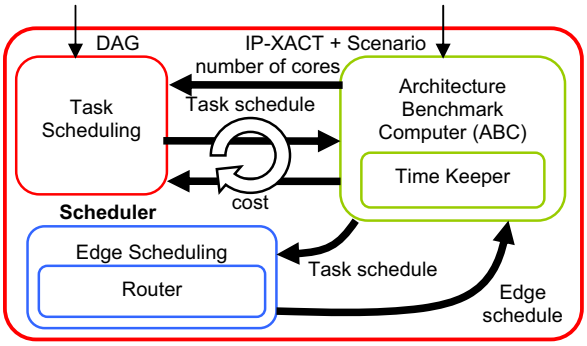



FIGURE 5.14: Structure du module de placement/ordonnancement

Lors de la phase de **génération de code**, suite au placement/ordonnancement, un code est automatiquement généré afin d'exécuter le déploiement sur une cible. De plus, il est possible de générer un graphe de Gantt très utile pour anticiper le fonctionnement d'une application lorsque le matériel n'est pas encore disponible : cela permet par exemple de dimensionner une architecture et de mieux la choisir parmi les solutions disponibles sur le marché.

## 5.4 Conclusion et perspectives sur le projet Preesm

Nos travaux depuis 2007 nous ont permis de mettre en place un environnement de développement sous Eclipse appelé Preesm. Il s'agit d'un logiciel pour le prototypage rapide qui a été pensé de manière modulaire. Les éléments clés de cette modularité sont : l'utilisation de l'environnement Eclipse, les notions de plug-in et de workflow, l'utilisation de plusieurs représentations de données utilisant des formats XML standard (IP-Xact, GraphML) ou non (pour la génération de code), une interface graphique pour les graphes indépendante de l'outil Preesm. Cette modularité a pour objectif la collaboration entre les membres de l'équipe, mais également avec d'autres partenaires académiques ou industriels. Pour cette première phase de développement, la thèse de Jonathan Piat vise à développer SDF4J et la phase de transformation dans Preesm pour traiter des boucles de calculs. La thèse de Maxime Pelcat est axée sur la modélisation d'une application spécifique appelée le LTE (Long Term Evolution, future norme de télécommunication), sur la mise en œuvre d'algorithmes de placement/ordonnancement ainsi que sur la génération de code. Les aspects visualisation de graphe constituent un des aspects de la thèse de Matthieu Wipliez (la

seconde partie étant liée à RVC que nous verrons plus tard, et qui nécessite également la partie visualisation de graphes).

D'un point du vue plus théorique, nous proposons de nouveaux modèles de graphes pour la modélisation des applications et des architectures. Le modèle d'architecture est plus précis que le modèle utilisé dans SynDEx (ce qui permet de modéliser des DMA et d'étudier les contentions sur les bus par exemple), et le modèle d'application est basé sur le modèle SDF, connu et reconnu dans le domaine du placement/ordonnancement. Les travaux actuels visent à étendre les possibilités de description des algorithmes en conservant la prédictibilité du modèle SDF. Ces propositions restent cohérentes avec la méthodologie AAA et vont nous permettre de mieux prendre en compte les architectures multicœurs et les MPSoC dans le placement/ordonnancement en particulier, et dans le processus de prototypage rapide en général.

L'outil Preesm est actuellement disponible sur SourceForge. Les descriptions des applications sont réalisées avec le modèle SDF, les algorithmes de placement/ordonnancement sont issus des meilleurs résultats connus sur le modèle DAG. Ces algorithmes avaient été présentés dans un contexte purement théorique. Ils sont maintenant intégrés dans l'outil Preesm et peuvent désormais être utilisés dans le cadre d'une méthodologie complète de prototypage rapide. La comparaison des résultats entre ces algorithmes est directe grâce à la visualisation sous la forme d'un diagramme de Gantt. Les fonctionnalités offertes par Preesm permettent de mettre en évidence rapidement les lacunes des algorithmes de placement/ordonnancement et de proposer des modifications au niveau de ces algorithmes pour un résultat optimal. Enfin, un code fonctionnel pour plateformes hétérogènes multicœurs est généré automatiquement avec des premiers résultats sur la génération de code optimisé pour une plateforme constituée de deux processeurs comportant trois cœurs chacun.

Comme nous avons pu le voir dans ce chapitre, nous ne pouvons pas réutiliser directement dans Preesm les applications décrites sous SynDEx. L'ensemble des travaux sur l'optimisation du placement/ordonnancement ont donc été réalisés sur des DAG générés aléatoirement. Les résultats obtenus sur les DAG aléatoires seront réutilisables lorsque nous devrons développer de nouvelles applications. Etant donné que le modèle implanté actuellement dans Preesm est le SDF hiérarchique, mais que nous cherchons encore à améliorer les fonctionnalités de description, nous avons jugé inutile à cette étape de développement de décrire toutes nos applications en SDF dans Preesm. Ceci constitue une perspective importante pour notre équipe.

Lorsque le modèle d'application de Preesm sera stabilisé, nous pourrons donc modéliser les applications complexes et concrètes présentées dans la section 4.1, comme nous avons pu le faire avec SynDEx V6. Nous pourrons alors juger définitivement de l'intérêt des méthodes proposées dans la thèse de Pengcheng Mu. Il est tout de même possible de tester ces algorithmes dans Preesm avec un plugin spécifique, mais leur intégration dans tous les plugins de Preesm se fera progressivement et définitivement lorsque ces applications complexes auront été étudiées.

Les résultats obtenus sur l'optimisation du placement/ordonnancement nous aideront à modéliser correctement nos applications. Ils pourront également être utilisés pour guider les choix des transformations du graphe SDF vers DAG (choix du niveau de hiérarchie, gestion optimisée des boucles). On pourra par exemple choisir un nombre fixe de nœuds du DAG en fonction du nombre de processeurs, ce qui permettra de sélectionner un niveau de hiérarchie (granularité intermédiaire) et de ne pas optimiser des boucles de trop bas niveaux (très bien traitées par un processeur VLIW). De plus, ceci aurait pour intérêt de fixer les temps d'adéquation dans Preesm et de ne pas dépasser des limites raisonnables. Une autre possibilité serait de choisir les techniques de placement/ordonnancement en fonction des caractéristiques de l'application, notamment en

estimant l'importance des communications d'une application modélisée sur une topologie donnée.

# Chapitre 6

# Normalisation MPEG RVC

Initié en 2005 par MPEG, le groupe de travail MPEG RVC a été chargé de définir une norme dans le but de lever les difficultés de spécification des précédentes normes MPEG. Contrairement aux autres normes MPEG déjà évoquées dans ce document, RVC ne définit pas de nouvelles technologies pour la compression vidéo. L'objectif est de pouvoir concevoir un décodeur à partir d'une spécification faite à un plus haut niveau d'abstraction que l'actuelle spécification définie en code C monolithique. Une spécification qui prône la modularité, la concurrence et la réutilisation est en effet de loin un meilleur point de départ à la fois pour accélérer l'adoption et la normalisation de nouvelles technologies et pour faciliter le processus de développement d'un décodeur. À cet effet, RVC a pour but de fournir une nouvelle spécification basée sur une modélisation à flux de données des normes de codage vidéo, modélisation bien adaptée aux algorithmes de traitement du signal et des images.

Comme nous l'avons évoqué précédemment, nos travaux nous ont apporté une expertise certaine dans les codecs MPEG. Nous avons notamment réalisé des descriptions flux de données pour les profiles ***Simple Profile*** et ***Advanced Simple Profile*** de la norme MPEG-4 Part2, ainsi que pour les décodeurs MPEG-4 AVC et SVC. Nous avons utilisé ces descriptions pour réaliser des implémentations optimisées sur plateformes embarquées multiprocesseurs. De plus, les enjeux affichés du nouveau standard RVC concordent avec ceux que nous avons dans nos activités sur le prototypage rapide, à savoir : accélérer le processus de développement d'une application à partir de sa description dans la norme. Notre investissement dans le groupe RVC a donc quelque chose de naturel.

Au moment où nous avons intégré le processus de normalisation, un formalisme faisant appel à un modèle flux de données avait été choisi dans MPEG RVC pour modéliser les codecs vidéo. En particulier, le langage RVC-CAL pour décrire les algorithmes avait été choisi. Ce langage semble être un bon candidat pour cette méthodologie. Un objectif de RVC est de fournir les descriptions des algorithmes MPEG décrites en RVC-CAL par des spécialistes du traitement des images. Ces descriptions améliorent la phase de compréhension des algorithmes (descriptions textuelles et logiciels de références) inhérente à la prise en main du standard. En outre, la phase de construction d'un graphe flux de données à réaliser à partir de la norme n'aurait plus lieu d'être, d'où un gain de temps non négligeable dans le flot de conception. D'après nos précédents travaux, on peut estimer que ces phases durent entre 1 et 2 ans avant d'avoir des solutions conformes aux tests requis par les standards MPEG. Ces délais pourraient être largement réduits par l'approche systématique proposée par RVC pour concevoir des décodeurs.

Cependant, le lien entre le langage RVC-CAL et la méthodologie de prototypage développée au laboratoire n'est pas direct pour deux principales raisons. Avec le langage RVC-CAL, il est possible de décrire un graphe flux de données constitué d'acteurs ainsi que de décrire le comportement interne de chacun de ces acteurs. La sémantique de RVC-CAL est très dynamique alors que celle des SDF de Preesm reste statique. De ce fait, le passage du graphe flux de données décrit en RVC-CAL en un modèle SDF n'est pas trivial. D'autre part, le comportement interne des acteurs est également décrit en RVC-CAL alors que dans Preesm, ce fonctionnement est décrit directement par l'utilisateur dans un langage compilable ou synthétisable (C, C++, VHDL...). L'utilisation des descriptions RVC-CAL nécessite alors une traduction vers un langage compilable ou synthétisable, que cela soit dans Preesm ou pour toute implantation faite à la main. Le langage RVC-CAL étant très expressif, l'élaboration d'une phase de traduction automatique n'est pas simple non plus.

En retour, pour RVC, l'utilisation de Preesm a un intérêt afin de pouvoir proposer des implantations optimisées sur plateformes multicœurs et MPSoC. Dans cette partie, nous allons présenter le contexte de MPEG RVC et ses problématiques. Nous allons également voir comment nous avons pu apporter nos connaissances à ce groupe de travail. Ces travaux ont permis de rédiger de nombreuses contributions pour la mise au point de la norme RVC (26 pour l'ensemble de l'équipe).

## 6.1 Présentation du standard MPEG RVC

### 6.1.1 Motivations

L'initiative RVC est motivée par l'analyse de l'historique des standards de compression. Les premiers standards MPEG ont été mis au point en 1993, et depuis, MPEG-2, MPEG-4, AVC (Advanced Video Coding) et SVC (Scalable Video Coding) ont été normalisés. Chaque nouveau codec a eu pour objectif de diviser d'un ordre deux le taux de compression à qualité constante. Cette amélioration des performances de compression est liée à l'utilisation de techniques de plus en plus complexes, qui ne peuvent être intéressantes qu'à partir du moment où les architectures de calcul peuvent les supporter. L'évolution des normes est donc complètement liée à l'évolution technologique des composants.

En termes de spécification, le groupe MPEG a considérablement évolué également : les premières normes étaient uniquement rédigées sous une forme textuelle. Devant les difficultés d'interprétation, MPEG a ensuite proposé la description de ses normes sous une double forme : un texte associé à un logiciel de référence, généralement codé en langage C ou C++. Cette association a amélioré la compréhension des normes et a ainsi accéléré leur utilisation dans les produits industriels.

