Optimisations Mémoire dans la méthodologie " Adéquation Algorithme Architecture " pour Code Embarqué sur Architectures Parallèles

Mickaël Raulet

N° d'ordre : D 06 - 05

Thèse

présentée devant
l'INSTITUT NATIONAL DES SCIENCES APPLIQUÉES DE RENNES

pour obtenir le titre de

Docteur

spécialité : *Electronique et Traitement du Signal*

Optimisations Mémoire dans la Méthodologie AAA pour Code Embarqué sur Architectures Parallèles

par
Mickaël RAULET

Soutenue le 18 mai 2006 devant la commission d'Examen

Composition du jury

Président du jury	
Olivier SENTIEYS	Professeur à l'ENSSAT de Lannion
Rapporteurs	
Michel PAINDAVOINE	Professeur à l'Université de Bourgogne
Jean-Luc PHILIPPE	Professeur à l'Université de Bretagne Sud
Examinateurs	
Alexis BISIAUX	Ingénieur de Recherche Mitsubishi ITE
Olivier DEFORGES	Directeur de la thèse, Professeur à l'INSA de Rennes, IETR
Yves SOREL	Co-directeur de la thèse, Directeur de Recherche INRIA
Invités	
Christophe MOY	Maître Associé à Supélec
Joseph RONSIN	Professeur à l'INSA de Rennes

IETR UMR CNRS 6164 - Groupe Image et Télédection
Mitsubishi Electric ITE - Equipe Sofware Radio

Table des matières

Remerciements

Mes premiers remerciements s'adressent tout naturellement aux personnes qui ont accepté d'examiner ce travail : Jean-Luc PHILIPPE et Michel PAINDAVOINE en qualité de rapporteurs, Alexis BISIAUX, Christophe MOY et Olivier SENTIEYS en tant que membres du jury.

Je tiens également à exprimer toute ma gratitude à Joseph RONSIN et à Olivier DEFORGES, pour m'avoir accueilli au sein de leur équipe dans le laboratoire de l'IETR. De plus, j'exprime toute ma reconnaissance à Olivier DEFORGES pour son encadrement. Je remercie également toute l'équipe "permanente" de *Software Radio (radio logicielle)* de MITSUBISHI ITE : Christophe MOY, Apostolos KOUNTOURIS, et Alexis BISIAUX ainsi que les prestataires que j'ai pu rencontrer dans cette entreprise et avec qui j'ai pu confronter différents avis : Pascal LECORRE, Philippe DELAGRANGE. Je remercie également Yves SOREL de l'INRIA Rocquencourt, pour m'avoir permis de collaborer avec l'INRIA dans un outil comme SynDEx.

Un grand merci également à Eric LAVILLONIERE de MITSUBISHI, Jean-Luc FOUQUET du LIFO à l'université d'Orléans, et Jean-François NEZAN de l'IETR, pour m'avoir aidé dans différentes parties de ma thèse : théorie des graphes, algorithmes de traitement d'images.

Ce mémoire ne prend toute sa valeur que grâce aux étudiants que j'ai pu encadrer et avec lesquels j'ai eu le plaisir de travailler. Un grand merci donc à Fabrice URBAN, Gurvan LE QUELLENEC, Alexis BRUMAUD, Rémi OLLIVIER et David JACQUINET, merci également aux thésards comme Marie BABEL-FOUQUET qui a commencé sa thèse en même temps que moi.

Un remerciement tout particulier pour les personnes qui ont eu le grand courage de relire (même partiellement) ce manuscrit (Olivier DEFORGES, Jean-François NEZAN, Christophe MOY, Joseph RONSIN, Yves SOREL).

Enfin mes derniers remerciements, qui n'en sont pas pour autant les moins importants, vont à ma famille...

Introduction générale

Les téléphones mobiles offrent aujourd'hui des prestations de services qui les assimilent de plus en plus à un mini-ordinateur. Toutefois, le caractère embarqué de tels systèmes, limitant notamment les ressources matérielles (unité de calcul, mémoire, consommation), implique que l'intégration de nouvelles applications reste une tâche très complexe. La bande passante limitée des communications, même pour les téléphones dits de *troisième* et *quatrième* générations, rend l'intégration de services de plus en plus difficile, notamment la vidéo embarquée.

Les principales difficultés soulevées dans le développement de systèmes sont liées à l'hétérogénéité à la fois des applications et des cibles matérielles. L'ensemble des traitements à effectuer sur un téléphone mobile peut être décomposé en différentes couches fonctionnelles correspondant soit à des couches de service multimédia de haut niveau (codage de la voix, des images ou des vidéos, accès Internet...) soit à des couches de transport (communication numérique) de plus bas niveau. La difficulté de développement est encore augmentée par la mise en réseau de tels systèmes : l'application doit alors être validée en tenant compte de son caractère distribué.

Le problème posé par l'implantation consiste à trouver la meilleure répartition des traitements à effectuer en tenant compte de la cible. Dans le domaine de la téléphonie, le nombre de cibles et d'applications s'avère important. Il faut ainsi pouvoir projeter une application donnée sur un ensemble de cibles, ou inversement un ensemble d'applications sur une cible donnée.

Dans le cadre de la *radio logicielle*, la reconfiguration partielle ou totale du téléphone portable doit être possible, supposant une mise en adéquation dynamique du système. Le processus d'implantation doit pouvoir dès lors s'effectuer de manière automatique et autonome.

Le prototypage rapide se réfère à des méthodologies permettant de passer d'une description d'entrée de l'application, à une implantation. Il vise à atteindre trois objectifs principaux.

1. Réduction du temps de cycle de développement : par une approche systématique, voire un processus automatique, les différentes phases du développement devront être fortement accélérées.
2. Sécurisation du développement : un processus automatique et sûr, au sens validité des résultats produits, exclura l'intrusion des erreurs d'une démarche manuelle et moins bien construite.
3. Exploration de différentes solutions d'implantation : l'espace d'exploration des solutions sera d'autant plus large que la description d'entrée sera fournie à un haut niveau d'abstraction.

La méthode de prototypage est d'autant plus bénéfique qu'elle permet une description de l'implantation à un haut niveau d'abstraction, tout en aboutissant à une solution optimisée de la façon la plus automatique possible.

Les téléphones mobiles dits de troisième génération, rendent possible la transmission de l'image à son interlocuteur. Toutefois, la bande passante limitée pour la transmission oblige à compresser très fortement les vidéos, d'où des qualités de service médiocres. De plus, la solution matérielle actuelle consiste à utiliser un composant dédié (ASIC) pour effectuer les opérations de codage/décodage d'images, proscrivant ainsi toute modification ou remise à jour de cette partie du système.

L'objectif principal de nos travaux est de trouver une chaîne de prototypage adaptée aux traitements des images et aux télécommunications dans le domaine embarqué. L'équipe Image du laboratoire IETR de Rennes travaille depuis plusieurs années sur des méthodologies de prototypage permettant un passage quasi-automatique de la description fonctionnelle d'une application de traitement d'images à son implantation sur une architecture multi-DSP pouvant comporter des FPGA. L'équipe radio logicielle de MITSUBISHI ITE explore quant à elle différentes voies permettant d'automatiser (ou du moins de simplifier) la conception de systèmes complexes et reconfigurables tels les systèmes de radio logicielle. L'objectif pour les deux entités (INSA et Mitsubishi) vise à définir un processus de portage autour de SynDEx pour des applications complexes hétérogènes embarquées.

Dans la première partie de ce mémoire, les travaux présentés cherchent à intégrer des critères d'optimisation mémoire dans la méthodologie AAA, pour le partitionnement des tâches et la génération automatique du code distribué. Plus spécifiquement, dans nos travaux, nous cherchons à obtenir une génération de code automatique sur architectures distribuées optimisée sur le plan de la mémoire allouée. Cela concerne aussi bien la minimisation de l'espace mémoire associé aux données lors de la génération de code, que l'expertise et la vérification des performances sur des implantations effectives. Nous montrons également une chaîne de prototypage construite autour de la méthodologie AAA permettant d'effectuer efficacement l'implantation automatique d'une application, et réduisant ainsi le gouffre entre les concepteurs d'un algorithme et ceux de l'architecture.

Dans la seconde partie du mémoire, le traitement du signal constitue le cadre applicatif général de la recherche, plus particulièrement celui de la "*radio logicielle*" pour les activités de MITSUBISHI ITE (UMTS, FM) et celui du traitement de la vidéo pour les activités du laboratoire IETR, équipe Image (MPEG4, LAR). Nous illustrons dans cette partie le prototypage rapide de nos applications optimisées selon le critère mémoire.

Le mémoire se termine par des conclusions globales sur les travaux effectués, en dégageant les directions futures de recherche.

Première partie

Optimisation mémoire dans la méthodologie AAA

Chapitre 1

Introduction/ Etat de l'art

1.1 Introduction/contexte de l'étude

Un des axes de recherche de l'équipe Image du laboratoire IETR de Rennes concerne la mise au point de méthodologies de prototypage. Il s'agit d'obtenir un passage quasi-automatique d'une description fonctionnelle d'une application de traitement d'images à son implantation sur une architecture multi-composants. Un des éléments majeurs du processus de portage est l'outil de génération de code distribué appelé SynDEx, développé à l'INRIA de Rocquencourt. Ce dernier, basé sur la méthodologie AAA (Adéquation Algorithme Architecture), a pour objectif de trouver pour le graphe logiciel représentant l'application la meilleure projection sur le graphe matériel modélisant la plate-forme cible. Les travaux antérieurs du laboratoire IETR groupe Image se sont essentiellement situés en aval de SynDEx avec la définition d'une couche d'entrée de description et de simulation fonctionnelle. Plus récemment, en collaboration avec MITSUBISHI ITE, la contribution s'est aussi située en amont avec le développement d'un noyau temps-réel SynDEx pour des architectures de type multi-C6x (composées de DSP TMS320C6201 de *TEXAS INSTRUMENTS*).

Les travaux de l'équipe radio logicielle de MITSUBISHI ITE portent sur l'élaboration de méthodes de conception évoluées pour des systèmes complexes et reconfigurables tels les systèmes de radio logicielle. Il faut comprendre par complexe l'association de composants permettant de faire :

- du calcul intensif programmable sur des cibles de plusieurs natures (DSP, GPP, FPGA, ASIC paramétrable)
- des communications de différents types (FIFO, passage de messages, mémoire partagée, bus)
- des traitements de natures différentes : traitement flot de données, contrôle rapide lié au traitement du signal, contrôle de plus haut niveau pour la gestion et la reconfiguration du système

Reconfigurable signifie ici qu'il est nécessaire de se doter d'une architecture logicielle adéquate afin de tirer parti de la reconfigurabilité potentielle des composants programmables (au moins paramétrables).

L'objectif à plus long terme pour les deux entités (INSA et Mitsubishi) vise à définir un processus de portage autour de SynDEx pour des applications complexes hétérogènes embarquées comportant un fort conditionnement et/ou de la reconfiguration.

Cette problématique se retrouve communément dans les domaines des télécommunications mobiles concernant la "radio logicielle", ainsi que dans des standards multimédia type MPEG4.

La thèse s'est déroulée en partenariat entre MITSUBISHI ITE et le laboratoire IETR, et a fait également l'objet d'une étroite collaboration avec l'INRIA de Rocquencourt et Y.SOREL. L'objectif essentiel de ces travaux est de s'intéresser à la minimisation de la mémoire allouée dans une chaîne globale de prototypage.

La bonne utilisation de la mémoire associée à un processeur est un élément essentiel d'un point de vue performance d'un système. Ainsi un DSP dispose généralement d'une mémoire interne, caractérisée par des temps d'accès très courts mais d'une capacité limitée, et d'une ou plusieurs mémoires externes de plus grande capacité mais à temps d'accès important.

L'optimisation du code d'un point de vue mémoire a donc un double objectif :

- minimisation de la taille du code programme et des données pour une implantation au maximum en mémoire interne,
- partitionnement du code programme et des données entre mémoire interne et si nécessaire externe pour un temps d'exécution minimal.

Les travaux à réaliser visent à intégrer ces critères d'optimisation sous SynDEx, pour le partitionnement des tâches et la génération automatique du code distribué. Cela concerne aussi bien :

- la minimisation de l'espace mémoire associé aux données lors de la génération de code,
- l'expertise et la vérification des performances sur des implantations effectives.

Le traitement du signal constitue le cadre applicatif général de la recherche, plus particulièrement la "*radio logicielle*" (activités MITSUBISHI ITE) et le traitement de la vidéo (activités laboratoire IETR, équipe image).

Le problème général du prototypage rapide sur cible nécessite la connaissance de différents domaines d'expertise :

- les méthodes et outils associés de conception avancée,
- les architectures de calculateurs numériques (embarqués ou non),
- l'aspect algorithmique des applications considérées.

Nous présentons ainsi dans le reste de ce chapitre, un rapide état de l'art portant sur les deux premiers points précédents, à savoir les outils de prototypage niveau système, suivi des architectures dans le domaine embarqué.

1.2 Prototypage - Les outils niveau système

Pour le prototypage d'applications, il existe un certain nombre d'outils niveau système. Toutefois ces outils présentent des caractéristiques différentes qui font que chacun d'eux est plus spécifiquement adapté à une classe d'applications : orientée flot de données, orientée flot de contrôles...

1.2.1 MCSE - Cofluent Studio

La méthodologie MCSE, pour *Méthodologie de Conception de Systèmes Electroniques* [Cal91], permet la conception complète de systèmes temps réel en suivant les différentes étapes :

- rédaction du cahier des charges pour la définition des exigences,
- élaboration des spécifications du système,
- conception fonctionnelle indépendante de la technologie,
- conceptions exécutive et architecturale,
- le prototypage et réalisation.

CoFluent Studio(1) est un outil industriel qui repose sur la méthodologie MCSE et qui supporte les trois dernières étapes.

Les deux premières phases permettent de dégager les spécifications fonctionnelles (ce que réalise le système), les spécifications opératoires (performances attendues), et les spécifications technologiques (contraintes de réalisation).

CoFluent Studio s'exécute à un niveau haut d'abstraction système comme peu d'outils de co-conception ou de conception au niveau du système peuvent le faire. *CoFluent Studio* permet aux concepteurs de capturer des informations via des graphiques, du code C et des spécifications d'attributs. Les utilisateurs commencent par créer et simuler un modèle fonctionnel intégralement temporisé et exécutable. Ensuite, ils définissent une architecture physique, en ayant recours à une analyse des performances et une co-simulation. Enfin, ceux qui le souhaitent peuvent générer un code C pour des noyaux temps réel et un VHDL pour les portions matérielles de la conception.

Algorithmes et architectures sont conçus indépendamment dans les phases de conception fonctionnelle et exécutive. Ces conceptions se font selon un modèle unifié du type graphe flot de contrôle. La conception architecturale permet d'étudier le portage de la solution fonctionnelle sur le support d'exécution. Des exemples d'application de cette méthodologie dans le contexte du CoDesign peuvent être trouvés dans [CPH97].

Les concepteurs peuvent vérifier le comportement du modèle architectural en analysant et en co-simulant les performances. *CoFluent Studio* traduit le modèle en un programme C++ ou SystemC. Le moteur de simulation exécute directement les algorithmes des opérations au lieu d'émuler chaque instruction d'une unité centrale.

Pendant la simulation, les concepteurs peuvent visionner des informations comme le tableau d'exécution d'un système, l'évolution des variables d'un système, des indices de performances comme l'utilisation mémoire et le débit de traitement d'images. Dans la dernière étape, celle du prototypage, les utilisateurs peuvent générer un code C pour certains systèmes d'exploitation en temps réel, tels que *VxWorks* de *Wind River*. Ils peuvent également produire un code VHDL RTL.

Pour conclure, *Cofluent Studio* est un outil niveau système bien adapté pour des applications à fort contrôle pour réaliser des systèmes temps réel réactifs ou encore pour l'intégration de systèmes dans des circuits. En revanche, cet outil est moins bien adapté pour le prototypage d'applications orientées données. De plus, l'outil ne permet pas de distribuer automatiquement une application sur une architecture multi-processeurs.

(1) http ://www.cofluentdesign.com/

1.2.2 CODEF

CODEF est un outil d'exploration d'architectures de systèmes sur puce réalisé en liaison avec *Philips Semiconductors* et le laboratoire I3S à Sophia Antipolis [ABCG00]. A partir d'une spécification de type flots de données conditionnés, l'outil CODEF est capable d'effectuer la sélection des IPs logiciels et/ou matériels, et réalise l'allocation et l'ordonnancement des fonctionnalités de l'application dans le but de satisfaire des contraintes de temps et de minimiser la surface de silicium. Les architectures ciblées sont de type multi-processeurs hétérogènes avec des co-processeurs et des accélérateurs matériels. La méthode de partitionnement proposée opère sur un modèle mixte basé sur un graphe de flots de données conditionnels contrôlés par une ou des machines d'états finis. Une autre étude, dans CODEF, a pour objectif de produire des architectures systèmes qui vérifient les contraintes temporelles de l'application, minimisant la consommation d'énergie et respectant des contraintes de puissance. L'étape de partitionnement/ordonnancement vise à minimiser soit les temps d'exécution, soit la surface ou une combinaison de ces deux critères.

L'outil est bien adapté pour des applications comportant à la fois du contrôle et beaucoup de données. Les modèles de description de l'architecture et de l'algorithme ont une granularité fine. Ceux-ci ne peuvent pas abordés de ce fait des applications complexes. Dans l'outil, il faut également une grande connaissance et maîtrise de l'architecture sur laquelle l'algorithme est embarqué.

1.2.3 POLIS

La méthodologie *Polis*, décrite dans [BCG+97], propose un contexte unifié de développement, couvrant aussi bien les phases de spécification, de synthèse et de validation. Dans cette méthodologie, on retrouve les différentes étapes du co-design. Les spécifications sont écrites dans un langage haut niveau comme, par exemple, *ESTEREL*. Ces spécifications sont ensuite traduites sous forme de machines d'état. Une étape de vérification par l'utilisation des méthodes formelles est utilisée afin de valider à un haut niveau d'abstraction le comportement du système.

L'étape de partitionnement dépend ensuite de l'utilisateur. L'expérience de celui-ci est alors nécessaire afin de proposer des partitionnements judicieux. La co-simulation vise à valider les choix du concepteur en tenant compte aussi bien des algorithmes que des effets de l'architecture sur ceux-ci. L'outil PTOLEMY est le plus souvent utilisé.

Les machines d'état peuvent ensuite être soit traduites en langage matériel (niveau RTL), soit synthétisés en code pour système d'exploitation temps réel. Cette méthodologie est le plus souvent utilisée dans le cas d'applications de contrôle-commande [TSLJ00]. La méthodologie Polis, mise au point par des universitaires, a été proposée comme environnement de développement par la société Cadence, sous le nom VCC (*Virtual Component Codesign*). Cet environnement reprend les phases de développement préconisées par Polis : conceptions comportementale et architecturale séparées, partitionnement et simulation de performances, synthèses logicielle, matérielle, et des interfaces.

Une des étapes importantes dans le processus de co-design concerne généralement le partitionnement des algorithmes sur une architecture distribuée, qui est hétérogène.

Le problème dans la méthode proposée est de ne pas partitionner automatiquement une application.

1.2.4 Design Trotter

Dans une architecture distribuée, la qualité de l'intégration repose pour beaucoup sur celle du partitionnement. De plus pour la majorité des outils de prototypage, la définition de l'architecture est indépendante de l'application. L'approche développée au LESTER dans l'outil *Design Trotter [Mou03]* cherche à guider la construction de l'architecture en fonction de ces propriétés. L'architecture est définie itérativement en précisant progressivement les hypothèses architecturales à mesure que l'on progresse dans l'analyse du système.

Les entrées de l'environnement *Design Trotter* se présentent sous la forme de fonctions décrites en langage de haut niveau (le C pour le moment). Ces fonctions peuvent être issues soit de la décomposition en une machine d'état de l'application (comme par exemple à partir du langage *ESTEREL*), soit à partir d'un graphe de tâches. L'environnement *Design Trotter* fournit en sortie différents types d'estimations et d'informations visant à guider le concepteur de systèmes embarqués.

Cet outil permet d'optimiser les opérations de type FFT mais ne permet pas de réaliser l'estimation d'un système complet composé de FFT par exemple. Cet outil est complémentaire d'une approche de niveau système.

1.3 Domaine d'application : systèmes embarqués pour le traitement de l'information

1.3.1 Système embarqué dans le domaine multimédia

Un système embarqué est un système intégré dans un système plus large avec lequel il est interfacé, et pour lequel il réalise des fonctions particulières (contrôle, surveillance, communication). De tels systèmes possèdent souvent la caractéristique principale de fonctionner en temps réel, car ils doivent gérer des informations et en déduire des actions avec un délai maîtrisé (soit un délai connu, soit un délai borné). De plus, ils sont souvent dits "critiques", ce qui signifie qu'ils ne doivent jamais faillir (notion de sécurité de fonctionnement).

Les systèmes embarqués se démarquent des ordinateurs traditionnels (PC ou station de travail) en premier lieu par le fait qu'ils soient développés pour une fonction particulière. La seconde grande différence tient dans les contraintes supplémentaires apportées par leur environnement : limitation de l'énergie, de la puissance de calcul et de l'espace mémoire, réactivité pouvant être importante. Comme le plus souvent, il existe de plus des contraintes temporelles pour le temps de réactivité du système, on parle alors de systèmes temps réel [Sta96]. Le temps réel est dit strict ou critique si la réponse du système doit impérativement être quasi-instantanée. C'est le cas par exemple des systèmes de contrôle équipant des véhicules. En revanche, le temps réel est qualifié de souple si les contraintes temporelles sont moins fortes, et les conséquences d'un retard de réactivité moins préjudiciables (cartes bancaires, PDA, téléphones portables). Une autre caractéristique importante pour juger de la complexité de la mise en œuvre des systèmes, est bien sur liée à sa charge, autrement dit à la quantité de calcul

à effectuer. De manière approximative, les applications de type contrôle possèdent une forte réactivité à l'environnement, un temps réel strict, et une charge moyenne. Les applications plutôt de type service ou multimédia au contraire ont plutôt une réactivité moyenne ou faible, un temps réel souple mais une charge importante. L'introduction de la vidéo dans le multimédia vient changer cette donne : si la réactivité du système reste limitée, et que le temps de réponse (latence) n'est pas forcément critique, en revanche les images doivent être traitées à un débit moyen (cadence) qui doit être dans l'absolu de 25 images par seconde pour du temps réel au sens cadence vidéo du terme. C'est pour cette raison que le décodage, et surtout le codage de la vidéo, est aujourd'hui l'un des enjeux majeurs du domaine de l'embarqué.

Un système se définit comme un ensemble de fonctions coopérantes, échangeant en général des informations. Dès lors, même l'implantation d'un système dans un seul processeur peut s'avérer complexe puisqu'il s'agit de partager la cible d'exécution. Dans le domaine du logiciel, de telles applications sont appelées multi-tâches. Elles se caractérisent en fonction du degré de réactivité, et de l'importance des liens et des ressources à partager. Par exemple, Windows est un système d'exploitation (OS) supportant le multi-tâches dans le sens où plusieurs applications peuvent s'exécuter en pseudo parallèle. Mais ces applications ne constituent pas un système global. Elles restent en général largement indépendantes, et le rôle de l'OS est surtout de devoir partager le temps du processeur. Dans le domaine du temps réel, il existe des OS dits RTOS (*Real Time Operating Systems*), spécifiquement conçus pour l'intégration de systèmes multi-tâches [Jos01].

Un RTOS propose d'une part un ensemble de primitives de haut niveau pour la synchronisation ou l'envoi de données entre tâches, d'autre part le plus souvent un ordonnanceur comme couche logicielle de supervision, permettant de déterminer à chaque instant et en fonction d'une politique d'ordonnancement donnée, quelle tâche doit être exécutée [SG90]. Un RTOS entraîne toujours un coût à la fois en terme de place mémoire et de temps d'exécution, coût pouvant être prohibitif pour des systèmes embarqués. Il existe donc des versions allégées de RTOS, spécifiquement conçues pour ce domaine (ex. : *DSP/BIOS* de TEXAS INSTRUMENTS [Dar01], *lightweight* fondée sur le noyau LINUX pour DSP *TEXAS INSTRUMENTS* [KB05]).

1.3.2 Système embarqué dans le domaine de la radio logicielle

La radio logicielle ou *Software Radio* a été introduite par J.MITOLA [Mit95] dans les années 90 pour désigner une radio qui peut s'adapter à différentes techniques de modulation et formes d'ondes radio, ainsi qu'aux standards présents et futurs, grâce au contrôle logiciel. Dans un système *radio logicielle,* la partie matérielle se limite à la conversion du signal haute fréquence autour d'une Fréquence Intermédiaire (FI) et à sa numérisation brute par un Convertisseur Analogique Numérique (CAN). Les traitements qui suivent peuvent être réalisés de façon logicielle : filtrage, décimation, démodulation, décodage... Ces traitements sont réalisés par des processeurs dédiés au traitement du signal (DSP *digital signal processor*), par des composants électroniques programmables (FPGA *Field Programmable Gate Array*), ou directement par le processeur d'un PC traditionnel. Cela confère une grande adaptabilité tant à l'émission qu'à la réception. En effet, il suffit de changer ou d'adapter le logiciel pour fonctionner avec un système radio différent.

Dans un système radio classique, l'émission/réception est assurée par des composants matériels (oscillateurs, filtres...) spécifiques et adaptés aux systèmes auxquels il est destiné. Il n'est donc souvent pas possible d'utiliser d'autres systèmes sans changer le matériel et donc l'intégralité du récepteur.

Les récents développements en matière de technologie radio permettent donc d'offrir de nouvelles capacités aux systèmes radio, qui jusqu'à présent étaient traditionnellement à fréquence unique et fixe pour une modulation donnée. L'avenir appartient aux radios programmables, qui peuvent être reconfigurées pour s'adapter aux différents protocoles de communication et à des bandes de fréquence variables. Cependant de tels systèmes nécessitent une forte puissance de calcul, par exemple des architectures multi-processseurs avec de fortes contraintes de mémoire. Dans le contexte de l'embarqué, il est nécessaire de minimiser la mémoire, même celle externe et lente, afin de réduire les coûts d'exploitation du produit. La mémoire est ce qu'il y a de plus coûteux dans un terminal mobile.

1.4 GPP - processeur d'usage général

Un GPP est un processeur d'usage général dont l'image est souvent associée à celle de l'ordinateur personnel (PC). Cette association, si elle reste vraie, n'est pas celle occupant la plus grande part de marché. Ainsi environ 90% des processeurs se trouvent aujourd'hui dans des systèmes embarqués. De tels processeurs sont différents du PC en effet. D'abord de concept CISC (*Complex Instruction Set Computer*), ils ont subi au cours des années une "cure d'amincissement" afin d'arriver à des architectures RISC (*Reduced Instruction Set Computer*) dans lesquelles instructions et modes d'adressage complexes étaient bannis. Des principes de simplification ont été mis en œuvre (comme par exemple le codage uniforme des instructions) afin de permettre notamment d'avoir des compilateurs fournissant de bonnes performances. Les principaux fournisseurs de ces architectures :

- la "famille" X86 : commencée avec le processeur 8086 d'INTEL, ses 2 représentants les plus connus sont aujourd'hui le P4 d'INTEL et l'Athlon d'AMD,
- SPARC (Sun), ARM (Acorn), Mips (Mips) sont des architectures licenciées permettant à différents fournisseurs de les proposer,
- les fabricants de FPGA s'intéressent également au marché : MicroBlaze (Xilinx), NIOS (Altera),
- d'autres encore : Motorola 68K (toujours utilisés dans les systèmes embarqués), PowerPc...

Toutes ces architectures mettent en œuvre le classique cycle VON NEUMANN (lecture de l'instruction, décodage de celle-ci, exécution, stockage des résultats produits et on passe à l'instruction suivante). Si ce modèle de base n'a guère évolué ces 30 dernières années, de nombreuses améliorations visant quasiment toutes un accroissement des performances ont été apportées.

D'un format initial de 4 *bits* (largeur du chemin de données) comme le 4004 d'Intel, on est arrivé à des architectures 64 *bits* comme par exemple l'Opteron d'AMD, ou le PowerPC 970 d'IBM. Un des buts visés est également d'aller au-delà des 4 *Go* (Gigaoctets) de RAM des processeurs 32 *bits*. Les premiers processeurs CISC demandaient plusieurs cycles pour exécuter une instruction, celle-ci n'ayant pas toutes la

même durée. L'intégration du *pipeline* en permettant de faire chevaucher simultanément plusieurs instructions à des étapes diverses du cycle VON NEUMANN a ramené ce nombre à théoriquement un cycle quelques soit l'instruction. L'accroissement de la profondeur du *pipeline* (appelé encore super pipeline) a permis d'atteindre des fréquences de fonctionnement dépassant le Giga Hertz (Pentium 4 d'INTEL cadencé à 4 Ghz). Classiquement ce nombre d'étages, qu'on appelle aussi la profondeur du *pipeline*, est de l'ordre de 10 voire plus pour certaines instructions.

Afin d'outrepasser cette barrière (une instruction par cycle), l'approche super scalaire en rajoutant des unités fonctionnelles (par exemple une deuxième unité de calcul sur les entiers, une unité de calcul flottant, des générateurs d'adresse,...) permet de lancer plusieurs instructions à chaque cycle, et sous réserve qu'il n'y ait pas de dépendance entre ces instructions, d'obtenir plusieurs résultats à chaque cycle. Ainsi, un super scalaire d'ordre 2 avec 2 UAL permettra de faire 2 calculs sur des nombres entiers à chaque cycle. Cette évolution a été accompagnée par l'introduction de jeux d'instructions complémentaires permettant d'adresser des unités fonctionnelles spécialisées. Ainsi INTEL (puis d'autres) il y a quelques années a rajouté des instructions multimédia ou *intrinsèques* (MMX) permettant de tirer pleinement parti de son architecture Pentium MMX.

Partant du constat que généralement le matériel est sous utilisé quand il exécute un seul flot d'instructions, l'hyper-threading consiste à émuler plusieurs processeurs logiques (chacun exécutant un flot d'instructions) sur un même processeur physique (par exemple l'INTEL Xeon).

Cet accroissement des performances a fait surgir de nouveaux problèmes. Les performances sont limitées par celle de l'étage du *pipeline* dont le temps de traversée est le plus long. On tombe ainsi sur le classique problème du goulot d'étranglement. Dans les processeurs, c'est l'étage d'accès à la mémoire qui constitue ce goulot. Pour en limiter l'impact les constructeurs ont très tôt mis en œuvre le principe du cache avec des mécanismes permettant d'assurer le va-et-vient des données entre mémoire et cache. Ces caches se sont ensuite hiérarchisés en des mémoires cache de premier niveau (quelques kilo-octets), de second niveau (quelques centaines de kilo-octets). La taille et la gestion de ces caches se sont également améliorées (organisation associative du cache). Le contrôleur mémoire externalisé dans des chipsets se trouve intégré dans le processeur, réduisant ainsi la latence sur les transferts. Par ailleurs, lancer des instructions en parallèle (super scalaire) n'est efficace que si ces instructions sont indépendantes les unes des autres. La détection de cet aléa a conduit à alourdir l'architecture par des mécanismes tels : les tampons de prédiction de branchement, l'exécution spéculative (l'ordre d'exécution ne respectant pas l'ordre d'écriture des instructions dans le programme).

On peut noter que l'approche VLIW (*Very Large Instruction Word*), présente également dans les GPP, est un retour au concept RISC original, élargi à l'approche super scalaire en étendant celle-ci (plus d'unités fonctionnelles). On garde la possibilité d'exécuter concurremment plusieurs instructions, mais l'électronique de gestion des aléas est enlevée, et leur gestion est confiée au compilateur. Des processeurs RISC relativement conventionnels sont également commercialisés sous forme d'IP (*Intellectual Property*) : ARM, NIOS, PowerPC. Associés bien souvent à des DSP, ils se retrouvent intégrés dans des SOC (System On Chip) pour des applications embarquées : multimédia, au-

tomobile. Ils ont alors en charge le contrôle du système. Là aussi, on peut constater la tendance vers l'augmentation du parallélisme. On peut ainsi citer l'architecture OMAP chez Texas (ARM + DSP C55).

1.5 Architectures embarquées

1.5.1 DSP

Un DSP (*Digital Signal Processor*) est un composant électronique programmable de type processeur. Il est utilisé dans bon nombre de domaines d'application qui nécessitent l'utilisation de filtres numériques ou adaptatifs, des FFTs, dans l'instrumentation (analyse transitoire, spectrale), dans le domaine médical (monitoring, échographie, imagerie médicale), dans les applications de contrôle (asservissement, robotique), le multimédia et l'imagerie, le militaire (radar, guidage de missile), les télécommunications (modems radio, cryptage de données, répéteurs de ligne) et le grand public (automobile, électroménager).

Son architecture est figée et comprend un ensemble d'éléments qui, suivant les modèles, permettent d'effectuer des calculs sur des données codées en virgule *fixe* ou *flottante*. Le codage en virgule fixe signifie qu'un nombre réel voit sa partie entière codée en puissance positive de 2 sur un nombre de bits suffisant et sa partie décimale codée en puissance négative de 2 sur le reste des bits non utilisés (ex : $0.75 = 0.5 + 0.25$ soit $2^{-1} + 2^{-2}$). Pour le codage en virgule flottante, le nombre est codé avec une mantisse et un exposant. Le calcul en virgule flottante permet une programmation plus souple, mais s'avère plus coûteux en terme de consommation. Le calcul en virgule fixe nécessite une unité arithmétique et logique plus simple, d'où une puissance de calcul plus élevée pour un prix d'achat du DSP plus bas. Les DSP ont une mémoire programme et une mémoire donnée séparées (architecture HARVARD), liées par un chemin de données (registres). Les registres sont reliés à des unités fonctionnelles permettant d'effectuer des opérations arithmétiques et/ou logiques : des comparaisons, des opérations de chargement et stockage de données.

Une architecture nommée VLIW (*Very Large Instruction Word*) permet de réaliser plusieurs instructions en un seul cycle. En effet les DSPs fonctionnent presque tous en mode *pipeliné*. L'instruction traduit une opération qui va être décomposée en un ensemble de sous-opérations à exécuter en parallèle par les différentes unités fonctionnelles. Par exemple : une instruction est tout d'abord lue en mémoire programme, décodée puis exécutée. Donc pour un ensemble d'instructions, chaque étape peut s'effectuer en parallèle sur des unités différentes suivant leurs disponibilités. Ceci accélère l'exécution du programme (proportionnelle au nombre d'étages du *pipeline*).

L'évolution technologique permet l'obtention de composants de plus en plus performants en terme de puissance de calcul ce qui, pour le DSP, se traduit par un nombre d'instructions par seconde dépassant le million (*MIPS* million d'instructions par seconde) et ne cessant de progresser. De même, la taille des données manipulées est passée de 16 à 32 *bits* pour des résultats pouvant aller jusqu'à 64 *bits*. Tout ceci favorise son utilisation dans des applications de traitement du signal nécessitant de fortes capacités de traitement.

Les architectures des DSP sont de plus en plus complexes (Fig. 1.1), et ressemblent presque à des Pentium. Les architectures des DSP (C62x et C64x de chez *"TEXAS*

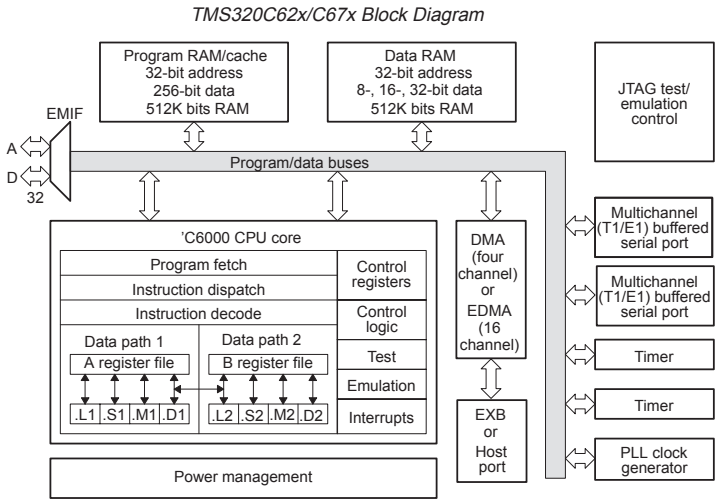


FIG. 1.1 – Architecture interne du C62x

INSTRUMENTS" utilisés par la suite sont détaillées sur la figure 1.2. La différence principale entre ces 2 DSP est le nombre d'étages du *pipeline*, et l'ajout d'une mémoire cache pour accélérer les transferts d'une mémoire externe vers une mémoire interne.