Cependant, plusieurs aspects restent problématiques :
- certains algorithmes (appelés outils de compression) peuvent être utilisés dans plusieurs standards MPEG sans que cela ne soit clairement précisé, chaque norme étant indépendante,
- les logiciels de référence sont proposés à l'aide de descriptions séquentielles (C, C++) qui ne permettent pas d'exposer le parallélisme potentiel des codecs. C'est une difficulté que j'avais pu rencontrer pendant mes travaux de thèse pour Mpeg-4 partie 2. Il faut noter que le premier travail du développement dans ce cas est de faire l'étude du parallélisme disponible dans l'application qui est fournie dans un langage de programmation séquentiel, ce qui se révèle complexe,
- les logiciels de référence ayant été programmés par de nombreuses personnes et sans le souci de fournir une version optimisée, la prise en main du code est très difficile, au point que toute

personne désirant produire une version optimisée pour processeur ou pour des composants de logique programmable, doit repartir d'une page blanche,
- la difficile évolution d'un standard : une modification d'un algorithme, même mineure, ne peut être prise en compte que si un nouveau standard est à l'étude alors que ce processus dure plusieurs années (entre 3 et 5 classiquement). L'évolution d'un standard est donc très limitée,
- l'utilisation des standards dans des produits industriels : pour être satisfaisant, le produit ne peut pas se permettre de ne prendre en charge que le dernier standard. Il doit être rétro-compatible avec les anciens formats de fichiers et les techniques associées. Il doit aussi supporter, en plus des standards MPEG, d'autres formats comme WM9 de Microsoft et ses évolutions. On peut rapidement constater que cette tendance ne pourra pas continuer de la sorte, puisque juxtaposer les standards est en contradiction totale avec les principes des systèmes embarqués où l'on cherche à minimiser les ressources mises en œuvre.

### 6.1.2 Cadre de développement RVC

Pour toutes ces raisons, MPEG a initié le processus de standardisation RVC. Le travail consiste à lever tous les verrous précédemment cités en fournissant une spécification des standards avec un modèle à haut niveau d'abstraction. L'objectif est alors de pouvoir reconfigurer un décodeur au fil du temps, comme illustré sur la figure 6.1. MPEG RVC est en cours de normalisation dans le cadre des standards ISO/IEC23001-4 (ou MPEG-B part 4) [ISO09a] et ISO/IEC23002-4 (ou MPEG-C part 4) [ISO09b].

L'utilisation de langages de programmation classiques (codes monolithiques) n'étant pas appropriée à ce genre de descriptions modulaires, MPEG a opté pour l'utilisation du modèle de programmation flux de données. Pour cela, RVC définit une bibliothèque appelée VTL (Video Tool Library). La VTL est constituée d'outils de codage vidéo appelés FU pour *Functionnal Unit*. MPEG-B donne le cadre de développement des futurs codecs MPEG ainsi que la manière de spécifier les FU. Ainsi, MPEG-B définit que chacun des FU doit être associé à une description textuelle ainsi qu'à une description dans le langage de spécification RVC-CAL. Quant à MPEG-C, ce document définit la bibliothèque VTL utilisée pour les standards MPEG déjà développés [LAM09].

Dans la figure 6.1, les trois types de décodeurs sont conformes au cadre RVC. Le décodeur `Type-1` est construit uniquement à partir de FU de la VTL standard. Ce type de décodeur est alors en conformité avec les standards MPEG-B et MPEG-C. Le décodeur `Type-2` est construit à partir de la VTL mais également avec des bibliothèques propriétaires ($VTL\,\{1-n\}$). Ce type de décodeur est uniquement conforme au standard MPEG-B. Enfin, le décodeur `Type-3` est construit en utilisant une ou plusieurs bibliothèques propriétaires, et sans l'aide de la VTL standard. Tout comme le décodeur `Type-2`, il sera conforme uniquement à la norme MPEG-B. Avec ces éléments, RVC procure à des fournisseurs de services les éléments pour réaliser tous les codecs vidéo constitués à partir de FU, qu'ils soient propriétaires ou bien définis dans MPEG-C. Ceci va simplifier la définition des futurs standards MPEG et accélérera la création de solutions dédiées par la réutilisation des FU dans plusieurs standards.

Une plateforme de décodage RVC est illustrée sur la figure 6.2. Cette plateforme doit utiliser une description de décodeur (Decoder Description) qui spécifie l'architecture du décodeur et la structure du flux compressé indispensable au décodage. Cette description de décodeur est constituée de :

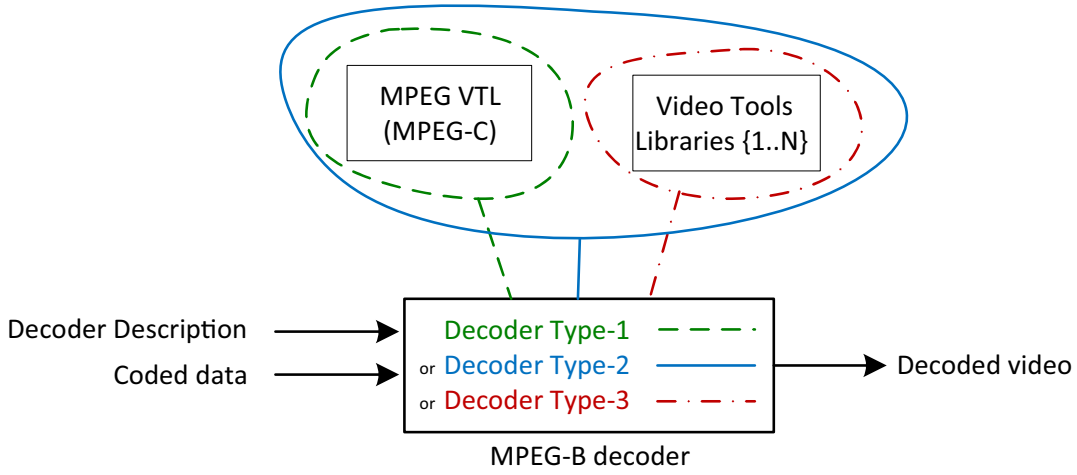



FIGURE 6.1: Le concept RVC.

- **la description du réseau de FU** (*FU Network Description*), écrite dans un format de description XML appelé FNL [DYLM08, JMP+08, RWR+08].
- **la description de la syntaxe du flux codé** (*Bitstream Syntax Description*), écrite dans le langage appelé RVC-BSDL. Ceci permet de générer un parseur capable de retrouver les données codées dans le flux entrant et de les interpréter. [Int, RPLM08],

L'association de la description d'un décodeur et de la VTL constitue le modèle de fonctionnement du décodeur (*Abstract Decoder Model*). Une fois que le modèle de fonctionnement du décodeur est spécifié, il possède toutes les données pour réaliser une instance de décodeur adapté au flux entrant. Pour créer cette instance de décodeur, il faut noter que les VTL (écrites en RVC-CAL) peuvent être optimisées pour chacune des plateformes envisagées (***MPEG Tool Library Implementation*** et ***Non MPEG Tool Library Implementation*** de la figure 6.2), à partir du moment où les fonctionnalités des FU sont respectées par rapport aux VTL (VTL standard et VTL propriétaire). On peut imaginer des implantations de VTL optimisée pour le multicœur ou pour FPGA par exemple.

Il faut préciser que dans le standard RVC, le décodeur ne peut être instancié qu'avec des éléments dont il possède les VTL, de sorte à éviter l'exécution de tout code malveillant. Seul l'assemblage de ces éléments est reconstruit à partir de données extérieures.

Dans le processus RVC, les descriptions de décodeurs en réseaux d'acteurs doivent être utilisées pour reconfigurer un décodeur. Un décodeur ne recevra plus un unique flux vidéo, mais trois flux différents. Le flux vidéo sera accompagné du RVC-BSDL précisant la structure du flux codé et du FNL précisant la structure du décodeur pouvant décoder le flux. Cette structure générique permettra de faire évoluer les décodeurs vidéo en les reconfigurant au fil des évolutions des algorithmes. Un exemple de modèle de fonctionnement RVC pour un décodeur MPEG-4 Part2 est donné dans la figure 6.3.

### 6.1.3 Le langage RVC-CAL

Le langage RVC-CAL est un sous-ensemble du langage CAL (Caltrop Actor Language). CAL est un langage créé dans le cadre des recherches sur Ptolemy II [Lee01, EJ03b]. RVC-CAL restreint

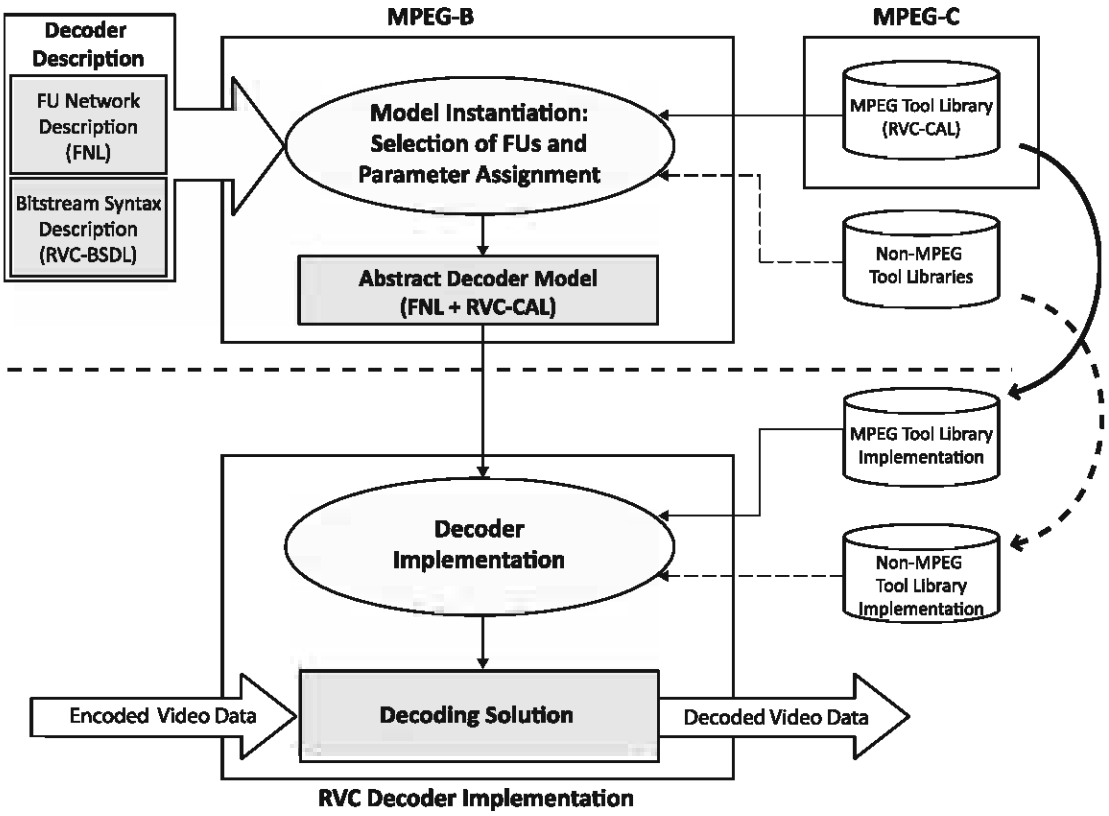



FIGURE 6.2: Illustration d'un décodeur RVC.

les types, les opérateurs et certaines fonctionnalités de CAL de manière à simplifier leur utilisation dans des générateurs de code. Les acteurs sont reliés entre eux dans un réseau d'acteurs : nous retrouvons donc une description flux de données. Le réseau d'acteurs est décrit dans un fichier FNL. Dans cette description, il est possible de déclarer des variables locales, de créer des instances de blocs et de leur associer des paramètres, de créer des descriptions hiérarchiques (un bloc peut être un FU ou un réseau d'acteurs). Contrairement à RVC-CAL, le FNL est un format XML et ne possède pas de syntaxe pour la programmation (calculs et boucles *for* par exemple).

La figure 6.4 illustre le principe de CAL et de sa sémantique. Un acteur est un composant modulaire qui encapsule son propre état. L'état d'un acteur ne peut être modifié par un autre acteur. Les interactions entre acteurs ne sont permises que par l'échange de données via des canaux au comportement de FIFO. Chaque acteur voit son comportement défini par un ensemble d'actions. Chaque action peut consommer des données sur les canaux entrants et produire des données sur les canaux sortants. Les actions d'un même acteur ne peuvent être exécutées parallèlement alors que les acteurs d'un même réseau peuvent l'être. Les transitions à l'intérieur d'une action (et d'un acteur) sont séquentielles.

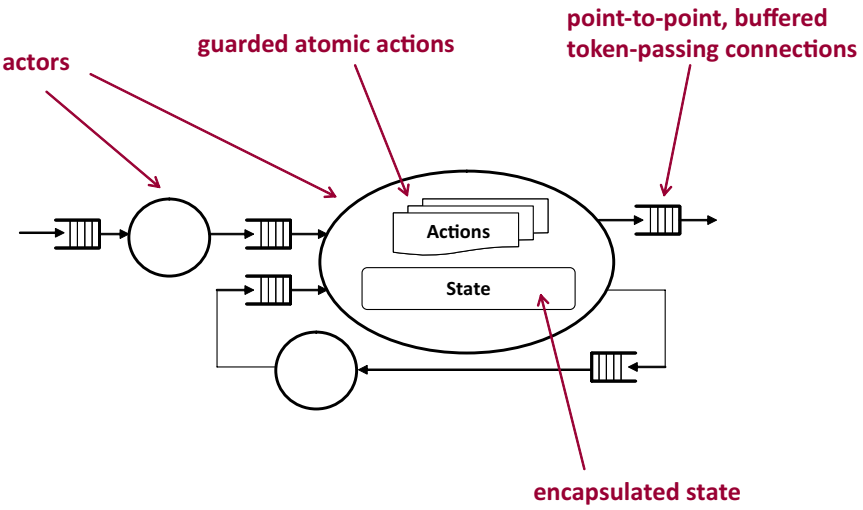



FIGURE 6.4: Représentation de la sémantique du langage CAL.

Le modèle CAL est un modèle flux de données basé sur le modèle Dataflow Process Network model (DPN) [LP95], dans lequel un acteur peut avoir plusieurs comportements en fonction des données entrantes. CAL étend le DPN en permettant l'indéterminisme. Ainsi, il est possible de spécifier en CAL des modèles DPN, mais également tous les autres modèles qui en sont des restrictions, comme le SDF qui est un DPN consommant et produisant un nombre de jetons identique à chaque itération. En revanche, le DPN étant plus expressif, les descriptions RVC ne sont jamais complètement conformes au modèle SDF.

Le code donné sur la figure 6.5 montre un acteur `A` utilisant à la fois des fonctionnalités des langages impératifs et fonctionnels pour extraire les composantes rouge, verte et bleue d'un pixel au format RVB. Cet acteur utilise un paramètre `COMPUTE_Y` qui conditionne les calculs de luminance, ainsi que plusieurs contraintes employées pour les masquages et les décalages intervenant dans les calculs. L'acteur `A` ne possède qu'une action qui nécessite `COUNT` jetons sur le port d'entrée `PIX` pour être exécutée.

Le contenu d'une action est similaire au contenu d'une procédure dans n'importe quel langage de

```
actor A (bool COMPUTE_Y) uint(size=24) PIX ⇒
  uint(size=8) R, uint(size=8) G, uint(size=8) B,
  uint(size=8) Y:

  int RSHIFT = 16; int RMASK = 255;
  int GSHIFT = 8; int GMASK = 255;
  int BSHIFT = 0; int BMASK = 255;
  int COUNT = 8;

  action: PIX:[pix] repeat COUNT ⇒
    R:[r] repeat COUNT,
    G:[g] repeat COUNT,
    B:[b] repeat COUNT,
    Y:[y] repeat COUNT
  var
    int i := 0
  do
    // imperative version to compute R, G, B
    while i < COUNT do
      r[i] := bitand(rshift(pix[i], RSHIFT), RMASK);
      g[i] := bitand(rshift(pix[i], GSHIFT), GMASK);
      b[i] := bitand(rshift(pix[i], BSHIFT), BMASK);

      i := i + 1;
    done

    // functional version to compute Y
    y :=
      if COMPUTE_Y then
        [rshift(
          66 * r[i] + 129 * g[i] + 25 * b[i] + 128,
          8) + 16 :
          for int i in Integers(1, COUNT) ]
      else
        [ 0 : for int i in Integers(1, COUNT) ]
      end;
  end
end
```

FIGURE 6.5: Un exemple de programmation en RVC-CAL

programmation impératif (langage C ou Java). Des variables peuvent être déclarées et initialisées avant les calculs. Les calculs peuvent inclure du conditionnement (`if/then/else`), des boucles (`for/while`), des appels de fonctions ou de procédures, des affectations de variables locales ou de variables d'état, qu'elles soient scalaires ou en tableaux. Le langage est structuré dans le sens où les branchements (`goto`, `break`, `continue`, `return` et `exit`) ne sont pas admis.

CAL utilise d'autre part des fonctionnalités des langages fonctionnels. La programmation fonctionnelle est un paradigme de programmation qui considère le calcul en tant qu'évaluation de fonctions mathématiques et rejette le changement d'état et la mutation des données. Elle souligne l'application des fonctions, contrairement au modèle de programmation impérative qui met en avant les changements d'état. Des exemples de langages fonctionnels sont ***Scheme*** ou ***OCaml*** ou encore ***Lisp***. Les fonctions dans ces langages sont dites sans effet de bords, c'est-à-dire qu'elles n'influent pas sur les autres fonctions de sorte que chacune d'entre elles peut être testée unitairement. En CAL, les expressions peuvent être écrites dans un conditionnement et avec l'affectation d'une seule variable. Les expressions conditionnelles (`if/then/else`) sont autorisées, tout comme les ***générateurs***. Les générateurs sont une sorte de boucle `for` pour la création de listes dont chaque élément est décrit par une expression. L'ensemble de ces fonctionnalités est relativement proche d'une description structurelle en VHDL.

Il est possible de programmer des machines d'états en CAL. Un exemple de multiplexeur est donné figure 6.6 et le code CAL correspondant dans la figure 6.7. Dans cet exemple, la machine d'états possède trois états (`init` étant l'état initial) et quatre transitions. Pour cela, chaque calcul de la machine est codée comme une action et chaque transition est définie après les mots clés `schedule fsm init`. On remarquera que les actions sont appelées sur les transitions du graphe, contrairement à ce qui est généralement fait dans d'autres langages comme le VHDL. Par exemple, l'action `readT` est réalisée sur la transition entre `init` et `waitA`. Nous avons donc dans cet exemple quatre actions. Les deux premières (`readT` et `readF`) permettent de tester la valeur de `S` à l'aide du mécanisme de garde (`guard`), les deux autres sélectionnent les données en entrée de l'acteur pour la recopier en sortie. Les actions `readT` et `readF` utilisent un jeton sur l'entrée `S` et ne produisent pas de jetons en sortie, l'action `copyA` (resp. `copyB`) va prendre la valeur du jeton sur le port `A` (resp. `B`) et le recopier en sortie. Nous avons donc bien un fonctionnement de multiplexeur. On vérifie sur cet exemple que le nombre de jetons pris en entrée et produits en sortie va dépendre des itérations des activations : nous sommes bien dans un modèle DPN.

Les principales caractéristiques de RVC-CAL qui diffèrent du langage CAL sont les suivantes :

1. RVC-CAL restreint l'utilisation des types à 6, dont 4 primitifs (**bool**, **float**, **int**, **uint**) et 2 étendus (**List**, **String**),
2. toutes les variables doivent être typées, ce qui signifie par exemple que le format des entiers doit être donné (signé ou non, nombre de bits) et que les tailles des listes et le type de leurs éléments doivent également être spécifiés,
3. le polymorphisme paramétrique (***generics*** en Java ou ***templates*** en C++) ne doit pas être utilisé, c'est-à-dire que tous les types doivent être déterminés lors de la définition de valeurs,
4. les fonctionnalités avancées de CAL sont interdites, comme la sélection des canaux entrant, les multiports ou les expressions lambda.

Une expression lambda est une fonction qui peut contenir des expressions et des instructions. Toutes les expressions lambda utilisent l'opérateur lambda $\Longrightarrow$, qui se lit ***conduit à***. Le côté gauche de l'opérateur lambda spécifie les paramètres d'entrée (le cas échéant) et le côté droit

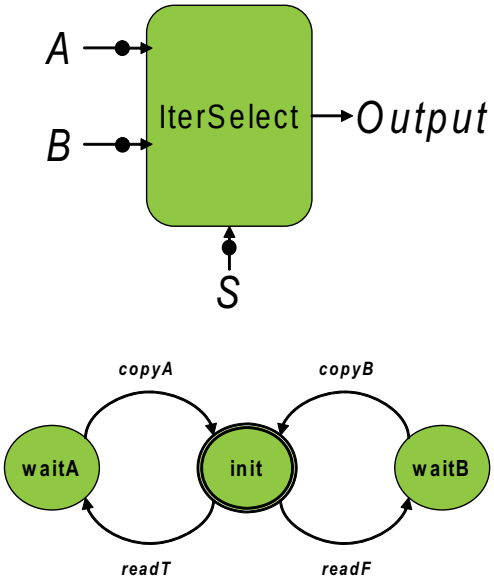



FIGURE 6.6: Représentation d'un multiplexeur et sa machine d'état.

```
actor IterSelect () S, A, B ⇒ Output :
        readT: action S: [s] ⇒
                guard s end

        readF: action S: [s] ⇒
                guard not s end

        copyA: action A: [v] ⇒ Output : [v] end
        copyB: action B: [v] ⇒ Output : [v] end

        schedule fsm init:
                init (readT) --> waitA;
                init (readF) --> waitB;
                waitA (copyA) --> init;
                waitB (copyB) --> init;
        end
end
```

FIGURE 6.7: Un exemple d'un multiplexeur en CAL

contient le bloc d'expression ou d'instructions. L'expression lambda x $\Longrightarrow$ x * x se lit ***x conduit à x fois x***.

Un exemple concret d'expression lambda est **l'évaluation partielle d'une fonction**. Prenons la fonction $f(x,y) = x + y$. La fonction $f$ peut être évaluée partiellement si elle peut être appelée avec un unique paramètre ($x$ par exemple) au lieu de deux ($x$ et $y$). Dans ce cas, la fonction $f$ va retourner une nouvelle fonction $g(y)$ qui lors de son appel calculera la somme $x + y$. Par exemple, si l'on appelle la fonction $f(x)$ avec $x = 5$, $f(x)$ retournera la fonction $g(y) = 5 + y$. L'évaluation partielle correspond à l'expression lambda $f(x,y) = functionx \Longrightarrow (functiony \Longrightarrow x + y)$. Les expressions lambda comme l'évaluation partielle, ne font pas partie des possibilités du langage C, mais sont à la base des langages fonctionnels comme CAL.

Le code de la figure 6.7 n'est donc pas du code RVC-CAL dans la mesure où les entrées ne sont pas typées.

## 6.2 RVC dans un contexte multicœur

Le standard MPEG utilise dorénavant des descriptions flux de données dans la spécification de ses algorithmes. Ceci permet d'exposer le parallélisme potentiel des standards, ce qui peut théoriquement être exploité dans un contexte multiprocesseur. Le parallélisme peut apparaître entre plusieurs éléments de la VTL puisque cette boîte à outils exprime des technologies de compression comme des acteurs indépendants. De plus, chaque élément de la VTL peut posséder une version propriétaire de chaque élément de la VTL standard optimisée pour le multicœur, comme l'illustre la figure 6.8.

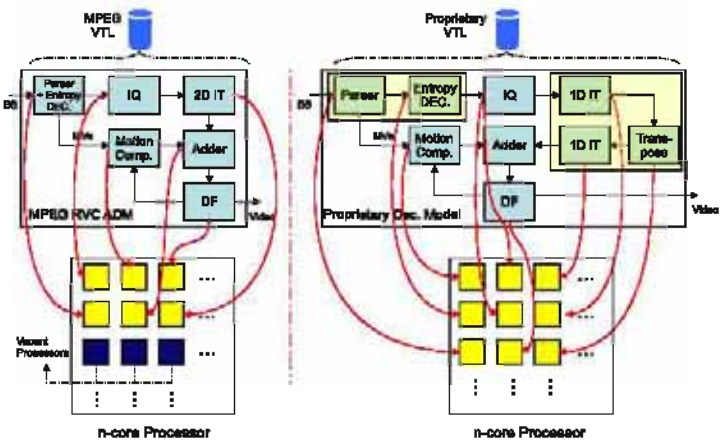



FIGURE 6.8: Illustration du placement d'un algorithme RVC sur une plateforme multicœur.

D'autre part, les fonctionnalités offertes par RVC vont contribuer à modéliser des décodeurs plus complexes et mieux adaptés au multicœur. Comparativement à SVC où la scalabilité a été définie de manière relativement rigide au niveau d'un décodeur, il sera possible avec RVC d'envoyer des flux dont les niveaux de scalabilité seront plus importants et qui pourront évoluer au fil du temps. Alors que les précédents standards étaient spécifiés d'une manière séquentielle, RVC ouvre la voie pour spécifier de nouveaux codecs vidéo plus parallèles et qui tireront parti des nouvelles architectures multicœurs.