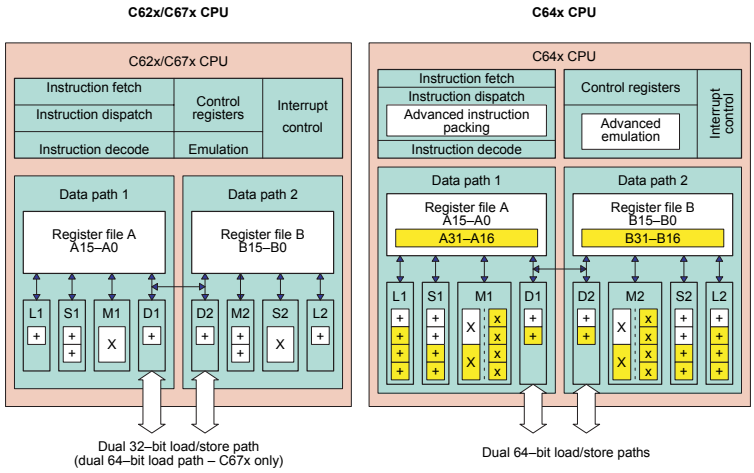


FIG. 1.2 – Comparaison entre les architectures du C62x et du C64x

1.5.2 FPGA

Les FPGA (*Field Programmable Gate Array*) sont des composants électroniques programmables de la famille des PLD (*Programmable Logic Device*). L'avantage de leur technologie est leur grande souplesse permettant une réutilisation à volonté et en un temps très court (quelques millisecondes) dans des algorithmes différents. Le progrès

permet de faire des composants toujours plus rapides et à plus haute intégration, autorisant la programmation d'applications importantes.

1.5.2.1 L'approche co-processeur

Une implémentation classique des FPGA est celle les utilisant comme des co-processeurs pour réduire le chemin critique logiciel. Les applications de traitement du signal, de codage/décodage et de chiffrement/déchiffrement particulièrement gourmandes en ressources processeur, et soumises au caractère ininterrompu du flot de données, sont de parfaites candidates pour une accélération câblée de type FPGA ou ASIC. L'utilisation de FPGA pour implémenter ces co-processeurs se justifie par la possibilité d'utiliser des accélérateurs différents sur un même composant d'où une utilisation extrêmement souple même si le changement de contexte reste un peu lent (de l'ordre de 250 ms). La nouvelle génération de composants reconfigurables, les FPGAs, reconfigurables partiellement, permettra de résoudre en partie ce problème. Au niveau du parallélisme, cette approche n'est constructive que pour accélérer localement le traitement dans une machine parallèle ou un réseau de stations utilisé comme tel. Par contre elle ne permet pas d'effectuer du calcul parallèle sur les FPGA.

1.5.2.2 L'approche systolique

La programmation systolique est utilisée dans les composants VLSI (*Very Large Scale Integration*) : elle consiste à découper le calcul en blocs similaires et à les placer au sein d'un pipeline de données (les résultats des blocs précédents servant d'entrées aux suivants). Cela permet de paralléliser des calculs critiques localement (co-processeurs systoliques) ou de faire de gros calculs parallélisés sur une carte contenant beaucoup de composants FPGA (massivement reconfigurable).

1.5.2.3 Programmation des FPGA

Il existe plusieurs façons de programmer un système incluant des FPGA : on peut discerner trois grandes tendances correspondant à des générations successives de compilateurs/implanteurs.

1.5.2.3.1 L'approche hardware

Cette méthode consiste à décrire le calcul sous sa forme la plus élémentaire possible : on forme des composants par composition de portes, puis on forme des circuits par composition de composants (on utilise le langage VHDL, acronyme de VHSIC[(2)] Hardware Description Language). Cette méthode présente l'avantage d'offrir une très grande optimisation du code car il est très proche de l'implémentation. En revanche, la conception peut être très longue si elle n'est pas associée à des bibliothèques standard pour les composants les plus courants.

1.5.2.3.2 L'approche impérative

La deuxième méthode consiste à écrire des programmes dans un langage séquentiel

[(2)] Very High Speed Integrated Circuit

proche du langage C (par exemple le langage Handel-C, SystemC) très proche des méthodes habituelles de développement. Le problème se situe au niveau du compilateur devant être très performant en raison de la taille très réduite des composants qui doit être compensée par découpage des programmes sur plusieurs puces et une bonne optimisation du code. En revanche, le parallélisme inhérent au modèle systolique permet d'optimiser les codes présentant du parallélisme.

1.5.2.3.3 L'approche *Softcore* La dernière approche intègre dans le FPGA des processeurs dénommés logiciels comme le NIOS (ALTERA) et le MICROBLAZE (XILINX). Ces processeurs peuvent être configurés et personnalisés en fonction de l'application visée en personnalisant en fonction de l'application les instructions RISC du processeur. Ensuite, ce processeur exécute de façon transparente un programme compilé en langage C.

1.5.3 ASIC

Un ASIC (*Application Specific Integrated Circuits*) est un circuit électronique intégrant sur une même puce tous les éléments actifs nécessaires à la réalisation d'une fonction ou d'un ensemble électronique. Il s'agit d'un circuit intégré conçu exclusivement pour le projet ou l'application qui l'utilise. Grâce aux ASIC les concepteurs peuvent désormais regrouper dans un seul boîtier toutes les fonctions nécessaires à leur application en occupant une surface comparable à celle d'un simple composant. Dans un produit, le circuit intégré est un élément porteur d'avenir, fiable, performant et peu encombrant, de plus il assure la confidentialité du savoir-faire.

Grâce à l'ASIC, il est possible de compléter (ou remplacer) des fonctions mécaniques, pneumatiques, hydrauliques ou électriques, abaissant les coûts de production. En particulier, il permet de concevoir les circuits électroniques de liaison entre le monde physique : capteur (analogique), et l'unité de traitement d'information et de commande (très souvent numérique). C'est l'élément sensible de la chaîne où la compétence et la maîtrise des outils restent fondamentales.

Comme pour toute décision industrielle, la justification de la réalisation d'un ASIC est d'ordre économique, mais également technique et stratégique. Le fonctionnement d'un ASIC doit être validé et l'amortissement de l'investissement de la réalisation du masque ne peut exister que sur une grande série de composants.

1.5.4 Communicateurs

1.5.4.1 FIFO

Une FIFO (*First In First Out*) est une ressource de mémorisation particulière dont le mécanisme de stockage est du type premier entré, premier sorti. Par rapport à une mémoire une FIFO ne possède pas de bus d'adresse puisque le stockage des données s'effectue toujours dans le même ordre. En revanche, une FIFO possède deux bus de données unidirectionnels, les données entrent par le bus d'entrée et sortent par le bus de sortie. Une FIFO est caractérisée par le nombre de données qu'elle peut stocker et par leur format. A chaque fois qu'une nouvelle donnée est stockée dans une FIFO, l'ensemble des données déjà présentes dans la FIFO est décalé vers la sortie. Lorsque

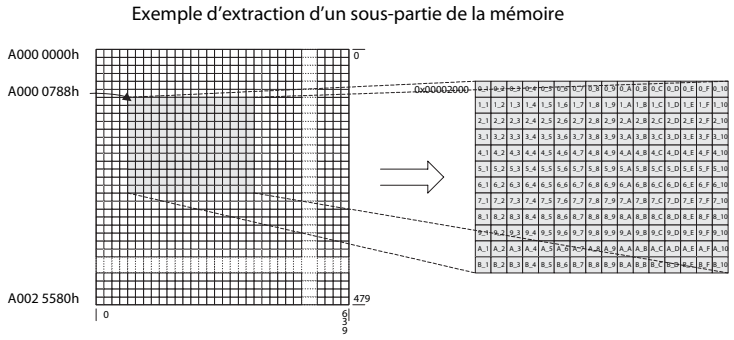


FIG. 1.3 – Extraction d'une sous-partie de l'image grâce à l'EDMA

la FIFO est pleine un signal l'indique. De la même manière lorsque la FIFO est vide un signal l'indique. Ces signaux sont utilisés afin de permettre aux unités accédant à la FIFO de connaître son état. Si la FIFO est pleine, l'écriture d'une nouvelle donnée va conduire soit à l'écrasement de la dernière donnée écrite dans la FIFO, soit à la perte de la donnée sortant de la FIFO, cela dépendant des mécanismes de contrôle utilisés par la FIFO.

1.5.4.2 Accélérateurs Matériels

L'accès direct mémoire ou DMA (*Direct Memory Access*) est un accélérateur matériel permettant un transfert des données d'un périphérique (port de communication, disque dur, mémoire externe) vers un autre, sans l'intervention du processeur si ce n'est pour initialiser ou conclure le transfert. La conclusion du transfert ou la disponibilité du périphérique peut être signalée par interruption. On l'oppose ainsi à des techniques de *polling* où le processeur reste en attente de chaque donnée.

Le DMA est nécessaire pour conserver la fluidité d'utilisation d'un système multi-tâches lors de l'accès à des périphériques rapides. En effet, en l'absence de DMA, le système est bloqué pendant les transferts de données. Par ailleurs, pour des périphériques rapides, il est impossible de transférer donnée par donnée sous interruption. Une alternative est que le périphérique ait une mémoire tampon partagée avec le système, et que le remplissage soit signalé par une interruption comme avec une FIFO.

Sur les nouveaux DSP C64x de *"TEXAS INSTRUMENTS"*, les accélérateurs matériels sont pourvus de fonctionnalités intéressantes. Il est par exemple possible de recopier un bloc d'une image 2D dans un buffer 1D, tout ceci en 5 instructions (Fig 1.3).

1.5.5 Conclusion

Dans cette partie, nous avons présenté les méthodes et outils permettant de réaliser le prototypage rapide d'applications. Par la suite, nous avons évoqué les architectures dans le domaine embarqué sur lesquelles des applications orientées données ou contrôle sont portées. Peu de ces outils ou méthodes existent au niveau système, et chacun d'eux présente des fonctionnalités différentes. Ces outils sont plus spécifiquement adaptés à une classe d'applications : orientée flot de données, orientée flot de contrôles...

Dans une approche de niveau système, des modélisations hautes (abstraites) de l'architecture et de l'application sont nécessaires dans un premier temps, afin de garantir une certaine généricité et d'accélérer les mises en adéquation. Toutefois une implantation finale optimisée sur cible suppose également une très bonne connaissance du matériel. Une architecture embarquée impose de plus des contraintes sur la consommation, sur la puissance de calcul des processeurs, sur la taille mémoire nécessaire...

Les travaux réalisés durant cette thèse concernent toutes les étapes de la chaîne de prototypage : depuis le niveau système optimisé sur un critère mémoire, jusqu'à une mise en œuvre sur cible automatique et optimisée sur le plan des ressources matérielles utilisées. Pour l'approche niveau système, nous nous appuyons sur la méthodologie AAA/SynDEx faisant l'objet du chapitre suivant. Pour les architectures cibles, notre choix s'est porté sur des solutions programmables et embarquées, c'est-à-dire de type DSP et FPGA. L'essentiel de nos expérimentations a plus précisément concerné des architectures multi-DSP, à la fois performantes et plus souples à mettre en œuvre. Le passage entre la couche système et l'exécution sur cible est abordé plus en détail dans le chapitre 3.

Chapitre 2

Méthodologie AAA/SynDEx

2.1 Introduction

Au sein du laboratoire IETR Groupe Image, un thème de recherche sur le prototypage rapide a débuté en 1996. Toujours actuellement, l'objectif du thème de recherche 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*). Cette appellation regroupe un ensemble de recherches menées au niveau national, visant à développer des méthodes systématiques de meilleure mise en correspondance entre l'algorithme d'une part, et l'architecture d'autre part. Cette méthodologie est adaptée à notre problématique, à savoir des applications d'image (systèmes orientés données), et une cible matérielle fixe essentiellement composée de plusieurs processeurs (DSP). Nous avons ainsi choisi la méthodologie AAA/SynDEx, pour la réalisation de la phase d'adéquation et de génération de code. Notre méthode d'optimisation mémoire présentée dans la suite s'appuie sur la méthodologie AAA/SynDEx. Dans ce chapitre, il est au préalable nécessaire de montrer les fondements de cette approche.

2.2 Présentation générale de la méthodologie AAA/SynDEx

SynDEx, logiciel de CAO au niveau système, est une concrétisation de la méthodologie AAA pour le prototypage rapide et l'implantation optimisée d'applications temps réel embarquées. Il permet en premier lieu de spécifier l'algorithme d'application et l'architecture multi-composants. Il réalise ensuite une adéquation correspondant à une implantation optimisée de l'algorithme sur l'architecture, dont le résultat est 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 faire une présentation de la méthodologie AAA/SynDEx, qui est plus détaillée dans [Vic99] et [Gra00], pour pouvoir appréhender les mécanismes que nous allons mettre en place dans la minimisation de la mémoire.

2.2.1 Modèle d'algorithme

La spécification fonctionnelle d'une application consiste en général en un algorithme obtenu par composition de plusieurs sous-algorithmes. A.TURING [Tur39] et E.POST [Pos48] définissent un algorithme comme une séquence finie (ordre total) d'opérations directement exécutable par une machine à états finis. Cette définition doit être étendue afin de prendre en compte :

- le parallélisme des architectures distribuées (composées de plusieurs machines à états finis interconnectées),
- l'interaction de l'application avec son environnement infiniment répétitive ou répété un nombre fini de fois (systèmes réactifs).

Le modèle d'algorithme AAA/SynDEx est un graphe de dépendance de données qui est hiérarchique, conditionné et factorisé [LS97] : il s'agit d'un graphe direct orienté acyclique (DAG) [BR00], dont les sommets sont des opérations partiellement ordonnées [Pra86] (parallélisme potentiel) par les dépendances de données inter-opérations ("diffusion de données" à travers des hyperarcs orientés pouvant avoir pour une seule origine plusieurs extrémités).

L'algorithme dans AAA/SynDEx est modélisé par un graphe de dépendance de données infini (Fig. 2.1), mais dans lequel un motif infiniment répété est identifié (graphe contracté Fig. 2.2). Le graphe est réduit par factorisation à son motif répété appelé *graphe flot de données* [Den75]. Les événements valués "circulent" sur les arcs de ce graphe formant les flots de données.

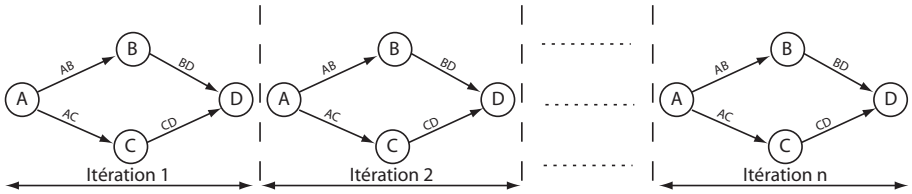


FIG. 2.1 – Graphe de dépendance sur n itérations

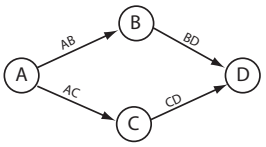


FIG. 2.2 – Graphe flot de données factorisé (contracté)

Le graphe d'algorithme peut être décrit de manière hiérarchique : chaque opération du graphe peut contenir un sous-graphe permettant une spécification hiérarchique de l'algorithme jusqu'aux "opérations atomiques" (Fig. 2.3). Une opération atomique est une opération élémentaire ne contenant que des ports d'entrée, de sortie, ou d'entrée-sortie. Sur la figure 2.3, les opérations atomiques sont A, B, C,D, E, C_1, C_2, tandis

que F et G sont des opérations hiérarchiques. Les opérations atomiques que l'on peut définir sous SynDEx sont :

l'opération *Constant* qui produit des données identiques lors de chaque itération de l'algorithme. Il suffit donc de l'exécuter lors de la première itération du graphe d'algorithme.

l'opération *Delay* (anciennement *Memory*) qui permet de spécifier les dépendances inter-itérations du graphe d'algorithme. Il s'agit d'une dépendance de donnée entre chaque motif du graphe flot de données.

l'opération *Function* qui est une opération atomique, quand elle contient uniquement des ports d'entrée et de sortie. Cependant cette opération peut elle-même contenir des opérations de calcul, de conditionnement et de répétition, elle-mêmes hiérarchiques.

l'opération *Sensor* qui active les entrées du graphe flot de données (capteur). Cette opération peut uniquement produire des données, elle ne contient que des ports de sortie.

l'opération *Actuator* qui active les sorties du graphe flot de données (actionneur). Cette opération peut uniquement consommer des données, elle ne contient que des ports d'entrée.

Le graphe d'algorithme peut contenir des dépendances de conditionnement (sorties de B et C sur la figure 2.3). La valeur portée par une dépendance de conditionnement détermine, à chaque réaction (répétition infinie), le choix parmi un ensemble de sous-graphes alternatifs. Sur la figure 2.3, dans le cas où la sortie de A vaut 1 alors le sous-graphe contenant G est exécuté. Dans le cas où la valeur de A est 2, c'est alors le sous-graphe contenant D qui est exécuté. L'imbrication des conditionnements est traduit par une représentation hiérarchique du graphe d'algorithme. À chaque interaction avec l'environnement, concrétisé 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 sommet non conditionné produit ses événements sur ses sorties dès que tous les événements sur les entrées sont arrivés.

Un sous-graphe du graphe d'algorithme peut être répété un nombre fini de fois et peut contenir, à son tour, un sous-graphe lui aussi répété un nombre fini de fois correspondant à des "nids de boucles". L'imbrication des boucles conduit aussi à de la hiérarchie dans le graphe d'algorithme. Un sous-graphe répété un nombre fini de fois peut aussi être réduit par factorisation à son motif répétitif.

AAA/SynDEx fournit la possibilité de spécifier les opérations répétées sous forme factorisée. La spécification d'algorithme répété sous une forme factorisée repose sur des opérations particulières, appelées *sommets frontières* car elles délimitent des *frontières de factorisation*. Les parties du graphe d'algorithme, appelées motifs, délimités par ces frontières sont les parties répétées du graphe. Les différents motifs répétés d'un graphe sont nécessairement disjoints. Les *opérations* ou *sommets frontières* sont créés automatiquement par l'outil SynDEx, et sont définies implicitement lors de la création du graphe d'algorithme.

Les *sommets frontières* sont les suivants :

– **sommet *Diffuse*** : diffuse la donnée en entrée aux motifs factorisés en sortie.

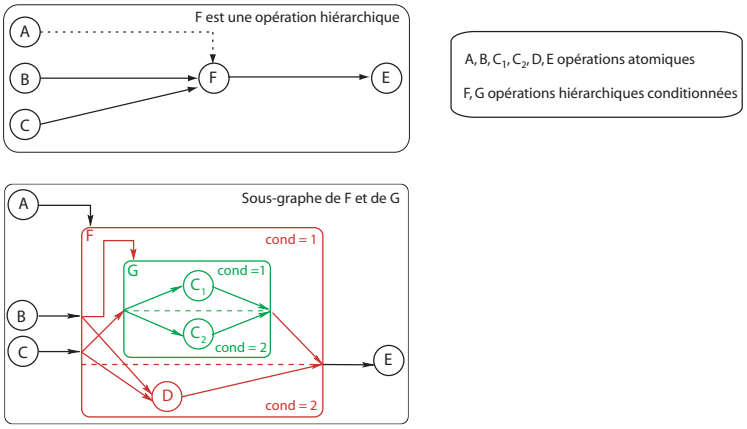


FIG. 2.3 – Graphe hiérarchique et conditionné

- **sommet *Fork*** : partitionne la donnée en entrée et distribue les parties aux motifs factorisés en sortie.
- **sommet *Join*** : regroupe les données partitionnées en entrée par les motifs factorisés en un vecteur en sortie.
- **sommet *Iterate*** : prend en entrée la sortie d'un des motifs factorisés ; sa sortie est redirigée vers le motif factorisé suivant. Une valeur d'initialisation est nécessaire.

Sur la figure 2.4, les sommets frontières sont représentés et permettent de réaliser le produit scalaire $A.B = s$ où s est un scalaire dont la valeur est :

$$s = A.B = a * c + b * d \text{ avec } A = \begin{bmatrix} a \\ b \end{bmatrix} \text{ et } B = \begin{bmatrix} c \\ d \end{bmatrix}.$$

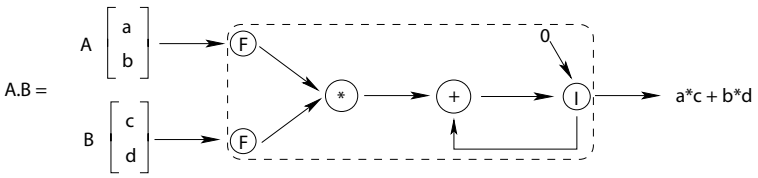


FIG. 2.4 – Graphe flot de données factorisé avec les sommets frontières

La figure 2.5 illustre la représentation sous SynDEx de la figure 2.4. A et B sont des opérations *sensor*, dont les ports de sortie sont de dimension 2. s est un *actuator* dont le port d'entrée est de dimension 1. *Prod_scalaire* est une opération *function* hiérarchique dont les ports d'entrée et de sortie sont de dimension 1. Les opérations *Fork* (F) sont donc créées automatiquement (implicitement) à la frontière entrante de *Prod_scalaire*, à cause du facteur de factorisation entre les sorties de A et B et l'entrée de *Prod_scalaire*. *Zero* est une constante nulle dont la valeur est définie à l'initialisation du graphe. *mul* et *ajout* sont des opérations *function* atomiques. L'opération *ajout* a une spécificité : elle a le même nom de port *io* sur une entrée et

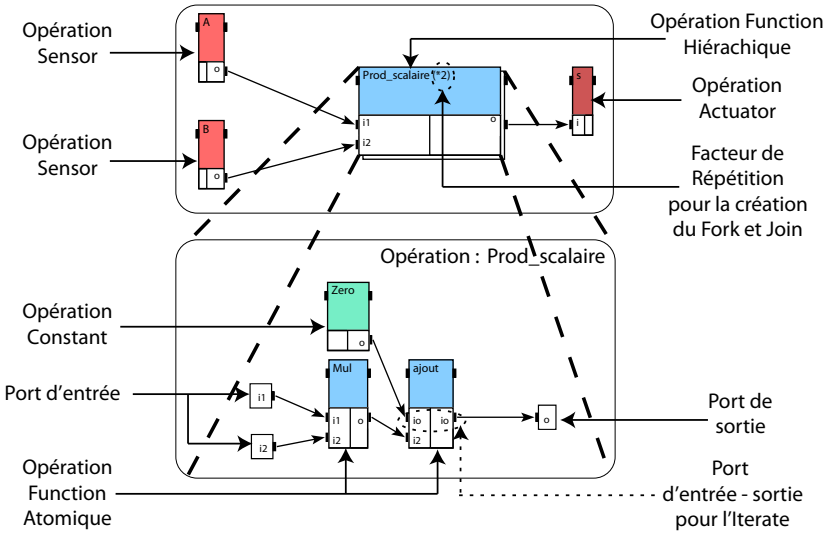


FIG. 2.5 – Graphe flot de données factorisé sous SynDEx

une sortie, ce qui crée l'opération frontière *Iterate*. Pour la première addition *ajout*, la valeur sur l'entrée de cette opération est la constante de valeur nulle, ensuite l'opération *ajout* est itérée et la sortie de $ajout_i$ est rebouclée sur l'entrée de $ajout_{i+1}$.

Le conditionnement et la factorisation sont les équivalents, en termes de graphe de dépendances de données, des structures de contrôle que l'on trouve dans les langages impératifs :

- If... Then... Else (*conditionnement*)
- For i=1 to N Do... (*factorisation*)

Les sommets frontières, principalement *Fork* et *Join*, ont comme principal avantage de permettre d'exprimer du parallélisme potentiel de données (par opposition au parallélisme potentiel plus général d'opération).

2.2.2 Modèle d'architecture

Les modèles les plus classiquement utilisés pour spécifier une architecture multi-processeurs (multi-composants ne comportant pas de circuit intégré spécifique) parallèle ou distribuée, sont les PRAM (*Parallel Random Access Machines*) et les DRAM (*Distributed Random Access Machines*) [Zom96]. Le premier modèle correspond à un ensemble de processeurs communiquant par mémoire partagée, alors que le second correspond à un ensemble de processeurs à mémoire distribuée communiquant par passage de messages. Si ces modèles sont suffisants pour décrire la distribution et l'ordonnancement des opérations de calcul de l'algorithme sur une architecture homogène, ils ne permettent pas :

- de prendre en compte des architectures hétérogènes,
- de décrire précisément la distribution et l'ordonnancement des opérations de communication inter-processeurs souvent critiques pour les performances temps réel.

Le modèle d'architecture hétérogène multi-composants [Gra00, GLS98, GLS99, GS03] choisi dans AAA/SynDEx est donc un graphe orienté, dont chaque sommet est une machine à états finis (machine séquentielle) et chaque arc une connexion physique entre deux machines à états finis. Dans AAA/SynDEx, il existe cinq types de sommets :

- l'opérateur pour séquencer des opérations de calcul (séquenceur d'instructions),
- le communicateur pour séquencer des opérations de communication (canal DMA),
- le *bus/mux/démux* avec ou sans arbitre pour sélectionner, diffuser et éventuellement arbitrer des données,
- la mémoire pour stocker des données,
- la mémoire pour stocker des programmes.

La mémoire peut aussi être considérée comme une machine séquentielle. Dans AAA/SynDEx, il existe deux types de sommets mémoire :

- la mémoire RAM (à accès aléatoire) pour stocker les données ou programmes locaux à un opérateur,
- la SAM (à accès séquentiel).

La RAM et la SAM sont toutes les deux utilisées pour les données communiquées entre opérateurs ou/et communicateurs.

L'arbitre dans un bus/mux/démux/arbitre constitue aussi une machine à états finis bus/mux/démux/arbitre. Cet arbitre décide de l'accès aux ressources partagées que sont les mémoires. Les différents sommets ne peuvent pas être connectés entre eux de n'importe quelle manière, il est nécessaire de respecter un ensemble de règles. Par exemple deux opérateurs ne peuvent pas être connectés directement, il en est de même pour deux communicateurs. Ces processeurs pour communiquer peuvent chacun être connectés à une RAM partagée ou à une SAM, en passant, ou non, par l'intermédiaire de communicateurs assurant le découplage entre calcul et communication.

L'hétérogénéité signifie non seulement que les sommets peuvent avoir chacun des caractéristiques différentes (durée d'exécution des opérations et taille mémoire des données communiquées par exemple), mais aussi que certaines opérations peuvent n'être exécutées que par certains opérateurs, ce qui permet de décrire tout autant des composants programmables (processeurs-FPGA) que des composants non programmables (ASIC) (pas seulement des processeurs).

Un processeur dans AAA/SynDEx est décrit par un sous-graphe contenant un seul opérateur, une ou plusieurs RAM de données et de programmes locaux. Ce processeur, n'ayant qu'un seul opérateur, n'est pas efficace pour la description d'un FPGA nécessitant généralement plusieurs machines à états finis ou opérateurs pour exprimer le parallélisme architectural de ce processeur. Un moyen de communication direct (sans routage) entre deux processeurs est un sous-graphe contenant au moins une RAM (données communiquées) et des bus/mux/démux/arbitre, ou bien un sous-graphe composé au minimum des sommets (bus/mux/démux/arbitre, RAM, communicateur, RAM ou SAM, communicateur, RAM, bus/mux/démux/arbitre).

Les architectures Pentek (fig. 2.6) et Sundance (Fig. 2.7) sont décrites suivant le formalisme de la méthodologie AAA/SynDEx. Sur les figures, les nœuds décrivent : $OPRi$ les opérateurs, $COMi$ les communicateurs, Sij les SAM entre le DSP_i et le DSP_j, Ri_DP la RAM de données et programme, Ri_SB mémoire externe de données $SBRAM$, Ri_SD mémoire externe $SDRAM$.

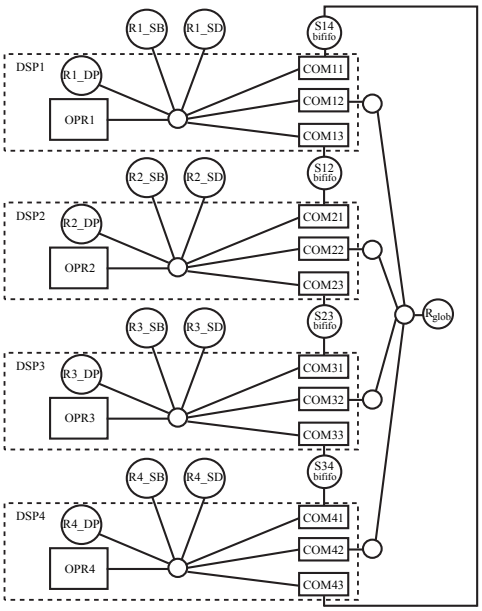


FIG. 2.6 – Graphe d'architecture dans la méthodologie AAA : modélisation d'une plate-forme pentek 4292 (4 DSP)

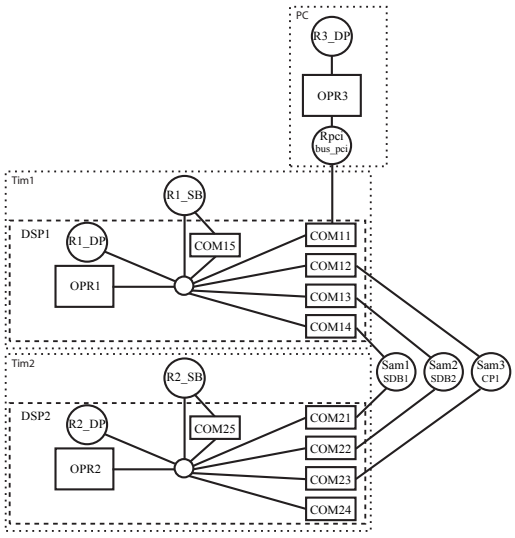


FIG. 2.7 – Graphe d'architecture dans la méthodologie AAA : modélisation d'une plate-forme Sundance (2 DSP) reliée à un PC via un bus PCI

Le modèle d'architecture décrit précédemment peut être simplifié. On peut abstraire la description interne d'un processeur en un seul et unique sommet. Il s'agit justement du modèle présent dans la version actuelle de SynDEx, qui sera également utilisé par la suite dans les exemples d'application montrés dans la partie II. Deux processeurs communiquant sont représentés par deux sommets processeur et un sommet

médium de communication SAM ou RAM. Il faut noter que SynDEx ne permet pas actuellement de décrire la mémoire interne du processeur.

2.2.3 Transformation de graphe et adéquation

2.2.3.1 Transformation du graphe d'algorithme

Le graphe d'algorithme spécifié par l'utilisateur est transformé par SynDEx avant d'effectuer l'adéquation. L'adéquation consiste à trouver la meilleure distribution et le meilleur ordonnancement d'un graphe flot de données sur une architecture cible. La transformation du graphe a pour objectif de résoudre les références et de défactoriser les définitions répétées ou conditionnées, mettant ainsi à plat sa hiérarchie : expansion du graphe d'algorithme. Les références sur des définitions hiérarchiques sont remplacées récursivement par le graphe d'algorithme de ces définitions. La récursion traite successivement les différents niveaux de hiérarchie. Le graphe obtenu après mise à plat est un graphe de dépendance portant uniquement sur des opérations atomiques.

La mise à plat d'un "nid de boucles" dans un graphe d'algorithme nécessite la création des différentes opérations implicites situées sur les sommets frontières :

sommet *Fork* : séparation (explosion) d'un buffer en plusieurs sous buffers utilisés par la même opération ($Fork$). La figure 2.8 illustre l'expansion de l'opération $Fork$, l'opération $Explode$ est ajoutée. L'utilisateur décrit sous SynDEx deux opérations $o_3 \rightarrow o_2$ dont le rapport entre la donnée sortante de o_3 et la donnée entrante de o_2 spécifie implicitement le facteur de répétition de o_2 (ici 3).

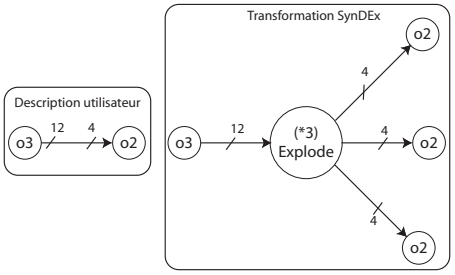


FIG. 2.8 – Explosion d'un buffer

sommet *Join* : regroupement (implosion) dans un buffer utilisé par la même opération de plusieurs sous buffers ($Join$). La figure 2.9 illustre l'expansion de l'opération $Join$, l'opération $Implode$ est ajoutée. L'utilisateur décrit sous SynDEx deux opérations $o_2 \rightarrow o_3$ dont le rapport entre la donnée sortante de o_3 et la donnée entrante de o_2 spécifie implicitement le facteur de répétition de o_2 (ici 3).

sommet *Iterate* : itération d'une opération récursivement sur elle-même grâce aux ports d'entrée-sortie. La figure 2.10 illustre l'expansion de l'opération *iterate*. L'utilisateur décrit explicitement un facteur de répétition de l'opération o_2. Pour avoir la récursivité de l'opération o_2, il faut spécifier en plus sur un port d'entrée et un port de sortie de o_2 le même nom et les mêmes caractéristiques sur ce

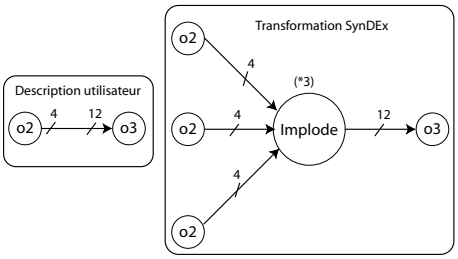


FIG. 2.9 – Implosion d'un buffer

port que l'on qualifiera d'entrée-sortie. L'expansion du sommet frontière *iterate* donne : $o_3 \rightarrow o_2 \rightarrow o_2 \rightarrow o_2$.

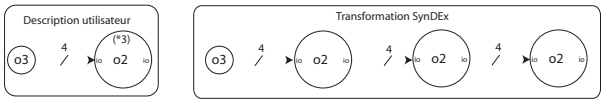


FIG. 2.10 – Itération d'une opération

sommet *Diffuse* : diffusion d'une donnée sur une opération explosée ou itérée. Sur la figure 2.11, la donnée entre o_1 et o_2 est diffusée sur chaque opération répétée o_2.

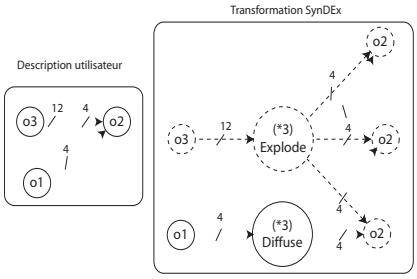


FIG. 2.11 – Diffusion d'une donnée

Le graphe d'algorithme de la figure 2.5 est transformé par SynDEx, pour ne contenir après la défactorisation que des opérations élémentaires (Fig. 2.13). L'opération *Fork* est traduite par l'opération élémentaire *Explode*, pour séparer (exploser) les vecteurs A et B. L'opération *Iterate* du graphe de la figure 2.5 est itérée sur l'opération *ajout* du graphe d'algorithme, de manière implicite comparée à l'exemple 2.10, et la dépendance de donnée $inter - ajout$ est créée automatiquement par l'outil SynDEx lors de la mise à plat du graphe d'algorithme.