SVC permet la difFUion de contenu multimédia en satisfaisant au mieux les utilisateurs en

fonction des débits disponibles et de la capacité de leurs terminaux. Alors que SVC utilise un certain nombre de niveaux de scalabilité définis une fois pour toute dans un codeur, RVC cherche à étendre cette scalabilité puisque la structure du décodeur pourra elle même évoluer et s'adapter à de nouveaux flux. On peut alors concevoir des codecs ayant plus de trois degrés de scalabilité (spatial, temporel, qualité et complexité). La figure 6.9 donne un exemple illustrant ce concept : le codeur RVC utilise trois niveaux qui seront envoyés sur plusieurs média de communication différents. Le décodeur qui pourra recevoir les données mettra à jour sa structure interne pour décoder les différents niveaux. L'utilisation d'un ou de plusieurs cœurs pourra être envisagée pour le décodage temps-réel de la vidéo complète avant affichage par un cœur chargé de la reconstruction finale. La reconfiguration du décodeur en fonction du flux émis augmente l'efficacité puisque le terminal n'intégrera plus un décodeur par type de flux émis.

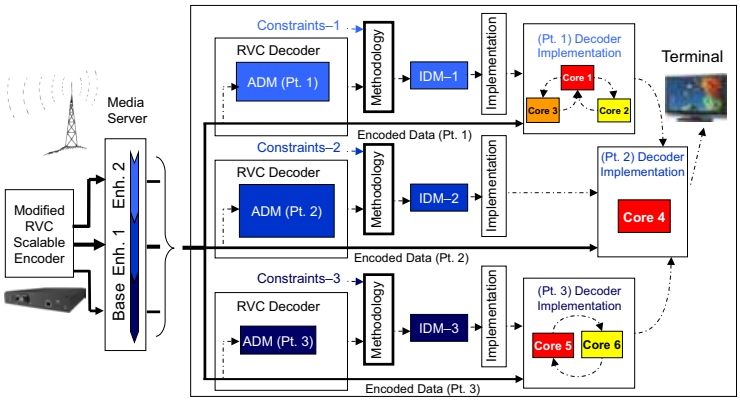



FIGURE 6.9: Exemple d'un décodeur RVC scalable dans un contexte multicœur.

Réaliser l'implantation d'un décodeur RVC sur plateforme multicœur consistera à analyser les différents acteurs du décodeur et à réaliser le placement de ces acteurs sur les cœurs disponibles. Chacun des acteurs pourra être transformé en langage C si ils ne font pas partie de la VTL, et compilé avant une phase d'édition de lien entre tous les acteurs et l'exécution du décodeur. Les acteurs faisant partie de la VTL auront pu être optimisés à la main et intégrés dans le terminal.

## 6.3 Synthèse logicielle de réseaux d'acteurs CAL

Les descriptions RVC-CAL réalisées dans RVC n'ont de sens que s'il existe des outils capables d'utiliser ce langage de conception haut-niveau. Les premiers outils ont été proposés pour simuler des descriptions RVC-CAL par les développeurs de ce langage dans le cadre du projet. L'idée est de favoriser la vérification fonctionnelle et comportementale des acteurs ainsi que des réseaux d'acteurs. Un autre objectif important de RVC est de permettre des implantations efficaces de programmes flux de données sur des architectures logicielles, matérielles et mixtes. D'autre part, dans le processus d'instanciation d'un décodeur RVC, il est intéressant de posséder un outil de transformation automatique de code RVC-CAL en code compilable. Des travaux pour la génération automatique de code HDL ont été réalisés par Xilinx [JMP+08]. Cette problématique a été initiée au laboratoire lors de la thèse de Ghislain Roquier [Roq08] et constitue le cœur de la thèse de Matthieu Wipliez [Wip10]. De plus amples détails pourront être trouvés dans [RWR+08, WRR+08].

Le langage CAL n'est pas uniquement un formalisme de description. Il est supporté par plusieurs environnements de simulation (Fig. 6.10). L'interprète est intégré dans les environnements de

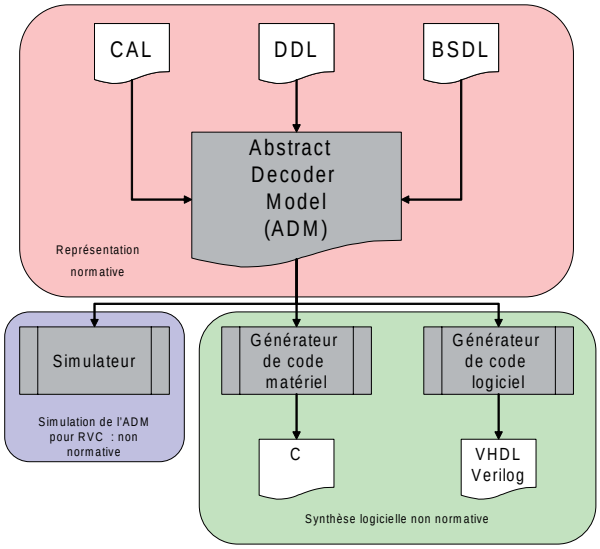



FIGURE 6.10: Outils dans RVC

modélisation et de simulation Open Dataflow, Ptolemy II et Moses. Le projet Open Dataflow (ou OpenDF, [opeb]) est une plateforme dédiée à la construction ainsi qu'à la simulation de modèles flux de données. Il supporte le langage CAL ainsi que le langage NL. OpenDF n'est cependant pas dédié exclusivement à ces deux langages et vise à supporter tous les langages qui se basent sur des descriptions structurelles et comportementales de modèles d'acteurs flux de données. Utiliser le langage CAL nécessite un interprète pour exécuter dynamiquement le programme. Un interprète écrit en Java est donc fourni dans l'outil OpenDF. Il est également intégré dans un environnement de modélisation et de simulation appelé Moses [mos] et développé à l'ETH de Zurich. Une description de décodeur peut être générée automatiquement par composition d'acteurs à l'aide d'un éditeur graphique. Il est possible avec cet outil de vérifier l'exécution au cours du temps de réseaux d'acteurs CAL et de connaître l'état courant des différentes FIFO ainsi que leur contenu.

Un premier objectif est donc pour nous de pouvoir transformer un code RVC-CAL en un code C, compilable sur une architecture monoprocesseur. Le propos n'est pas ici de proposer un générateur de code C optimal, mais de générer du code automatiquement qui pourra servir d'une part à la vérification fonctionnelle d'un programme CAL tout en accélérant le temps de simulation par rapport à l'interprète, et d'autre part servir de base pour une future génération de code optimisé. Ce code généré pourra alors être intégré dans une chaîne de développement logiciel classique basé sur le langage C, comme peut l'être actuellement un logiciel de référence. Un spécialiste des normes MPEG pourra alors passer facilement aux descriptions RVC sans devoir apprendre un nouveau langage de programmation et rendre l'adoption de RVC dans le groupe MPEG plus facile. De plus, une partie de ce code généré pourra être utilisé comme le code relatif aux opérations du graphe d'application utilisé dans une méthodologie de prototypage rapide comme Preesm que nous avons présenté.

La figure 6.11 décrit les étapes successives qui, à partir de sources CAL et NL, mènent à la génération de code C. L'ensemble des acteurs à été traduit en langage C alors que le réseau d'acteurs a été traduit en SystemC pour pouvoir réaliser l'ordonnancement des acteurs sur un processeur.

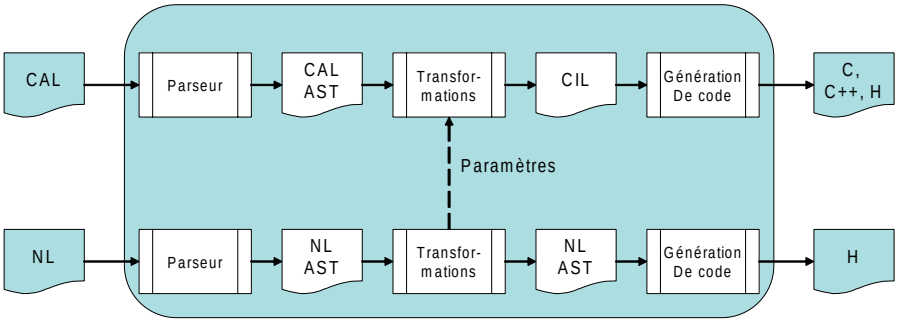



FIGURE 6.11: Processus de synthèse logicielle dans Cal2C

Cette première synthèse logicielle a apporté une nette amélioration des performances vis-à-vis du simulateur intégré à l'outil OpenDF comme l'illustre le tableau 6.1. Ce tableau montre les différents temps d'exécution (deuxième colonne) et la taille du programme du décodeur MPEG-4 Part2 (troisième colonne) en nombre de lignes de code (SLOC pour *Source Line Of Code*). Ces résultats sont donnés pour la simulation (deuxième ligne) avec OpenDF, pour la synthèse logicielle avec CAL2C (troisième ligne) et pour la synthèse matérielle avec Cal2HDL (quatrième ligne). Il faut noter que les résultats de la synthèse matérielle ont été donnés par Xilinx. L'exécutable issu de la synthèse logicielle a une fréquence (en macroblocs par seconde) d'environ 130 fois plus élevé que pour la simulation. Le décodeur est proche du temps réel pour un format QCIF ($176 \times 144$ pixels), le débit est alors de 20 images par seconde au lieu de 25 pour atteindre le temps-réel. En comparaison, la simulation a un débit de 0,15 image par seconde. La synthèse sur FPGA de la description matérielle a quant à elle un débit très important par rapport à la synthèse logicielle. Le décodeur est temps-réel pour la résolution haute définition (full HD) 1080p30 ($1920 \times 1080$ pixels en balayage progressif à une fréquence de 30 images par seconde).

| **Décodeur MPEG4 SP** | **temps d'exécution** (MB/s) | **taille du code** (SLOC×1000) |
|---|---|---|
| simulateur CAL | 15 | 3.4 |
| Cal2C | 2000 | 10.4 |
| Cal2HDL | 290000 | 4 |

TABLE 6.1: Temps d'exécution et taille du code pour le décodeur MPEG-4 Part2

Les performances de la synthèse matérielle nous ont toutefois ouvert les yeux sur les travaux qui restent à réaliser pour obtenir une solution plus efficace sur des architectures formées de processeurs. En effet, la solution de décodage fournie par Cal2C dans cette première version est plus proche de la simulation et de la vérification fonctionnelle que d'une réelle solution temps-réel de l'application implantée sur des architectures multiprocesseurs. Cependant, cette première étape est nécessaire pour mener à bien ce défi. Les avantages fournis par Cal2C sont multiples. Tout d'abord la traduction des actions en langage C est totalement indépendante d'un quelconque environnement de programmation : cette traduction s'avère donc tout à fait pérenne pour une future synthèse logicielle plus efficace. Ensuite, la spécification de la structure du code généré pour l'ordonnanceur des actions a été réalisée afin d'être le plus conforme possible au C. Seules les fonctions SystemC relatives aux transferts de données au travers de FIFO ont été ajoutées. Finalement, la construction du réseau a permis de comprendre le langage CAL en vue d'une mise

en œuvre efficace.

## 6.4 Implantation d'un décodeur MPEG-4 Part2 sur plateforme multicœur

Nous avons étudié l'implantation du décodeur MPEG-4 Part2 décrit en RVC-CAL sur une plateforme multicœur. Nous avons pu illustrer l'utilisation de RVC dans ce contexte et étudier les performances obtenues. Alors que l'étude du programme séquentiel avait été problématique lors de ma thèse, la réutilisation de la description RVC-CAL a été très rapide, ce qui constitue un gain indéniable pour un processus de développement. La structure du décodeur en RVC est donnée dans la figure 6.3. Le décodeur est constitué de quatre parties : un parseur et le traitement de chaque composante Y, U et V. Chacune des composantes de l'image passe par deux FU que sont le décodeur de texture et la compensation de mouvement. Ces deux FU sont décrites de manière hiérarchique.