Les opérations de conditionnement nécessitent également la création de sommets *CondI* et *Cond*0 (opérations atomiques) pendant la phase de mise à plat :

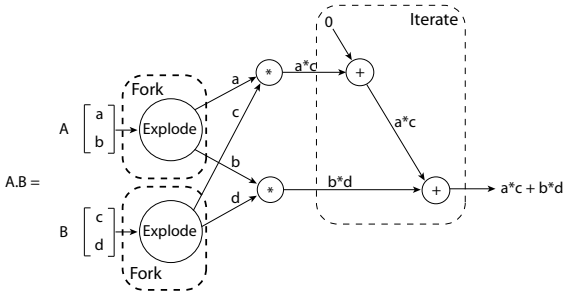


FIG. 2.12 – Mise à plat du graphe du produit scalaire

CondI pour la phase d'initialisation du conditionnement afin d'aiguiller la sortie d'une opération vers le sous-graphe qui doit être exécuté. Il permet surtout de savoir d'où vient la valeur de conditionnement lorsque l'on distribue l'application.

Cond0 pour la phase de finalisation du conditionnement afin de stocker les sorties des sous-graphes conditionnés vers une donnée unique consommée par une opération.

Le graphe d'algorithme de la figure 2.3 est transformé et expansé par SynDEx pour donner la figure 2.13, pour ne contenir que des opérations élémentaires. La sortie de l'opération C est conditionnée, l'opération $CondI_{CF}$ envoie la donnée suivant la valeur de A vers l'opération D ou $CondI_{CFG}$. L'entrée de E ou la sortie de $Cond_{Fo}$ prend la valeur du sous-graphe conditionné qui vient d'être exécuté. La sortie de $Cond_{Fo}$ prend soit la valeur de D, soit la valeur $Cond0_{Go}$.

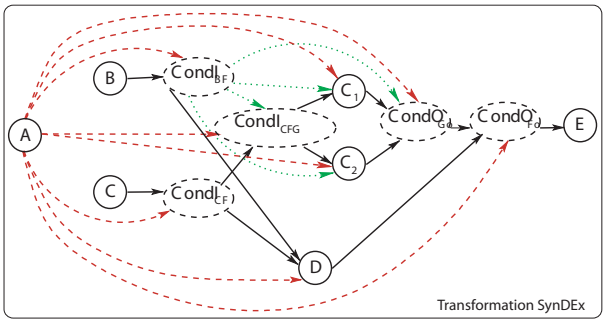


FIG. 2.13 – Sommets de conditionnement : *CondI* et *CondO*

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 flots de données habituels. La structure de données est elle aussi transformée au cours de cette mise à plat pour passer à une structure à accès plus rapide, la vitesse de calcul étant un soucis majeur de l'adéquation.

Le critère considéré pour le choix de la meilleure implantation dans l'espace des implantations possibles du graphe d'algorithme sur le graphe d'architecture est de retenir l'implantation donnant une latence globale minimale (minimisant la durée globale de l'application en tenant compte des coûts de communication inter-processeurs).

2.2.3.2 Implantation

L'implantation d'un algorithme sur une architecture multi-composants est une distribution et un ordonnancement non seulement des opérations de l'algorithme sur les opérateurs de l'architecture, mais aussi des opérations de communication, qui découlent de la première distribution, sur les communicateurs, les bus/mux/démux/arbitre et les mémoires.

La distribution consiste à affecter chaque opération de l'algorithme à un opérateur capable de l'exécuter (Fig. 2.14). Ceci conduit à une partition de l'ensemble des opérations de l'algorithme en autant de sous-graphes que d'opérateurs. Ensuite pour chacune de ces opérations, il faut ajouter un sommet d'allocation mémoire programme locale (resp. données locales) et affecter ce sommet à une RAM programme (resp. données) connectée à l'opérateur qui exécute l'opération. Enfin, il faut affecter chaque dépendance de données inter-opérateurs (c'est-à-dire entre opérations affectées à des opérateurs différents), à une route reliant les deux opérateurs (chemin dans le graphe de l'architecture). Il faut ainsi créer et insérer, entre les deux opérations de l'algorithme, autant d'opérations de communication que de communicateurs, autant de sommets identité [Gra00] que de sommets bus/mux/démux/arbitre et autant de sommets d'allocation de mémoires données qui sont communiquées que de sommets mémoire SAM et RAM sur la route (Fig. 2.15). Enfin, il faut affecter ces éléments aux sommets correspondants du graphe de l'architecture. Ceci conduit à une partition de l'ensemble des sommets de communication, des sommets identité et des sommets d'allocation respectivement en autant de sous-graphes que de communicateurs, de bus/mux/demux et de mémoires. Pour la suite il est important de noter que les sommets d'allocation sont ceux qui permettent de déterminer la taille des mémoires nécessaire pour l'application.

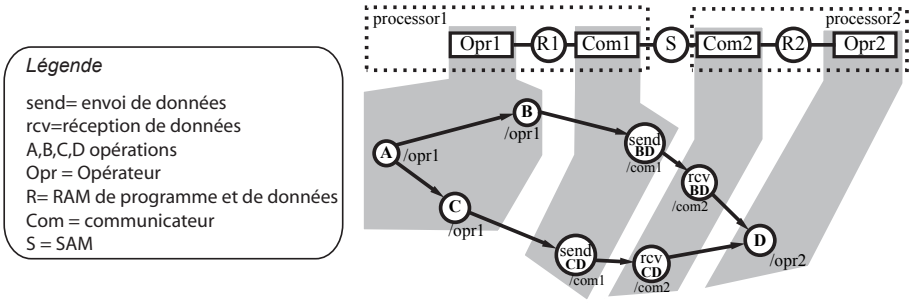


FIG. 2.14 – Affectation du graphe d'algorithme sur le graphe d'architecture

L'ordonnancement consiste à linéariser (rendre total par ajout d'arcs Fig. 2.16) l'ordre partiel associé à chaque sous-graphe de l'algorithme formé d'opérations, d'opérations de communication, de sommets identité et d'allocations, affectés à un sommet respectivement opérateur, communicateur, bus/mux/demux, et mémoires du graphe de l'architecture, car ceux-ci sont des machines séquentielles. Une implantation est donc le résultat d'une transformation du graphe de l'algorithme (ajout de nouveaux sommets 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

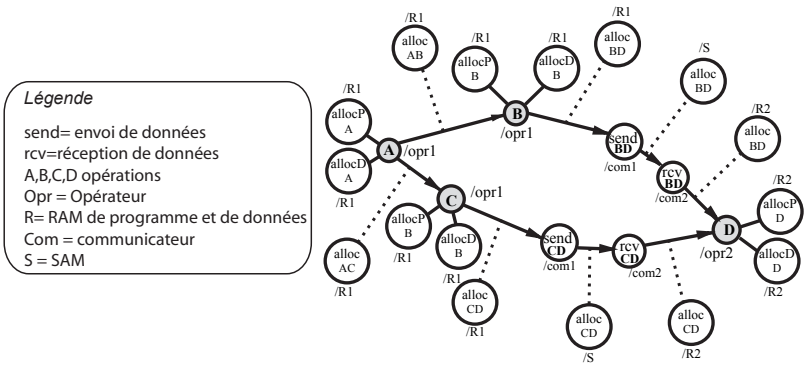


FIG. 2.15 – Sommet d'allocation sur un graphe d'algorithme

comme une composition de trois relations binaires : le routage, la distribution et l'ordonnancement, chacune d'elles mettant en correspondance deux couples de graphes (algorithme, architecture) [Vic99]. On peut aussi la voir comme une loi de composition externe où un graphe d'algorithme est composé avec (influencé par) un graphe d'architecture pour conduire à un graphe d'algorithme transformé (distribué et ordonnancé). Chacune de ces implantations possibles a des performances (latence, cadence) différentes. Ces performances sont obtenues par calcul de chemins critiques (pour les latences) et/ou de boucles critiques (pour les cadences) sur le graphe de l'implantation étiqueté par les durées d'exécution caractéristiques des opérateurs, des communicateurs, des bus/mux/démux/arbitre et des mémoires de l'architecture.

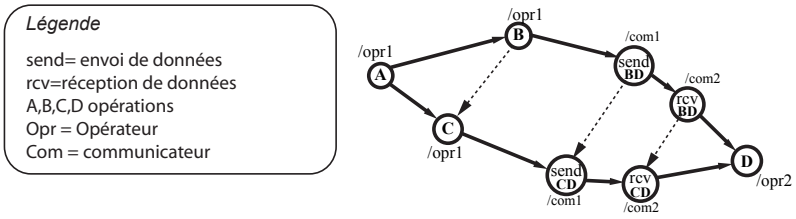


FIG. 2.16 – Ordre total sur un graphe d'algorithme

2.2.3.3 Pression d'ordonnancement

Afin de minimiser la latence globale de l'application, à un pas donné de l'algorithme d'ordonnancement, on choisit l'opération de calcul à ordonnancer et l'opérateur sur lequel elle est distribué en fonction de la *pression d'ordonnancement* de cette opération sur cet opérateur (pour plus de détails, voir [GLS99]). Nous retiendrons ici uniquement la forme finale de la pression d'ordonnancement servant de base aux calculs effectués dans SynDEx, corresondant à une heuristique gloutonne. La pression d'ordonnancement simplifiée $\sigma(o_i, p_j, n)$ pour une opération o_i distribuée sur un opérateur p_j à l'étape n de l'adéquation est :

$$\sigma(o_i, p_j, n) = S(o_i, p_j, n) + \Delta(o_i, p_j) + \bar{S}(o_i, p_j, n)_{p_j} - R(n-1)$$

où $\Delta(o_i, p_j)$ est la durée de l'opération o_i sur l'opérateur p_j. $S(o_i, p_j, n)$ $(S = Start)$ est la date de début au plus tôt de o_i sur p_j depuis le début, $\bar{S}(o_i, p_j, n)$ est la date de début au plus tard de o_i sur p_j depuis la fin. $E(o_i, p_j, n)$ $(E = End)$ est la date de fin au plus de tôt de o_i sur p_j depuis le début à l'étape n, $\bar{E}(o_i, p_j, n)$ est la date de fin au plus tard de o_i sur p_j depuis la fin à l'étape n. $R(n-1)$ est la longueur du chemin critique du graphe d'algorithme à l'étape $n-1$, c'est-à-dire la durée du chemin le plus long de ce graphe.

On note déjà que $R(n-1)$ ne dépend ni de o_i ni de p_j. Ainsi, pour choisir l'opérateur optimal pour une opération donnée, puis pour choisir le meilleur candidat $(operation, operateur_{optimal})$ à ordonnancer, la valeur R (le chemin critique du graphe d'algorithme) n'a pas à être prise en considération (quel que soit le choix à faire, de toute façon cette valeur ne varie pas). On ne calculera donc jamais la valeur de R dans l'adéquation de SynDEx.

La figure 2.17 [Gra00] montre la pression d'ordonnancement de l'opération C sur le graphe d'algorithme composé des opérations A, B, C, D.

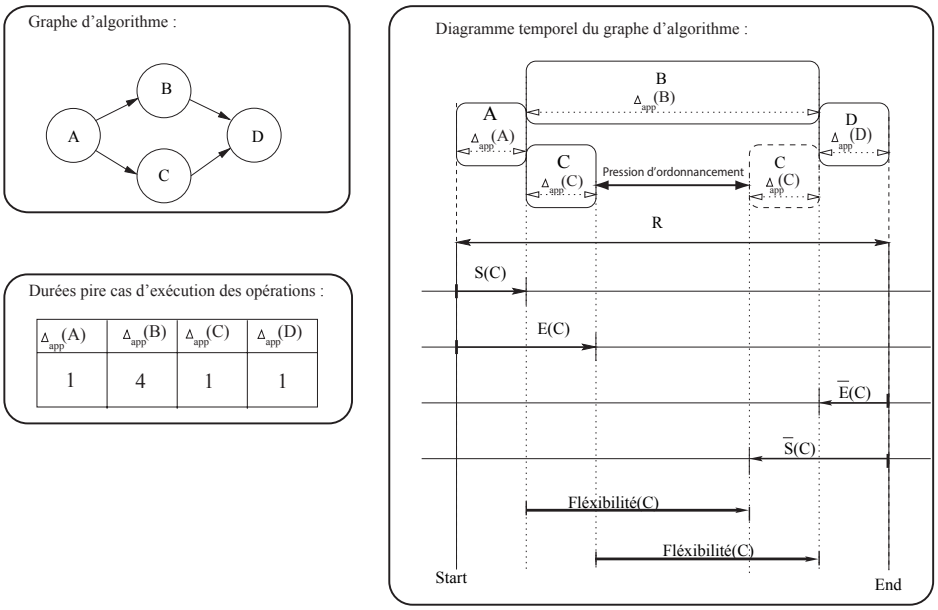


FIG. 2.17 – Pression d'ordonnancement

2.2.3.4 Choix de l'opérateur optimal pour une opération donnée

L'opérateur optimal pour une opération donnée est l'opérateur minimisant la pression d'ordonnancement pour cette opération. On note ici que dans le calcul de la pression d'ordonnancement d'une opération o_i, la valeur $\bar{E}(o_i, n)$ ne dépend pas de l'opérateur sur lequel o_i est distribuée. Ainsi, il n'est pas nécessaire de calculer cette valeur pour

choisir l'opérateur optimal pour o_i (cette valeur ne sera calculée qu'une fois le choix de l'opérateur effectué).

Le critère de choix de l'opérateur optimal $operateur_{optimal}$ pour une opération *operation* est donc uniquement la minimisation de la valeur suivante :

$$S(o_i, p_j, n) + \Delta(o_i, p_j)$$

Le principe du choix de l'opérateur optimal pour une opération donnée est décrit de manière simplifiée dans l'algorithme 2.1.

Algorithme 2.1 Choix de l'opérateur optimal pour une *opération* sur les *opérateurs*

```
opr_min ← tte[operateurs]
// esfs= earliest start from start = date de début depuis le début
esfs_min ← +∞
spfast_min ← +∞
pour tout operateur de operateurs faire
    si △(operation, operateur) ≠ NULL & Architecture.capable_d_executer
       spfast ← E(operation)_operator + △(operation, operateur)
       si spfast < spfast_min alors
          opr_min ← oprateur
          esfs_min ← oprateur
          sfpfast_min ← spfast
       fin_si
    fin_si
fin_pour
si esfs_min = +∞ alors
   Lever une erreur : aucun operateur n'est capable d'effectuer operation
sinon
    sp ← (esfs_min + E(opr_min)‾)
    retourner (opr_min, esfs_min, sp)
fin_si
```

L'ensemble *operateurs* passé en paramètre à la fonction de choix n'est pas l'ensemble complet des opérateurs de calcul du graphe d'architecture. Il est réduit à l'ensemble respectant les éventuelles contraintes de distribution imposées sur l'opération.

En fait, le corps de la boucle de l'algorithme 2.1 est un peu plus compliqué que ce qui est présenté ici. Les complications viennent principalement du fait que pour pouvoir calculer $E(operation, operateur)$, il faut d'abord calculer le temps que prendront les communications requises par *operation* lorsqu'elle est placée sur *operateur*.

Si l'on entre plus dans le détail, avant de calculer $E(operation, operateur)$, il faut donc d'abord :

- initialiser à vide (si ça n'a pas déjà été fait) l'ordonnancement pour la condition $cond(operation)$ sur *operateur* pour éviter une exception au moment du calcul de $E(operation, operateur)$,
- considérer temporairement que *operation* est ordonnancée sur *operateur*,
- créer des copies de sauvegarde des ordonnancements qui vont être modifiés par le calcul des communications et devront être rétablis une fois $E(operation, operateur)$ calculé,
- construire les communications des prédécesseurs de *operation* vers *operateur*.

Ensuite, une fois $E(operation, operateur)$ calculée, il faut rétablir les ordonnancements dans l'état où ils étaient avant leur modification provisoire pour pouvoir calculer $E(operation, operateur)$. Il faut ainsi, symétriquement à ce qui a été fait ci-dessus :

- supprimer les communications des prédécesseurs de *operation* vers *operateur*,
- rétablir les ordonnancements modifiés par le calcul des communications à partir de leurs sauvegardes,
- signaler que *operation* n'est plus ordonnancée.

2.2.4 La génération de code

La génération automatique d'exécutif distribué se fait suivant des règles décrivant la transformation d'un graphe d'implantation optimisé en un graphe d'exécution.

Pour chaque opérateur (resp. communicateur) on construit un programme séquentiel formé de la séquence des opérations de calcul (resp. communication) qu'il doit exécuter. Les opérations de communication sont :

- des "*Send*" et des "*Receive*" de données transmises entre communicateurs via une SAM (communication par passage de messages),
- des "*Write*" et des "*Read*" quand les données sont transmises via des RAM (communication par mémoire partagée).

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 les opérations de ces séquences, on ajoute des opérations de synchronisation avant et après chaque opération qui lit (resp. écrit) une donnée écrite (resp. lue) par une opération appartenant à une autre séquence. Ces opérations de synchronisation utilisent des sémaphores générés automatiquement. Il a été montré à l'aide des réseaux de Petri que ces sémaphores permettent à l'exécutif de respecter l'ordre partiel du graphe d'algorithme initial, n'introduisant ainsi pas d'inter-blocage dans une itération infinie entre la séquence de calcul et celles de communication, ou entre deux itérations infinies.

Les principes majeurs de la génération d'exécutif de SynDEx sont très bien détaillés dans le document [GLS98]. Cependant quelques mises à jour sont présentées ici, en relation avec nos contributions à l'outil.

2.2.4.1 Synchronisations inter-séquences

L'exécutif d'un opérateur de calcul est constitué d'une séquence de calcul principale et d'autant de séquences de communication que de média auxquels cet opérateur est connecté. Chaque séquence est exécutée par un séquenceur *(thread)* différent (Fig. 2.18). Le mécanisme des réseau de Petri pour le cheminement des jetons est expliqué en annexe C. Ce mécanisme est fondamental et est repris des travaux de T.GRANDPIERRE [Gra00]. Ces séquenceurs sont synchronisés au niveau des dépendances les reliant. Des synchronisations sont générées pour chaque dépendance reliant deux opérations exécutées par deux séquenceurs différents (communication-communication, calcul-communication, communication-calcul).

Les synchronisations générées par SynDEx ne synchronisent que les séquences d'un même opérateur de calcul. Les synchronisations entre deux opérateurs de calcul communiquant des données de l'un vers l'autre sont assurées par les couples d'opérations d'envoi/réception. Le fonctionnement de ces synchronisations inter-opérateurs est complètement dépendant de l'implantation qui sera donnée aux opérations d'en-

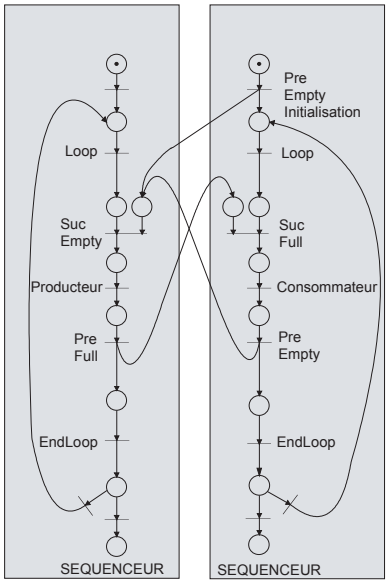


FIG. 2.18 – Réseau de Petri avec le maximum de parallélisme

voi/réception dans le noyau d'exécutif du médium les supportant (sémaphores partagés ou autre).

Les synchronisations inter-séquences sont de deux types :

- intra-itération : on assure qu'une donnée n'est pas consommée avant d'être produite (synchronisations de type "tampons pleins")
- inter-itération : on assure qu'une donnée produite par une opération à l'itération n du graphe d'implantation 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 soient exécutées (synchronisations de type "tampon vide").

Les synchronisations sont assurées par un mécanisme de sémaphores dont la gestion consiste en deux phases :

- déterminer les sémaphores nécessaires et générer des macros d'allocation pour ces sémaphores ;
- générer les macros d'attente notées *Suc* (équivalentes à *Wait ou P*) et de libération notées *Pre* (équivalentes à *Signal ou V*) sur les sémaphores avant et après les opérations de calcul ou de communication à synchroniser.

2.2.4.2 Les communicateurs

AAA permet l'utilisation de deux types de média de communication :

- SAM : transmission par paquets, FIFO
- RAM : mémoire partagée

Ces modèles ont été développés lors de la conception de la méthodologie, dans la thèse de T.GRANDPIERRE [Gra00]. Comme leurs fonctionnements sont différents, le modèle synchronisé de ces deux types de médium diffère d'un type à l'autre.

2.2.4.2.1 SAM

Un médium de communication de type SAM fonctionne comme une FIFO bidirectionnelle : les données produites sont envoyées par paquets vers le médium ("*Send*"), et le séquenceur de communication du consommateur vient les lire ("*Receive*"). Les synchronisations entre l'émetteur ("*Send*") et le récepteur ("*Receive*") sont transparentes et gérées le plus souvent par le médium au niveau matériel (*full flag* ou *empty flag* de la FIFO), elles ne sont pas générées par le modèle AAA. Les données sont lues dans l'ordre où elles sont écrites. L'envoi et la réception sont exécutés simultanément sur l'émetteur et le récepteur. Le modèle AAA ne génère des synchronisations qu'entre le séquenceur de calcul et le séquenceur de communication (*Pre* et *Suc*) au sein d'un même processeur.

La figure 2.19 représente le réseau de Petri d'une application utilisant un médium de communication de type SAM. 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. Ces graphes font apparaître les synchronisations logicielles générées par AAA/SynDEx à l'intérieur des processeurs tandis que les synchronisations inter-processeurs sont matérialisées par la FIFO.

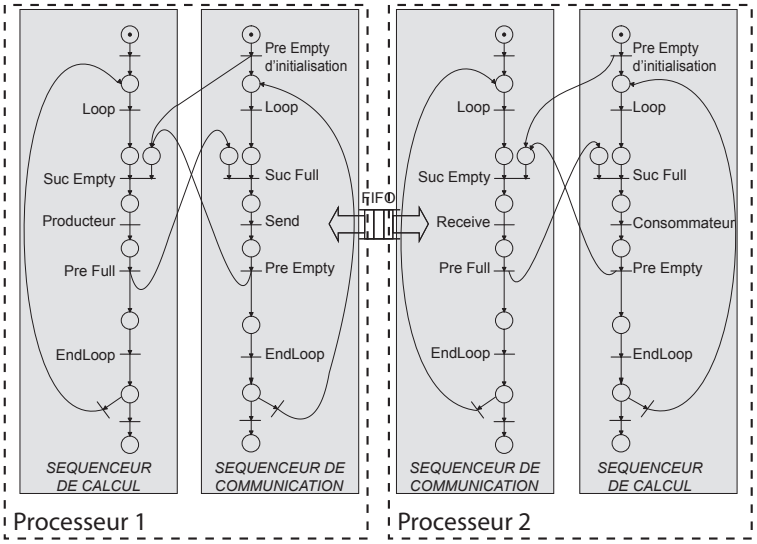


FIG. 2.19 – Réseau de Petri pour le modèle SAM

2.2.4.2.2 RAM

Le médium de communication de type RAM (Fig. 2.20) utilise une mémoire indexable à accès aléatoire, on peut donc venir lire les données dans un ordre différent de celui où elles ont été écrites. Les synchronisations entre l'émetteur et le récepteur sont maintenant gérées par logiciel, et générées par le modèle AAA. Il n'y a plus d'envoi ni de réception, mais des lectures et écritures séquentielles (les données sont écrites en mémoire, puis lues). Les ordonnancements sur les séquenceurs de communication n'est plus forcément symétriques ou duaux, car les données ne sont pas forcément lues dans le même ordre que celui dans lequel elles sont écrites (accès aléatoire).

La figure 2.20 représente le réseau de Petri de la même application que précédemment, en utilisant un médium de communication de type RAM. Ces graphes font apparaître les synchronisations inter-processeurs prises en compte pour la génération de code.

Il y a deux types de synchronisations pour la modélisation d'un médium de type RAM :

- Les synchronisations entre le séquenceur de calcul et le séquenceur de communications (*Pre* et *Suc*) au sein d'un même processeur qui sont les mêmes que précédemment,
- Les synchronisations partagées entre les processeurs connectés à la RAM (*PreR* et *SucR*).

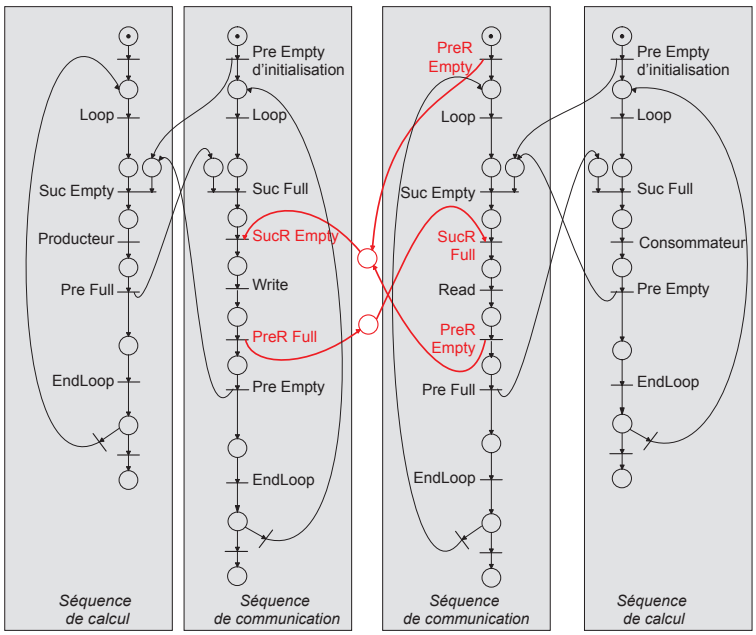


FIG. 2.20 – Réseau de Petri pour le modèle RAM

La figure 2.21 représente une mémoire partagée entre trois processeurs. Les processeurs 2 et 3 accèdent à toutes les données en RAM partagée (données $i = 1...3$), tandis que le processeur 1 n'accède qu'aux données 1 et 2. Le processeur 1 doit cependant connaître l'emplacement de chaque donnée par rapport l'adresse initiale de la RAM. Dans le référentiel de chaque processeur, il est nécessaire de situer l'emplacement de chaque donnée écrite en mémoire RAM, même si chaque processeur n'accède pas forcément à toutes les données écrites. Ce mécanisme n'existe pas dans la génération de code de la méthodologie AAA/SynDEx. Nous l'avons donc rajouté dans l'outil supportant la méthodologie.

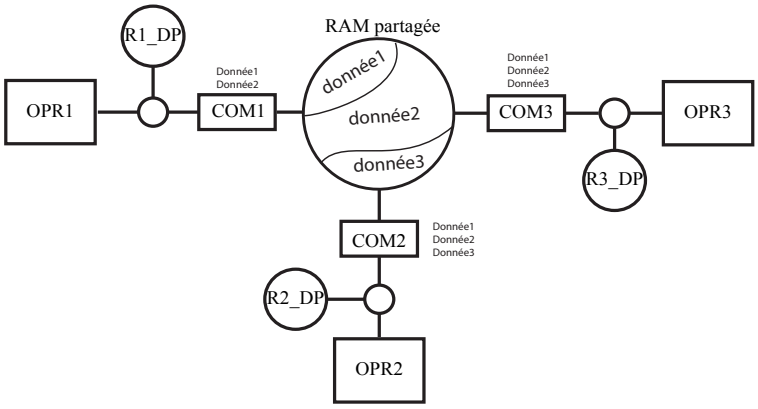


FIG. 2.21 – Implantation d'un mémoire RAM entre 3 opérateurs de calcul

2.3 SynDEx

L'outil CAO de niveau système SynDEx concrétise la méthodologie AAA/SynDEx pour le prototypage rapide et l'implantation optimisés d'applications temps réel embarquées. Il permet de spécifier l'algorithme d'application et l'architecture multi-composants de faire une adéquation correspondant à une implantation optimisée de l'algorithme sur l'architecture. Le résultat est une simulation temporelle de l'exécution de l'algorithme sur l'architecture. Il génère automatiquement pour chaque processeur un exécutif temps réel dédié, ou un fichier de configuration pour un exécutif temps réel résident standard. Les exécutifs dédiés sont produits à partir de bibliothèques de noyaux d'exécutif extensibles et portables dépendant des processeurs de l'architecture. Actuellement il supporte les architectures multi-processeurs à base de stations de travail UNIX, de processeurs i80x86, de processeurs de traitement du signal TMS320C40 et ADSP21060, de microcontrôleurs MPC555, MC68332 et i80C196. Il permet de faire naturellement de la conception conjointe logiciel/matériel en connectant à ces processeurs des circuits intégrés spécifiques ASIC ou FPGA contenant une interface de communication adéquate. SynDEx est utilisé aussi bien par des universitaires que par des industriels. Il est distribué gratuitement sur http ://SynDEx.org `http://SynDEx.org`.

2.4 Conclusion

Nous avons présenté la méthodologie AAA/SynDEx et l'outil SynDEx de niveau système principalement orientée flot de données avec un peu de contrôle, qui la supporte. L'utilisation de cette méthodologie réduit le cycle de développement et apporte un gain de temps important dans le développement d'applications multi-processeurs. L'ordre d'exécution garanti (pas d'inter-blocages), 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 de SynDEx apporte un processus automatique et optimisé en terme de placement/ordonnancement mais également en terme de minimisation spatiale (mécanismes non résidents) et temporelle (optimisation de la

latence). Cette présentation de SynDEx a permis de détailler les différentes phases du processus, depuis la phase de description de l'algorithme et de l'architecture, jusqu'à celle de la génération de code.

Chapitre 3

Chaîne de prototypage autour de la méthodologie AAA

3.1 Introduction : positionnement des travaux

3.1.1 Chaîne de prototypage AVS/SynDEx

Nous décrivons ici la chaîne de prototypage AVS/SynDEx (Fig. 3.1) développée et utilisée auparavant dans l'équipe Image de l'IETR pour l'implantation d'algorithmes sur architectures parallèles homogènes, constituées de processeurs usuels (DSP), ou sur architectures mixtes, constituées de DSP et de FPGA. La mise au point d'AVS/SynDEx au sein de l'IETR a débuté en 1996. Principalement réalisée dans le cadre de la thèse de V.FRESSE de 1997 à 2001 [FDN02, FD99, Fre01], elle s'est poursuivie jusqu'en 2002 lors de la thèse de J.-F.NEZAN [Nez02]. Grâce à cette chaîne de prototypage, les applications se sont focalisées sur des chaînes complètes de traitement d'images. La distribution sur les différentes unités de calcul est automatique. L'expertise requise pour un tel processus de portage se limite uniquement à la connaissance des algorithmes de traitement d'images. Les outils mis en jeu sont relativement simples à manipuler, accélérant de ce fait le processus d'implantation. Les phases de vérification fonctionnelle et de simulation de la description sont effectuées dès le début de la conception, à travers un outil de développement propre au traiteur de signal et d'images. En théorie il en résulte au niveau algorithmique qu'aucune erreur ne peut apparaître lors du prototypage. La chaîne de prototypage AVS/SynDEx est basée sur l'utilisation de deux logiciels : AVS[(1)] et SynDEx. Nous reviendrons sur AVS ci-après. La particularité de cette chaîne de prototypage est la création quasi-automatique d'exécutifs distribués statiques performants pour architectures hétérogènes en conservant l'utilisation des environnements.

AVS est un outil de haut niveau de visualisation de données permettant le développement et la simulation d'applications. Ses fonctionnalités sont principalement destinées aux créations graphiques. AVS est constitué d'un environnement de programmation visuel (Fig. 3.2), d'un large ensemble de fonctions graphiques et de visualisation, ainsi que d'un ensemble d'outils permettant de développer des applications complexes graphiques de manière plus aisée. Il propose des modèles d'applications et d'outils de

[(1)] Advanced Visual System

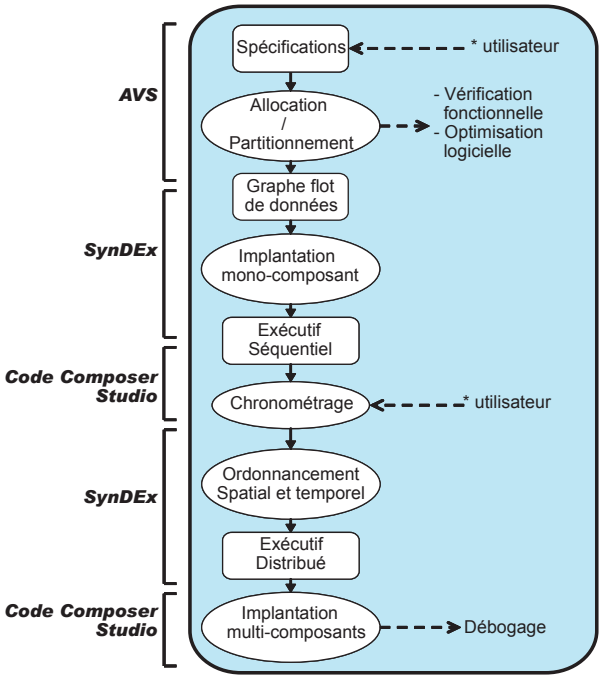


FIG. 3.1 – Chaîne de prototypage AVS/SynDEx

développement pour les environnements UNIX, Windows NT, 2000 et XP... AVS se sert de méthodes avancées de visualisation de données, telles que les couleurs, le mouvement, la forme et la taille pour des indications précises concernant les données. Cet outil s'avère relativement performant pour un large éventail de domaines. Il est par exemple fréquemment utilisé en imagerie biomédicale pour des visualisations obtenues au moyen de techniques 2D et 3D (l'objectif principal dans ce type d'application est la fusion d'images obtenues avec différentes sources et la visualisation des résultats). AVS est également utilisé pour des applications d'ingénierie : on peut citer notamment les domaines des télécommunications, le traitement d'images, ainsi que l'analyse financière... Il peut aisément s'interfacer avec d'autres outils, un exemple illustrant bien ceci est celui de la banque italienne, l'AIS de Milan, (Artificial Intelligence Software), qui interface AVS avec Oracle DBMS. Dans notre chaîne de prototypage, nous avions interfacé AVS avec SynDEx.

L'analogie existante entre la description flot de données sous AVS et celle sous SynDEx permet un interfaçage facilité entre les deux outils, ce qui a conduit l'équipe Image à développer un traducteur automatique, pour le passage d'AVS vers SynDEx. A cette époque, il s'agissait de la version 4 de SynDEx (6 aujourd'hui). Le rôle du traducteur est d'une part de transformer la description flot de données sous AVS en une description flot de données compatible avec SynDEx, et d'autre part de récupérer les fonctions associées aux modules créés sous AVS puis d'enlever les instructions propriétaires d'AVS. Le traducteur a été écrit à l'aide des macro-processeurs Lex (analyseur lexical) et Yacc (analyseur syntaxique).

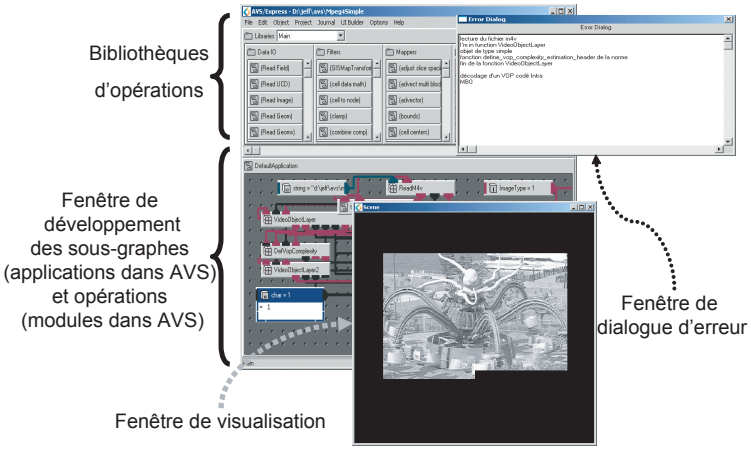


FIG. 3.2 – L'environnement de développement AVS

3.1.2 Limites des travaux précédents

Ce travail effectué en amont de SynDEx a donné de bons résultats pour l'extension du processus de prototypage dès la phase de mise au point du système. Mais AVS s'est avéré un logiciel commercial coûteux et les applications ainsi développées à partir d'AVS ont souvent nécessité, après portage vers SynDEx, d'être à nouveau déboguées. De ce fait, le portage sur les architectures cibles n'était plus direct.