Il existe donc un parallélisme potentiel que nous pouvons exploiter dans un contexte multicœur. Cependant, le placement de ces éléments permettant d'optimiser l'implantation multicœur doit être déterminé. Pour cela, nous pouvons utiliser CAL2C pour générer un code C, et intégrer ce code dans plusieurs tâches connectées par des FIFOS. Nous tests ont été effectués sur des PC multicœur avec un système d'exploitation *Windows*. Ce système d'exploitation est alors capable de placer de manière dynamique les tâches créées sur les cœurs disponibles. Il faut noter que cette technique engendre des changements de contexte et potentiellement des préemptions entre tâches. Comme un acteur n'est pas sensé se faire interrompre tant qu'il n'a pas terminé son calcul, une implantation multitâche n'est pas la plus appropriée. D'autre part, le nombre de tâches créées ne doit pas être trop important pour qu'un OS reste performant : ainsi la granularité de la description utilisée doit être limitée [Lee06].

Une autre solution plus adaptée est d'utiliser des techniques d'ordonnancement comme le *round-robin*. Cette technique appelle les acteurs les uns après les autres dans une tâche de l'OS. Une tâche par cœur disponible est créée et les acteurs sont placés manuellement dans les tâches. Il est alors facile de tester plusieurs configurations pour sélectionner la plus performante. Etant donné qu'il existe une tâche par cœur, et que chacune de ces tâches doit être exécutée continuellement, nous sommes dans un cas où l'OS doit assigner une tâche à un cœur et ne pas réaliser de préemption.

Le tableau 6.2 montre quelques résultats sur une plateforme dual-core. Toutes ces solutions sont valides fonctionnellement, même si elles diffèrent par leur efficacité.

TABLE 6.2: Chronométrage du décodeur MPEG-4 Part2 avec un processeur dual-core 2.5GHz sur une séquence SD (720x576 pixels).

| Core 1 | Core 2 | images/sec |
|---|---|---|
| Parser + TextureYUV + MotionYUV + Merger | | 5.9 |
| Parser + TextureYUV + MotionYUV | Merger | 5.7 |
| Parser + TextureYUV | MotionYUV + Merger | 8.5 |
| Parser + TextureUV + MotionUV | TextureY + MotionY + Merger | 9.2 |

L'objectif du placement effectué est de minimiser au maximum les transferts entre cœurs. Le meilleur résultat sur cette expérience est obtenu lorsque le parseur et les calculs sur les chrominances (décodage de texture et compensation de mouvement) sont exécutés sur un cœur, et que le second cœur réalise à lui seul le décodage de la luminance et la reconstruction de l'image à afficher. Ceci a pu être déterminé assez facilement en sachant que les données utilisées pour le décodage de la luminance sont 4 fois plus nombreuses que celles utilisées pour chacune des chrominances (format 4 :2 :0). Nous avons alors une implantation parallèle sous la forme d'un pipeline.

Le tableau 6.3 montre les différentes vitesses de décodage du décodeur MPEG4-SP sur un et sur deux cœurs respectivement. Cette expérience est prometteuse puisque des accélérations ont pu être chronométrées. Ainsi, nous pouvons espérer trouver un algorithme de placement/ordonnancement automatique optimisant ce processus.

TABLE 6.3: Accélération du décodeur MPEG-4 Part2 en utilisant plusieurs cœurs.

| | Throughput (images/sec) | | Speedup Factor |
|---|---|---|---|
| | Single Core | Dual Core | |
| QCIF (176x144) | 78.5 | 122.7 | 1.56 |
| CIF (352x288) | 21.4 | 32 | 1.49 |
| SD (720x576) | 5.9 | 9.2 | 1.57 |

## 6.5 Conclusion et perspectives sur le projet MPEG RVC

Notre implication dans le groupe de travail RVC a été importante depuis 2007. Cela constitue une des missions principales de Mickael Raulet, ingénieur de recherche titularisé dans notre équipe en décembre 2008. Outre le fait de participer à toutes les réunions de normalisation, Mickael Raulet a participé à la rédaction de la norme RVC et propose des techniques permettant de valider les développements réalisés dans le groupe de travail. En effet, pour la création de la VTL, les membres qui participent à son développement testent chaque FU de manière indépendante et sans les intégrer dans un décodeur complet. Une approche systématique a donc été proposée pour intégrer un FU dans un décodeur de référence de manière à valider ce FU avec l'ensemble des séquences de test fournies pour chacun des décodeurs MPEG. Cette technique a permis de retrouver des erreurs dans la VTL et d'accélérer les tests de conformité. Le groupe de travail RVC centralise de nombreuses initiatives et de très nombreuses discussions autour du prototypage rapide et des méthodes de conception. L'implication dans MPEG RVC demande beaucoup d'investissement en termes de temps, mais en retour, cela apporte à l'équipe beaucoup d'informations sur les travaux en cours dans notre domaine et donne au laboratoire une visibilité au niveau international.

En ce qui concerne l'avenir du standard, RVC n'apportera une véritable solution que si la norme peut être utilisée directement pour la conception de systèmes optimisés. Le développement d'outils de synthèse qui permettent le passage à des solutions d'implantations efficaces reste un problème ouvert. Les premiers résultats présentés dans ce chapitre ont contribués à transformer des descriptions RVC-CAL en un code C pour les calculs effectués à l'intérieur des acteurs, et en SystemC pour la structure globale du programme équivalent au fonctionnement d'un réseau d'acteurs. Ceci correspond à une première étape dans l'automatisation du processus de conception d'un système embarqué basé sur des descriptions MPEG RVC. SystemC étant principalement utilisé dans un contexte monoprocesseur pour simuler des systèmes électroniques complexes, notre outil de synthèse devra évoluer pour éliminer cette partie et la remplacer par des primitives d'OS

embarqués (ordonnancement dynamique) et/ou par des fonctions écrites en langage C (ordonnancement statique).

Par ailleurs, les outils de synthèse sont actuellement distincts en fonction des cibles visées : notre outils de synthèse logicielle pour des cibles logicielles (DSP, ARM) et des outils de synthèse matérielle (VHDL, Verilog) pour les cibles matérielles (FPGA, ASIC) [JMP+08]. Ces outils doivent encore progresser de façon a être plus rapides pour la génération de code (ce qui influera sur l'instanciation des décodeurs RVC), en générant un code plus optimal, et en prenant en compte des plateformes multiprocesseurs, multicœurs et hétérogènes. Un objectif sera alors d'intégrer des algorithmes de partitionnement logiciel/matériel en amont des outils existants. Nos travaux autour de Preesm pourront être utilisés dans ce contexte. Deux thèses sont actuellement en cours sur cette thématique [Sir11, Wip10] à l'IETR, et une thèse réalisée par Jérôme Gorin a débuté avec l'INT Evry où Mickael Raulet participe à l'encadrement.

# Chapitre 7

# Conclusion et perspectives

Ce document de synthèse a montré comment, à partir de la méthodologie AAA, nous avons participé à l'évolution du logiciel SynDEx et comment nous mettons maintenant en œuvre le logiciel Preesm. Initialement, nos travaux se sont focalisés sur l'utilisation du logiciel SynDEx pour utiliser cet outil dans un processus complet de conception de systèmes embarqués, en nous basant sur des compétences dans les applications de traitement des images, ainsi que dans l'optimisation de code pour systèmes embarqués (DSP/FPGA). De nombreux algorithmes ont été étudiés dans nos travaux : MPEG-4 Part2, AVC, SVC, estimateurs de mouvement et LAR pour la compression vidéo, mais également UMTS, MC-CDMA et LTE pour les communications numériques via des collaborations. Des optimisations du processus de génération de code portant sur la taille mémoire, sur l'utilisation des mémoires caches ou encore sur l'utilisation d'OS multitâches, ont été mises en œuvre. Des résultats tangibles ont été obtenus sur de nombreuses plateformes matérielles : cartes multiprocesseurs (Vitec, Sundance ou Pentek), processeurs multicœurs Texas Instruments, plateformes de développement Xilinx, PC en réseau et PocketPC. Ces résultats ont été utilisés dans des démonstrateurs lors de projets collaboratifs nationaux comme Mobim@ges, Scalim@ges et le seront encore par exemple dans le projet SVC4QoE.

Les applications de traitement des images développées sous SynDEx (V6) pourront être reprises sous l'outil Preesm pour démontrer sur ces cas les gains de la méthodologie de conception. Les premiers cas applicatifs portés par Preesm sont liés à LTE (*Long Term Evolution*), une application de télécommunication pour la téléphonie 4G. De même, la génération de code pour plateformes embarquées reste un objectif fort. La génération de code à partir de Preesm est désormais fonctionnelle et a été testée pour les derniers composants multicœurs Texas Instruments. L'utilisation d'architectures spécifiques MPSoC sur FPGA sera prise en compte ultérieurement. Une opération du graphe d'architecture pourra être prise en charge par un IP sur une topologie spécifiée dans le graphe d'architecture, ce qui représentera une amélioration par rapport à l'utilisation que nous avions pu faire de SynDEx où une seule opération pouvait être prise en charge par un FPGA.

Le point fort de Preesm est la modularité de la solution, qui divise ce problème complexe en sous-parties. Chacune des sous-parties pourra faire l'objet de nouveaux travaux dans l'avenir, à travers des stages, thèses, contrats de collaborations avec des industriels ou projets universitaires. Il sera alors plus facile pour une personne spécialiste d'une sous-partie donnée de valider ses résultats en montrant l'impact de ses travaux sur le processus de conception complet. Si l'on prend la problématique du placement/ordonnancement, un travail dans ce domaine est classiquement validé sur des graphes décrits à la main. Ces graphes sont généralement assez simples et ne reflètent pas des applications réelles comme celles utilisées au laboratoire (algorithmes vidéo ou télécom).

Avec notre approche, les modifications sur les algorithmes de placement/ordonnancement sont directement testées sur ces applications pour des tests intensifs. Cet exemple peut être étendu aux autres aspects comme la modélisation d'applications, la génération de code, ou l'optimisation de boucles. La structure logicielle de Preesm a été pensée pour la capitalisation des résultats à long terme.

Le futur pour un logiciel comme Preesm est d'aller encore plus loin dans les modèles de spécification d'algorithmes. A l'heure actuelle, l'outil prend en charge le modèle SDF, et donne de bons résultats pour des applications déterministes. Etendre les résultats en permettant aux utilisateurs de spécifier leurs algorithmes à l'aide de modèles plus dynamiques (comme les modèles RVC-CAL par exemple) va remettre en cause l'ensemble des phases de placement/ordonnancement actuels. Il faudra alors ajouter des phases dans le *workflow* de Preesm en amont et en aval des algorithmes déjà développés. Encore une fois, la structure logicielle modulable adoptée a été pensée pour aboutir à ce résultat. Le principal avantage de cette option est de pouvoir prendre en compte les spécificités de l'architecture dans le placement/ordonnancement ainsi que les temps de calcul des acteurs (mesurés ou estimés). L'analyse d'un réseau d'acteurs pour en extraire des parties statiques [BSL+08, GJRB09] constitue là encore également une première étape pour rapprocher le modèle CAL du logiciel Preesm. Si nous pouvons étendre les modèles pris en compte dans le placement/ordonnancement de manière à traiter des modèles de programmation complexes, il sera alors possible d'envisager de prendre une description d'entrée dans Preesm sous un format UML MARTE. Une autre solution serait de créer un profil UML spécifique au modèle utilisé dans Preesm.

Toujours dans les perspectives de l'outil Preesm, il est possible de faire un lien entre OpenMP ou OpenCL et le graphe d'implantation que nous utilisons. Dans cette optique, ces langages pourraient être utilisés dans une phase de génération automatique de code. A l'heure actuelle, les performances pouvant être atteintes avec ces langages sur des applications réelles doivent être évaluées avant d'envisager une telle automatisation.

D'autre part, ce document a montré comment nous pouvons mettre en œuvre nos compétences en prototypage rapide dans le contexte de la normalisation MPEG RVC. Notre première contribution est de fournir automatiquement, à partir des descriptions standard, un code en C/SystemC utilisable sur plateforme monoprocesseur. Il n'y a pas d'étape de placement puisqu'il n'y a qu'un processeur, et l'ordonnancement des acteurs est laissé à la charge de SystemC (ordonnancement dynamique monoprocesseur). Le processus devra être optimisé en analysant le graphe avant compilation pour créer un placement sur plusieurs unités de calcul (logicielles et/ou matérielles) et optimiser la génération de code pour plus de performances, à savoir : rendre statique une partie de l'ordonnancement, remplacer SystemC par un ordonnanceur optimisé. L'objectif alors est de conduire l'analyse d'acteurs et de réseaux d'acteurs pour déterminer lesquels d'entre eux peuvent être ordonnancés statiquement ou transformés pour isoler des régions statiques d'un réseau. Des techniques dédiées à l'analyse d'acteurs et de réseaux d'acteurs CAL pour en déterminer des propriétés statiques sont présentées dans [EJ03c, PE08]. Un certains nombre de travaux en cours sont réalisés dans ce sens. On peut citer entre autres les travaux de Ruirui Gu [GJRB09] à l'université du Maryland sous la direction de Shuvra Bhattacharyya et par Jani Boutellier à l'École Polytechnique Fédérale de Lausanne sous la direction de Marco Mattavelli [BSL+08]. Encore une fois, on peut également penser à l'utilisation des primitives OpenMP ou OpenCL. Le dynamisme lors de l'exécution permettrait alors d'optimiser l'exécution du décodeur. Nous avons déjà beaucoup discuté des avantages et des inconvénients de cette approche dans le chapitre 2. L'évolution des solutions proposées dans ce domaine est rapide et pourra apporter des résultats intéressants.

A plus long terme, il s'agira de pouvoir présenter une solution de reconfiguration des décodeurs selon le standard RVC. On peut dire à l'heure actuelle que notre solution constitue une première version de configuration initiale dans un contexte logiciel. Il faudra adapter le processus pour prendre en charge des instructions de reconfiguration au format BSDL, mais également en visant des plateformes hétérogènes. Nos approches étant parties du domaine multiprocesseur, leur extension pour le domaine FPGA devra être poursuivie avec éventuellement un couplage avec d'autres approches utilisant le profil UML MARTE [AKL+09]. Il faudra notamment prendre en charge la reconfiguration dynamique partielle pour l'instanciation optimisée des décodeurs RVC [DPML07].

Nous pouvons constater que nos travaux se trouvent de plus en plus en phase avec les préoccupations industrielles pour la conception de systèmes embarqués. Aucune solution proposée actuellement (commerciale ou académique) n'est encore suffisamment mature pour être utilisée à grande échelle. D'autre part, les concepteurs doivent faire face à de nombreuses initiatives pour accélérer les cycles de développement. Ils doivent les évaluer en même temps qu'ils doivent concevoir leurs systèmes. Même s'il nous reste des défis pour aboutir à une solution automatique satisfaisante, il apparaît alors que l'expertise acquise sur les modélisations d'applications, la programmation de systèmes hétérogènes et sur le placement/ordonnancement peuvent être bénéfiques dès aujourd'hui pour des industriels. Ce constat nous a amené au projet de création de la société Modaë Technologies. Cette société sera portée par deux ex-ingénieurs de Thomson Silicon Components en collaboration avec notre équipe. Cette société est actuellement dans une phase de maturation grâce au financement de la région Bretagne, pour un an à partir du mois d'octobre 2009. Le but de cette démarche, soutenue par Rennes Atalante, est de créer, courant 2010, une société d'aide à la conception de systèmes embarqués multiprocesseurs dans les domaines de la compression vidéo et du traitement d'image. L'originalité du projet réside dans une démarche systématique de modélisation amont des applications, incluant les notions de parallélisme dès les premières phases de conception. Pour l'IETR, Modaë pourra être une structure industrielle de valorisation et de diffusion. Les résultats obtenus au laboratoire pourront être utilisés par l'entreprise et ainsi être mis à la portée de leurs clients sans phase de formation spécifique. En contrepartie, les projets concrets traités dans l'entreprise devront apporter des retours d'expérience utilisables pour la définition de nouveaux thèmes de recherche.

Dans tous les cas, nous gardons pour objectif la modélisation des applications complexes selon les modèles flux de données les plus adaptés à l'aide de SynDEx V6, de Preesm ou des outils liés à RVC-CAL comme openDF, CAL2C et CAL2HDL. La maîtrise de l'ensemble du processus de prototypage, de la modélisation des applications aux optimisations matérielles sur des composants récents, en passant par les algorithmes de placement/ordonnancement, nous permet d'envisager ces nombreuses perspectives.

# Chapitre 8

# Bibliographie

## 8.1 Revues Internationales avec comité de lecture


[ALM+09] Ihab Amer, Christophe Lucarz, Marco Mattavelli, Ghislain Roquier, Olivier Déforges, and Jean-François Nezan. Reconfigurable video coding : an overview of its main objectives. *IEEE Signal processing Magazine*, (Special Issue on Signal Processing on Platforms with Multiple Cores : Part 1 - Overview and Methodology), 2009.

[FDN02] Virginie Fresse, Olivier Déforges, and Jean Francois Nezan. AVSynDEx : a rapid prototyping process dedicated to the implementation of digital image processing applications on Multi-DSP and FPGA architectures. *EURASIP Journal on Applied Signal Processing*, 2002(Issue 9) :pp 990–1002, May 2002.

[NDR04] Jean François Nezan, Olivier Déforges, and Mickaël Raulet. Fast prototyping methodology for distributed and heterogeneous architectures : application to mpeg-4 video tools. *Springer Design Automation for Embedded Systems Journal*, 9 :141–154, 2004.

[PPW+09] Maxime Pelcat, Jonathan Piat, Matthieu Wipliez, Slaheddine Aridhi, and Jean-François Nezan. An open source rapid prototyping framework for signal processing applications. *Eurasip Design and Architectures for Signal and Image Processing*, (Special issue on Design and Architectures for Signal and Image Processing), 2009.

[RKNS06] Markus Rupp, Thomas Kaiser, Jean François Nezan, and Gerhard Schmidt. Signal processing with high complexity : Prototyping and industrial design. *EURASIP Journal on Embedded Systems*, 2006 :2 pages, 2006. Article ID 90363.

[RUN+06] Mickaël Raulet, Fabrice Urban, Jean François Nezan, Christophe Moy, Olivier Déforges, and Yves Sorel. Rapid prototyping for heterogeneous multicomponent systems : An MPEG-4 stream over a UMTS communication link. *EURASIP Journal on Advances in Signal Processing*, 2006 :13 pages, 2006. Article ID 64369.

[UNR09] Fabrice Urban, Jean François Nezan, and Mickaël Raulet. Hds, a real-time multi-dsp motion estimator for mpeg-4 h.264 avc high definition video encoding. *Springer Journal of Real-Time Image Processing*, 4(1) :23–31, 2009.

[UPND08] Fabrice Urban, Ronan Poullaouec, Jean François Nezan, and Olivier Déforges. A flexible heterogeneous Hardware/Software solution for Real-Time HD h.264 motion estimation. *IEEE Transactions on Circuits and Systems for Video Technology (TCSVT)*, 18(12) :1781–1785, 2008.

[WRN09] Matthieu Wipliez, Ghislain Roquier, and Jean-François Nezan. Software code generation for the rvc-cal language. *Springer Journal of signal processing systems*, (Special Issue on Reconfigurable Video Coding : technologies, tools and methodologies for video codec reconfiguration), 2009.

## 8.2 Conférences Internationales avec comité de lecture

[MARN07] Sylvain Moreau, Slaheddine Aridhi, Mickael Raulet, and Jean François Nezan. On modeling the RapidIO communication link using the AAA methodology. In Eurasip, editor, *Workshop on Design and Architectures for Signal and Image Processing (DASIP'07)*, Grenoble France, 2007.

[MNRC09] Pengcheng Mu, Jean-François Nezan, Mickael Raulet, and Jean-Gabriel Cousin. A list scheduling heuristic with new node priorities and critical child technique for task scheduling with communication contention. In Eurasip, editor, *Workshop on Design and Architectures for Signal and Image Processing (DASIP'09)*, Nice France, 2009.

[MRN+02] Yann Le Mener, Mickael Raulet, Jean François Nezan, Apostolos Kountouris, and Christophe Moy. SynDEx executive kernel development for DSPs TI c6x applied to real-time and embedded multiprocessors architectures. In Eurasip, editor, *European Signal Processing Conference (Eusipco'02)*, 2002.

[MRNC07] Pengcheng Mu, Mickaël Raulet, Jean François Nezan, and Jean Gabriel Cousin. Automatic code generation for Multi-Microblaze system with SynDEx. In *European Signal Processing Conference (Eusipco'07)*, pages 1644–1648, Poznan ; Poland, 2007. ISBN : 978–83-921340-2-2.

[MURN07] Alain Maccari, Fabrice Urban, Mickael Raulet, and Jean François Nezan. Automatic code generation for interconnected distributed RAM in the AAA methodology : H264 motion estimation case study. In Eurasip, editor, *Workshop on Design and Architectures for Signal and Image Processing (DASIP'07)*, Grenoble France, 2007.

[NDR02] Jean François Nezan, Olivier Déforges, and Mickael Raulet. Rapid prototyping methodology for multi-DSP TI C6X platforms applied to an mpeg-2 coding application. In ACM Federated Computing Research Conference (FCRC), editor, *Symposium on Parallel Algorithms and Architectures (SPAA'02)*, Avril 2002.

[NDUP06] Jean François Nezan, Olivier Déforges, Fabrice Urban, and Ronan Poullaouec. Real-time multi-DSP motion estimator for MPEG-4 AVC/H.264 high definition video encoding. In IEEE, editor, *International Conference on Signals ans Electronics Systems (ICSES'06)*, pages 305–308, 2006.

[NFD01] Jean François Nezan, Virginie Fresse, and Olivier Déforges. Fast prototyping of parallel architectures : An mpeg-2 coding application. In *International Conference on Imaging Science, Systems, and Technology (CISST'01)*, Las Vegas USA, may 2001.

[NFDR02] Jean François Nezan, Virginie Fresse, Olivier Déforges, and Mickael Raulet. AV-SynDEx methodology for fast prototyping of Multi-C6x DSP architectures. pages 990–1002, 2002.

[NRD02] Jean François Nezan, Mickael Raulet, and Olivier Déforges. Integration of mpeg-4 video tools onto Multi-DSP architectures using AVSynDEx fast prototyping methodology. In *IEEE Workshop on Signal Processing Systems (SIPS'02)*, pages 207–212, United States, Octobre 2002.

[PAN08] Maxime Pelcat, Slaheddine Aridhi, and Jean François Nezan. Optimization of automatically generated multi-core code for the LTE RACH-PD algorithm. In *Workshop on Design and Architectures for Signal and Image Processing (DASIP'08)*, Bruxelles Belgium, 2008.

[PMAN09a] Maxime Pelcat, Pierrick Menuet, Slaheddine Aridhi, and Jean-François Nezan. Scalable compile-time scheduler for multi-core architectures. In IEEE, editor, *Proceedings of Design, Automation and Test in Europe (DATE'09)*, Nice France, 2009. IEEE.

[PMAN09b] Maxime Pelcat, Pierrick Menuet, Slaheddine Aridhi, and Jean-François Nezan. A static scheduling framework for deploying applications on multicore architectures. In IASTED, editor, *Proceedings of Parallel and Distributed Computing and Networks (PDCN'09)*, Innsbruck, Autriche, 2009.

[PNP+09] Maxime Pelcat, Jean-François Nezan, Jonathan Piat, Jerome Croizer, and Slaheddine Aridhi. A system-level architecture model for rapid prototyping of heterogeneous multicore embedded systems. In Eurasip, editor, *Workshop on Design and Architectures for Signal and Image Processing (DASIP'09)*, Nice France, 2009.

[RBD+03] Mickael Raulet, Marie Babel, Olivier Déforges, Jean François Nezan, and Yves Sorel. Automatic Coarse-Grain partitioning and automatic code generation for heterogeneous architectures. In *IEEE Workshop on Signal Processing Systems (SIPS'03)*, pages 316–321, République de Corée, september 2003.

[RRND06] Ghislain Roquier, Mickael Raulet, Jean François Nezan, and Olivier Déforges. Using RTOS in the AAA methodology automatic executive generation. In *European Signal Processing Conference (Eusipco'06)*, Genova Italia, september 2006.

[RUN+05] Mickael Raulet, Fabrice Urban, Jean François Nezan, Christophe Moy, and Olivier Deforges. Syndex executive kernels for fast development of applications over heterogeneous architectures. In *European Signal Processing Conference (Eusipco'05)*, Antalaya Turkey, 2005.

[RWR+08] Ghislain Roquier, Matthieu Wipliez, Mickael Raulet, Jean François Nezan, and Olivier Déforges. Software synthesis of CAL actors for the MPEG reconfigurable video coding framework. In *IEEE International Conference on Image Processing (ICIP'08)*, pages 1408–1411, San-Diego USA, october 2008.