Une autre approche académique a été développée par V.FRESSE en utilisant Ptolemy à la place d'AVS en amont de SynDEx, offrant des possibilités de simulation de systèmes hétérogènes intéressantes [FD02]. Dans cette seconde approche, l'objectif a été la mise en œuvre d'une phase de vérification fonctionnelle à partir d'une co-simulation C (pour cible DSP) et VHDL (pour cible FPGA). L'arrivée d'une nouvelle version de SynDEx, dont la sémantique des descriptions des graphes a radicalement changé, est venue interrompre ces développements. Le traducteur "descendant" permettant le passage d'AVS ou de Ptolemy vers SynDEx nécessitait désormais une mise à jour logicielle pour être compatible avec chaque évolution de SynDEx, d'AVS ou de Ptolemy. C'est pourquoi le portage d'une application développée sur station de travail vers une architecture cible n'est plus automatique avec la dernière version de SynDEx ou d'AVS. De plus, de nombreuses erreurs sont apparues lors de la compilation, du fait du passage d'un compilateur pour le langage C pour processeur à usage général à celui pour DSP. Leur cause tenait souvent aux problèmes de description de l'application flot de données (typages de données erronés ou d'erreur logicielle). La correction des erreurs effectuées sur l'architecture cible nécessitait alors un traducteur "ascendant" de SynDEx vers AVS ou Ptolemy pour pourvoir remonter les modifications apportées au code source, et faire une vérification fonctionnelle de l'algorithme sous l'environnement de simulation.

La réunion de ces deux outils était devenue inconfortable puisque toute modification d'un des outils appelait à de nouveaux développements. Cela nous a conduit à une ré-orientation de nos recherches. Nous avons finalement défini une autre stratégie (Section 3.3) qui est de créer les applications directement sous SynDEx grâce aux

bibliothèques d'opérations. Une bibliothèque est un regroupement d'opérations communes à différents graphes développés sous SynDEx (par exemple : une bibliothèque d'affichage).

Par ailleurs, l'obtention d'un code exécutable nécessite des noyaux d'exécutif SynDEx propres à la cible. En 2001, un certain nombre de noyaux étaient disponibles (pour le noyau du C4x notamment, ancienne génération de DSP "*TEXAS INSTRUMENTS*", développé par C.LAVARENNE). Les laboratoires de MITSUBISHI et de l'IETR groupe image ayant fait l'acquisition de nouveaux DSP de la famille des C6x, le problème du développement de nouveaux noyaux s'est alors posé pour nous. Il s'en est suivi la création d'un ensemble de bibliothèques pour processeurs et média décrites dans la section suivante.

3.2 Noyaux d'exécutifs SynDEx

La création de tels noyaux est nécessaire pour traduire les macro-codes génériques créés par SynDEx suite à l'adéquation en des exécutifs compilables pour des processeurs logiciels (DSP, pentium...) ou matériels (FPGA) reliés par divers liens de communication. Historiquement, c'est lors de stages personnels en tant qu'étudiant que le développement des noyaux d'exécutifs SynDEx a débuté en 2001 par un noyau dédié au DSP de la famille C6x de "TEXAS INSTRUMENTS" avec la participation de Y.LE MENER [LR01] sous la tutelle de J.-F.NEZAN, en collaboration avec l'équipe radio logicielle de MITSUBISHI ITE. Cela a continué en 2002 lors de mon stage de fin d'études [Rau02]. Ainsi, toutes les applications décrites sous SynDEx peuvent être portées automatiquement sur des architectures hétérogènes principalement multi-C6x [LRN+02, RND02]. Notre approche est générique dans le sens où nos travaux ont été réalisés sur des plates-formes n'appartenant pas aux mêmes constructeurs. Nous décrivons dans la suite les plates-formes ainsi que les noyaux utiles pour le portage de nos applications.

3.2.1 Principe de fonctionnement

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. Ce fichier est composé d'une liste d'appels de macros qui seront traduites par un macro-processeur en autant de programmes respectifs dans le langage source préféré de l'opérateur correspondant (C, ou assembleur par exemple). Chacun de ces programmes sources sera ensuite compilé puis chargé dans la mémoire programme de chaque opérateur. 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 fixe 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.

3.2.2 plates-formes cibles

3.2.2.1 Plate-forme Sundance

La plate-forme de prototypage Sundance est constituée d'une carte mère sur laquelle se trouve 4 emplacements (slots) pouvant accueillir des modules DSP ou FPGA (Fig. 3.3). La carte mère est connectée sur le bus PCI du PC. Cette plate-forme est modulable. En effet, les modules utilisés peuvent être choisis parmi les DSP et FPGA disponibles, de plus les liens de communication entre les modules sont également interchangeables.

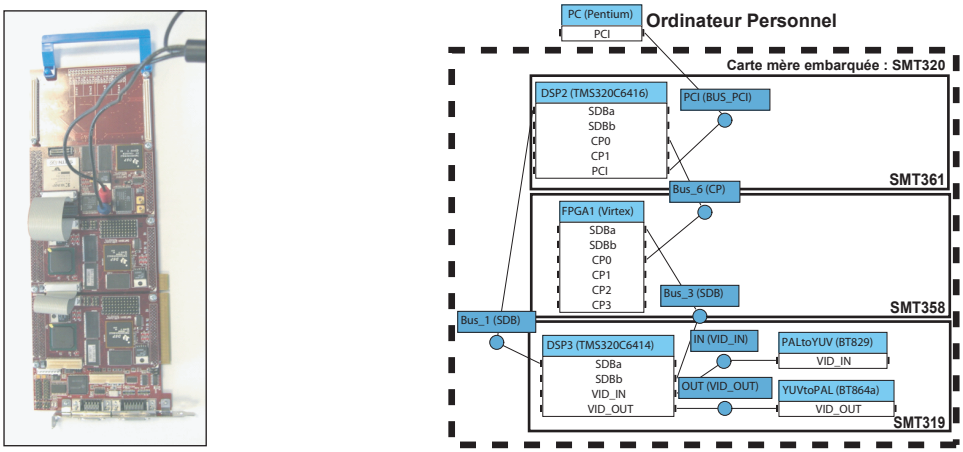


FIG. 3.3 – Exemple de topologie de plate-forme Sundance reliée à un PC via un bus PCI sous SynDEx

On retrouve sur cette plate-forme différents liens de communication :

- les ComPorts (CP) de type "TEXAS INSTRUMENTS" C4x fonctionnant à 20 Mo/s
- les SDB (Sundance Digital Bus) et SHB (Sundance High Digital Bus) ; ce sont des liens de communication propres à Sundance fonctionnant respectivement à 200 Mo/s et à 400 Mo/s,
- Le bus PCI entre un DSP sur le slot 0 et le PC hôte fonctionnant à 50 Mo/s avec la carte mère SMT320 et à 100 Mo/s avec la SMT310Q.

Les différents modules disponibles au sein du groupe image du laboratoire IETR sont :

- 2 SMT335 : Modules avec DSP *TMSC6201* à 200 MHz, mémoire interne 64 Ko données et 64 Ko programme, 2 ports SDB et 6 ports CP (Annexe A.2.1),
- 2 SMT361 : Modules avec DSP *TMSC6416* à 400 MHz, mémoire interne 1 Mo données et programme, 2 ports SDB et 6 ports CP (Annexe A.2.2),
- 2 SMT395 : Modules avec DSP *TMSC6416T* à 1 GHz, mémoire interne 1 Mo données et programme, 2 ports SHB et 6 ports CP (Annexe A.2.5),
- 1 SMT319 : Module avec DSP *TMSC6416* à 600 MHz, mémoire interne 1 Mo données et programme, 2 ports SDB, 2 ports de conversion numérique $\Longleftrightarrow$ analogique et 6 ports CP. Ce module est une *framegrabber* et permet l'acquisition d'une vidéo analogique et la restitution d'une vidéo numérique grâce à ses convertisseurs (Annexe A.2.4),
- 1 SMT358 : Module avec FPGA *Virtex 2 Pro*, 4 ports SDB et 6 ports CP (Annexe A.2.3),
- 2 cartes mères :1 SMT320 et 1 SMT310Q avec 1Mo de mémoire embarquée (Annexe A.1).

3.2.2.2 Plate-forme Pentek

La plate-forme de prototypage Pentek p4292 (Fig. 3.4 et Annexe A.3) possède une architecture fixe et constitue une évolution de la plate-forme p4290 (Annexe A.4) utilisée lors de mes stages en 2001 et 2002.

Elle est composée de quatre DSP *TMSC6203* (fonctionnement à 300 MHz, mémoire interne 512 Ko donnée et 384 Ko programme) dont la topologie est en anneau. Les liens de communication entre les DSP sont des FIFO bidirectionnelles fonctionnant à 300 Mo/s. Il est possible d'ajouter des cartes filles sur la plate-forme, telles qu'un module FPGA ou des convertisseurs numérique $\Longleftrightarrow$ analogique.

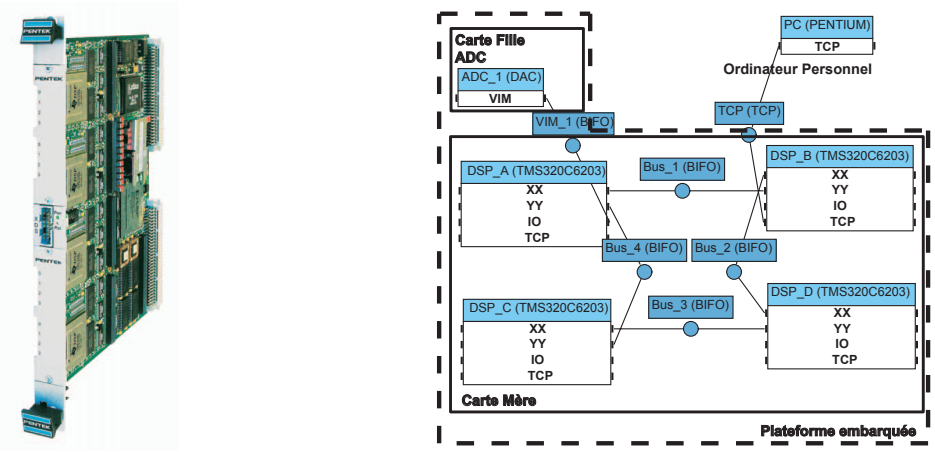


FIG. 3.4 – Exemple de topologie de plate-forme Pentek p4292 reliée à un PC via un bus TCP sous SynDEx

Cette plate-forme, se présentant sous la forme d'un boîtier autonome, est reliée aux PC par le réseau (lien TCP). Elle peut donc être programmée à partir de n'importe quel

ordinateur du réseau. La connexion TCP permet aussi de réaliser des communications entre les DSP et des PC, grâce à un processeur se trouvant sur la plate-forme.

3.2.3 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. Ces bibliothèques sont classées de façon hiérarchique (Fig. 3.7) afin d'être les plus génériques possible et d'être facilement réutilisables pour la définition d'une nouvelle bibliothèque.

Ces bibliothèques sont le cœur de notre chaîne de prototypage, du portage de nos applications dans le monde embarqué (architecture multi-DSP) ainsi que de nos partenariats industriels (société INNES) et académiques (projet PALMYRE). Par la suite des améliorations à ces bibliothèques ont été effectuées principalement par F.URBAN (stage de fin d'études) et moi-même, essentiellement sur les nouveaux composants acquis ainsi que sur les bus reliant les cartes embarquées au PC (station de travail).

3.2.3.1 Bibliothèques pour les processeurs

3.2.3.1.1 Bibliothèques C6x : C62x et C64x

Principes

Une bibliothèque pour les DSP C4x, ancienne génération de DSP "TEXAS INSTRUMENTS", existe. Cependant l'assembleur spécifique au noyau d'exécutif C4x n'est pas utilisable sur le C6x, du fait que ces DSP possèdent des architectures internes très différentes. De plus les DSP C4x intègrent des ports de communication (CP) fournissant des communications inter-C4x. Cette interface spécifique aux C4x permet de construire rapidement des architectures parallèles. A l'inverse, les DSP C6x ne possèdent pas de CP intégrés dans l'architecture interne du processeur, c'est pourquoi, afin de rendre possible les communications, les fabricants doivent ajouter des ressources matérielles entre les C6x. Ainsi une logique additionnelle doit être adoptée par les fabricants de plate-forme à base de multi-C6x : il s'agit généralement d'une FIFO bidirectionnelle inter-DSP. Cependant, suivant les plates-formes, la taille, les informations de contrôle et d'état de la FIFO peuvent changer. Les drapeaux ou *flags* d'une FIFO (FIFO vide, FIFO pleine) peuvent interrompre une séquence de calcul sur le DSP : début et fin d'un transfert. La manière de générer ces interruptions peut changer suivant les spécifications du fabricant.

Les interfaces entre les C6x n'étant pas standardisées, le noyau d'exécutif créé est obligatoirement dépendant de l'architecture cible, d'où la division du noyau en bibliothèques correspondant d'une part aux processeurs C6x, et d'autre part aux média de communication de Sundance et de Pentek.

Le macro-code générique fait apparaître plusieurs séquenceurs distincts par opérateur :

- un séquenceur de calcul pour les opérations,
- des séquenceurs de communication pour les communicateurs reliés à cet opérateur.

Ces derniers peuvent potentiellement être exécutés en parallèle selon les travaux décrits dans [Gra00]. Mais physiquement un processeur possède en général un unique séquenceur appelé CPU. Les séquenceurs (calcul et communication) doivent alors être exécutés par le CPU, interdisant tout parallélisme réel entre ces séquenceurs. Il est cependant possible d'utiliser les DMA du C6x pour prendre en charge les communications pendant que le CPU calcule les opérations de l'algorithme.

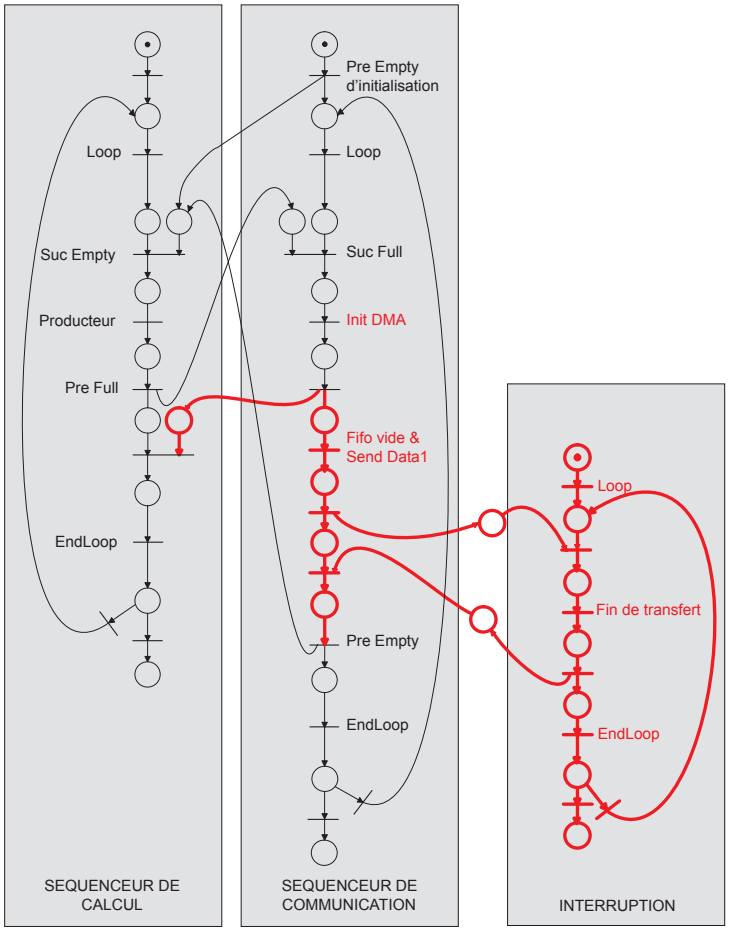


FIG. 3.5 – Réseaux de Petri complets d'un transfert effectué sur C6x

Le DMA doit être programmé par le CPU pour une communication donnée. Ensuite, il est vu comme un co-processeur qui va gérer l'envoi ou la réception des données sur le médium de communication. Il en résulte que la séquence de calcul est réalisée entièrement par le CPU et que la séquence de communication est partagée entre le CPU et le DMA. Nous avons toutefois mis en place une solution qui s'en rapproche basée sur le formalisme décrit sur la figure 3.5. En conséquence, une exécution des séquences de calcul et de communication totalement parallèle comme spécifié dans le modèle SynDEx n'est pas strictement réalisable, la technique utilisée s'en approche néanmoins au maximum. Les fonctions de transfert de chaque médium (envoi ou réception sur la FIFO) sont structurées de la façon suivante :

- calcul du nombre de paquets à transmettre (partitionnement de la donnée en paquets en fonction de la taille de la FIFO),
- sélection de la FIFO (nécessaire uniquement pour la p4290 à cause d'un multiplexeur en entrée des FIFO),
- configuration de la FIFO,
- autorisation des interruptions,
- démarrage du DMA.

Une fonction de transfert ne dispose du séquenceur d'instructions (CPU) que pour lancer un transfert qui s'exécute, ensuite, en parallèle des opérations de calcul. Le séquenceur de calcul se fait interrompre ensuite en fin de transfert. La sauvegarde de contexte de l'instruction suivante à exécuter après la fin de transfert qui était en cours est nécessaire. Ne pouvant réaliser cette sauvegarde en langage C standard, une solution a été de construire les séquenceurs de communication sous la forme d'une machine à états finis. Cette approche permet une cohérence maximale entre les graphes d'exécution sous SynDEx et leur implantation.

Le code généré est totalement écrit en langage C. Les DSP peuvent être programmés soit en assembleur, soit en langage C. Dans le cas d'une architecture VLIW comme celle des C6x, le développement d'un programme en assembleur optimisé nécessite une connaissance très pointue de l'architecture interne du processeur, et de longs temps de développement pour tirer parti des huit unités fonctionnelles du CPU. D'autre part, l'utilisation de l'assembleur se justifie surtout pour des calculs spécifiques, répétitifs (DCT par exemple). Dans le cas des mécanismes de la séquence de communication, les opérations doivent généralement être réalisées de manière séquentielle. La puissance des compilateurs pour les récents DSP est telle qu'un programme en langage C sera pratiquement aussi performant qu'un programme assembleur pour ce type d'opérations de communication. Dans ces conditions, l'utilisation de l'assembleur n'apparaît pas nécessaire en termes de performances.

La différence d'architecture entre les deux familles de DSP C62x et C64x implique une différence de programmation, et met en évidence la nécessité de créer deux noyaux différents *C62x.m4x* et *C64x.m4x,* qui prennent en compte la différence de programmation de ces deux familles de DSP :

- le *timer* : un tic (coup d'horloge du *timer*) tous les 4 coups d'horloge pour le C62x, contre un tic tous les 8 coups d'horloge pour le C64x,
- les accès mémoire : le C62x intègre un DMA (***D**irect **M**emory **A**cces*) avec 4 canaux alors que le C6416 intègre un EDMA (***E**nhanced **D**irect **M**emory **A**cces*) avec 64 canaux.

Cette différence de programmation des accès mémoire due à l'utilisation du module EDMA a nécessité l'évolution des bibliothèques de communication dépendant du C64x.

Caractéristiques du code généré pour C6x

Puisque le macro-code générique doit être transformé en un code compilable pour C6x, les techniques de programmation vont influer sur les performances finales.

Le code généré est au maximum indépendant des RTOS. Les développements sur plates-formes multi-C6x reposent souvent sur l'utilisation de RTOS, et dans notre cas *3L* de "Diamond" ou *DSP/BIOS* de "TEXAS INSTRUMENTS". Dans le cas de *3L*, la description de l'application est effectuée sous la forme d'un graphe, où chaque

opération est placée manuellement sur chaque opérateur. Ensuite, les communications inter-processeurs sont gérées automatiquement, mais les transferts sont bloquants et le surcoût de *3L* est important. *3L* est un RTOS multi-tâches, chaque opération est une tâche. Les tâches communiquent par des liens de communication ou FIFO logicielles qui représentent un coût temporel.

DSP/BIOS quant à lui n'est pas un RTOS multi-processeurs. Il est alors plus simple de traduire l'exécution d'un séquenceur "ordonnanceur" sous SynDEx par la création d'un *thread* sous DSP/BIOS, ainsi que les mécanismes de synchronisation inter-séquenceurs. Cette solution est particulièrement simple à mettre en œuvre puisqu'il faut uniquement remplacer le macro-code générique en un appel à des primitives. Nous avons dans le cas de la plate-forme Pentek, utilisé *DSP/BIOS* car les primitives disponibles sur cette plate-forme disposent de BSP (***B**oard **S**upport **P**ackage*) nécessitant l'utilisation de cet RTOS (Section 3.2.3.2.4).

Pour la plate-forme Sundance, tous nos développements se sont effectués dans un premier temps sans l'utilisation d'un RTOS. L'utilisation d'un RTOS à ensuite été envisagée pour traduire le macro-code, induisant un surcoût spatial et temporel.

Spatialement, DSP/BIOS prend 55 Ko de plus que la version sans RTOS.

Temporellement, l'impact de DSP/BIOS se fait principalement ressentir sur les petits transferts de données inter-processeurs.

3.2.3.1.2 Bibliothèque Pentium

Cette bibliothèque est devenue nécessaire pour l'interconnexion des plates-formes embarquées avec les stations de travail, ainsi que la simulation fonctionnelle sur PC (station de travail). Cette bibliothèque est entièrement fondée sur les travaux du C6x, grâce à l'utilisation du langage C qui présente de nombreux avantages :

- réutilisation directe pour d'autres noyaux,
- débogage simple, sans perte notable de temps.

Le noyau créé en langage C s'avère partiellement réutilisable pour de futurs DSP et processeurs, la tendance actuelle chez les fournisseurs de DSP étant de fournir des composants totalement programmables en C.

La bibliothèque pour *Pentium* est également disponible avec différents types d'OS : *Windows API* ou le standard *POSIX*. L'avantage de *POSIX* est l'interopérabilité entre les différents systèmes d'exploitation Linux ou Windows.

3.2.3.1.3 Bibliothèque FPGA

Le groupe CPR (Communications Propagation Radar) de l'IETR a également travaillé sur l'utilisation de modules FPGA des cartes Sundance à partir de SynDEx[Le 03]. Ceux-ci dialoguent avec les DSP à travers les mêmes bus CP et SDB. Leurs études ont permis d'établir une architecture interne du FPGA autorisant les traitements et l'échange de données (blocs IP des opérations, mémoires tampon, arbitres). La génération de code automatique sur une architecture composée d'un DSP et d'un FPGA, a été finalisée par F.URBAN, lors de ses stage de $4^{\text{ème}}$et $5^{\text{ème}}$année ingénieur sur les plates-formes Sundance et Pentek.

3.2.3.2 Bibliothèques pour les média de communication

3.2.3.2.1 Média SHB, SDB et CP (Sundance)

Les média de communication sont des FIFO bidirectionnelles et le mécanisme de transfert respecte strictement le modèle SAM décrit sur la figure 2.19.

Les bibliothèques pour les média CP et SDB ont d'abord été développées pour le C62x. Une mise à jour de ces bibliothèques a été nécessaire due à l'évolution du C64x : différence de programmation entre un DMA et un EDMA. Ces média suivent tous le fonctionnement décrit par la figure 3.5.

Les bibliothèques du C64x sont similaires à celles du C62x, seules les fonctions de transfert ont été développées avec l'utilisation de l'EDMA :

- les fonctions *CSL (***C**hip **S**upport **L**ibrary) sont utilisées pour programmer l'EDMA. Ce sont des fonctions fournies dans une librairie par "*TEXAS INSTRUMENTS*", elles simplifient la programmation en permettant notamment d'accéder aux registres et de les modifier de façon transparente grâce des fonctions (telles *EDMA_ open()* ou *EDMA_ config()*).
- le choix de l'interruption extérieure qui contrôle le démarrage d'un transfert n'est plus libre ; les interruptions extérieures (interruptions 4 à 7) contrôlent les canaux EDMA (canaux 4 à 7).
- l'interruption générée par l'EDMA est toujours la même (CPU_INT8) quelque soit le canal utilisé, par opposition à une interruption par canal DMA pour le C62x. Il n'y a donc plus qu'une seule fonction d'interruption liée à l'EDMA, un registre de statut permettant de savoir quel canal EDMA a généré l'interruption.
- l'arrêt d'un transfert avec l'EDMA doit être réalisé en créant un lien vers un canal EDMA nul, c'est à dire un canal EDMA dont tous les champs sont à 0. Donc à l'initialisation du DSP, un canal EDMA nul est créé (le canal 0), et lors de la configuration d'un EDMA, un lien est créé vers le canal nul.

	Buffer transféré (octets)			***théorique***
	1	***2500***	***10000***	
CP (C6201-200Mhz)	0,25 Mo/s	16 Mo/s	19 Mo/s	20 Mo/s
CP (C6416 - 400Mhz)	0,5 Mo/s	17 Mo/s	19 Mo/s	20 Mo/s
SDB (C6201 - 200Mhz)	0,25 Mo/s	173 Mo/s	190 Mo/s	200 Mo/s
SDB (C6416 - 400Mhz)	0,5 Mo/s	182 Mo/s	193 Mo/s	200 Mo/s
SHB (C6416T - 1Ghz)	1 Mo/s	364 Mo/s	387 Mo/s	400 Mo/s

TAB. 3.1 – Taux de transfert du lien de communication SDB avec SMT310Q

Le tableau 3.1 montre que pour un nombre de données important, on se rapproche rapidement du débit théorique. Les buffers de petites tailles restent éloignés du débit théorique car ces mesures (Tab. 3.1) prennent en compte le temps d'initialisation d'une fonction de transfert (configuration du DMA). Les média Sundance permettent un transfert au débit théorique, on peut donc représenter le transfert par une fonction affine $f(x) = ax + b$, ou a est le débit théorique et b est le temps d'initialisation d'un transfert.

3.2.3.2.2 Médium BIFO et Médium BIFO_DMA (Pentek)

La première génération de plate-forme Pentek (p4290) ne permettait pas d'accéder directement à l'espace mémoire de chaque FIFO, il fallait sélectionner l'adressage par l'intermédiaire d'un démultiplexeur situé entre le DSP et les FIFOs. La dernière génération de plate-forme Pentek (p4292) possède un accès direct à chaque FIFO permettant d'affecter un canal DMA par FIFO et donc de maximiser les parallélismes entre une opération de calcul et plusieurs transferts. Auparavant, on ne pouvait exécuter qu'un seul transfert à la fois. Comme précédemment, le débit théorique sur la plate-forme Pentek est atteint pour un grand nombre de données.

3.2.3.2.3 Médium Bus PCI (Sundance)

Sur les systèmes d'exploitation protégés (Windows, Linux, Solaris,...), un programmeur ne peut pas accéder directement au matériel depuis la couche logicielle où il développe. L'accès au matériel n'est autorisé qu'au sein du système d'exploitation par des modules logiciels (pilotes de périphériques). L'utilisation d'un pilote de périphérique est donc nécessaire pour communiquer avec la carte de développement par bus PCI. Le pilote utilisé ici a été développé au départ par Sundance, grâce à l'outil *WinDriver* permettant d'accéder aux périphériques sous *Windows* puis adapté par Fabrice URBAN, sous mon encadrement, pour une utilisation sous SynDEx. Il fournit les fonctions d'allocation de mémoire PCI et gère les transferts avec la carte PCI.

SynDEx permet de définir un lien de communication soit de type SAM, soit de type RAM. Le macro-code généré et sa traduction en C sont alors différents suivant l'utilisation de l'un ou l'autre des modèles. Ce médium est l'illustration de la première implantation matérielle d'un modèle RAM de la méthodologie AAA.

Bus PCI : Modèle SAM

Le premier modèle utilisé pour nos transferts via le bus PCI s'appuie sur le modèle SAM décrit sur la figure 2.19. Cependant, pour un communicateur de type SAM, les synchronisations inter-processeurs sont supposées matérielles (*full flag*, *empty flag*), et donc déjà câblées. Le bus PCI ne comportant pas ces bits de contrôle, il faut donc synchroniser les transferts afin d'être sûr de la disponibilité du buffer de transfert (vide dans le cas d'un "Send" et plein dans le cas d'un "Receive"). La figure 3.6 montre pour le bus PCI la nouvelle implantation du modèle SAM comme mécanisme de *rendez-vous :* à l'aide des sémaphores qui passent par une boîte aux lettres (CP 3) du DSP. Ce mécanisme est un mécanisme de *rendez-vous au plus tôt* car le CP 3 ne dispose que d'une profondeur de boîte aux lettres égale à 1 (= 1 registre). Cette synchronisation a pour effet de ralentir énormément les transferts.

Lors de l'initialisation, un espace mémoire est alloué et réservé pour les transferts. La taille de ce buffer est celle du plus grand buffer alloué sur le processeur, de façon à pouvoir réaliser un transfert d'un seul bloc réduisant ainsi le nombre de paquets à 1 et limitant ainsi les besoins de synchronisation.

Le macro-code généré par SynDEx comporte une partie initialisation suivie d'une boucle. Lors de l'initialisation, en plus de libérer les sémaphores prévus par SynDEx, un buffer est alloué dans la RAM du PC, et un pointeur permettant d'y écrire est transmi au DSP via la boîte aux lettres CP 3. Les pointeurs de la mémoire vue du

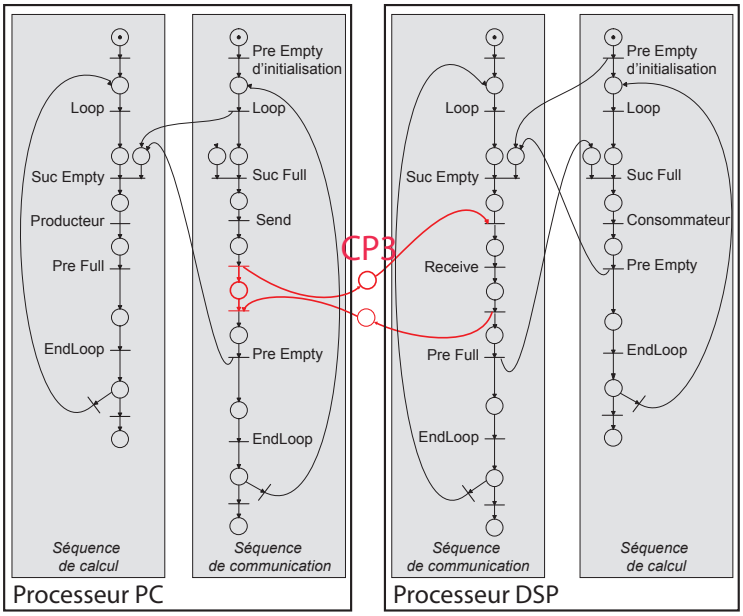


FIG. 3.6 – Synchronisation d'une séquence de communication pour le bus PCI suivant le modèle SAM

PC et de la mémoire vue du DSP ne sont pas les mêmes. Ceci est dû à l'utilisation du logiciel *WinDriver* qui conditionne tout cela :

- sur le PC, les données sont écrites et lues dans la RAM (espace d'adressage RAM), *WinDriver* se charge de transférer les données entre la RAM et le bus PCI (espace d'adressage PCI) par DMA, sur requête du bus PCI. Ceci est rendu transparent par l'utilisation des fonctions fournies par *WinDriver*.
- sur le DSP les données sont écrites et lues sur le *Global Bus* par le DMA. Le *Global Bus* se charge des transferts avec le bus PCI.

Bus PCI : Modèle RAM

Il existe beaucoup de similarités entre l'utilisation du bus PCI avec le modèle RAM et le modèle SAM ; dans cette partie, seules les différences sont abordées. Le modèle RAM utilisé est celui décrit sur la figure 2.20. La mise en œuvre du modèle RAM a été très délicate mais nécessaire à l'élaboration d'une plate-forme complète. Dans cette partie nous détaillons très précisement le fonctionnement et l'implantation de ce modèle sur le bus PCI.

Dans le modèle du communicateur de type RAM, il n'y a plus de synchronisation physique. SynDEx génère un buffer partagé alloué (sur le bus PCI) par donnée à transférer, ainsi que les sémaphores pour synchroniser individuellement chaque buffer. Sur chaque processeur, un tableau de sémaphores est alors alloué pour synchroniser la mémoire partagée, celui pour le PC étant directement alloué dans la RAM partagée.

Lors de l'initialisation, le buffer contenant les données partagées est alloué dans la RAM du bus PCI côté PC, et comme précédemment un pointeur permettant d'y écrire est transmis au DSP via la boîte aux lettres CP 3. La taille de ce buffer est la somme

des tailles des données transférées et de la taille du tableau de sémaphores permettant de synchroniser les "*Write*" et les "*Read*" de la mémoire partagée.

La macro *shared_* modifie temporairement les macros *Semaphores_* et *alloc_*. Sur le DSP, un tableau de sémaphores est alloué pour les synchronisations partagées, puis ils sont initialisés. Les allocations de mémoire ne seront générées que plus tard sur le PC, et sur le DSP, seuls les pointeurs vers la mémoire PCI allouée sur le PC seront définis. Sur le PC, la macro *thread_* génère l'allocation des buffers de transfert :

– *PCI.Port=init_BUS_PCI_SAM(TAILLE);*

Cette fonction initialise *WinDriver*, alloue la mémoire PCI, transmet l'adresse du bus PCI vers le DSP et initialise les sémaphores partagés. La taille de la mémoire allouée est la somme des buffers transmis et de la taille du tableau de sémaphores. Cette mémoire est ensuite partagée : le début contient les sémaphores, et chaque buffer transféré possède un pointeur correspondant.

De même pour le DSP, la macro *thread_* définit le pointeur des buffers de transfert :

– *PCI.Port=PCIRecover();*

Cette fonction récupère l'adresse PCI allouée par *WinDriver* sur le PC, les pointeurs de chaque buffer de transfert étant ensuite initialisés. Lors de l'initialisation, sur le DSP, les registres du DMA gérant l'envoi par paquets sont configurés.

Les macros *Suc1_* et *Pre0_* se voient traitées de la même façon que précédemment. Les macros *SucR1_* et *PreR1_* concernent les sémaphores partagés, il faut les transmettre à l'autre processeur lors de la macro *PreR1_*. Celle-ci génère :

– *sem_PCI[source__RAM_DSP1_PC_host_empty]= 1;*

– *Send_Semaphore_PCI(PCI.Port + source__RAM_DSP1_PC_host_empty);*

La fonction *Send_Semaphore_PCI()* diffère suivant l'endroit où elle s'exécute :

– sur le DSP, le sémaphore est écrit au début de la mémoire PCI, le PC viendra directement lire l'état de ce sémaphore lors du *SucR1_* correspondant,
– sur le PC, le sémaphore est envoyé au DSP par lien CompPort. Le DSP génère une interruption qui débloque la séquence de communication.

Les macros *read_* et *write_* génèrent l'appel des fonctions de transfert. Elles ressemblent à celles du cas précédent, à deux différences près :

– les synchronisations étant déjà présentes, ces fonctions ne les prennent plus en compte,
– les transferts sur les DSP sont traités par interruption, c'est à dire que les fonctions de transfert initialisent les bus et le DMA, puis rendent la main à la séquence de calculs.

La fin du transfert est détectée par une interruption générée par le DMA, celle-ci force la séquence de communication à se réactiver. Ce mécanisme permet un réel parallélisme entre les communications et les calculs.