[UPND07] Fabrice Urban, Ronan Poullaouec, Jean François Nezan, and Olivier Déforges. H.264 fractional motion estimation refinement : a real-Time and low complexity hardware solution for HD sequences. In *European Signal Processing Conference (Eusipco'07)*, pages 836–840, Poznan, Poland France, 2007.

[URND06] Fabrice Urban, Mickael Raulet, Jean François Nezan, and Olivier Déforges. Automatic DSP cache memory management and fast prototyping for multiprocessor image applications. In *European Signal Processing Conference (Eusipco'06)*, page nc, Florence Italia, 2006.

[VNRD03] Nicolas Ventroux, Jean François Nezan, Mickael Raulet, and Olivier Déforges. Rapid prototyping for an optimized mpeg-4 decoder implementation over a parallel heterogeneous architecture. In IEEE Signal Processing Society (SPS'03), editor, *28th IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP '03)*, pages 433–436. IEEE Signal Processing Society, 2003.

[WRR+08] Matthieu Wipliez, Ghislain Roquier, Mickael Raulet, Jean François Nezan, and Olivier Déforges. Code generation for the MPEG reconfigurable video coding frame-

work : From CAL actions to c functions. In *IEEE International Conference on Multimedia and Expo (ICME'08)*, pages 1049–1052, Hanover Germany, june 2008.

## 8.3 Conférences Internationales en tant qu'invité

[NR06] Jean François Nezan and Ghislain Roquier. Lar image codec onto a multiprocessor architecture. In IEEE Signal Processing Society, editor, *"DSP Campus", showcase of Universities during 28th IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP '06)*, Mai 2006.

[NWR+08] Jean François Nezan, Matthieu Wipliez, Ghislain Roquier, Mickael Raulet, and Olivier Déforges. Software synthesis from the CAL language. In *Workshop on DataFlow Modeling for Embedded Systems Using the CAL Actor Language*, Pise Italy, Mai 2008.

## 8.4 Conférences Nationales avec comité de lecture

[DNR04] Olivier Déforges, Jean François Nezan, and Mickael Raulet. Développement d'un codec vidéo MPEG-4 temps-réel embarqués sur architectures distribuées. In *International Symposium on Image/Video Communications (ISIVC)*, Brest France, 2004.

[MCNR09] Pengcheng Mu, Jean-Gabriel Cousin, Jean-Francois Nezan, and Michael Raulet. Heuristique statique améliorée d'ordonnancement de tâches : impact sur le tri des tâches et sur l'allocation de processeur. In *Colloque GRETSI - Traitement du Signal et des Images*, Dijon, septembre 2009.

[NDF01] Jean François Nezan, Olivier Déforges, and Virginie Fresse. Intégration rapide de services vidéo mpeg sur architectures parallèles. In CNRS GDR ISIS, editor, *Gretsi 2001*, september 2001.

[NDR02a] Jean François Nezan, Olivier Déforges, and Mickael Raulet. Développement d'un noyau d'exécutif SynDEx pour DSP TI c6x appliqué aux architectures multiprocesseurs temps-réel et embarquées. In CNRS GdR ISIS, editor, *Journées Francophones sud l'Adéquation Algorithme Architecture (JFAAA)*, Monastir Tunisie, december 2002.

[NDR02b] Jean François Nezan, Olivier Déforges, and Mickael Raulet. Intégration d'un décodeur mpeg-4 sur architecture multi-C6x. In CNRS GdR ISIS, editor, *Journées Francophones sur l'Adéquation Algorithme Architecture (JFAAA)*, Monastir Tunisie, december 2002.

[RRND06] Ghislain Roquier, Mickael Raulet, Jean François Nezan, and Olivier Déforges. Génération automatique de code distribué à l'aide de RTOS : application au codage d'images LAR. In France Telecom R&D, editor, *COmpression et REprésentation des Signaux Audiovisuels (CORESA)*, France, october 2006.

[UNDP06] Fabrice Urban, Jean François Nezan, Olivier Déforges, and Ronan Poullaouec. Estimateur de mouvement temps réel multi-DSP pour l'encodage vidéo MPEG-4 AVC/H.264 haute définition. In France Telecom R&D, editor, *COmpression et REprésentation des Signaux Audiovisuels (CORESA)*, october 2006.

[VNRD03] Nicolas Ventroux, Jean François Nezan, Mickael Raulet, and Olivier Déforges. Prototypage rapide d'un décodeur mpeg-4 optimisé sur architectures hétérogènes parallèles. In CNRS GdR ISIS, editor, *GRETSI 2003*, november 2003.

## 8.5 Contributions à la normalisation MPEG

[NRD07] Jean-François Nezan, Mickaël Raulet, and Olivier Déforges. M14650 : Preesm in the rvc framework, July 2007.

[PBR+07] Maxime Pelcat, Médéric Blestel, Mickaël Raulet, Jean-François Nezan, and Olivier Déforges. M14463 : Evolutions of rvc so as to handle svc decoding, April 2007.

[PRD+07] Maxime Pelcat, Mickaël Raulet, Olivier Déforges, Médéric Blestel, and Jean-François Nezan. M14965 : Implementing svc from rvc avc : description of the specific svc fus, November 2007.

[RPR+07] Ghislain Roquier, Maxime Pelcat, Mickaël Raulet, Matthieu Wipliez, Jean-François Nezan, and Olivier Déforges. M14457 : A scheme for implementing mpeg-4 sp codec in the rvc framework, April 2007.

[RRW+08] Mickaël Raulet, Ghislain Roquier, Matthieu Wipliez, Jean-François Nezan, and Olivier Déforges. M15167 : Update of cal2c code generation, January 2008.

[WRN09] Matthieu Wipliez, Mickaël Raulet, and Jean-François Nezan. M16145 : Proposed changes for rvc-cal annex a of iso-iec 23001-4, February 2009.

[WRR+07] Matthieu Wipliez, Ghislain Roquier, Mickaël Raulet, Jean-François Nezan, Marco Mattavelli, and Ian Miller. M14981 : Status of cal2c code generation, November 2007.

[WRR+08] Matthieu Wipliez, Mickaël Raulet, Ghislain Roquier, Jean-François Nezan, and Olivier Déforges. M15382 : A fast simulation of rvc mpeg4 sp decoder using cal2c code generation, April 2008.

## 8.6 Thèses soutenues

[Mu09] Pengcheng Mu. *Fast Prototyping Methodology for FPGA Based MPSoC.* Option électroniques, Institut National des Sciences Appliquées de Rennes, Juillet 2009.

[Nez02] Jean François Nezan. *Intégration automatique de services Vidéo Mpeg-4 sur architectures parallèles.* Option électroniques, Institut National des Sciences Appliquées de Rennes, Novembre 2002.

[Rau06] Mickael Raulet. *Optimisations Mémoire dans la Méthodologie AAA pour Code Embarqué sur Architectures Parallèles.* Option électroniques, Institut National des Sciences Appliquées de Rennes, Mai 2006.

[Roq08] Ghislain Roquier. *Étude de modèles flux de données pour la synthèse logicielle multiprocesseur.* Option électroniques, Institut National des Sciences Appliquées de Rennes, Novembre 2008.

[Urb07] Fabrice Urban. *Implantation optimisée déstimateurs de mouvement pour la compression vidéo sur plates-formes hétérogènes multiprocesseurs.* Option électroniques, Institut National des Sciences Appliquées de Rennes, Novembre 2007.

## 8.7 Thèses en cours

[Pel10] Maxime Pelcat. *Etude du déploiement optimal dún système de communication LTE sur une architecture multiprocesseur hétérogène.* Option électroniques, Institut National des Sciences Appliquées de Rennes, Prévue en 2010.

[Pia10] Jonathan Piat. *Optimisation de Boucles dans la Méthodologie AAA pour Code Embarqué sur Architectures Parallèles*. Option électroniques, Institut National des Sciences Appliquées de Rennes, Prévue en 2010.

[Sir11] Nicolas Siret. *Etude de la technologie RVC pour le déploiement dún décodeur SVC sur une plate-forme mixte matérielle-logicielle*. Option électroniques, Institut National des Sciences Appliquées de Rennes, Prévue en 2011.

[Wip10] Matthieu Wipliez. *Stratégies dórdonnancement et génération de code multiprocesseurs dans le cadre de la méthodologie de prototypage rapide*. Option électroniques, Institut National des Sciences Appliquées de Rennes, Prévue en 2010.

## 8.8 Liste des sites Internet

[ARM] Arm. http ://www.arm.com/.

[Cof] Société cofluent. http ://www.cofluentdesign.com/index.php/en_US/home.html.

[ecl] Eclipse Open Source IDE : Available Online. http ://www.eclipse.org/downloads/.

[graa] Grammatica parser generator : Available Online. http ://grammatica.percederberg.net/.

[grab] Graphiti Editor : Available Online. http ://sourceforge.net/projects/graphiti-editor/.

[IME] Imec svc decoder. http ://www.imec.be/ScientificReport/SR2007/html/1384303.html.

[inn] Innes, Technologies for Digital Media. http ://www.innes.fr/.

[ITR] Itrs, the international technology roadmap for semiconductors. http ://www.itrs.net/.

[JSV] Jsvm software. http ://ip.hhi.de/imagecom_G1/savce/downloads/SVC−Reference−Software.htm.

[mos] The Moses project. http ://www.tik.ee.ethz.ch/ moses/.

[opea] OpenCL. http ://www.khronos.org/opencl/.

[opeb] OpenDF project. http ://opendf.sourceforge.net/.

[opec] OpenMP. http ://openmp.org/wp/.

[Pde] Pdesigner. http ://www.cin.ufpe.br/ pdesigner/drupal2/.

[pol] PolyCore Software Poly-Mapper tool. http ://www.polycoresoftware.com/products3.php.

[pre] PREESM : Available Online. http ://sourceforge.net/projects/preesm/.

[sdf] SDF4J : Available Online. http ://sourceforge.net/projects/sdf4j/.

[SVC] Open SVC Decoder. https ://sourceforge.net/projects/opensvcdecoder/.

[Sys] Osci : Open systemc initiative. http ://www.systemc.org/home/.

## 8.9 Bibliographie

[AKL+09] D. Aulagnier, A. Koudri, S. Lecomte, P. Soulard, J. Champeau, J. Vidal, G. Perrouin, and P. Leray. Soc/sopc development using mdd and marte profile. In *Model Driven Engineering for Distributed Real-time Embedded Systems*. Hermes, 2009.

[Aud91] N.C. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. In *Real-Time Systems*, 1991.

[BEH+01] U Brandes, M Eiglsperger, I Herman, M Himsolt, and MS Marshall. Graphml progress report, structural layer proposal. In P. Mutzel, M. Junger, and S. Leipert, editors, *Graph Drawing - 9th International Symposium, GD 2001 Vienna Austria,*, pages 501–512, Heidelberg, 2001. Springer Verlag.

[BJK+95] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, and Y. Zhou. Cilk : An efficient multithreaded runtime system. In *Journal of parallel and distributed computing*, volume 37, pages 207–216, 1995.

[BM01] F. Bodin and A. Monsifrot. Performance issues in automatic differentiation on superscalar processors. In *Automatic Differentiation : From Simulation to Optimization*, 2001.

[BSL+08] J. Boutellier, V. Sadhanala, C. Lucarz, P.B., and M. Mattavelli. Scheduling of dataflow models within the Reconfigurable Video Coding framework. In *IEEE Workshop on Signal Processing Systems (SiPS'08)*, pages 182–187, 2008.

[Den74] J.B. Dennis. First version of a data flow procedure language. In *Proceedings of the Colloque sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 362–376. Springer, 1974.

[Déf04] Olivier Déforges. *Codage d'images par la méthode LAR et méthodologie Adéquation Algorithme Architecture : de la définition des algorithmes de compression au prototypage rapide sur architectures parallèles hétérogènes.* Habilitation à diriger des recherches, Université de Rennes 1, Novembre 2004.

[Dij68] E.W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages : NATO Advanced Study Institute*, pages 43–112. Academic Press, 1968.

[DM89] M.L. Dertouzos and A.K. Mok. Multiprocessor online scheduling of Hard-Real-Time tasks. In *IEEE Transaction on Software Engineering*, volume 15, pages 1497–1506, 1989.

[DPML07] J.-P. Delahaye, J. Palicot, C. Moy, and P. Leray. Partial reconfiguration of FPGAs for dynamical reconfiguration of a software radio platform. In *Proceedings of IST Mobile SUMMIT 2007*, Budapest Hongrie, July 2007.

[DYLM08] D. Ding, L. Yu, C. Lucarz, and M. Mattavelli. Video decoder reconfigurations and AVS extensions in the new MPEG reconfigurable video coding framework. In *IEEE Workshop on Signal Processing Systems (SiPS'08)*, pages 164–169, 2008.

[EJ03a] J. Eker and J.W. Janneck. CAL Language Report. Technical report, ERL Technical Memo UCB/ERL M03/48, University of California at Berkeley, December 2003.

[EJ03b] J. Eker and J.W. Janneck. CAL language report specification of the CAL actor language. Technical Report UCB/ERL M03/48, EECS Department, University of California, Berkeley, 2003.

[EJ03c] J. Eker and J.W. Janneck. A structured description of dataflow actors and its application. Technical Report UCB/ERL M03/13, ERL, UC Berkeley, May 2003.

[Fre01] Virginie Fresse. *Implantation de chaînes de traitement d'image sur des architectures dédiées et mixtes.* Option électroniques, Institut National des Sciences Appliquées de Rennes, Février 2001.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J.M. Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1994.

[GJRB09] R. Gu, J.W. Janneck, M. Raulet, and S. S. Bhattacharyya. Exploiting statically schedulable regions in dataflow programs. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, pages 565–568, Taipei, Taiwan, April 2009.

[GLS99a] T. Grandpierre, C. Lavarenne, and Y. Sorel. Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In *Proceedings of 7th International Workshop on Hardware/Software Co-Design, CODES'99*, Rome, Italy, May 1999.

[GLS99b] T. Grandpierre, C. Lavarenne, and Y. Sorel. Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In *Proceedings of the Seventh International Workshop on Hardware/Software Codesign (CODES)*, pages 74–78, 1999.

[Gra00] T. Grandpierre. *Modélisation d'architectures parallèles hétérogènes pour la génération automatique d'exécutifs distribués temps réel optimisés.* PhD thesis, Université de Paris Sud, Spécialité électronique, 2000.

[GS03a] T. Grandpierre and Y. Sorel. From algorithm and architecture specifications to automatic generation of distributed real-time executives : a seamless flow of graphs transformations. In *Proceedings of the first ACM and IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE'03)*, pages 123–132, 2003.

[GS03b] T. Grandpierre and Y. Sorel. From algorithm and architecture specifications to automatic generation of distributed real-time executives : a seamless flow of graphs transformations. In *First ACM and IEEE International Conference on Formal Methods and Models for Co-Design, Mont Saint-Michel, France*, Juin 2003.

[HCAL89] J.-J. Hwang, Y.-C. Chow, F.D. Anger, and C.-Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times. In *SIAM J. Comput.*, volume 18, pages 244–257, Philadelphia, PA, USA, 1989. Society for Industrial and Applied Mathematics.

[HKB05] C.-J. Hsu, M.-Y. Ko, and S.S. Bhattacharyya. Software synthesis from the dataflow interchange format. In *Proceedings of the 2005 workshop on Software and compilers for embedded systems (SCOPES'05)*, pages 37–49, New York, NY, USA, 2005. ACM.

[HKK+04] C.-J. Hsu, F. Keceli, M.-Y. Ko, S. Shahparnia, and S.S. Bhattacharyya. Dif : An interchange format for dataflow-based design tools. 2004.

[Int] International Standard ISO/IEC FDIS 23001-5. *MPEG systems technologies - Part 5 : Bitstream Syntax Description Language (BSDL).*

[ISO09a] ISO/IEC FDIS 23001-4. *MPEG systems technologies – Part 4 : Codec Configuration Representation*, 2009.

[ISO09b] ISO/IEC FDIS 23002-4. *MPEG video technologies – Part 4 : Video tool library*, 2009.

[ITR07] ITRS. Design. Technical report, International Technology Roadmap For Semiconductors, 2007.

[Jan07] J.W. Janneck. NL - a Network Language. Technical report, ASTG Technical Memo, Programmable Solutions Group, Xilinx Inc., July 2007.

[JMP+08] J.W. Janneck, I.D. Miller, D.B. Parlour, G. Roquier, M. Wipliez, and M. Raulet. Synthesizing hardware from dataflow programs : An MPEG-4 simple profile decoder case study. In *IEEE Workshop on Signal Processing Systems (SiPS'08)*, pages 287–292, 2008.

[JP86] M. Joseph and P. Pandya. Finding response times in a Real-Time system. In *The Computer Journal*, volume 29, pages 390–395, May 1986.

[KA95] Y.-K. Kwok and I. Ahmad. Bubble scheduling : A quasi dynamic algorithm for static allocation of tasks to parallel architectures. In *SPDP '95 : Proceedings of the 7th IEEE Symposium on Parallel and Distributeed Processing*, page 36, Washington, DC, USA, 1995. IEEE Computer Society.

[KA96] Y.-K. Kwok and I. Ahmad. Dynamic critical-path scheduling : An effective technique for allocating task graphs onto multiprocessors. In *IEEE Transactions on Parallel and Distributed Systems*, volume 7, pages 506–521, May 1996.

[KA99] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. In *ACM Computing Survey*, volume 31, pages 406–471, New York, NY, USA, 1999. ACM.

[Kao04] L. Kaouane. *Formalisation et optimisation d'applications s'exécutant sur architecture reconfigurable*. PhD thesis, Université de Marne-La-Vallée, Spécialité Informatique, 2004.

[Ker09] O. Kermia. *Ordonnancement temps réel multiprocesseur de tâches non préemptives avec contraintes de précédence, de périodicité stricte et de latence*. PhD thesis, Université de Paris Sud, Spécialité Physique, 19/03/2009.

[KK04] F. Kelly and A. Kokaram. Fast image interpolation for motion estimation using graphics hardware. In *IST/SPIE Electronic Imaging - Real-Time Imaging VIII*, San Jose, California, USA, January 2004.

[KL02] F. Kordon and Luqi. An introduction to rapid system prototyping. In *IEEE Transaction on Software Engeneering*, volume 28, pages 817–821, 2002.

[KM66] R.M. Karp and R.E. Miller. Properties of a model for parallel computations : Determinacy, termination, queueing. In *SIAM Journal on Applied Mathematics*, volume 14, pages 1390–1411. SIAM, 1966.

[Kwo97] Y.-K. Kwok. *High-performance algorithms of compile-time scheduling of parallel processors*. PhD thesis, Hong Kong University of Science and Technology (People's Republic of China), 1997.

[LAM09] C. Lucarz, I. Amer, and M. Mattavelli. Reconfigurable Video Coding : Concepts and Technologies. In *initially accepted in IEEE International Conference on Image Processing, Special Session on Reconfigurable Video Coding*, Cairo, Egypt, 2009.

[Lee01] E.A. Lee. Overview of the ptolemy project. In *Technical memorandum UCB/ERL M01/11, University of California at Berkeley*, 2001.

[Lee06] E.A. Lee. The problem with threads. Technical report, Technical Report No. UCB/EECS-2006-1, January 2006.

[LH93] B. Lee and A.R. Hurson. Issues in dataflow computing. In *Advances in Computers*, volume 37, pages 285–333, 1993.

[LL73] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a Hard-Real-Time environment. In *ACM Journal*, volume 20, pages 46–61, 1973.

[LM87a] E.A. Lee and D.G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. In *IEEE Transactions on Computers*, volume 36, pages 24–35, Washington, DC, USA, 1987. IEEE Computer Society.

[LM87b] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. volume 75, pages 1235–1245, 1987.

[LP95] E.A. Lee and T.M. Parks. Dataflow Process Networks. In *Proceedings of the IEEE*, volume 83, pages 773–801, May 1995.

[LS97] C. Lavarenne and Y. Sorel. Modèle unifié pour la conception conjointe logiciel-matériel. In *Traitement du Signal*, volume 14, pages 569 – 578, 1997.

[LW82] J.Y.-T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks*1. In *Performance Evaluation*, volume 2, pages 237–250, December 1982.

[Mud01] T. Mudge. Power : a first-class architectural design constraint. In *IEEE Computer Magazine*, volume 34, pages 58–52, 2001.

[Mur89] T. Murata. Petri nets : Properties, analysis and applications. In *Proceedings of the IEEE*, volume 77, pages 541–580, 1989.

[NVi06] NVidia. Nvidia geforce 8800 architecture technical brief. Technical report, NVIDIA Corporation, November 2006.

[OLG+05] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A.E. Lefohn, and T.J. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, aug 2005.

[PaMR09] J. Piat and S.S. Bhattacharyya ans M. Raulet. Interface-based hierarchy for synchronous data-flow graphs. In *IEEE Workshop on Signal Processing Systems (SiPS'09)*, 2009.

[PBL95] J.L. Pino, S.S. Bhattacharyya, and E.A. Lee. A hierarchical multiprocessor scheduling framework for synchronous dataflow graphs. In *Laboratory, University of California at Berkeley*, 1995.

[PE08] C. Von Platen and J. Eker. Efficient Realization of a CAL video decoder on a mobile terminal (position paper). In *IEEE Workshop on Signal Processing Systems (SiPS'08)*, 2008.

[PHLB95] J.L. Pino, S. Ha, E.A. Lee, and J.T. Buck. Software synthesis for DSP using Ptolemy. In *Journal of VLSI Signal Processing Systems*, volume 9, pages 7–21, Hingham, MA, USA, 1995. Kluwer Academic Publishers.

[RPLM08] M. Raulet, J. Piat, C. Lucarz, and M. Mattavelli. Validation of bitstream syntax and synthesis of parsers in the MPEG Reconfigurable Video Coding framework. In *IEEE Workshop on Signal Processing Systems (SiPS'08)*, pages 293–298, 2008.

[RWR+08] G. Roquier, M. Wipliez, M. Raulet, J.W. Janneck, I.D. Miller, and D.B. Parlour. Automatic software synthesis of dataflow program : An MPEG-4 simple profile decoder case study. In *IEEE Workshop on Signal Processing Systems (SiPS'08)*, pages 281–286, 2008.

[Sar89] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* MIT Press, Cambridge, MA, USA, 1989.

[Sin07] O. Sinnen. *Task Scheduling for Parallel Systems (Wiley Series on Parallel and Distributed Computing).* Wiley-Interscience, 2007.

[SL93] G.C. Sih and E.A. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. In *IEEE Transactions on Parallel and Distributed Systems*, volume 4, pages 175–187, Feb. 1993.

[spi08] The SPIRIT consortium, IP-XACT v1.4 : A specification for XML meta-data and tool interfaces, March 2008.

[SRL90] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols : an approach to real-time synchronization. In *IEEE Transactions on Computers*, volume 39, pages 1175–1185, 1990.

[SS05] O. Sinnen and L.A. Sousa. Communication contention in task scheduling. In *IEEE Transactions on Parallel and Distributed Systems*, volume 16, pages 503–515, 2005.

[Stu07] S. Stuijk. *Predictable Mapping of Streaming Applications on Multiprocessors*. PhD thesis, Technische Universiteit Eindhoven, 2007.

[Tex6a] Texas Instruments. TMS320C6000 DSP Cache User's guide, spru656a.

[TGDT07] F. Thomas, S. Gérard, J. Delatour, and Francois Terrier. Software real-time resource modeling. In *International Conference Forum on Specification and Design Languages (FDL'07)*, 2007.

[The07] B.D. Theelen. A performance analysis tool for Scenario-Aware streaming applications. In *Quantitative Evaluation of Systems, 2007. QEST 2007. Fourth International Conference on the*, pages 269–270, 2007.

[TM93] M. Tomasevic and V. Milutinovic. A survey of hardware solutions for maintenance of cache coherence in shared memory multiprocessors. In *Proceeding of the Twenty-Sixth Hawaii International Conference on System Sciences*, volume 1, pages 863–872, August 1993.

[Vic99] A. Vicard. *Formalisation et optimisation des systèmes informatiques distribués temps réel embarqués*. PhD thesis, Université de Paris Nord, Spécialité informatique, 1999.

[WG90] M.-Y. Wu and D. Gajski. Hypertool : A programming aid for message-passing systems. In *IEEE Transactions on Parallel and Distributed Systems*, volume 1, pages 330–343, 1990.

[YG94] T. Yang and A. Gerasoulis. Dsc : scheduling parallel tasks on an unbounded number of processors. In *IEEE Transactions on Parallel and Distributed Systems*, volume 5, pages 951–967, Sept. 1994.