Performances sur le Bus PCI et conclusion

Plusieurs tests de validation ont été effectués avec des cartes de développement Sundance SMT320 et SMT310Q ainsi que des modules DSP SMT335 (C6201), SMT361 (C6416) et SMT395 (C6416T). Les débits théoriques sont normalement de 132 Mo/s (bus de 32 *bits* à 33 MHz). Cependant les débits annoncés par Sundance sont inférieurs car de nombreux intermédiaires matériels (FIFO) sont nécessaires à la mise en place des communications via le bus PCI (Annexe A.1). Les débits théoriques sur les cartes

mères Sundance sont donc de 50 Mo/s pour une SMT320 et de 100 Mo/s pour une SMT310Q.

Les tableaux 3.2 et 3.3 répertorient les différents taux de transfert atteints selon le type RAM ou SAM et la taille des données transférées. Les résultats donnés sont assez loin des transferts théoriques car ces transferts prennent en compte le temps d'initialisation du DMA ainsi que les sémaphores de synchronisation.

	Buffer transféré (octets)			***Théorique***
	40	***4000***	***10000***	
bus PCI SAM (C6201)	3 Mo/s	12 Mo/s	16 Mo/s	50 Mo/s
bus PCI RAM (C6201)	3 Mo/s	35 Mo/s	36 Mo/s	50 Mo/s
bus PCI RAM (C6416)	5 Mo/s	37 Mo/s	42 Mo/s	50 Mo/s

TAB. 3.2 – Taux de transfert du lien de communication PCI entre PC et carte Sundance SMT320

	Buffer transféré (octets)			***Théorique***
	40	***4000***	***10000***	
bus PCI SAM (C6201)	6 Mo/s	30 Mo/s	32 Mo/s	100 Mo/s
bus PCI RAM (C6201)	6 Mo/s	68 Mo/s	72 Mo/s	100 Mo/s
bus PCI RAM (C6416)	8 Mo/s	69 Mo/s	72 Mo/s	100 Mo/s

TAB. 3.3 – Taux de transfert du lien de communication PCI entre PC et carte Sundance SMT320

Lorsque les transferts sont gérés par interruption (médium bus PCI RAM), des calculs peuvent être effectués en même temps que les communications, après initialisation du bus et du DMA. Ces initialisations demandent 2.5 μs, quelle que soit la taille à transférer, ceci explique la chute du débit lors du transfert de petits buffers. Ces taux de transfert prennent en compte les initialisations avant chaque transfert et la reconfiguration du DMA avant envoi du dernier paquet lorsque sa taille reste inférieure à 128 *mots*. Dans le cas du PCI SAM, ils prennent aussi en compte les synchronisations.

Nous disposons donc de deux modèles de média de communication pour le bus PCI : le modèle SAM exploite le bus PCI sous forme FIFO, et le modèle RAM sous forme de mémoire partagée. Le code C des communications est généré automatiquement suivant l'un ou l'autre des modèles grâce à SynDEx, en créant soit un lien de communication BUS_PCI_SAM de type SAM, ou BUS_PCI_RAM de type RAM. L'arborescence des bibliothèques M4 utilisées pour la génération du code correspondant est décrite sur la figure 3.7. Les taux de transfert obtenus sont environ deux fois plus élevés pour le modèle RAM, dans lequel existe une meilleure gestion des synchronisations.

Le modèle RAM du bus PCI offre la possibilité de recevoir les données dans un ordre différent de celui dans lequel elles ont été transmises. Ceci permet de faire apparaître du parallélisme dans une application séquentielle partagée (similitudes avec le mécanisme

de *double buffering* [ABD+01, LaM98]) et ainsi de diminuer le temps de traitement en diminuant les attentes.

3.2.3.2.4 Médium TCP (Pentek et PC)

Le lien de communication TCP peut être utilisé pour échanger des informations entre deux PC, entre deux applications sur un même PC, ou encore entre un DSP de la plate-forme Pentek et un PC, ou un autre DSP. Le protocole TCP ne change pas, que les processeurs interconnectés soient des PC ou des DSP, la différence réside dans la programmation pour l'un ou l'autre de ces processeurs.

Pour établir une connexion TCP, il faut un serveur qui attend ou écoute une connexion sur un numéro de port donné (on parle aussi de "socket TCP"), et un client qui se connecte au serveur sur ce port. Pour la génération de code il a donc fallu résoudre deux problèmes :

- le choix du processeur serveur et du processeur client pour l'établissement de la connexion,
- le choix du numéro de port sur lequel s'établit la connexion, sachant qu'il n'est pas possible d'établir deux connexions différentes sur un même port au sein d'un processeur (ou d'une plate-forme).

Il faut donc étudier le macro-code généré par SynDEx pour trouver une solution générique à ces problèmes.

Lorsque SynDEx génère le macro-code pour un processeur, il crée une macro par lien de communication connecté sur ce processeur :

thread_(type_du_medium, nom_du_port, processeur_1, processeur_2)

L'ordre des paramètres *processeur_1* et *processeur_2* est le même dans le macro-code des deux processeurs, ceci est donc exploité : le *processeur_1* est le serveur et *processeur_2* est le client.

Le deuxième problème se résoud en imposant certaines contraintes lors de la description de l'architecture sous SynDEx : le nom d'un port TCP créé doit être *TCP#* ou *TCP_#* et un lien TCP entre deux processeurs doit être connecté entre deux ports portant le même numéro (comme sur la figure 3.10). Lors de la traduction du macro-code en C, le numéro # du port est extrait de la chaîne de caractère *TCP_#*, et la connexion est établie sur le port 5000+#.

Les adresses IP des serveurs doivent être fournies aux clients pour la procédure de connexion, ceci est réalisé en ajoutant la ligne suivante au fichier *applicationName.m4x* : *define('nom_processeur_IP','adresseIP')* où *nom_processeur* est le nom du processeur dans le graphe d'architecture SynDEx, et *adresseIP* l'adresse IP correspondante.

Remarque : adresseIP peut aussi être le nom de la machine si elle n'est pas connectée à un DSP. Les commandes utilisées, développées sur PC pour les communications TCP, fonctionnent aussi bien avec l'adresse IP qu'avec le nom de la machine.

Utilisation de TCP sur PC

La génération automatique de code pour le TCP permet d'émuler une plate-forme complexe sur un PC (ou plusieurs PC). L'architecture décrite dans SynDEx comporte autant de processeurs pentium que nécessaire, reliés par des liens TCP. Une fois le

code généré, les programmes des différents processeurs sont exécutés sur un PC, dans des tâches séparées (ou sur des PC différents).

La génération de code sur PC est ici utilisée pour faire de l'émulation. L'objectif du laboratoire étant de faire de l'embarqué, le code n'est pas optimisé sur PC, notamment concernant les transferts qui sont tous bloquants sur PC (pas de parallélisme transferts/traitements).

Utilisation de TCP sur DSP

L'utilisation de TCP est possible sur la plate-forme Pentek du laboratoire MITSUBISHI ITE. Cette plate-forme intègre une mémoire globale de $32Mo$ partagée entre les quatre DSP. Un processeur supplémentaire (i960) intégré sur la plate-forme permet de gérer les communications avec l'extérieur. Ce processeur fonctionne par messages envoyés par les DSP. Ces messages sont des paramétrisations des tâches qu'il doit exécuter.

Pour réaliser les transferts TCP, des fonctions fournies par Pentek dans le BSP (***B**oard **S**upport **P**ackage*) Readyflow 4292 sont utilisées. Ces fonctions nécessitent l'utilisation de l'OS (**O**peration **S**ystem) temps réel de TEXAS INSTRUMENTS DSP/BIOS.

L'implantation de DSP/BIOS sur le DSP doit être prise en compte lors de l'utilisation de TCP pour la génération automatique de code en ajoutant à la bibliothèque *applicationName.m4x* la ligne suivante :

- *ifelse(processorName_,nom,'define('USE_DSP_BIOS')')* où *nom* est le nom donné dans SynDEx au processeur utilisant DSP/BIOS.

L'utilisation de DSP/BIOS modifie la gestion des interruptions. Les noyaux des média de communication de la plate-forme Pentek ont été modifiés pour prendre en compte ces modifications lors de l'utilisation de DSP/BIOS :

- les interruptions de fin de transfert sont *mappées* par DSP/BIOS,
- les fonctions TCP doivent être appelées à partir d'une tâche, alors que le séquenceur de communication est débloqué à partir d'une interruption dans le cas des transferts non bloquants.

Il faut donc créer une tâche rattachée à chaque interruption de fin de DMA. L'interruption libère un sémaphore débloquant cette tâche, puis cette tâche relance le séquenceur de communication.

Performances du Bus TCP sur DSP

Le débit de la liaison TCP de la plate-forme Pentek est de 1.5 Mo/s (12 $Mbit/s$). Ce débit théorique est atteint, cependant il reste faible et ne permet pas de transférer de la vidéo non compressée sur la plate-forme ($352 \times 288 \times 25 = 2.5\ Mo/s$ pour une vidéo CIF couleur), il est préalablement nécessaire de compresser la vidéo sur PC. De plus ces transferts sont bloquants (pas de génération d'interruption avec les fonctions fournies), le temps alloué au traitement se voit donc largement diminué.

L'utilisation de DSP/BIOS normalement contradictoire avec l'utilisation de SynDEx dont le but est de se passer de RTOS, induit une taille de programme plus importante. Par contre ce noyau permet d'analyser le taux d'erreur d'une chaîne de télécommunications, de transférer de l'audio (qualité CD) ou de la vidéo compressée.

3.2.4 Conclusion sur les noyaux SynDEx

Le développement de nouveaux noyaux SynDEx induit un nombre croissant de fichiers différents qu'il est nécessaire de classer. Ce classement réalisé afin d'offrir le moins de redondance possible entre les différentes bibliothèques, doit préserver une utilisation simple, la plus générique possible.

Dans cette optique les bibliothèques des média de communication des différents processeurs pour un type de médium donné sont regroupées en un seul fichier. Le choix de la génération de code pour un type de processeur se fait suivant la macro SynDEx *processorType_* . La figure 3.7 rappelle l'avancement des développements effectués.

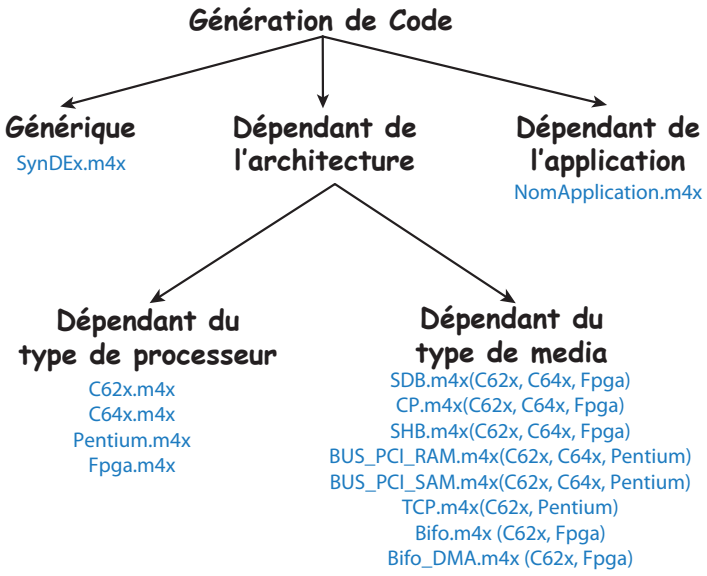


FIG. 3.7 – Arborescence des bibliothèques SynDEx

L'utilisation de la méthodologie AAA et de SynDEx depuis quelques années a permis le développement d'un noyau temps réel pour C6x, pour FPGA, et de noyaux pour les liens de communication inter-DSP et DSP-FPGA. Le développement des noyaux PCI et TCP permet aujourd'hui d'intégrer des PC à la plate-forme de développement. Des ressources PC (telles que l'affichage ou la lecture d'un fichier sur disque dur) peuvent donc maintenant être facilement utilisées.

Les noyaux de génération automatique de code concernant les plates-formes Sundance et Pentek 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 gérés automatiquement. De plus les différentes fonctions sont ordonnancées automatiquement par SynDEx de manière optimisée sur les différents composants de l'architecture, afin d'utiliser au mieux le matériel disponible.

Le changement de plate-forme de prototypage ou l'ajout de nouveaux composants nécessite bien sûr le développement de nouveaux noyaux. Cette tâche est rendue plus

rapide grâce aux connaissances acquises au cours du développement de ces divers noyaux [RUN+05].

3.3 Méthode de développement

3.3.1 Vérification fonctionnelle

Afin de tenir compte des spécificités du traitement des images d'un point de vue entrée/sortie, nous avons créé sur PC, puis encapsulé, des fonctions d'acquisition d'images (par *webcam* de la figure 3.8 ou par lecture de fichier), et d'affichage (Fig. 3.8) (une fenêtre associée à chaque appel de fonction de visualisation), sous environnement standard de développement (*Visual C++*, dev-cpp). L'utilisateur est ainsi en mesure d'instancier directement des opérations spécifiques de visualisation dans son graphe d'algorithme. Si le graphe d'architecture comporte un ou plusieurs PC, l'exécutif généré fera apparaître automatiquement les séquences désirées par le développeur dans des fenêtres de visualisation. Ces visualisations peuvent être utilisées en association avec les outils de débogage classiques sur PC et sur DSP. Sur DSP, l'exécutif généré réalisera le transfert de l'image sur le PC puis son affichage à l'écran.

FIG. 3.8 – Primitives de visualisation

La visualisation fonctionne en mode débogage et autorise ainsi les vérifications au fur et à mesure des optimisations faites jusqu'au prototype final par le développeur. Cette fonctionnalité permet d'intégrer automatiquement des opérations d'acquisition et de visualisation des images et nous autorise à une vérification fonctionnelle du code autrement que par des outils comme AVS ou Ptolemy. Partant directement du graphe d'algorithme sous SynDEx, en premier lieu une toute première implantation monoprocesseur sur PC, sous environnement standard, est générée pour le test fonctionnel. Cette démarche est plus cohérente que la précédente, évitant tout intermédiaire entre la description d'entrée et SynDEx. Bien que SynDEx soit principalement un outil de conception pour la distribution/ordonnancement et la génération de code, nous démontrons ici son utilisation à la place d'AVS comme point d'entrée de notre chaîne de prototypage.

3.3.2 Processus général

Le processus de prototypage (Fig. 3.9) sur architecture hétérogène multi-processeurs comporte ainsi 3 étapes :

Etape 1 : le graphe flot de données (GFD) de l'application est créé. La génération automatique de code fournit un programme en C pour une première implantation sur PC (mono-processeur). Ainsi, l'utilisateur peut créer les fonctions en langage C associées à chacune des opérations du GFD et tester les fonctionnalités de son application avec un environnement de compilation standard. Nous utilisons des primitives de visualisation remplaçant avantageusement celles d'AVS, permettant d'accélérer la mise au point des algorithmes d'image.

Etape 2 : le GFD est ensuite directement utilisé pour des implantations mono-processeur avec chronométrages. Le code généré insère automatiquement les fonctions de chronométrage. L'utilisateur doit juste compiler les sources et lancer l'exécution du programme avec l'outil adéquat à la cible. Les durées de chacune des fonctions (opérations 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. Cette phase est aussi l'occasion de tester diverses optimisations en fonction des cibles (optimisation de l'écriture du programme, utilisation du langage assembleur ou de bibliothèques optimisées livrées avec le composant). Les différents temps obtenus, après optimisations éventuelles, peuvent alors être reportés sous SynDEx.

Etape 3 : SynDEx réalise l'adéquation sur l'architecture parallèle et génère un exécutif distribué temps réel dans lequel les chronométrages ne sont pas présents. L'exécutif est optimisé en fonction de la plate-forme multi-processeurs. 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.

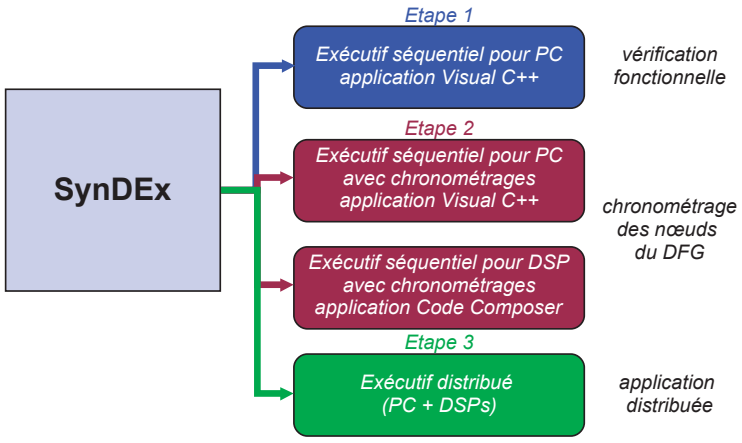


FIG. 3.9 – Etapes de développement

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.

Toutes les tâches complexes (adéquation, synchronisations, transferts de données et chronométrages) sont effectuées automatiquement. De plus, SynDEx étant téléchargeable gratuitement, le processus ne nécessite plus de logiciels commerciaux.

Avant mon arrivée à l'IETR comme dans l'équipe radio logicielle de MITSUBISHI ITE, les logiciels permettant la gestion de projet et le contrôle de versions étaient inexistants. Un logiciel de gestion de versions est un logiciel de gestion de configuration permettant de stocker des informations pour une ou plusieurs ressources informatiques, et permettant de récupérer toutes les versions intermédiaires des ressources, ainsi que les différences entre les versions (Annexe B).

Dans les phases de prototypage rapide de tels logiciels sont d'une nécessité primordiale, dans notre cas, c'est le logiciel de contrôle de versions *subversion* qui articule :

- le portage de nos applications pour la correction d'erreur,
- la collaboration des divers intervenants à un projet au sein d'un même laboratoire,
- la collaboration entre plusieurs laboratoires, l'IETR groupe Image, MITSUBISHI ITE, l'INRIA et divers organismes académiques et industriels pouvant intervenir dans un projet.

3.3.3 Vérification de la distribution de l'architecture

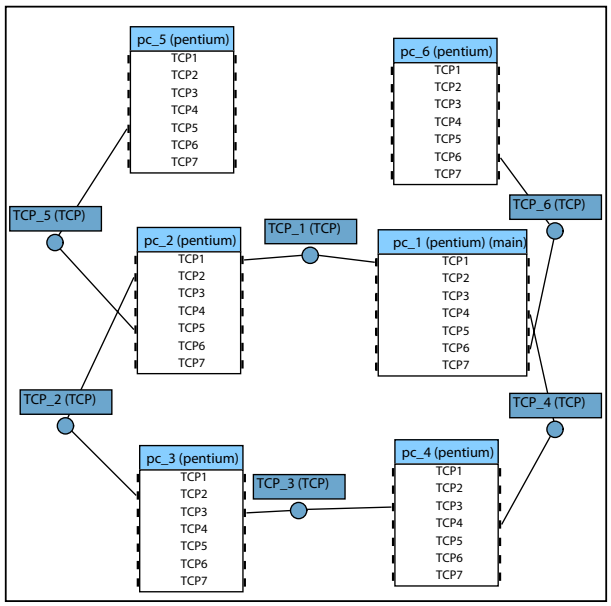


FIG. 3.10 – Utilisation de TCP pour la conversion de la plate-forme Pentek

SynDEx permet, avant l'acquisition d'une plate-forme de prototypage, de faire de l'exploration architecturale, c'est-à-dire de déterminer au mieux le nombre d'opérateurs (processeurs) nécessaires pour le portage de l'application. La bibliothèque TCP précédemment décrite nous permet donc, après cette phase d'exploration architecturale, de vérifier la distribution/ordonnancement du GFD de l'application sur une plate-forme émulée sous la forme d'une grappe de PC (*cluster*). Par exemple, la grappe de

PC sur la figure 3.10 représente la plate-forme *Pentek p4292*. Ici les transferts TCP sont bloquants dans le sens où l'on ne peut pas exécuter d'opération en parallèle des communications.

Nous avons donc modifié la génération de code SynDEx pour prendre en compte le portage sur une grappe de PC (Fig. 3.10). Cette modification garantit l'ordonnancement, pour cela il suffit de placer au mieux les sémaphores dans le séquenceur de calcul, et de considérer cette communication comme étant de durée nulle :

- pour un *send*, on déplace le sémaphore *Suc_ empty* juste après le sémaphore *Pre_ full* ce qui bloque le séquenceur de calcul tant que la communication n'est pas terminée. On place le *Pre_ full* au début du *send*.
- pour un *receive*, on déplace le sémaphore *Pre_ empty* juste après le sémaphore *Suc_ full* ce qui garantit de faire le *receive* entre ces deux sémaphores. On place le *Pre_ empty* à la date de début du *send* correspondant au *receive*.

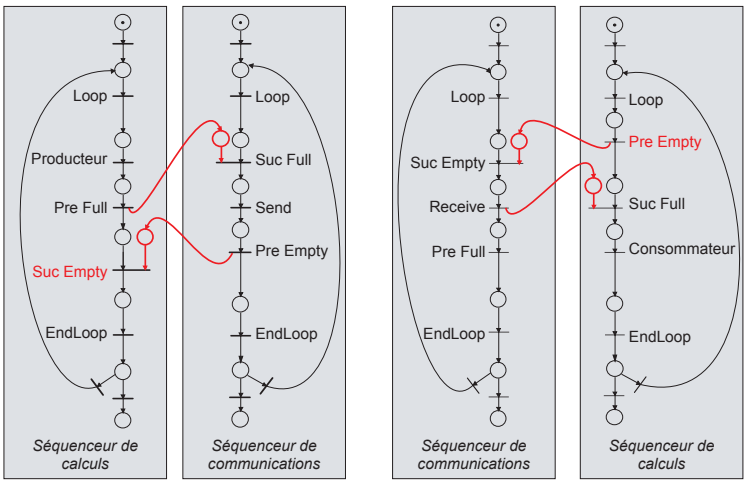


FIG. 3.11 – Modifications du réseau de Petri pour émuler une plate-forme avec TCP

3.3.4 Implantation sur systèmes embarqués

Des optimisations sur des systèmes embarqués ont été effectuées notamment sur des DSP *"TEXAS INSTRUMENTS"*. Les DSP principalement utilisés sont des C6416, plus rapides et plus récents. Les optimisations furent donc réalisées spécifiquement pour celui-ci. Certaines modifications plus générales furent appliquées aux C62x. Dans le code, les optimisations globales aux C6x (C64x et C62x) sont précédées par la variable préprocesseur *"TI_ OPTIM"*, tandis que celles caractéristiques du C6416 sont, elles, précédées par la variable préprocesseur *"CHIP_ 6416"* : par exemple dans l'utilisation d'intrinsèques ou de bibliothèques optimisées par "*TEXAS INSTRUMENTS*" spécifiques au C64x. Quant au code permettant la simulation pour PC, il est toujours inclus dans les sources C et est utilisé quand aucune variable de préprocesseur n'est définie.

Pour implanter efficacement une application sur DSP, il faut respecter des règles lors de l'écriture des fonctions. Voici celles qui ont été appliquées :

Règle 1 Dans les options de compilation, il existe quatre niveaux d'optimisation différents, le niveau *-o3* étant le plus élevé. C'est le niveau d'optimisation retenu par la suite lors de la compilation sous "*Code Composer Studio*" (compilateur et outil de *TEXAS INSTRUMENTS*).

Règle 2 Pour optimiser une fonction C, l'objectif est d'écrire les fonctions de telle sorte que le compilateur puisse réaliser un code assembleur avec le plus de parallélisme et de mise en *pipeline* possible dans les boucles pour une vitesse de calcul accrue. Pour cela, nous pouvons lui fournir certaines informations :

- Certains paramètres d'une fonction sont utilisés sans être modifiés. Il est préférable de le notifier en ajoutant le mot "const". Dans l'exemple suivant de la fonction "*demo*", c est calculé à partir de a et b. Les valeurs de a et b n'étant pas modifiées dans la suite de la fonction, il est plus intéressant d'écrire :

```
void demo(const int *a, const int *b, int *c)
```

- Il est également intéressant pour le compilateur de savoir si deux tableaux fournis en paramètres se chevauchent ou non. Quand ils se recouvrent, le compilateur prévoit des dépendances de données entre les entrées/sorties de la fonction et ne peut pas réaliser certaines opérations en parallèle sous peine de modifier le résultat final. Il faut alors utiliser l'instruction *restrict* dans le cas où ces buffers ne se chevauchent pas. Dans l'exemple de la fonction "*demo*", si a et b pointent sur des tableaux distincts (sans recouvrement mémoire), il est préférable d'utiliser le prototype suivant :

```
void demo(const int *restrict a, const int *restrict b, int *c)
```

Règle 3 Une dernière étape importante dans le développement de nos applications sous SynDEx est la meilleure utilisation de l'espace mémoire. Il est vrai que les accès mémoire ont une importance capitale pour le temps de traitement des opérations sur DSP.

L'utilisation du QDMA (*Quick Direct Memory Access*) se révèle particulièrement efficace pour transférer les données rapidement d'une mémoire à une autre (interne-externe). Ce co-processeur peut être utilisé pour effectuer des transferts de mémoire en parallèle des calculs réalisés par le CPU et surtout ne nécessite aucune instruction CPU une fois configuré. Comme nous pouvons le remarquer sur le tableau 3.4, l'utilité du QDMA se confirme pour les transferts mémoires externe-interne et externe-externe, mais pas interne-interne, ceux-ci étant plus rapides quand ils sont réalisés uniquement par l'ALU du CPU. Par la suite les accès mémoire réalisés par le QDMA seront considérés comme des transferts à partir ou vers une mémoire externe.

Le QDMA est une alternative à l'EDMA (Enhanced Direct Memory Access, préconisé pour les transferts entrées/sorties (FIFO) - mémoires), les deux permettant la réalisation des transferts de mémoire en parallèle des calculs effectués par le CPU. Dans notre cas le QDMA s'utilise pour rapatrier les données en mémoire interne, sans parallélisme entre les calculs. L'EDMA est quant à lui utilisé pour transférer les données sur une FIFO ou médium de communication. La différence entre l'EDMA et le QDMA se situe au niveau des registres. En fait le QDMA est plus approprié pour transférer des

recopie d'un Bloc 64*64 entiers de 32bits	Nb cycles CPU	
	CPU	QDMA
Mémoire interne -> interne	2 864	8 520
Mémoire interne -> externe	104 464	5 576
Mémoire externe -> interne	17 128	4 976
Mémoire externe -> externe	138 680	17 400

TAB. 3.4 – Taux de transfert entre une mémoire interne et une mémoire externe avec ou sans QDMA

données en "bloc", pendant que le code s'exécute sur le CPU. Un transfert QDMA requiert seulement de 1 à 5 cycles CPU pour le lancer, dépendant du nombre de registres à configurer. Un transfert QDMA typique se fait en affectant 4 des valeurs de paramétrage à leurs registres, suivi par la cinquième à son pseudo-registre correspondant, celui-ci démarrant le transfert. Tous les registres du QDMA retiennent leur valeur après la requête. Donc si un second transfert est demandé, avec n'importe laquelle des mêmes valeurs de paramétrage, celles-ci n'ont pas besoin d'être réécrites. Seuls les registres à modifier doivent être réécrits, avec le paramètre final affecté au pseudo-registre approprié pour lancer le transfert. En conséquence, des requêtes QDMA à la suite peuvent nécessiter seulement 1 cycle pour démarrer le transfert.

Sous SynDEx, pour accélérer les temps de calcul sur DSP, chaque transfert mémoires externe-interne et externe-externe est donc réalisé avec le QDMA (avec les transferts en *Array Synchronized*) et est considéré comme une opération SynDEx. Ainsi, dans l'exécutif produit par SynDEx, les opérations de *memcpy()* (recopie des buffers) ont été remplacées par des *QDMA32copy()*, réalisant la recopie des buffers mis en mémoire externe. Si nous comparons une recopie réalisée uniquement par le CPU (*memcpy()*) et un transfert QDMA, de mémoire interne à mémoire interne, le *memcpy()* est 3 fois plus efficace. Cependant, dès qu'une mémoire externe entre en jeu, les transferts QDMA sont jusqu'à 25 fois plus rapides que les transferts réalisés par un *memcpy()* utilisant uniquement le CPU. D'autres fonctions ont été aussi créées, réalisant la recopie d'un bloc 8x8 dans une image, d'un bloc 8x8 d'une image vers une autre image, ou encore celle d'un bloc 8x8 d'une image dans un bloc.

3.4 Conclusion

Dans ce chapitre, nous avons présenté les différentes étapes de notre chaîne de prototypage et la mise en place de la gestion des différents projets par des logiciels adéquats pour une collaboration des développements au sein d'un groupe de travail. Grâce à cette chaîne, différentes équipes peuvent maintenant collaborer de manière efficace et minimiser le gouffre qu'il peut y avoir entre les concepteurs d'un algorithme et les intégrateurs sur une plate-forme multi-composants. Par ailleurs, de nombreux noyaux ont été développés et optimisés pour différents types de processeurs (pentium, C62x, C64x, FPGA) et média (SDB, CP, bus PCI suivant le modèle RAM ou SAM, BIFO, TCP).

Cette chaîne de prototypage sert au développement de différentes applications dans le domaine du traitement des images et des télécommunications, nous les présentons dans la seconde partie de ce mémoire.

Chapitre 4

Optimisation mémoire dans AAA

4.1 Introduction

Dans le chapitre précédent, nous avons présenté une chaîne de prototypage complète dans laquelle notre contribution a consisté à venir l'interfacer avec SynDEx. En tant qu'utilisateur, nous avons constaté un défaut majeur lors de la génération de code : la taille mémoire allouée. Nous avons donc décidé de collaborer directement avec l'INRIA dans l'outil SynDEx pour y intégrer des optimisations mémoires. La recherche de ces optimisations est l'objet de la présente thèse.

Dans le chapitre 2, nous avons montré que SynDEx, après la mise à plat du graphe flot de données, alloue un buffer sur chaque sortie de chaque opération atomique. Dans un contexte temps réel embarqué, il est nécessaire de trouver une méthode adéquate des minimisations des buffers. Dans la littérature, nous n'avons pas trouvé d'existant sur la minimisation de la mémoire par rapport à des outils de conception système de haut niveau tel que SynDEx. Nous nous sommes donc dirigés vers des travaux existants dans le domaine de la compilation ou dans le domaine de la résolution d'emploi du temps. Des solutions ont donc été trouvées grâce à la théorie des graphes et notamment du coloriage du graphe pour minimiser le nombre de buffers utiles : nous montrerons ici deux méthodes efficaces.

Dans une seconde étape, nous nous sommes intéressés à la minimisation mémoire pendant l'ordonnancement et la distribution, dans cette partie nous n'avons pas eu le temps d'aller au bout de cette méthode mais nous donnons les directions de nos prochaines recherches. Les algorithmes génétiques constituent une bonne approche pour minimiser une fonction de coût multi-critères.

4.2 Coloriage de graphe

4.2.1 Introduction

Un processus fondamental en mathématiques est la séparation d'un ensemble d'objets en classes suivant certaines règles. Ces règles permettent par exemple de préciser pour toute paire d'objets s'ils peuvent appartenir à une même classe. Il est naturel de modéliser ce problème par un graphe. Les objets considérés sont les sommets de ce graphe,

deux sommets sont adjacents si et seulement si ils ne peuvent être contenus dans une même classe.

Exemple 1 Organisation du planning des examens de fin d'année.

Une bonne organisation des examens de fin d'année impose (au moins !) qu'un étudiant puisse assister à tous les examens pour lequel il est inscrit. On représente ce problème par un graphe dont les sommets sont les matières, deux matières sont adjacentes si et seulement si des étudiants y sont inscrits en même temps. Un ensemble d'épreuves qui peuvent se dérouler en parallèle est un stable *du graphe (on dit aussi un ensemble* indépendant*). On tentera d'optimiser le planning des épreuves en minimisant le nombre de sessions parallèles, c'est-à-dire que l'on cherchera une partition des sommets du graphe en un nombre minimum de stables. C'est ce que nous appellerons une* coloration *du graphe.*

4.2.1.1 Définitions et notations

Des présentations générales et classiques sur les graphes peuvent être trouvées dans [Ber70, BM79]. Un graphe, dans cet exposé, sera toujours *simple* (pas d'arête multiple entre 2 sommets, ni de boucle sur le même sommet) et *non-orienté* et sera noté $G = (V, E)$ (V ou $V(G)$ pour les sommets, E ou $E(G)$ pour les arêtes), n désignera en général le nombre de sommets, tandis que m précisera le nombre d'arêtes. Un *coloriage* de G est une application ϕ telle que :

$$\begin{cases} \phi \ : \ V(G) \rightarrow C \ tq \\ x \ et \ y \in E(G) \Rightarrow \phi(x) \neq \phi(y) \end{cases}$$

C (ensemble de couleurs) est en général l'ensemble des entiers de 1 à k et ϕ est alors un $k-coloriage$. Si G admet un $k-coloriage$, il admet aussi un $k+1-coloriage$ (toutes les couleurs ne sont pas nécessairement utilisées). Le plus petit entier k pour lequel il existe un $k-coloriage$ de G est le *nombre chromatique* de G, il est noté $\chi(G)$.

4.2.1.2 Coloriage et complexité

Les graphes $1-coloriables$ n'ayant qu'un intérêt restreint (il s'agit de l'ensemble de tous les stables), le résultat suivant caractérise les graphes $2-coloriables$ (on dit aussi *bipartis*).

Théorème 4.1 *Un graphe est biparti si et seulement si tous ses cycles sont pairs.*

Preuve 4.2 *Si G contient un cycle impair C, alors $\chi(G) \geq \chi(C) = 3$. Inversement, si G n'a aucun cycle impair, l'ensemble des sommets situés à distance paire d'un sommet v quelconque est un stable, de même pour l'ensemble des sommets situés à distance impaire. On a ainsi une partition naturelle des sommets de G en deux stables, c'est à dire un $2-coloriage$ de G.*

Pour $k \geq 3$, la question de savoir si un graphe G possède un $k-$coloriage s'avère un problème difficile. C'est un des problèmes NP-complets classiques de la théorie des graphes (voir [GJ79] pour plus d'informations sur ce sujet).

La théorie de la complexité repose sur la définition de classes de complexité permettant de classer les problèmes en fonction de la complexité des algorithmes qui existent pour les résoudre. Parmi les classes les plus courantes, on distingue :

Classe P : un problème de décision est dans P s'il peut être décidé (réponse oui) par un algorithme déterministe en un temps polynomial par rapport à la taille de l'instance. On qualifie alors le problème de polynomial.

Classe NP : c'est la classe des problèmes de décision pour lesquels la réponse oui peut être décidée par un algorithme non-déterministe en un temps polynomial par rapport à la taille de l'instance.

Un problème est dit polynomial de classe **P** si il existe un algorithme pour résoudre ce problème dont la durée maximale (nombre d'opérations à effectuer dans le pire des cas) est majorée par une fonction polynomiale de la taille des données.

Un problème est dit de classe **NP** s'il existe un algorithme polynomial pour vérifier la validité d'une solution candidate. Par exemple, considérons le problème de savoir si un graphe contient un cycle hamiltonien (un cycle passant par tous les sommets une fois et une seule). Ce problème est de classe **NP**, car on peut vérifier en $\mathcal{O}(n)$ qu'un ordonnancement donné des sommets induit un cycle et a le bon nombre de sommets.

Un problème est dit **NP**-complet si tout problème **NP** peut se ramener par une transformation polynomiale à ce problème. Il est conjecturé qu'il n'existe pas d'algorithme polynomial pour résoudre un problème **NP**-complet.

Théorème 4.3 *Les problèmes suivants sont **NP**-complets :*

- *3-coloriage*
  donnée : *un graphe G*
  question : *G admet-il un 3-coloriage ?*
- *3-coloriage des graphes planaires*
  donnée : *un graphe planaire G dont tous les sommets ont un degré au plus 4*
  question : *G admet-il un 3-coloriage ?*

Les affirmations ci-dessus pourraient nous inciter à viser des résultats plus modestes. Par exemple, peut-on trouver un algorithme de coloriage approché raisonnable ? Hélas M.R. GAREY et D.S. JOHNSON ([GJ79]) montrent qu'il est aussi difficile à obtenir qu'un algorithme de coloriage optimal !

Théorème 4.4 *Soient r et d deux constantes avec $r < 2$. S'il existe un algorithme polynomial A pour colorier un graphe quelconque G en au plus $r.\chi(G)+d$ couleurs, alors il existe un algorithme polynomial permettant de colorier ce graphe en $\chi(G)$ couleurs.*

Il est donc a priori désespéré de vouloir résoudre efficacement le problème du coloriage d'un graphe (sauf si $P = NP$!). On peut, dès lors, attaquer ce problème sous divers angles : donner des bornes "raisonnables" du nombre chromatique, décrire des heuristiques efficaces permettant de l'approcher, étudier la classe des graphes pour laquelle le problème est polynomial...
Supposons que nous disposions d'un algorithme A de coloriage d'une famille de graphes $\mathcal{F}$ (A ne fournissant pas nécessairement un coloriage optimal des graphes de la famille), il est intéressant de mesurer l'écart éventuel entre le nombre chromatique d'un graphe

et le nombre de couleurs obtenues par application de l'algorithme A à ce graphe $\chi_A(G)$. On s'intéressera donc au rapport $\chi_A(G)/\chi(G)$. On notera alors

$$\chi_A(n, \mathcal{F}) = \max_{G \in \mathcal{F}} \{\chi_A(G)/\chi(G), \mid V(G) \mid \leq n\}$$

et, lorsque $\mathcal{F}$ est la famille de tous les graphes

$$\chi_A(n) = \max_{G} \{\chi_A(G)/\chi(G), \mid V(G) \mid \leq n\}.$$

Clairement on a $\chi_A(n) \leq n$ pour tout algorithme de coloriage A.

4.2.1.3 Des bornes sur le nombre chromatique

Donnons-nous un graphe G quelconque et essayons de le colorier de la manière suivante : affectons la couleur 1 à un premier sommet v_1 et considérons un sommet v_2. Si v_2 est adjacent à v_1 affectons lui la couleur 2, sinon la couleur 1. Examinons ainsi les sommets $v_1, v_2 ....v_i$ et imaginons que ces sommets soient déjà coloriés en utilisant k couleurs. S'il n'y a pas d'autres sommets dans le graphe, nous avons un $k-coloriage$ de G, sinon il existe au moins un sommet v_{i+1} non encore colorié. Si les couleurs présentes dans le voisinage de ce sommet ne couvrent pas l'ensemble $\{1...k\}$, on peut affecter une des couleurs manquantes au sommet v_{i+1}, sinon l'on affecte la couleur $k+1$ (donc on ajoute une nouvelle couleur). On poursuit alors jusqu'à épuisement des sommets. L'algorithme décrit est l'algorithme *glouton* (*greedy*). C'est une heuristique classique qui nous donne une borne immédiate sur ce nombre chromatique :

Théorème 4.5 *Le nombre chromatique d'un graphe est au plus son degré maximum Δ plus un (ie $\chi(G) \leq \Delta + 1$).*

Preuve 4.6 *En effet, à chaque étape de l'algorithme ci-dessus, il y a au plus Δ couleurs présentes dans le voisinage du sommet considéré.*

En 1941 ce résultat a été amélioré par R.L.BROOKS [Bro41] :

Théorème 4.7 *Soit G un graphe quelconque, alors $\chi(G) \leq \Delta + 1$. Si $\Delta = 2$, on a l'égalité si et seulement si G est un cycle impair. Si $\Delta \geq 3$ on a l'égalité si et seulement si G est un graphe complet sur $\Delta + 1$ sommets.*

Le nombre maximum de sommets d'un sous-graphe *complet* de G, $\omega(G)$ est clairement une borne inférieure du nombre chromatique puisque dans un coloriage quelconque tous les sommets d'une *clique* doivent recevoir une couleur distincte. Dans la théorie des graphes, une clique est un sous-graphe complet, c'est-à-dire un sous-graphe dont les sommets sont tous connectés deux à deux. On a donc (sauf pour les cliques et les cycles impairs) :

$$\omega(G) \leq \chi(G) \leq \Delta(G).$$

Entre ces deux bornes, on peut avoir n'importe quelle valeur pour $\chi(G)$. En effet, un graphe biparti vérifie $\omega(G) = \chi(G) = 2$, tandis que le degré maximum est quelconque et, d'un autre côté, on a le théorème suivant [Zyk52] :

Théorème 4.8 *Pour tout k, il existe un graphe G_k $k-$chromatique sans triangle (ie $\chi(G_k) = k$ et $K_3 \subseteq G_k$).*

Preuve 4.9 *Supposons que nous connaissions $G_1, G_2..., G_{k-1}$, construisons G_k. Soit V un ensemble de $\mid V(G_1) \mid . \mid V(G_2) \mid ... \mid V(G_{k-1}) \mid$ nouveaux sommets qui correspondent à tous les choix possibles d'un sommet dans chacun des $G_1, G_2..., G_{k-1}$. On obtient G_k, à partir de $G_1, G_2..., G_{k-1}$ en joignant un sommet de V aux sommets qui lui correspondent dans $G_1, G_2..., G_{k-1}$. Le graphe G_k est $k-$coloriable. S'il était $(k-1)-$coloriable, alors on pourrait trouver dans G_1 un sommet v_1 de couleur i_1, dans G_2 un sommet v_2 de couleur i_2 distincte de i_1, dans G_3 un sommet v_3 de couleur i_3 distincte de i_1 et i_2 ... etc. Le sommet de V relié aux sommets $v_1, v_2...v_{k-1}$ ne pourrait alors avoir aucune des couleurs $1..k-1$, d'où contradiction.*

4.2.2 Aspects algorithmiques.

Comme déjà remarqué, un coloriage est une partition des sommets en stables. Ceci va nous conduire à une heuristique de coloriage basée sur les indépendants d'un graphe. Dans une deuxième partie nous reviendrons sur l'approche séquentielle (algorithme glouton).

4.2.2.1 Heuristique des indépendants.

Si G admet un $k-$coloriage, on a une partition des sommets de G en k indépendants $V_1, V_2...V_k$. Inversement une telle partition est un coloriage de G.

Théorème 4.10 *Pour tout coloriage de G en k couleurs $V_1, V_2...V_k$, il existe un $l-$coloriage $V_1^{'}, V_2^{'}...V_l^{'}$ tel que $l \leq k$ et $V_1^{'}$ est un indépendant maximal de G.*

Preuve 4.11 *Si V_1 n'est pas un indépendant maximal de G alors soit $V_1^{'}$ un indépendant maximal contenant V_1. Soient $V_i^{'} = V_i - V_1^{'}$ $(2 \leq i \leq k)$, certains de ces ensembles peuvent être vides, on obtient donc un coloriage de G en au plus k couleurs tel que $V_1^{'}$ soit indépendant maximal.*

Il existe donc un ensemble indépendant maximal S tel que

$$\chi(G) = \chi(G - S) + 1$$

et S étant toujours un stable de G, on obtient :

$$\chi(G) = \min_{S \subseteq V(G)} \chi(G - S) + 1$$

Si un graphe est $k-$*chromatique*, on peut le colorier en considérant un stable maximal V_1 (auquel on affecte la couleur 1), puis un stable maximal V_2 de $G - V_1$ (que l'on colorie 2) etc. Il faut donc considérer tous les stables maximaux S de G, puis ceux de $G - S$ etc. Cette remarque nous conduit à l'algorithme 4.1 (dû à N.Christofides [Chr75]) .
L'intérêt de cet algorithme est qu'il nous donne un coloriage optimal de notre graphe au prix fort : il faut en effet pouvoir exhiber tous les stables maximaux du graphe. La

Algorithme 4.1 Algorithme de Christofides

```
début
  k := 0
  G_0 := ∅
  tant que ∀H ∈ G_k V(H) ≠ V(G)
    F := ∅
    pour tout H ∈ G_k faire
      pour tout stable maximal S de G − H faire
        F := F ∪ {H + S}
        k := k + 1
        G_k := sous-graphes l−chromatiques maximaux de F
      fin pour tout
    fin pour tout
  fin tant que
  k est le nombre chromatique de G
fin
```

technique de *backtracking* peut être optimisée de diverses manières, mais la complexité sera toujours exponentielle ($\mathcal{O}(mn2.445^n)$ au pire).
Au lieu d'engendrer tous les stables maximaux, on peut espérer obtenir une bonne approximation des coloriages optimaux en n'engendrant que les stables **maximums** (stables ayant le plus de sommets dans G, traditionnellement $\alpha(G)$ dénote ce paramètre). Malheureusement, la recherche du stable maximum d'un graphe est aussi un problème NP-complet [GJ79] ! Finalement, on peut rechercher non pas un stable maximum de G, mais une approximation de ce stable maximum (Algorithme 4.2).

Algorithme 4.2 Algorithme de Recherche d'un stable maximum approché

```
début
  U_1 := U
  W := ∅
  tant que U_1 ≠ ∅
    trouver un sommet u de degré minimum de U_1
    W := W ∪ u
    U_1 := U_1 − u − {v | v ∈ U_1 uv ∈ E(G)}
  fin tant que
  W est le stable cherché
fin
```

L'utilisation de cette stratégie nous donne un comportement en $O(n/logn)$ pour la fonction d'évaluation de performance $\chi_A(n)$. L'une des stratégies les plus efficaces dans la recherche d'une "bonne" coloration (F.T.LEIGHTON [Lei79]) utilise en partie cette dernière méthode.

4.2.2.2 Heuristiques séquentielles

L'algorithme glouton (Algo. 4.3) se traduit finalement par la donnée d'un ordre sur les sommets du graphe, $v_1 \leq v_2 \leq ... \leq v_n$. On colorie alors séquentiellement les sommets en affectant à v_i la première couleur disponible (ie la première couleur n'apparaissant pas parmi ses voisins appartenant à $\{v_1, v_2....v_{i-1}\}$).
L'utilisation de cette procédure se justifie par le théorème ci-dessous :

Algorithme 4.3 Algorithme Séquentiel ou Glouton

```
début
  φ(v_1) := 1
  pour i allant de 2 à n faire
    φ(v_i) := min_{k≥1}{∀v_j, 1 ≤ j ≤ i − 1, v_i v_j ∈ E(G), φ(v_j) ≠ k}
  fin pour
fin
```

Théorème 4.12 *Pour tout graphe G, il existe un ordre O des sommets pour lequel l'algorithme glouton produit un coloriage optimal.*

Preuve 4.13 *Considérons en effet un coloriage optimal de G, c'est-à-dire une partition en stables S_1, S_2....S_k. Ordonnons alors les sommets de G en classant d'abord les sommets de S_1 (dans n'importe quel ordre), puis ceux de S_2 .... puis ceux de S_k. L'algorithme glouton appliqué à cet ordonnancement des sommets nous redonne le coloriage $S_1, S_2....S_k$.*

On peut alors se demander quels sont les graphes pour lesquels n'importe quel ordre produit un coloriage optimal par application de l'algorithme glouton. Il n'y a pas, en fait, de réponse vraiment intéressante puisqu'un graphe G auquel on ajoute un graphe complet ayant au moins $\chi(G)$ sommets aura toujours cette particularité. En effet, on a le théorème suivant (où un graphe est dit *k-parti-complet* s'il existe une partition des sommets en k stables $S_1, S_2...., S_k$, deux sommets appartenant à des stables distincts étant toujours reliés) :

Théorème 4.14 *Si G est un graphe $k-$parti-complet, tout ordonnancement de ses sommets conduit à un coloriage optimal par application de l'algorithme glouton.*

Il est facile de trouver une borne supérieure $u_S(G; v_1, v_2....v_n)$ du nombre de couleurs $\chi_s(G)$ produit par l'algorithme séquentiel appliqué au graphe G avec l'ordonnancement $v_1 \leq v_2 \leq .... \leq v_n$. Le sommet v_i pourra toujours être colorié i et donc on a $\phi(v_i) \leq i$. D'un autre côté, au moins l'une des $d(v_i) + 1$ premières couleurs peut être affectée à v_i ($d(v_i)$ désigne le *degré* de v_i, ici le nombre de sommets auxquels il est adjacent). On a donc :

$$\phi(v_i) \leq \min\{i, d(v_i) + 1\}$$

et ainsi

$$\chi_s(G) \leq u_S(G; v_1, v_2....v_n) = \max_{1 \leq i \leq n} \min\{i, d(v_i) + 1\}$$

Ce qui nous donne une nouvelle borne supérieure du nombre chromatique d'un graphe (notons qu'en pratique le comportement de cet algorithme est meilleur).
Il est assez naturel d'essayer de trouver de "bons" ordonnancements pour cet algorithme. Dans [PW67], M.B.POWEL et D.J.A.WELSH ordonnent les sommets par degrés décroissants ($d(v_1) \geq d(v_2) \geq ..... \geq d(v_n)$) et décrivent ainsi une variante de l'algorithme séquentiel (LFS algorithm, *largest first*).

Théorème 4.15 *Soit $u_1, u_2...., u_n$ un LF-ordonnancement des sommets de G (ordonnancement par degrés décroissants) et soit $u_{LF}(G) = u_S(G; u_1, u_2....u_n)$. Alors*

$$u_{LF}(G) = \min u_S(G; v_1, v_2....v_n)$$

le minimum étant pris sur tous les ordonnancements des sommets de G.

Preuve 4.16 *Il suffit de remarquer que, si pour un ordre des sommets $v_1, v_2 ...., v_n$, il existe un indice i $(1 \leq i \leq n-1)$ pour lequel $d(v_{i+1}) > d(v_i)$ alors*

$$u_S(G; v_1, v_2 ...., v_i, v_{i+1} ....v_n) \geq u_S(G; v_1, v_2 ...., v_{i+1}, v_i ....v_n)$$

Malheureusement, cet algorithme (comme d'ailleurs toutes les heuristiques de coloriage) peut avoir un mauvais comportement. Considérons en effet le graphe biparti suivant $G_{2n} = (V_{2n}, E_{2n})$

$$V_{2n} = \{u_i, v_i : i = 1, 2..n\}, E_{2n} = \{u_i v_j : i, j = 1, 2..n \; i \neq j\}$$

et l'ordre des sommets

$$u_1, v_1, u_2, v_2 ....u_n, v_n.$$

Puisque tous les degrés sont égaux, cet ordre est bien de type LF. L'application de l'algorithme séquentiel sur ce graphe muni de cet ordre conduit à affecter la couleur i aux sommets u_i et v_i, et il faudra donc n couleurs pour colorier ce graphe !
Revenons à l'algorithme séquentiel classique, nous avons remarqué que la couleur attribuée à v_i était majorée par

$$\phi(v_i) \leq \min\{i, d(v_i) + 1\}$$

En fait, ce n'est pas le degré du sommet v_i qu'il faut prendre en compte, mais bien le degré de v_i dans le sous-graphe engendré par les sommets déjà coloriés, c'est à dire les sommets $v_1, v_2, ...v_{i-1}$. Notons donc $d_i(v_i)$ le nombre de voisins de v_i dans l'ensemble $\{v_1, v_2, ..., v_{i-1}\}$. On aura alors

$$\chi_s(G) \leq u_S^{'}(G; v_1, v_2 ....v_n) = \max_{1 \leq i \leq n} d_i(v_i) + 1.$$

Cette dernière idée est à la base de l'algorithme séquentiel SL (*Smallest Last*) de J.D.ISAACSON, G.MARBLE et D.W.MATULA [IMM72] La procédure suivante trouve un ordonnancement des sommets qui minimise $u_S^{'}(G; v_1, v_2 ....v_n)$.

1. v_n est un sommet de degré minimum de G
2. $i = n-1, n-2, ..., 1$, v_i est un sommet de degré minimum dans le sous-graphe de G induit par les sommets de $V - \{v_n, v_{n-1}, ...v_{i+1}\}$.

Par construction de la séquence $v_1, v_2, ..., v_n$, nous avons

$$d_i(v_i) = \min_{1 \leq j \leq i} d_i(v_j)$$

soit alors

$$u_{SL}(G) = 1 + \max_{1 \leq i \leq n} \min_{1 \leq j \leq i} d_i(v_j)$$

quand $v_1, v_2, ..., v_n$ est un SL ordonnancement des sommets de G. On peut prouver que

$$\chi_S(G) \leq u_{SL}(G) \leq u_{LF}(G).$$

L'inégalité de G.SZEKERES and H.S.WILF [SW68] est issue de cette technique :

$$\chi(G) \leq 1 + \max_{H \subseteq G} \min_{v \in V(H)} d_H(v).$$

L'idée de regarder le degré du sommet courant dans le sous-graphe des sommets déjà colorié conduit à une preuve du théorème de R.L.BROOKS [Bro41], due à L.LOVASZ [Lov75]. Supposons en effet que G soit connexe et qu'il existe un sommet v de degré strictement inférieur au degré maximum (i.e. G n'est pas régulier). On peut alors ordonner les sommets de G dans l'ordre inverse de leur distance à ce sommet v. Pour un tel ordre O, un sommet quelconque sera relié à au plus $\Delta - 1$ sommets qui le précèdent. L'algorithme séquentiel conduit alors à un $\Delta-$coloriage de G. En fait L.LOVASZ remarqua que même si le graphe est régulier une argumentation similaire convient, pourvu que le graphe soit 3−connexe. Des arguments spécifiques permettent alors de régler les cas restants.
Pour terminer cette étude succincte des heuristiques séquentielles, on peut remarquer qu'une couleur affectée à un sommet n'est jamais remise en cause dans la suite du déroulement de l'algorithme. Cette restriction peut être superflue. En effet, à un coloriage donné d'un graphe G en $k-$couleurs, on peut associer toute une famille de coloriages en au plus k couleurs de ce graphe. Il suffit pour cela de considérer les *composantes bicolores*, c'est-à-dire les sous-graphes de G engendrés par deux couleurs quelconques. Notons $G_{p,q}$ le sous-graphe de G engendré par les couleurs p et q, si ce sous-graphe n'est pas connexe, l'échange des couleurs p et q sur l'une de ses composantes conduit à un nouveau coloriage de G. Comme de nouvelles couleurs ne sont pas introduites, on a ainsi un nouveau $k-$coloriage de G. En réitérant de toutes les façons possibles ce procédé, on obtient une famille de $k-$coloriage de G, il est clair qu'il peut se produire, à l'issue de telles permutations, que deux classes de couleurs n'en fassent plus qu'une seule (si leur réunion est un stable de G). L'algorithme séquentiel peut être alors modifié comme dans l'algorithme 4.4.

Algorithme 4.4 Algorithme Séquentiel avec échanges

```
début
  k := 0
  G_0 := ∅
  tant que ∀H ∈ G_k V(H) ≠ V(G)
    F := ∅
    pour tout H ∈ G_k faire
      pour tout stable maximal S de G − H faire
        F := F ∪ {H + S}
      fin pour
    k := k + 1
    G_k := graphes maximaux de F
    fin pour
  fin tant que
  k est le nombre chromatique de G
fin
```

On peut montrer qu'un tel algorithme permet de colorier de manière optimale un graphe biparti. Il existe cependant des graphes pour lesquels le résultat est loin d'être optimal. ($\chi_A(n) = O(n)$). Il peut même se produire que la procédure d'échange à une

étape produise un coloriage moins bon que l'application de l'algorithme séquentiel sur le même ordre des sommets !

4.2.2.3 Autres méthodes

Une direction radicalement différente est de voir le problème du coloriage comme une minimisation d'une fonction. Supposons que l'on veuille trouver un $k-$coloriage d'un graphe G. Une solution envisageable s de ce problème est donc une partition en k sous-ensembles de sommets de G, $s = (S_1, S_2, ..., S_k)$. Une telle partition sera un coloriage si chacun des S_i est un stable du graphe. On peut donc chercher à minimiser la fonction

$$f(s) = \Sigma_{i=1}^{k} \mid E(S_i) \mid$$

où $E(S_i)$ désigne l'ensemble des arêtes du sous-graphe de G engendré par S_i. On aura un $k-$coloriage de G si et seulement si $f(s) = 0$. Si s est une solution envisageable, s' est une autre solution envisageable voisine de s si on peut obtenir s' à partir de s en transférant un sommet du stable S_i à un stable S_j. Nous avons ainsi tous les ingrédients nécessaires pour utiliser ces méthodes d'optimisation (cf [Hd87]).

L'idée étant alors d'utiliser les méthodes classiques d'optimisation comme le recuit simulé, la méthode Tabou (voir [CHdW87]), ainsi que les algorithmes génétiques qui vont être présentés dans la section suivante.

4.3 Les algorithmes génétiques

L'utilisation des algorithmes génétiques dans la résolution de problèmes est à l'origine le fruit des recherches de John HOLLAND et de ses collègues et élèves de l'Université du Michigan qui ont, dès 1960, travaillé sur ce sujet. La nouveauté introduite par ce groupe de chercheurs a été la prise en compte de l'opérateur de *crossing over* en complément des mutations. Et c'est cet opérateur qui permet le plus souvent de se rapprocher de l'optimum d'une fonction en combinant les gènes contenus dans les différents individus de la population. Le premier aboutissement de ces recherches a été une publication majeure en 1975 [Hol75].

4.3.1 Présentation

Les algorithmes génétiques reprennent la théorie de DARWIN : la sélection naturelle de variations individuelles. Les individus les plus adaptés (*the fitness*) tendent à survivre plus longtemps et à se reproduire plus aisément. L'amélioration de la population est très rapide au début (recherche globale) ; de plus en plus lente à mesure que le temps passe (recherche locale). La convergence de la valeur moyenne de la fonction d'adaptation a tendance à se rapprocher de celle de l'individu le plus adapté pour une uniformisation croissante de la population. Le temps de calcul des algorithmes génétiques croît en $n * ln(n)$, n étant le nombre de variables.

Les algorithmes génétiques [Gol94] étant basés sur des phénomènes biologiques, il convient de rappeler au préalable quelques notions de génétique.

Les organismes vivants sont tout d'abord constitués de cellules comportant des chromosomes qui correspondent en fait à des chaînes d'ADN. L'élément de base de

ces chromosomes (le caractère de la chaîne d'ADN) est un gène. Sur chacun de ces chromosomes, une suite de gènes constitue une chaîne qui code les fonctionnalités de l'organisme (la couleur des yeux...). La position d'un gène sur le chromosome est son locus. L'ensemble des chromosomes correspond au génome de l'individu. Les différentes versions d'un même gène sont appelées allèles.

On utilise aussi, dans les algorithmes génétiques, le principe de l'évolution des espèces émis par DARWIN ; il suppose qu'au fil du temps, les gènes conservés au sein d'une population donnée sont ceux qui sont plus adaptés aux besoins de l'espèce et à son environnement.

4.3.2 Principe

La génétique a mis en évidence l'existence de plusieurs opérations au sein d'un organisme donnant lieu au brassage génétique. Ces opérations interviennent lors de la phase de reproduction lorsque les chromosomes de deux organismes fusionnent.

Ces opérations sont imitées par les algorithmes génétiques afin de faire évoluer de manière progressive les populations de solutions (Fig. 4.1).

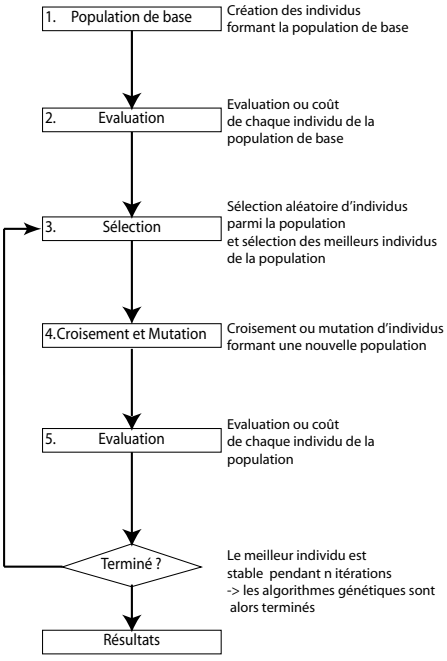


FIG. 4.1 – Schéma global des algorithmes génétiques

* **Les sélections** : Pour déterminer quels individus sont plus enclins à obtenir les meilleurs résultats, une sélection est opérée. Ce processus est analogue à un processus de sélection naturelle, où les individus les plus adaptés gagnent la compétition de la reproduction, tandis que les moins adaptés meurent avant la reproduction, ce qui améliore globalement l'adaptation.

* **Les *crossing over* ou recombinaisons ou hybridations** : Lors de cette opération, deux chromosomes s'échangent des parties de leurs chaînes, pour donner de

nouveaux chromosomes. Ces *crossing-over* peuvent être simples ou multiples. Dans le premier cas, les deux chromosomes se croisent et s'échangent des portions d'ADN en un seul point (locus). Dans le deuxième cas, il y a plusieurs points de croisement. Pour les algorithmes génétiques, c'est cette opération (le plus souvent sous sa forme simple) qui est prépondérante. Sa probabilité d'apparition lors d'un croisement entre deux chromosomes est un paramètre de l'algorithme génétique. En règle générale, on fixe la proportion d'apparition à 70%.

* **Les mutations** : De façon aléatoire un gène peut, au sein d'un chromosome, être substitué à un autre. De la même manière que pour les *crossing-over*, on définit ici un taux de mutation lors des changements de population qui est généralement compris entre 0,1% et 1%. Il est nécessaire de choisir pour ce taux une valeur relativement faible de manière à ne pas tomber dans une recherche aléatoire et conserver le principe de sélection et d'évolution. La mutation sert à éviter une convergence prématurée de l'algorithme. Par exemple lors d'une recherche d'extremum la mutation sert à éviter la convergence vers un extremum uniquement local.

4.3.2.1 Exemples d'applications

Le problème du *voyageur de commerce* est un problème classique des algorithmes génétiques. Son sujet concerne les trajets d'un voyageur de commerce. Celui-ci dispose de plusieurs points où s'arrêter et le but de l'algorithme est d'optimiser son trajet de façon à ce que celui-ci soit le plus court possible. Dans le cas où huit points d'arrêt existent, cela est encore possible par énumération (2520 possibilités [pour n arrêts, $n \geq 3$, il y a $(n-1)!/2$ chemins possibles) mais ensuite, l'augmentation du nombre d'arrêts fait suivre une croissance exponentielle au nombre de possibilités.

Par le biais d'algorithmes génétiques, il est possible de trouver des chemins relativement corrects. De plus, l'encodage de ce type de problèmes sous forme d'algorithme génétique reste relativement aisé. L'idée de base est de prendre la longueur comme fonction d'adaptation ou de coût d'un chemin. Puis une fois que le locus où doit avoir lieu le *crossing-over* est sélectionné, on effectue le croisement de deux chemins A et B. Ce croisement scinde chaque chemin en 2 parties. On garde pour chaque chemin les itinéraires respectifs à leurs premières parties jusqu'à ce locus. On place ensuite dans le chemin A les villes qui ne sont pas présentes dans la première partie de A, dans l'ordre où elles apparaissent dans le chemin B. On fait de même pour le chemin B avec le chemin de A.

Soit un itinéraire qui contient 9 villes, et supposons que l'on croise les deux chemins suivants (un chiffre représente une ville). Si l'on croise ces deux chemins après le locus 4, on obtient les deux chemins A' et B'.

Chemin	*Codage*
A	1234 :56789
B	4163 :98257
A'	1234 69857
B'	4163 25789

En partant de ce principe, de nombreux algorithmes génétiques ont été développés, chacun utilisant différentes variantes afin de se rapprocher le plus possible de l'optimum

dans tous les cas. Il existe d'ailleurs un concours sur internet qui propose de développer un algorithme à même de trouver le meilleur chemin sur un problème de voyageur de commerce de 250 villes.

4.3.3 Schéma simplifié

1. **Population de base générée aléatoirement**
   - n chaînes de caractères ou de bits
   - 1 chaîne correspond à 1 chromosome.
2. **Évaluation**
   - à chaque chaîne, une note correspondant à son adaptation au problème.
3. **Sélection**
   - tirage au sort de $n/2$ (paramétrable) couples de chaînes. Chaque chaîne a une probabilité d'être tirée proportionnelle à son adaptation au problème.
   - Optimisation possible : si l'individu le plus adapté n'a pas été sélectionné, il est copié d'office dans la génération intermédiaire à la place d'un individu choisi aléatoirement.
4. **Croisement (hybridation) et mutation**
   Chaque couple donne 2 chaînes filles.
   - Crossing-over :
     - Probabilité : 70 % (paramétrable). Emplacement du *crossing-over* choisi aléatoirement.
     - Croisement en 2 points plus efficace.
     - Exemple :
       Chaînes parents : A : 00110100 ; B : 01010010
       Chaînes filles : A : 00010010 ; B : 01110100
   - Mutations des chaînes filles :
     - Probabilité : de 0,1 à 1% (paramétrable).
     - Inversion d'un bit au hasard ou remplacement au hasard d'un caractère par un autre.
     - Probabilité fixe ou évolutive (auto-adaptation). On peut prendre probabilité = 1/nombre de bits.

Un des avantages incontestables des algorithmes génétiques est de pouvoir trouver un minimum local combinant plusieurs critères de minimisation. Nous verrons par la suite (section 4.5.3) qu'une résolution multi-critères par des algorithmes génétiques donne des résultats bons et rapides car le temps de recherche de la solution optimale n'augmente quasiment pas. Le nombre de critères n'est pas coûteux car la résolution de la fonction de coût n'est pas plus complexe quand le nombre de critères augmente.

4.3.4 Limites des algorithmes génétiques

4.3.4.1 Complexité de calcul

Bien que ces algorithmes soient plus performants que des algorithmes qui parcourent toutes les solutions, ils nécessitent tout de même de nombreux calculs, en particulier au niveau de la fonction d'évaluation ou de coût.

4.3.4.2 Choix des paramètres

Ils sont aussi souvent difficiles à paramétrer. Certains paramètres comme la taille de la population sont généralement difficilement évaluables, souvent de façon empirique. Un autre point souvent délicat à fixer concerne la fonction d'évaluation ou de coût. Celle-ci doit souvent prendre en compte plusieurs paramètres du problème qu'il faut considérer avec attention.

Il faut aussi noter l'impossibilité d'être assuré, même après un nombre important de générations, que la solution trouvée soit la meilleure. On peut seulement être sûr que l'on s'est approché de la solution optimale, sans la certitude de l'avoir atteinte. Cela dépend de la méthode de décision ou du cas d'arrêt.

Un autre problème important est celui des optimums locaux. En effet, lorsqu'une population évolue, il se peut que certains individus qui à un instant occupent une place importante au sein de cette population deviennent majoritaires. À ce moment, il se peut que la population converge vers cet individu et s'écarte ainsi d'individus plus intéressants mais trop éloignés de l'individu vers lequel on converge. Pour résoudre ce problème, il existe différentes méthodes comme l'ajout de quelques individus générés aléatoirement à chaque génération, des méthodes de sélection différentes de la méthode classique...

4.4 Minimisation mémoire mono-composant

Dans les sections 4.2 et 4.3, nous avons posé les bases pour la résolution des problèmes rencontrés. Les algorithmes de coloriage de graphes vont principalement permettre de résoudre la minimisation des buffers alloués par le générateur de code AAA/SynDEx, tandis que les algorithmes génétiques permettent une optimisation multi-critères de la distribution/ordonnancement dans AAA/SynDEx. Nous allons traiter de manière progressive les différents problèmes de minimisation :

1. minimisation mono-composant après la génération de code de AAA/SynDEx,
2. minimisation multi-composants après la génération de code de AAA/SynDEx,
3. optimisation multi-critères de la distribution/ordonnancement.

Dans la littérature, peu de travaux illustrent la réutilisation de buffers mémoire sur une architecture mono-composant [MB01]. Les solutions évoquées traitent principalement de la minimisation de buffers à taille fixe (minimisation de registre) sur une architecture mono-composant ou d'une utilisation dynamique de la mémoire qui se montre très peu efficace dans un contexte embarqué. La réorganisation automatique peut être assimilée à celle que l'on peut retrouver dans les "Garbage collector" (GC) ou ramasse-miettes.

Un ramasse-miettes, ou récupérateur de mémoire, ou glaneur de cellules est un sous-système informatique de gestion automatique de la mémoire. Il est responsable du recyclage de la mémoire préalablement allouée puis inutilisée.

Lorsqu'un système dispose d'un ramasse-miettes, ce dernier fait généralement partie de l'environnement d'exécution associé à un langage de programmation particulier. Le ramassage des miettes a été inventé par John McCarthy comme faisant partie du premier système *Lisp* [McC60]. On retrouve également un *garbage collector* dans le

langage *Objective Caml* qui est utilisé pour la programmation de l'outil SynDEx. *Objective Caml* est un langage fonctionnel développé à l'INRIA qui se prête bien à la manipulation de listes [CMP00, LRVD04].

Le principe de base de la récupération automatique de la mémoire est simple :

- déterminer quels objets dans le programme ne sont pas utilisés à un instant donné,
- récupérer la mémoire utilisé par ces objets.

Bien qu'en général il soit impossible de déterminer à l'avance à quel moment un objet ne sera plus utilisé, il est possible de le découvrir à l'exécution : un objet sur lequel le programme ne maintient plus de référence, donc devenu inaccessible, ne sera plus utilisé.

Dans un premier temps, nous allons traiter le problème de la taille mémoire allouée après la génération de code sous AAA/SynDEx sur une architecture mono-composant. Le principe est la minimisation du nombre de données allouées sur un processeur en considérant que les données sont de tailles variables et de types différents. On présentera des résultats de minimisation de la mémoire dans la partie II.

4.4.1 Résolution par une méthode gloutonne "Mono critère latence"

Dans une première approche, nous nous sommes intéressés uniquement à une minimisation mono-processeur après la génération de code de AAA/SynDEx. L'approche utilisée consiste à étudier la durée de vie des buffers mémoire alloués en vue d'une réutilisation future au cours du temps. La durée de vie des buffers communiqués entre fonctions est simple à déterminer dans AAA/SynDEx, car l'ordonnancement fourni par l'outil est statique et donc les temps d'exécution sont prédictibles et connus avant la compilation. Le principe adopté est de réutiliser au mieux la mémoire, de manière statique.

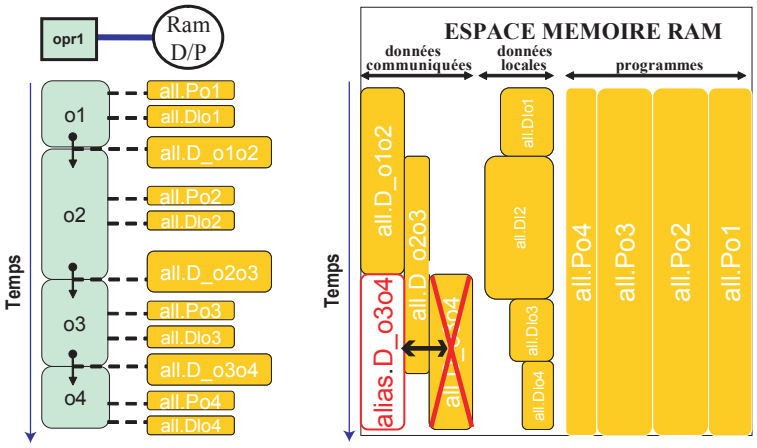


FIG. 4.2 – Schéma de principe de la réutilisation des buffers

Sur la figure 4.2, l'algorithme est exécuté sur une architecture mono-processeur. Les données allouées à l'intérieur des opérations o_i ne sont pas prises en compte pendant la phase de minimisation. Les 4 opérations atomiques sont ordonnancées selon l'ordre

total suivant : o_1, o_2, o_3, o_4. Les données inter-opérations $o_i \rightarrow o_{i+1}$ sont des données communiquées à l'intérieur du processeur. On remarque que la donnée entre $o_3 \rightarrow o_4$ n'interfère pas dans le temps avec la donnée entre $o_1 \rightarrow o_2$, cet espace mémoire est donc libre au moment de calculer la donnée entre $o_3 \rightarrow o_4$. Le principe va donc consister à réutiliser l'espace laissé libre de la donnée entre $o_1 \rightarrow o_2$.

La taille mémoire du code programme dans le pire des cas sera la somme du code programme de toutes les opérations atomiques o_i, car o_i et o_j peuvent exécuter la même référence. Dans le cas où o_i et o_j sont sur le même opérateur, on ne rajoute que la taille d'une des deux opérations. La mémoire programme n'est pas traitée dans les algorithmes de minimisation car elle n'est pas critique sur les algorithmes de la partie II.

4.4.1.1 Minimisation de registres

De nombreux travaux sont réalisés autour de la minimisation du nombre de registres [Bri92, BCT94, Zhu01] à utiliser pendant l'exécution d'une fonction. Ce problème peut être efficacement traité par la méthode de coloriage de graphe basée et implantée grâce à l'algorithme de CHRISTOFIDES et l'algorithme glouton ou séquentiel (Algo. 4.1 et 4.3). Nous décrivons ici le principe général.

Les compilateurs traduisent l'exécution d'une fonction en un graphe flot de données reliant les opérations élémentaires réalisées par l'ALU du processeur cible. Les données échangées entre ces opérations sont des registres dont la taille est fixe et déterminée par la capacité du bus de données. De ce graphe flot de données, on définit l'ordre total d'exécution des opérations sur le séquenceur de l'opérateur. Grâce à l'ordre total d'exécution, le compilateur analyse chaque registre r_i en déterminant la durée de vie T_{r_i} de ce registre auquel il associe un temps de début Td_{r_i} et un temps de fin Tf_{r_i}. La date de début Td_{r_i} est la date de début de l'opération créant le registre r_i et la date de fin Tf_{r_i}est déterminée par la date de fin de la dernière opération utilisant le registre r_i. Un graphe d'intervalle G est créé en associant à chaque registre r_i un sommet du graphe G, et en reliant chaque sommet du graphe G par un arc (r_i, r_j), dès que deux registres r_i et r_j interfèrent.

$$\text{si } Td_{r_i} \leq Td_{r_j} \text{ alors } Td_{r_j} < Tf_{r_i} \text{et } Tf_{r_i} \leq Tf_{r_j}$$

$$\text{si } Td_{r_j} \leq Td_{r_i} \text{ alors } Td_{r_i} < Tf_{r_j} \text{et } Tf_{r_j} \leq Tf_{r_i}$$

On peut ensuite appliquer l'algorithme glouton ou séquentiel (Algo. 4.3) pour minimiser le nombre de registres (= *couleurs*). Cet algorithme ne permet pas à lui seul de trouver le minimum de registres. En triant les registres $[r_1, ..r_n]$ suivant les dates de début Td_{r_i}, pour fournir finalement un ordre sur les sommets du graphe, on applique cette propriété sur l'algorithme glouton ou séquentiel pour trouver le minimum global de registres (= *couleurs*) (Algo. 4.3).

Le graphe d'intervalle G est créé suivant l'algorithme 4.5. Sur ce graphe G, nous pouvons utiliser l'algorithme 4.6 qui donne le nombre chromatique optimal $\chi(G)$ de registres utiles pour le graphe flot de données. On sait qu'un graphe d'intervalle est tel que $\chi(G) = \omega(G)$. Il se trouve que l'algorithme glouton donne effectivement l'optimum pour ces graphes. La raison est que l'ordonnancement choisi (par date de début) force l'augmentation de la clique maximum courante à chaque fois qu'une nouvelle couleur

Algorithme 4.5 Création du graphe d'intervalle

```
@ : concaténation d'un élément à une liste
donnée r_i est un registre
        - Td_{r_i}date_début : entier
        - Tf_{r_i}date_fin : entier
graphe :
       - liste_sommets =[s_1, s_2, ..., s_i, ..., s_n] avec s_i est un registre
       - liste_arcs[a_1, a_2, .., a_i, ...a_m] où a = (s_i, s_j)
  //[r_1, .., r_n] = liste_registres triée par ordre croissant de Td_{r_i}
  graphe fait_graphe ([r_1, .., r_n] = liste_registres)
  graphe G;
  G.liste_sommets = [];
  G.liste_arcs = [];
  début
    pour chaque r_i de liste_registres faire
    // G.liste_sommets=G.liste_sommets@r_i
    G.liste_sommets=ajout_sommet(r_i);
    pour chaque s_j de G.liste_sommets faire
      // les données interfèrent si les dates se recoupent
      //Td_{r_i} < Tf_{s_j} et Tf_{s_j} ≤ Tf_{r_i} car Td_{s_j} ≤ Td_{r_i}
      si interfère(s_j,r_i) alors
        //G.liste_arcs = G.liste_arcs@(s_j, r_i)
        G.liste_arcs=ajout_arc(s_j, r_i);
      fin si
    fin pour
  fin pour
  retourne G;
fin
```

est nécessaire. Au final, on a un coloriage de graphe en k couleurs et la preuve que c'est optimal puisqu'une clique de k sommets est exhibée. Le résultat est surtout l'expression de la réutilisation des registres exprimée en terme d'équivalence de couleur et donnant le minimum de mémoire utile.

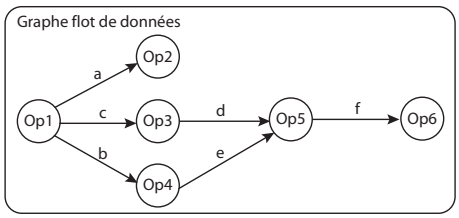


FIG. 4.3 – Exemple de graphe flot de données

Sur la figure 4.3, un ordre total du graphe flot de données est le suivant : op_1, op_2, op_3, op_4, op_5, op_6. La durée de vie des registres (a, b, c, d, e, f) est déduite de l'ordre total. La figure 4.4 illustre la création du graphe d'intervalle G du graphe flot de données. A chaque sommet du graphe est associé un registre, et un arc existe entre deux sommets dès que deux registres interfèrent : a interfère avec b et c, donc a est relié à b et à c.

Algorithme 4.6 Coloration de graphe

```
liste_couleur coloration (graphe G)
// retourne une liste de couleurs (= buffers)
// une couleur est une liste de données
liste_couleur = [[]]
booleen ajoute = faux ;
booleen existe_arc = faux ;
début
  pour chaque s_i de G.liste_sommets faire
    ajoute = faux ;
    pour chaque c_j de liste_couleur faire
      existe_arc = faux ;
      pour chaque r_i de c_j faire
        si (s_i, r_i)∈ G.liste_arcs alors
          existe_arc = vrai ;
        fin si
      fin pour
      si existe_arc= faux alors
        //ajout d'un registre dans une couleur existante
        c_i = c_i@s_i
        ajoute = vrai ;
      fin si
    fin pour
    si ajoute= faux alors
      // nouvelle couleur c_i = s_i
      liste_couleur = liste_couleur@[s_i];
    fin si
  fin pour
fin
```

On trie donc la liste de registres par ordre croissant de date de début Td_{r_i} : (a, b, c, d, e, f). Sur la liste triée, on applique le coloriage de graphe sur G dans l'ordre suivant :

1. On regarde le 1^{er} élément de liste, et on lui associe la première couleur (rouge).
2. Pour le 2^e élément b, comme il est connecté à a, on lui associe une autre couleur (vert).
3. Pour c, il est connecté à a et à b, on lui associe également une autre couleur (orange). On remarque à ce point que a, b, c engendrent une clique de 3 sommets.
4. Pour d, il est connecté à b et c, mais pas à a, alors il peut prendre la couleur de a (rouge).
5. Pour e, il est connecté à b et d, mais pas à a et c. La couleur de c est disponible mais pas celle de a (rouge) car elle est reliée à e par l'intermédiaire de d. e prend la couleur de c (orange).
6. f prend la couleur de b, car la couleur de b (vert) est laissée libre, elle n'est pas connectée à f. L'optimum est prouvé par le fait que nous avons utilisé 3 couleurs et la clique (a, b, c) prouve qu'il n'est pas possible de mieux faire.

Le nombre chromatique optimal $\chi(G)$ pour cet ordre total est de 3.

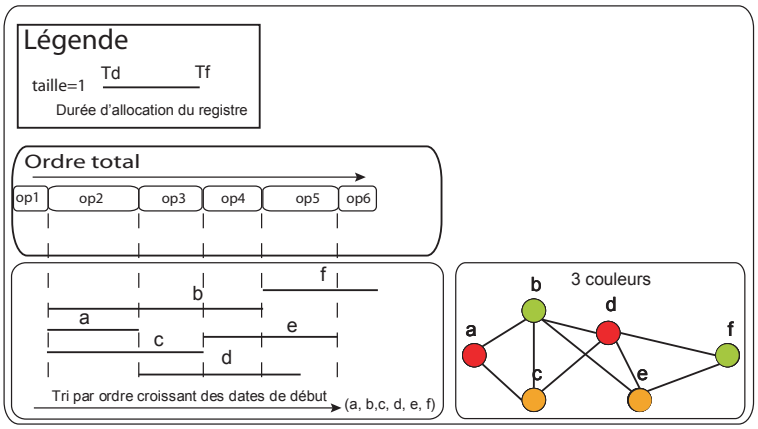


FIG. 4.4 – Graphe d'intervalle et minimisation des registres

4.4.1.2 Comment dimensionner le minimum de mémoire ?

AAA/SynDEx génère un exécutif avec un ordonnancement statique sans préemption. La durée de vie des buffers alloués peut donc être facilement déterminée. Chaque buffer B_i est représenté par une taille $Taille_{B_i}$ en nombre d'octets, une date de début d'allocation Td_{B_i} et une date de fin Tf_{B_i} (cf. tableau 4.1).

Taille du Buffer (octets)	Temps d'allocation
$Taille_{B_i}$	Td_{B_i} (temps de début) Tf_{B_i} (temps de fin)

TAB. 4.1 – Caractérisation des buffers

Une fois les dates de chaque buffer B_i déterminées, il faut trier par ordre croissant les dates de début et les dates de fin dans une même liste $L = [T_{B_1}, ..T_{B_i}, ...T_{B_n}]$ avec $T_{B_i} = Td_{B_i}$ ou Tf_{B_i} sachant que pour chaque date de début correspond une allocation $(+Taille_{B_i})$ et que pour chaque date de fin correspond une désallocation $(-Taille_{B_i})$. Pour N buffers, la liste L est donc de longueur $2N$. Il suffit de parcourir la liste en suivant l'algorithme glouton 4.7.

Algorithme 4.7 Minimum des minimums

$memoire_{max} = 0$
$memoire_{utile} = 0$
`tant que` $L \neq \varnothing$ `faire Parcourt de L`
  `si` $T_{B_i} \in L = Td_{B_i}$ `avec` T_{B_i} `est` Td_{B_i} `ou` Tf_{B_i}
    $memoire_{utile} = memoire_{utile} + Taille_{B_i}$
  `sinon`
    $memoire_{utile} = memoire_{utile} - Taille_{B_i}$
  `fin si`
  $memoire_{min_{utile}} = maximum(memoire_{utile}, memoire_{min_{utile}})$
`fin tant que` $L \neq \varnothing$

Grâce à l'algorithme 4.7, il est donc possible de définir le minimum de mémoire que l'on peut allouer. Cependant cet algorithme n'est réalisable que pour des cas simples quand tous les buffers sont de même type et de même taille.

Dans cet algorithme pour des buffers de taille variable, on ne tient pas compte du tout de l'implantation du graphe flot de données AAA/SynDEx sur une architecture cible. La mémoire minimum utile ne peut être modélisée que sous la forme d'un buffer circulaire (Fig. 4.5.a). Le buffer circulaire impose des contraintes importantes d'implantation logicielle. Le concepteur du graphe flot de données ne peut faire abstraction de la taille mémoire minimum obtenue et doit programmer ses opérations atomiques en conséquence. La discontinuité du buffer et le buffer circulaire nécessitent une programmation adéquate pour le concepteur du graphe flot de données.

On impose de plus une continuité de tous les buffers, dans le cas suivant, il faut que les buffers B_i et B_j ne se recoupent pas $\bigtriangledown i, j$ (Fig. 4.5.b).

FIG. 4.5 – Problème du buffer circulaire et de la discontinuité

Cette restriction impose, pour une minimisation globale de la mémoire allouée, une continuité de l'espace mémoire et une réallocation au mieux entre chaque opération. Il ne faut pas oublier une contrainte supplémentaire que l'on s'est fixée qui est de minimiser le temps d'exécution de notre algorithme. Chaque buffer est défini comme étant statique, et l'utilisation de chaque buffer est connu avant la compilation, sinon une réallocation dynamique de chaque buffer est nécessaire ce qui augmente le temps d'exécution de notre algorithme.

Dans le cadre des systèmes embarqués que l'on cherche à traiter, il faut donc s'orienter vers une solution moins efficace qui est déterminée hors-ligne "avant la compilation".

4.4.1.3 Minimisation des buffers basée sur la minimisation de registre

La première minimisation mise en œuvre est une extension de la méthode basée sur la minimisation du nombre de registres décrite dans la section 4.4.1.1. On définit le terme "données" d_i pour chaque sortie de l'algorithme après la mise à plat de celui-ci sous AAA/SynDEx. A chaque élément "donnée" d_i, on associe alors un couple : $(Type_i, T_{d_i})$ pour caractériser respectivement le type d'une valeur de la donnée d_i et le nombre de valeurs de d_i. Ce couple définit "une clé" permettant de stocker chaque élément "donnée" d_i associé à sa clé dans une table de hachage.

En informatique, une table de hachage est une structure de données qui permet une association clé-élément. On accède à chaque élément de la table via sa clé. A chaque clé, on a donc une liste de buffers de même taille et de même type qui lui est associée. Il est donc possible de considérer la liste des buffers associés à une clé comme une liste de registres et d'appliquer la méthode 4.4.1.1. On définit alors un graphe d'intervalle pour chaque clé noté G_{cle}. Cela revient à séparer le graphe d'intervalle global en sous-graphes homogènes (même taille et même type) pris distinctement. L'algorithme 4.3

donne donc le minimum de couleur $\chi(G_{cle_i})$ pour chaque sous-graphe G_{cle_i}. La mémoire utilisée pour cet algorithme est :

$$mem_{utile} = \sum_{i=0}^{n=nb_{cle}} (\chi(G_{cle_i}) * Td_i * Taillede(Type_i))$$

L'inconvénient majeur est que cette méthode ne permet pas de réutiliser des buffers de tailles différentes.

4.4.1.4 Minimisation des buffers "tétris" : réutilisation des buffers à taille variable

Une deuxième minimisation plus performante et originale a été mise en œuvre pour la minimisation des données allouées dans AAA/SynDEx. Le terme "buffer" b_j est défini comme un emplacement mémoire susceptible de contenir plusieurs "données" d_i. L'algorithme précédent s'exécute sur une table de hachage, qui à chaque élément donnée d_i associe une clé dont les champs sont un couple : $(T_{d_i}, Type_{d_i})$. Pour cet algorithme, la table de hachage est modifiée : on associe à chaque élément donnée d_i une clé à champ unique qui est le type $Type_{d_i}$.

Le principe de cette méthode consiste à réutiliser au mieux les données de taille variable mais de même type. On utilise maintenant des éléments "donnée" dont le type est le même mais pas nécessairement la taille.

Plusieurs méthodes ont été envisagées :

- considérer les données de taille n comme n données de taille 1 nécessitant n couleurs pour cette donnée : la difficulté est de trouver ensuite un espace contigu de taille k couleurs pour contenir une donnée de taille k, alors que k couleurs sont disponibles.
- considérer les plus grosses données comme prioritaires, et compléter les espaces disponibles par de plus petites données. Cette méthode utilise un mécanisme qui peut associer à n données $(d_1, .., d_n)$, un seul buffer b_i contenant toutes les données précédentes.

La méthode donnant les meilleurs résultats est de commencer par répartir les plus grosses données dans les buffers, puis de continuer sur l'ensemble des données par taille décroissante. L'algorithme précédent ne peut plus être utilisé tel quel car il est difficile à partir du graphe d'intervalle de pouvoir compléter efficacement un buffer b_j avec une donnée d_i. En effet deux données d_i et d_k dont les dates se recoupent (relié par un arc), n'interfèrent pas forcément dans un espace mémoire b_j (si $T_{d_i} + T_{d_k} \leq T_{b_j}$), cela dépend de la taille du buffer dans lequel on les place. Si deux données d_i et d_k interfèrent et sont contenues dans b_j, on définit $offset_{d_i}$ l'emplacement de la donnée d_i dans le buffer b_j, de même pour d_k. A chaque donnée d_i, on associe un offset dans un buffer b_j.

Prenons l'exemple du graphe d'algorithme de la figure 4.3, on détermine un ordre total (op_1, op_2, op_3, op_4, op_5, op_6) permettant de définir les dates de début et de fin de chaque donnée (a, b, c, d, e, f).

Le graphe d'intervalle de la figure 4.6 a un minimum global formé par la clique (b, a, c) dont la somme des poids de chaque élément est égale à 6.

Avant d'exécuter l'algorithme 4.8, il faut trier itérativement 3 fois cette liste suivant différents critères : dates de fin, dates de début et tailles des données. Nous obtenons donc sur ce graphe flot de données (a, b, e, c, d, f).

Appliquons sur cette liste l'algorithme 4.8 :

Etape 1 On part d'une liste vide pour les buffers, puis on crée a dans un buffer b_1 de taille 3 avec un offset nul.

Etape 2 b interfère avec a. b est dans un nouveau buffer b_2 de taille 2 avec un offset nul.

Etape 3 e n'interfère pas avec a, il peut donc être stocké dans b_1 comme a, avec un offset nul. b_1 contient (a, e).

Etape 4 c se chevauche avec a et b, c devient donc un buffer b_3.

Etape 5 d n'interfère pas avec a mais interfère avec e. d a une taille de 1, et peut prendre la place laissée libre par e. d est stocké dans b_1 avec un offset de 2. b_1 contient (a, e, d)

Etape 6 f n'interfère pas avec a, mais il interfère avec e puis avec d, le buffer b_1 ne contient plus de place pour f. f regarde dans b_2, et il n'interfère pas avec b. On peut donc stocker f dans b_2 avec un offset nul.

Le résultat de minimisation obtenu sur cet algorithme est donc de 6 sur ce graphe d'intervalle. Pour l'algorithme décrit dans la section 4.4.1.3, nous avons trois listes de données de même taille à minimiser : (a), (b, e) et (c, d, f), ce qui donne un minimum de mémoire allouée égale à 9.

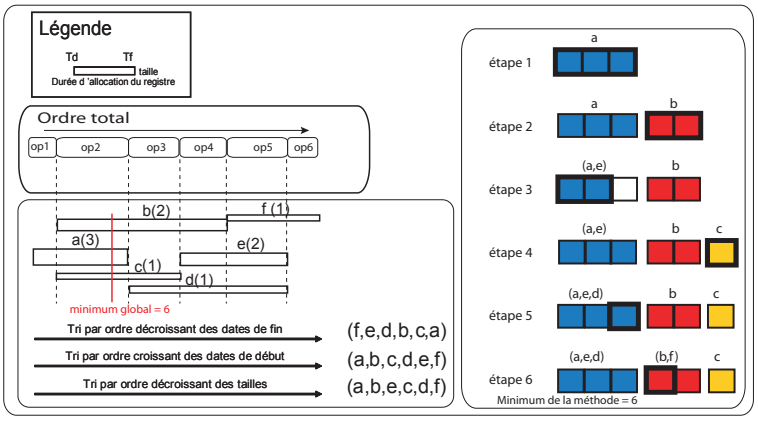


FIG. 4.6 – Graphe d'intervalle et Minimisation tétris

Algorithme 4.8 Minimisation tétris : réutilisation des buffers à taille variable

```
@ : concaténation d'un élément à une liste
donnée d_i est une donnée
        - Td_di date_début : entier
        - Tf_di date_fin : entier
        - T_di : entier
buffer b_i :  // avec off est l'offset de cette donnée
        - liste_donnees_offset =[(d_1, off_1), (d_2, off_2), ..., (d_i, off_i), ..., (d_n, off_n)]
        - Taille_bi

liste_buffer place_buffer_optim (liste_donnees : [d_1, ..., d_k] )
liste_bufffer= [] ;
booleen fait = faux ;
début
 liste_donnees triée par ordre décroissant de Tf_di
 liste_donnees triée par ordre croissant de Td_di
 liste_donnees triée par ordre décroissant de T_di
 pour chaque d_i dans liste_donnees faire
    fait = faux ;
    pour chaque b_j de liste_buffer faire
      si place(d_i,b_j) alors
        fait = vrai ;
        sortir pour ;
      fin si
    fin pour
    si non fait alors
    // ajoute un buffer à la liste
      liste_buffer = liste_buffer@ = liste_buffer @
            (nouveau buffer
            (liste_donnnees_offset = [(d_i, 0)], T_bk = T_di)) ;
    fin si
  fin pour
  retourne liste_buffer ;
fin
```

Algorithme 4.9 Vérification de la place disponible dans un buffer b_i

```
boolean place (d_k : donnée, b_l : buffer)
entier offset = 0 ;
début
  pour chaque (d_i, off_di) de b_l.liste_donnees_offset faire
    // les données interfèrent si les dates se recoupent
    //Td_di < Tf_dk et Tf_dk ≤ Tf_di car Td_dk ≤ Td_di
    si interfère (d_k, d_i) alors
      offset = offset + off_di;
    fin si
  fin pour
  si offset < T_bi alors
    b_i.liste_donnees_offset = b_i.liste_donnees_offset@(d_k, offset);
    retourne vrai ;
  sinon
    retourne faux ;
  fin si
fin
```

De façon générale, quand on ne peut pas placer une donnée dans un buffer, on alloue un nouveau buffer (algorithme 4.8 et 4.9). Ces 2 algorithmes sont des algorithmes gloutons, ne remettant jamais en cause les choix faits sur le placement d'une donnée dans un buffer : on ne parcourt qu'une seule fois la liste des données triées par taille décroissante. Cette méthode a fourni de bons résultats, meilleurs que l'optimisation précédente. L'algorithme 4.9 regarde sur le graphe d'intervalle si les données d_i et d_k interfèrent, pour un offset au départ nul du buffer b_j de sortie puis réitère l'opération tant que l'offset n'est pas égal à la taille du buffer de sortie.

Cette méthode ne fournit pas toujours le minimum global de la minimisation des données à effectuer sur l'algorithme mis à plat par AAA/SynDEx. Cependant elle aboutit à de meilleurs résultats que la méthode basée sur la minimisation de registre (section 4.4.1.3), sur l'ensemble des algorithmes de télécommunication et de traitement d'images de la partie II.

4.5 Minimisation mémoire multi-composants

L'objectif de AAA/SynDEx est de faire une implantation optimisée d'un graphe d'algorithme sur une architecture multi-composants. Dans le cas multi-composants, il existe une difficulté supplémentaire pour les allocations sur les communications. Il est intéressant de pouvoir étendre les méthodes précédentes au cas multi-composants en incorporant les données communiquées entre composants (Fig. 4.7).

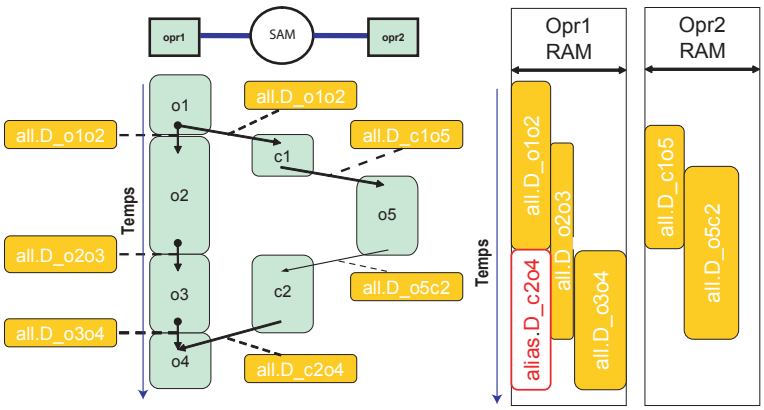


FIG. 4.7 – Schéma de principe de la réutilisation mémoire pour le muli-composant

4.5.1 Temps d'allocation sur les séquenceurs de communication

4.5.1.1 Communications dans AAA/SynDEx

La durée d'allocation d'une donnée d_i sur les séquenceurs de communication est d'une durée infinie. Pour le comprendre, nous prenons l'exemple du graphe d'algorithme de la figure 4.8, qui est le suivant : $Producteur \rightarrow (Consommateur_1, Consommateur_2)$. Le *Producteur* produit une donnée *data* pour $Consommateur_1$ et $Consommateur_2$. L'implantation de cet algorithme est effectuée sur une architecture multi-composants :

$((Producteur, Consommateur_1) \rightarrow op_1, Consommateur_2 \rightarrow op_2)$. Prenons comme référentiel de temps, le séquenceur de l'opérateur de calcul op_1. A la première itération, *data* est vide ou disponible (Suc_{empty}), le *Producteur* peut produire des données. *data* est utilisé dès le début du calcul du producteur. Quand la donnée *data* est pleine (Pre_{full}), le séquenceur de communication *com*1 est prévenu d'une donnée produite (Suc_{full}). Le séquenceur de calcul du *Producteur* attend sur Suc_{empty} que la donnée produite *data* soit consommée par l'opération *send* (Pre_{empty}). La donnée *data* est donc utilisée sur une itération entière du séquenceur du producteur, et nous considérons ce temps d'allocation de la donnée comme étant une durée infinie. Le macro-code pour l'opérateur op_1 équivalent à l'implantation du graphe flot de données sur l'architecture cible est décrit sur l'algorithme 4.10. Les sémaphores de synchronisation sont basés sur le réseau de Petri générique de la figure 2.18. Pour l'opérateur op_1, nous obtenons le réseau de Petri de la figure 4.9.

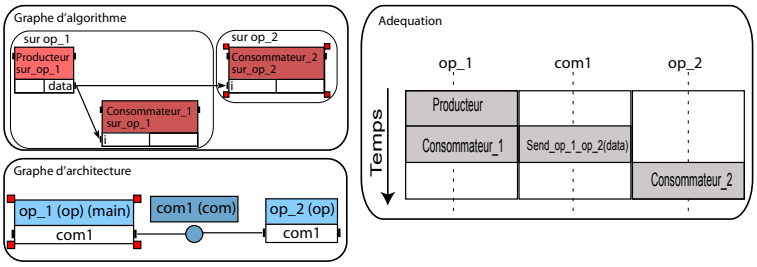


FIG. 4.8 – Exemple d'application multi-composants *producteur* → *consommateur*

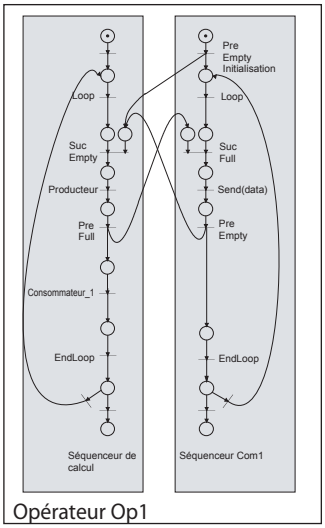


FIG. 4.9 – Réseau de Petri pour l'opérateur op_1 de l'algorithme *producteur* → *consommateur*

Algorithme 4.10 Macrocode AAA/SynDEx avant modification

```
// liste des sémaphores de synchronisations inter-séquenceurs
semaphores_ (Semaphore_Thread_com1, Producteur_data_op_1_com1_empty
                          , Producteur_data_op_1_com1_full)
alloc_ (int,Producteur_data,1) //allocation de la donnée data
//séquenceur de communication sur com1
thread_ (com,com1,op_1,op_2)
  Pre0_ (Producteur_data_op_1_com1_empty,,Producteur_data,empty) // Pre_empty
  loop_
    Suc1_ (Producteur_data_op_1_com1_full,,Producteur_data,full) //Suc_full
    send_(Producteur_data,op,op_1,op_2) // send_(data) vers consommateur_2
    Pre0_ (Producteur_data_op_1_com1_empty,,Producteur_data,empty) // Pre_empty
  endloop_    saveFrom_(,op_2)
endthread_
//séquenceur de calcul
main_
spawn_thread_ (com1)   // init du séquenceur de communication com1
   loop_
    Suc0_ (Producteur_data_op_1_com1_empty,com1,Producteur_data,empty) // Suc_empty
    Producteur(Producteur_data) // instance : Producteur produit (data)
    Pre1_ (Producteur_data_op_1_com1_full,com1,Producteur_data,full) // Pre_full
    Consommateur(Producteur_data) //instance : Consommateur_1 consomme data
  endloop_
endmain_
```

Dans le cas général, ce schéma est reproduit autour de chaque producteur de donnée qui est transférée vers un autre opérateur (Fig. 4.10). Un producteur d'une donnée d_i est directement précédé d'un sémaphore Suc_{empty} et suivi d'un sémaphore Pre_{full}, ces 2 sémaphores étant associés à d_i. Prenons un ensemble de données $(d_1, ..d_n)$ ordonnancées et envoyées par un ensemble de producteurs $(Prod_1...Prod_n)$ sur le médium de communication $com1$. Sur le séquenceur de calcul op_1, l'ordre total est défini par la liste triée $Prod_1$ à $Prod_n$. Pour chaque d_i, on associe sur ce séquenceur $(Suc_{empty_{di}}, Pre_{full_{d_i}})$. La donnée d_i est transférée avec certitude lorsque op_1 est bloqué sur $Suc_{empty_{d_i}}$. La donnée d_i est donc utilisée sur une itération entière.

Notre référentiel temps est limité à une itération du graphe d'algorithme, à chaque nouvelle itération le temps étant réinitialisé. La durée d'exécution d'une itération de l'algorithme sur une architecture cible est définie comme la *latence*de l'application distribuée. La durée moyenne de l'algorithme en régime permanent sur une infinité d'itérations caractérise la *cadence*.

La *cadence* a une durée plus faible que la *latence* : il peut y avoir chevauchement des transferts sur les calculs de l'itération suivante dans le meilleur des cas, avec/ou imbrication de l'itération courante dans l'itération précédente, ce qui réduit le temps d'exécution en régime permanent (*cadence*). Par exemple, sur la figure 4.8, la *latence* est de 3 unités de temps (en considérant le temps de chaque opération unitaire) et la *cadence* est de 2 unités de temps (imbrication de l'itération courante dans l'itération précédente sans chevauchement de transfert, Fig. 4.11).

$$cadence \leq latence$$

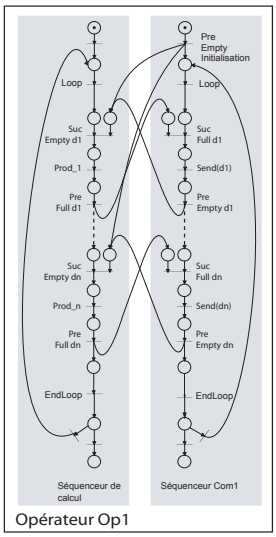


FIG. 4.10 – Réseau de Petri général pour l'opérateur op_1

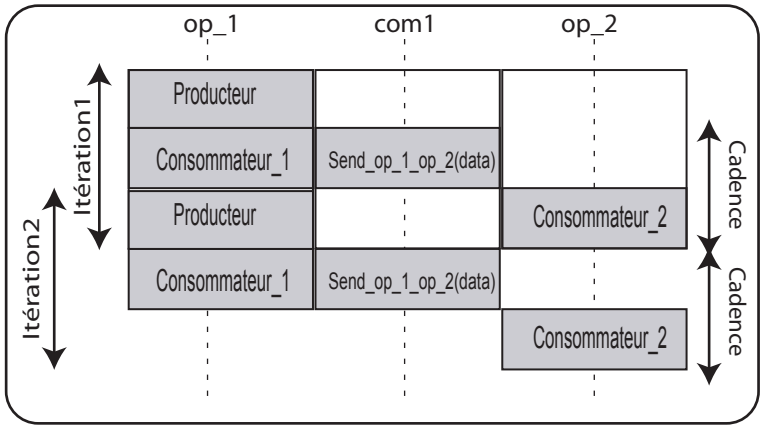


FIG. 4.11 – Cadence de l'application multi-composants *producteur* → *consommateur*

4.5.1.2 Communications bloquantes

Le réseau de petri prévu pour les allocations sur les séquenceurs de communications a donc une durée infinie (Fig. 2.18), et ce schéma de principe se répète sur toutes les données communiquées entre processeurs (Fig. 4.10). Les sémaphores de synchronisation inter-séquenceurs sont placés de manière à avoir le maximum de parallélisme entre les calculs et les communications :

- autorisant le maximum de flexibilité entre les séquenceurs,
- évitant le partage de données entre séquenceurs.

Dans le cas où la donnée d_i est envoyée sur le medium $com1$, le sémaphore d'attente Suc_{empty} est placé au plus tard dans le cas général AAA/SynDEx. Pour une donnée

envoyée d_i, il est possible de ne plus autoriser de parallélisme entre les calculs et les données d_i émises, en déplaçant au plus tôt le sémaphore Suc_{empty} dans le séquenceur de calcul.

Dans le cas où la donnée d_i est reçue sur le medium $com1$, le sémaphore Pre_{empty}, indiquant au séquenceur de communication que la donnée d_i peut être reçue de nouveau, est placé au plus tôt dans le cas général AAA/SynDEx, juste après le consommateur de la donnée d_i. Le principe est de fournir le maximum de parallélisme et de flexibilité entre les séquenceurs de calcul et de communication. Il est possible de borner au minimum le temps de réception d'une donnée d_i, en déplaçant le sémaphore Pre_{empty} pour une donnée reçue juste avant l'opération produite.

Pour les communications bloquantes, nous ne respectons pas le temps d'exécution du graphe temporel fourni par AAA/SynDEx, car nous modifions le positionnement des sémaphores du graphe temporel après la phase d'adéquation d'AAA/SynDEx. La phase d'adéquation n'est pas modifiée et prend toujours en compte le parallélisme entre les communications et les opérations de calcul sur un opérateur. Seul le macro-code est transformé en déplaçant les sémaphores Pre_{empty} pour un *receive* et Suc_{empty} pour un *send*. Sur la figure 4.8, le macro-code généré pour l'opérateur op_1 suit l'exécution suivante : $Producteur$, $Send(data)$, $Consommateur_1$. Le parallélisme entre l'opération $consommateur_1$ et $Send(data)$ n'existe plus dans le macro-code généré. Le temps d'exécution est de 3 unités de temps sur l'opérateur op_1, qui est la somme des temps d'exécution des opérations $Producteur$, $Send$, $Consommateur_1$ considérés unitaires (2 unités de temps avec parallélisme). La $latence_{bloquante}$ de la figure 4.8 avec les modifications sur le séquenceur de calcul est de 3 unités de temps comme pour le schéma général ($latence_{SynDEx}$). La $cadence_{bloquante}$ avec les communications bloquantes est ralentie sur l'exemple (2 unités de temps pour la $cadence_{SynDEx}$). D'une manière générale, on a :

$$\begin{cases} latence_{SynDEx} \leq latence_{bloquante} \\ cadence_{SynDEx} \leq cadence_{bloquante} \end{cases}$$

Pour garantir l'ordonnancement sur le séquenceur de calcul et éviter les interblocages, le couple (Pre_{full},Suc_{empty}) pour l'envoi d'une donnée est placé sur l'opérateur de calcul à la date de début d'un transfert (*send*) ; il ne faut pas uniquement déplacer le Suc_{empty} après le calcul producteur de la donnée émise. Le couple (Pre_{empty}, Suc_{full}) pour la réception d'une donnée est placé sur le séquenceur de calcul à la date de fin d'un transfert (*Receive*), le déplacement du Pre_{empty} avant le calcul consommateur de la donnée reçue ne suffit pas.

La durée de vie sur un opérateur d'une donnée communiquée c_i commence au début de l'opération produisant la donnée c_i et se termine au maximum de la date de fin du dernier calcul consommant la donnée c_i. L'opération produisant la donnée c_i peut être un *receive*, la date de début de cette opération correspond à la date de fin du transfert pour les communications bloquantes. Les opérations consommant la donnée peuvent être des *send*, la date de fin de ces opérations correspond à la date de début de transfert pour les communications bloquantes.

Le mécanisme des communications bloquantes (Fig. 4.12) permet une simulation plus aisée du système à implanter sur cible, à cause de la séquentialité des opérations de calcul et des communication.

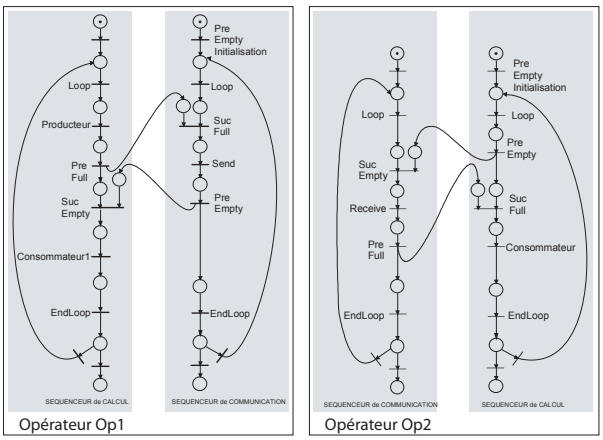


FIG. 4.12 – Exemple de réseau de Petri pour les communications bloquantes

4.5.1.3 Communications plus proches de l'adéquation

Pour borner les temps de transfert, et pouvoir réutiliser les espaces mémoire des données transférées, des modifications sont apportées au modèle précédent en tenant compte ici des temps de transfert des communications. Ce modèle est identique et plus proche du graphe temporel fourni après l'adéquation de AAA/SynDEx (Fig. 4.14). Dans ce modèle, on déplace les sémaphores de synchronisation en fonction de l'opération qui est effectuée sur le séquenceur de communication : envoi ou réception de données (*Send* ou *Receive*).

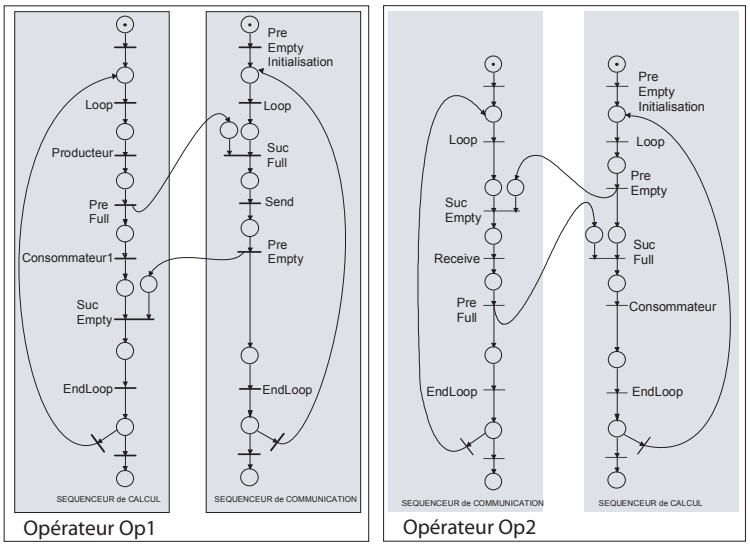


FIG. 4.13 – Exemple de réseau de Petri avec les communications plus proche de l'adéquation

Algorithme 4.11 Macrocode AAA/SynDEx : modifications du placement du Suc_{empty} pour un send

```
// liste des sémaphores de synchronisation inter-séquenceurs
semaphores_ (Semaphore_Thread_com1, Producteur_data_op_1_com1_empty
                          , Producteur_data_op_1_com1_full)
alloc_ (int,Producteur_o,1)  // allocation de la donnée produite
//séquenceur de communication sur com1
thread_ (com,com1,op_1,op_2)
  loop_
  Suc1_ (Producteur_data_op_1_com1_full,,Producteur_data,full) // Suc_full
  send_(Producteur_data,op,op_1,op_2) // send_(data) vers consommateur_2
  Pre0_ (Producteur_data_op_1_com1_empty,,Producteur_data,empty) // Pre_empty
  endloop_
endthread_
//séquenceur de calcul
main_
spawn_thread_(com1)  // init séquenceur de communication com1
loop_
  Producteur(Producteur_data) // instance : Producteur  produit (data)
  Pre1_ (Producteur_data_op_1_com1_full,com1,Producteur_data,full) // Pre_full
  Consommateur(Producteur_data) //instance : Consommateur_1 consomme data
  Suc0_ (Producteur_data_op_1_com1_empty,com1,Producteur_data,empty) //Suc_empty
endloop_
endmain_
```

Pour le graphe d'algorithme 4.8, on déplace le sémaphore Suc_{empty} sur le séquenceur de calcul de l'op_1 à la date de fin du transfert ($Send$) sur le séquenceur de communication $com1$. Pour l'op_2, on déplace le Pre_{empty} au début du transfert ($receive$) sur le séquenceur de $com1$: la date de début du $receive$ sur op_2 est la même que la date de début du $send$ sur op_1. L'utilisation de la donnée $data$, dans le référentiel de l'$op1$, commence à la date de début du $Producteur$ et se termine soit après la date de fin de la dernière opération utilisant la donnée sur op_1, ou soit après la date de fin de transfert sur $com1$. Dans le référentiel de temps de l'op_2, l'utilisation de $data$ se termine à la fin de l'opération $Consommateur$ et commence à la date de début de transfert sur le séquenceur $com1$.

Ce processus de déplacement des sémaphores est réitéré sur chaque donnée transférée (Fig.4.14). Pour ne pas avoir d'inter-blocage dans les séquenceurs de calcul, il faut que l'ordonnancement sur les communicateurs d'un opérateur op_i s'exécute comme l'ordonnancement spécifié par l'adéquation AAA/SynDEx. Les déplacements des sémaphores Suc_{empty} et Pre_{empty} ne se limitent pas à la durée d'un transfert, mais au début (Suc_{empty}) ou à la fin (Pre_{empty}) d'un transfert pour garantir l'ordonnancement de l'exécution sur un séquenceur de communication.

Le macro-code généré est conforme et plus proche de l'adéquation fournie par AAA/SynDEx. Le parallélisme entre le séquenceur de calcul et le séquenceur de communication est préservé. Le macro-code de l'$op1$ pour l'adéquation pour la figure 4.8, est donc modifié (Algo. 4.11). La *latence*, fournie par l'adéquation, est identique avec et sans modification des sémaphores sur la génération de code. Cependant, dans ce cas, la *cadence* de l'algorithme sur une architecture cible est supérieure ou égale à la *cadence* sans modification sur le placement des sémaphores, car les communications

de la fin d'une itération ne peuvent chevaucher l'itération suivante, mais l'imbrication de l'itération courante dans la précédente reste présente, d'où :

$$cadence \leq latence$$

Dans le domaine de l'embarqué où les contraintes mémoire sont fortes, on choisira souvent la solution où l'on borne les transferts pour pouvoir mieux minimiser la mémoire.

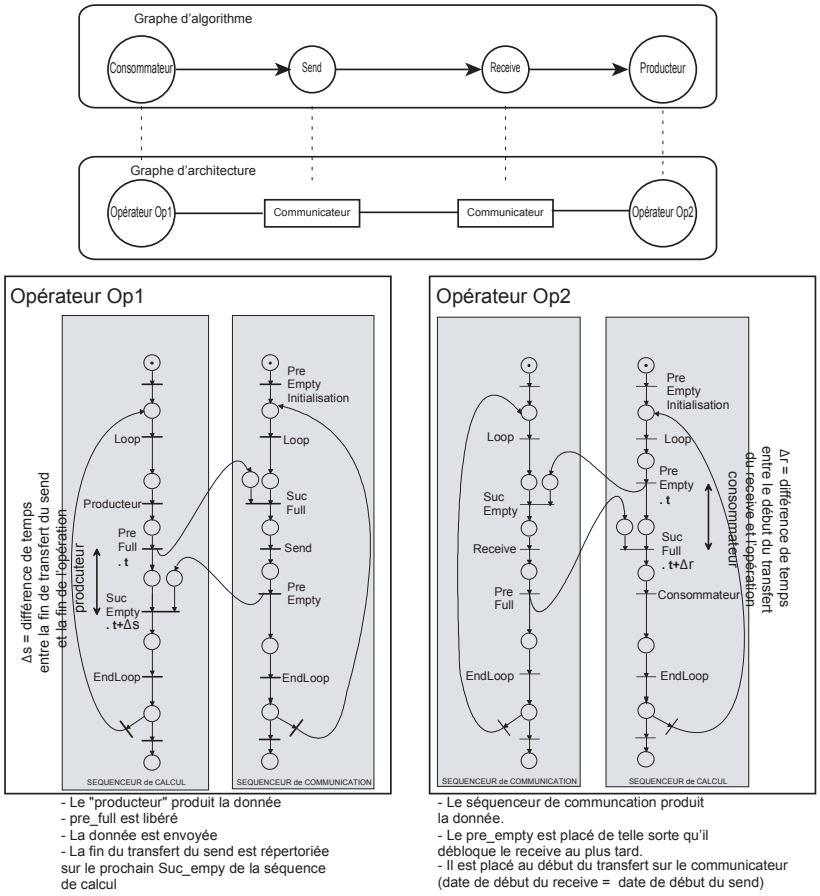


FIG. 4.14 – Modifications du placement des synchronisations

4.5.2 Minimisation par algorithme glouton

4.5.2.1 Minimisation après la génération de code AAA/SynDEx

Les minimisations décrites précédemment pour l'approche mono-composant sont étendues aux multi-composants. Dans l'approche multi-composants, on peut considérer deux cas :

1. minimisation des données communiquées inter-opérations sur un même composant,

2. minimisation des données communiquées inter-opérations et inter-composants.

Dans l'approche multi-composants, il existe deux types de données, les données communiquées d_i uniquement inter-opérations sur un même opérateur et les données communiquées c_i sur les séquenceurs de communication d'un opérateur. Sur l'opérateur op_1, les données allouées par AAA/SynDEx sont $data_{op_1} = \{d_1, ..., d_n\} \cup \{c_1, ..., c_m\}$.

Pour le sous-ensemble $data = \{d_1, ..., d_n\}$, il suffit d'utiliser les algorithmes de minimisation mono-composant. Pour cela, il suffit de ne pas déplacer les sémaphores de synchronisation, et de considérer les temps de transfert d'une donnée c_i comme infinis. Donc, chaque donnée c_i sur l'opérateur op_1 est allouée et n'est pas réutilisée par un autre buffer.

Le second sous-ensemble $data = \{c_1, ..., c_n\}$ est traité en déplaçant les sémaphores de synchronisation afin de dater chaque temps de début et de fin d'utilisation d'une donnée c_i. Toutes les données $data_{op_i} = \{d_1, ..., d_{n_i}\} \cup \{c_1, ..., c_{m_i}\}$ pour chaque opérateur op_i étant datées, les algorithmes de minimisation mono-composant sont directement transposés pour chaque opérateur op_i sur chaque liste $data_{op_i}$.

Cas particulier du routage

Sur la figure 4.15, l'opération *producteur* est affectée sur l'opérateur op_1, et l'opération *consommateur* sur op_3. La donnée *data* produite par l'opération *producteur* est transférée sur op_3 pour l'opération *consommateur* via l'opérateur op_2. On parle alors du routage de la donnée *data* sur l'opérateur op_2. Le routage représente le transfert direct d'une donnée d'un communicateur à un autre sans passer par le séquenceur de calcul : sur l'opérateur op_2 la donnée *data* passe directement du $communicateur_1$ au $communicateur_2$ (Fig. 4.16).

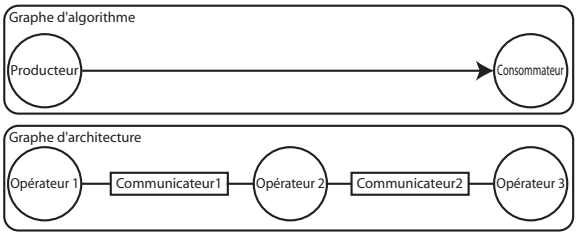


FIG. 4.15 – Le routage : graphes d'algorithme et d'architecture

Le mécanisme de routage sur l'opérateur op_2 n'a pas de référentiel de temps dans le séquenceur de calcul de op_2, c'est pourquoi la donnée reçue puis envoyée a une durée de vie infinie. Dès qu'une donnée est routée, le temps associé à d_i est infini. Le réseau de Petri associé au routage dans AAA/SynDEx est décrit sur la figure 4.17. Un sémaphore $Pre1_{empty}$ d'initialisation vient débloquer la réception de *data*. Le $communicateur_2$ est bloqué sur $Suc1_{full}$. Une fois la donnée *data* reçue, le sémaphore $Pre1_{full}$ vient débloquer $Suc1_{full}$, et la donnée *data* peut être ainsi émise. Le $communicateur_1$ se bloque sur $Suc1_{empty}$, qui sera débloqué lorsque le sémaphore $Pre1_{empty}$ sera envoyé.

Une autre approche a été envisagée pour pouvoir minimiser les données routées, mais elle modifie la latence calculée par la phase d'adéquation AAA/SynDEx, ce qui n'est pas satisfaisant. Cette méthode offre cependant une bonne minimisation de la

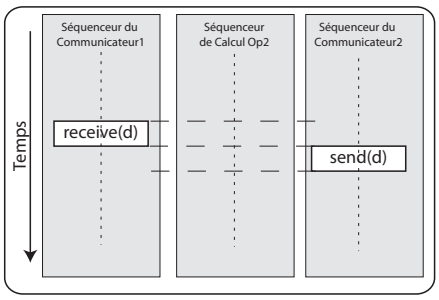


FIG. 4.16 – Le routage : graphe temporel fourni par AAA/SynDEx

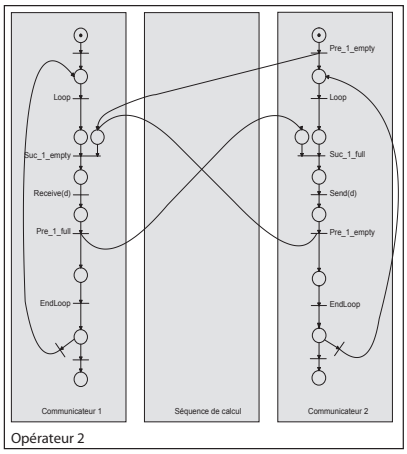


FIG. 4.17 – Réseau de Petri du routage dans AAA/SynDEx

mémoire lorsqu'il y a beaucoup de données routées. Le principe est de pouvoir dater les temps de début et de fin de vie d'une donnée routée dans le référentiel de temps du séquenceur de calcul. Pour cela, sur le séquenceur de calcul, il faut borner le temps de transfert de la donnée *data* reçue en rajoutant des sémaphores de synchronisation : $Pre1_{empty}$ et $Suc0_{full}$. De même, on borne le temps de transfert en rajoutant pour chaque émission routée de la donnée *data* deux sémaphores de synchronisation $Pre1_{full}$ et $Suc0_{empty}$. Le réseau de Petri de la figure 4.18 illustre les modifications apportées sur les sémaphores. Le principe consistant à rajouter des sémaphores fonctionne sur cet exemple : la latence reste la même.

Le graphe temporel de la figure 4.19.a illustre le parallélisme possible entre une opération et une donnée routée sans modification des sémaphores. Cependant un problème de création et de placement des sémaphores peut survenir sur cet exemple ; la figure 4.19.b montre les modifications temporelles apportées sur le séquencement des opérations. Il faut autoriser le transfert avant l'opération grâce à $Pre1_{empty}$. Pour garder du parallélisme entre la réception et l'opération, le transfert ne pourra se terminer qu'après l'opération sur $Suc0_{full}$. On peut ensuite autoriser l'envoi de la donnée grâce

au sémaphore $Pre1_{full}$. La donnée est envoyée dès que le sémaphore $Suc0_{empty}$ est reçu.

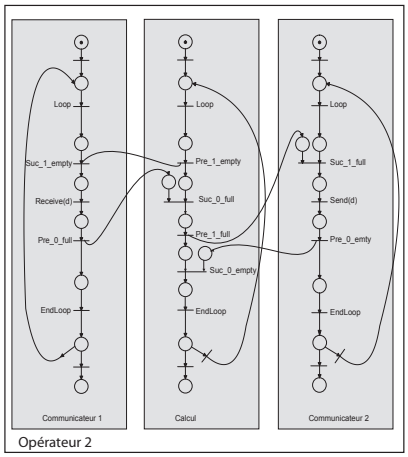


FIG. 4.18 – Modifications apportées au routage

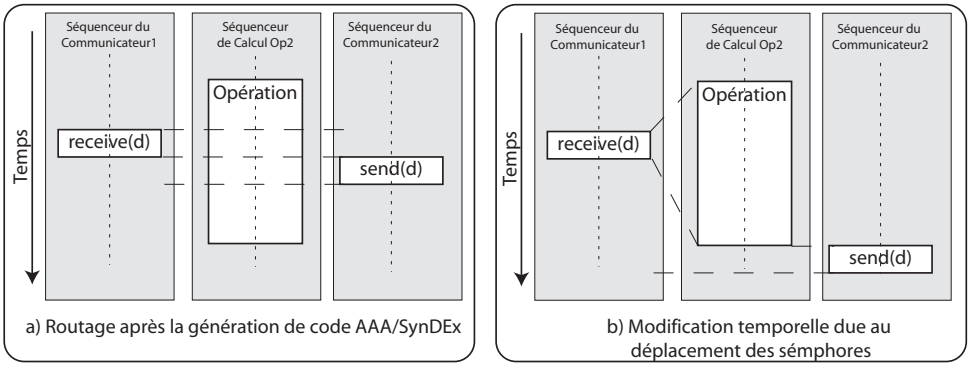


FIG. 4.19 – Le routage : graphe temporel fourni par AAA/SynDEx

4.5.2.2 Minimisation pendant la phase d'adéquation

Les algorithmes précédemment décrits sont des post-traitements sur le macro-code généré : il s'agit alors uniquement d'une minimisation de buffers effectuée indépendamment sur chaque processeur, une fois que la distribution et l'ordonnancement ont été effectués, c'est-à-dire lorsque l'ordre total est figé.

Nous avons ensuite cherché une nouvelle heuristique totalement fondée sur la minimisation de la mémoire au cours de la distribution et de l'ordonnancement. La phase de distribution/ordonnancement doit alors être déterminée en fonction d'une répartition des buffers sur les processeurs produisant un minimum d'allocation sur chacun d'eux. La méthode explorée remplace la fonction de coût mise en place dans AAA/SynDEx pour calculer la distribution/ordonnancement sur une architecture. Cette méthode

est gloutonne et basée sur le coloriage de graphe. A chaque nouvelle opération à ordonnancer sur un opérateur, on regarde la charge mémoire de chaque opérateur que l'on détermine grâce au coloriage de graphe. Cette charge mémoire est calculée pour chaque opérateur à chaque nouvelle opération à ordonnancer. On affecte l'opération sur l'opérateur ayant la charge mémoire minimale. Cet algorithme ne tient pas compte de l'évolution du graphe à implanter, car pour chaque nouvelle opération, il faut créer les transferts correspondants et calculer la charge de chaque opérateur en fonction des tables de routage utilisées. Cet algorithme est très coûteux, et ne donne pas de résultats satisfaisants.

Une heuristique gloutonne mixte, basée à la fois sur l'optimisation temporelle et la minimisation de mémoire a été envisagée, mais pour une approche multi-critères cela ne s'est pas avéré efficace, d'autant plus que l'ordonnancement fourni par AAA/SynDEx et le post-traitement effectué donnent déjà de bons résultats.

4.5.3 Minimisation par algorithme génétique

Cette approche ne traite pour l'instant que du problème de la minimisation temporelle. Pour optimiser les choix de la distribution/ordonnancement du graphe d'algorithme de notre application, nous nous sommes intéressés aux algorithmes génétiques, plus coûteux en temps de calcul sur un seul critère mais pour lesquels une approche multi-critères s'avère par contre plus facile à réaliser et moins coûteuse. Le temps d'exécution de la recherche d'une solution avec les algorithmes génétiques ne varie pas ou peu suivant le nombre de critères. A chaque individu fourni par l'algorithme génétique, on peut appliquer les algorithmes de post-traitement de la mémoire afin de savoir si l'individu s'avère potentiellement intéressant.

AAA/SynDEx fournit une solution performante pour résoudre le problème d'ordonnancement/distribution sur des architectures hétérogènes multi-processeurs, basée sur des heuristiques gloutonnes. Par contre AAA/SynDEx ne prend en compte qu'un critère de convergence, la latence. De plus son algorithme est déterministe et si la solution trouvée n'est pas la meilleure aucune autre ne sera cherchée. A MITSUBISHI ITE, avec E.LAVILLONNIERE et C.MOY, nous avons abordé la distribution/ordonnancement sous l'angle des algorithmes génétiques [Kwo97], ce qui permet une amélioration des résultats sur la minimisation de la latence, mais surtout de pouvoir facilement traiter l'optimisation de la cadence qui reste un critère important dans les applications de télécommunication et de traitement d'images, souvent répétitives.

Au sein du laboratoire de MITSUBISHI, l'approche AAA/SynDEx est utilisée pour dimensionner un processeur Renesas (alliance entre MITSUBISHI et Hitachi semi-conducteur), connu sous le nom M32R, il s'agit d'un processeur multi-cœurs de traitement du signal (plusieurs séquenceurs de calcul dans le même processeur). L'objectif est l'évaluation grâce à AAA/SynDEx du nombre de cœurs de processeur nécessaires. Ce dimensionnement doit être précis, d'où l'utilisation des algorithmes génétiques pour résoudre ce problème. L'implantation logicielle des algorithmes génétiques a été effectuée par E.LAVILLONNIERE en $C++$. Mon travail a été de spécifier le schéma de données de SynDEx et la description algorithmique précédant et durant la phase de mise à plat. L'implantation $C++$ de l'algorithme génétique se base sur le même fichier de description de scène compatible avec SynDEx, et permet une définition des critères de sélection combinant cadence et latence. La recherche de la solution

peut se distribuer sur un réseau d'ordinateurs grâce à des techniques développées par E.LAVILLONNIERE (utilisation de *Message Oriented Middleware* [Lav]) cela pour obtenir plus rapidement une solution.

La mise en œuvre d'un algorithme génétique nécessite la sélection des individus à différentes étapes. Le choix s'appuie sur le résultat d'une fonction d'évaluation ou de coût appliquée à chaque individu (Algo. 4.12).

La façon la plus immédiate pour déterminer la fonction de coût passe par une définition d'une probabilité proportionnelle au résultat de la fonction d'évaluation. Néanmoins, si les résultats sont mal répartis, la sélection risque de favoriser un tout autre groupe.

Une autre possibilité réside dans l'utilisation d'un mécanisme de *ranking*. Suite à leur évaluation les individus sont classés par résultat. On leur affecte alors une probabilité proportionnelle à leur classement. Cela permet de beaucoup moins exclure les cas ayant une très faible fonction de coût, et par voie de conséquence de beaucoup moins privilégier ceux qui en ont une très bonne, puisque finalement le rang compte pour la sélection (et pas la valeur). Si p est le classement d'un individu et n le nombre d'individus, on obtient la formule 4.1 (le meilleur individu est classé n) pour la probabilité $Proba(p, n)$ associée au mécanisme de *ranking*.

$$Proba(p, n) = p / \sum_{k=1}^{n} k \tag{4.1}$$

Dans notre cas les algorithmes génétiques sont utilisés pour :
- affecter un opérateur à une opération (par exemple un processeur à une opération),
- créer un ordre total d'un graphe d'algorithme dans la détermination de l'ordre des opérations à effectuer.

Les individus manipulés sont donc des descriptifs expliquant comment réaliser une suite d'opérations sur un certain nombre d'opérateurs. Ils indiquent :
- l'opérateur affecté à chaque opération,
- l'ordre dans lequel les opérations vont être exécutées sur les opérateurs, pour répondre aux dépendances de données inter-opérations.

Pour obtenir ces deux types d'information, on doit faire appel à deux techniques des algorithmes génétiques. Pour trouver les opérateurs, on décrit initialement l'ensemble des opérations sous un n-uplet $(opn_1, ..., opn_n)$. La solution est un n-uplet d'opérateurs $(un_1, ...un_n)$. On peut donc utiliser les algorithmes génétiques travaillant sur des gènes. Pour trouver la bonne distribution/ordonnancement des opérations sur les opérateurs, il faut parcourir (exécuter) l'ensemble des opérations en respectant certaines contraintes. Ce problème est proche de celui du *voyageur de commerce*.

Les algorithmes génétiques n'ont pas pour l'instant été testés avec un critère de minimisation de la mémoire. Ils ont été testés uniquement dans le contexte d'optimisation de la *latence* ou de la *cadence*.

Des applications types (Annexe D) ont été créées sous SynDEx de façon à traiter toutes les opérations possibles : conditionnement, explode, implode. La fonction de coût des algorithmes génétiques est réglée de telle sorte qu'elle favorise 5 fois plus l'optimisation de la cadence par rapport à celle de la latence. Dans les algorithmes génétiques, la population initiale a été fixée à 1200 individus et le critère d'arrêt est

Algorithme 4.12 Description détaillée de l'algorithme génétique pour la distribution/ordonnancement

```
début
  Sélection d'une première population au hasard
    - Sélection au hasard de l'affectation des opérations sur les opérateurs
    - Sélection au hasard d'une distribution/ordonnancement
      parmi les ordonnancements valides
  faire
    Hybridation sur l'affectation des opérateurs et sur l'ordonnancement
      - pour les affectations des échanges d'opérateurs sont faits
        sur les mêmes opérations de deux individus
      - pour l'ordonnancement dans la limite des ordonnancements valides
        des parties d'ordonnancement d'un individu sont reprises dans un autre
    Mutation des opérateurs ou de l'ordonnancement de certains individus
    Sélection par survie
      - On sélectionne certains individus parmi la population de base
       et les nouveaux venus, en prenant plus souvent les meilleurs
       individus (sélection par ranking)
    - Une partie des meilleurs individus est systématiquement gardée
      d'une génération à une autre
    - Les individus sélectionnés en même nombre que la population
      initiale définissent la nouvelle génération
  tant que certains critères d'arrêt ne sont pas vérifiés :
    - Le meilleur de la population n'a pas bougé depuis n-itérations
    - P générations ont été étudiées
```

fixé au bout de 3500 itérations stables. La cadence est le critère le plus intéressant à résoudre pour les applications de télécommunications, ce critère est donc prépondérant dans la fonction de coût. Les résultats sont donnés dans le tableau 4.20.

Programme	*SynDEx*	*Latence **	*Cadence**
exp-32	179	205	80
exp3	84	85	40
matrixmul	77	75	50
simpcond	280	285	260
exp-32c	178	240	172
exp-512	1279	1297	1193
exp-2048	4701	4722	4621

*la fonction de coût minimise à 80% la cadence et à 20% la latence sur une population initiale de 1200 individus et un critère de stabilité de 3500

FIG. 4.20 – Minimisation de la cadence avec les algorithmes génétiques

4.5.4 Optimisation de la génération de code

AAA/SynDEx crée des opérations qui lui sont propres pendant la phase de mise à plat de l'algorithme décrit par l'utilisateur :

Explode : explosion d'un buffer en plusieurs sous-buffers utilisés par la même opération ($Fork$).
Sur la figure 2.8, la relation entre les opérations $o_3 \rightarrow o_2$ est donnée implicitement par le facteur de défactorisation. Le facteur de défactorisation est un entier positif n égal au rapport entre la taille du buffer de sortie de o_3 et la taille du buffer de l'entrée de o_2. Le buffer de sortie de o_3 est séparé en n sous-buffers par l'opération $Explode$ ($n = 3$ sur la Fig. 2.8). L'implantation logicielle effectuée par SynDEx est traduite par n recopies de chaque énième sous-partie du buffer en sortie de o_3 vers chaque buffer en sortie de $Explode$. Cette implantation est due au fait que l'opération $Explode$ créée implicitement par SynDEx peut s'exécuter sur un autre opérateur que les opérations o_3, $o_{2_1}, ...o_{2_n}$. Sur l'opérateur de l'opération $Explode$, les durées de vie des n données émises en sortie de $Explode$ et la durée de vie de la donnée émise par o_3 et reçue par $Explode$ se chevauchent au moins pendant la durée de l'opération $Explode$. Les données consommées et produites par $Explode$ sont donc partagées. En appliquant les modifications apportées sur les réseaux de Petri pour les transferts de données, les déplacements de Suc_{empty} pour un envoi et de Pre_{empty} pour une réception font que les durées des données consommées et produites par $Explode$ ne se chevauchent plus. L'opération $Explode$ peut être optimisée par n simples pointeurs sur la sortie de l'opération o_3 (pour garder la contiguïté de l'espace mémoire de la donnée en sortie de o_3). Un gain est donc réalisé non seulement en terme de place mais aussi en terme de temps. La réallocation des n sous-buffers est effectuée de manière statique (hors-ligne), les n opérations de recopie ne sont plus effectuées (Fig. 4.21).

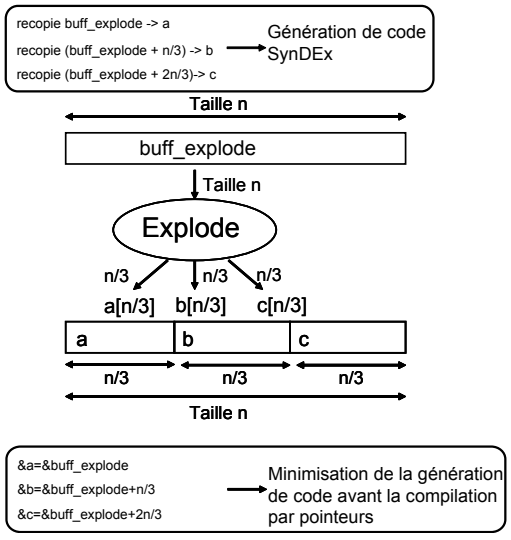


FIG. 4.21 – Minimisation de la génération de code de l'opération Explode

Implode : implosion dans un buffer utilisé par la même opération de plusieurs sous buffers ($Join$).
Comme précédemment la relation est donnée implicitement entre $o_2 \rightarrow o_3$ (Fig. 2.9). Les résultats obtenus sont équivalents à l'opération $Implode$, i.e.

une minimisation mémoire et temporelle de l'opération *Implode*. Pour garder la contiguïté de la donnée à regrouper, la réallocation statique est effectuée vers la donnée en sortie de *Implode* (en entrée de o_3).

CondI : conditionnement sur une donnée entrante. Sur la figure 4.22, la sortie de l'opération B est conditionnée pour aller suivant la valeur de la sortie de A vers l'opération $Cond_{CFG}$ ou C_1, ces deux opérations étant de fait mutuellement exclusives. On peut faire l'analogie entre l'opération *CondI* et les opérations *Implode* et *Explode*. Comme pour toute opération, SynDEx crée des entrées/sorties à l'opération *CondI*, par autant de sorties que la donnée est conditionnée (2 en sortie de B). De ce fait il est nécessaire de recopier l'entrée de *CondI* vers la sortie de *CondI* suivant le cas du conditionnement. Cette opération est minimisable dès que l'on utilise la technique de déplacement des sémaphores Pre_{empty} et Suc_{empty} : une simple réallocation des pointeurs de sortie de l'opération *CondI* vers l'entrée de cette opération est donc nécessaire. L'opération *CondI* était préalablement traduite par une recopie suivant l'état de la sortie de A vers l'opération $Cond_{CFG}$ ou C_1.

CondO : conditionnement sur une donnée sortante. La minimisation est dans ce cas un peu différente. La minimisation temporelle est effectuée de la même façon dès que l'on déplace les sémaphores. La figure 4.23 représente l'expansion de l'opération hiérarchique de l'opération G du graphe de la figure 4.22. L'opération *CondI* est minimisée par une réutilisation de l'espace mémoire des sorties de *CondI* vers l'entrée de *CondI*. Pour résoudre la recopie directe d'une opération *CondI* vers une opération *CondO* (Fig. 4.23), il n'est pas possible de faire pointer les buffers en entrée ($buffer_3$ et $buffer_4$) de *CondO* vers la sortie de *CondO* ($buffer_5$), ce qui crée dans ce cas une rupture de liaison (buffer avec 2 pointeurs différents par exemple sur le $buffer_4$) entre *CondI* et *CondO* par exemple quand la condition est égale à 2 dans l'opération hiérarchique G (Fig. 4.22 et 4.23). Une réallocation des pointeurs de sortie de *CondO* est alors réalisée vers l'entrée de cette opération, mais cette fois-ci il ne s'agit pas d'une réallocation statique mais dynamique. Les buffers finaux dans chaque branche de conditionnement sont différents ($buffer_3 \neq buffer_4$).

La sortie de *CondO* prend la valeur de son entrée à la volée suivant la condition à réaliser, afin de pouvoir prendre en compte le cas de la figure 4.22. Une fois la minimisation faite, l'opération *CondO* ne correspond pas à un temps nul comme les opérations précédentes, mais dure le temps de la réaffectation des pointeurs.

Les opérations hiérarchiques conditionnées (frontières) sont traduites lors de la transformation de graphe par des opérations coûteuses en temps et en mémoire. Grâce aux déplacements des sémaphores, ces opérations sont de durées nulles (*CondO*, *Implode*, *Explode*) ou quasi-nulles (*CondO*), et l'espace mémoire redondant entre l'entrée et la sortie de ces opérations est réutilisé.

La réutilisation mémoire pour les opérations *Implode*, *Explode* et *CondI* est réalisée par un alias des buffers de sortie sur les buffers alloués en entrée pour les opérations *Explode* et *CondI*, réciproquement pour l'opération *Implode*. Sur la figure 4.23, les deux buffers de sortie de *CondI* pointent sur le buffer en entrée de *CondI*.

De même, pour la minimisation des buffers par les algorithmes basés sur la durée des buffers (minimisation *registre* et *tétris*), les alias sont utilisés pour faire pointer

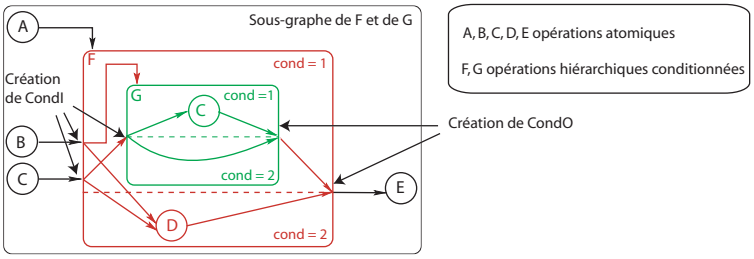


FIG. 4.22 – Minimisation de la génération de code du conditionnement *CondO*

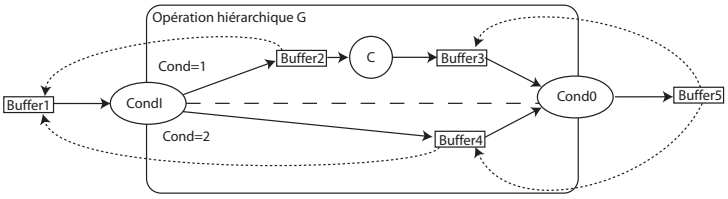


FIG. 4.23 – Expansion du graphe pour la minimisation de la génération du conditionnement

tous les buffers appartenant à une même liste sur le buffer d'entrée de cette liste de manière statique (hors-ligne). Une liste contient tous les buffers qui sont minimisables. Ces listes de buffers représentent la minimisation mémoire décrite dans les sections 4.4.1.3 et 4.4.1.4.

4.6 Conclusion et perspectives

La minimisation mémoire a donc été réalisée en différentes étapes. La première étape de la minimisation consiste à déterminer une quantité minimale de mémoire locale sans prendre en compte les buffers transférés entre processeurs (minimisation mémoire mono-processeur). Nous avons donc pour cela développé deux algorithmes de minimisation mémoire différents reposant sur la théorie des graphes (notamment du coloriage de graphe). Ces optimisations sont basées sur la durée de vie des buffers (intervalles) à partir de laquelle on peut construire un graphe d'intervalles. Le premier algorithme, appelé *registre*, utilise des buffers à la fois de même taille et de même type, le second, appelé *tétris*, exploite des buffers de même type mais pas nécessairement de même taille. La seconde étape de la minimisation complète la première, en intégrant les buffers communiqués. Pour cela, le placement des sémaphores est modifié lors de la génération de code tout en gardant les spécificités du graphe temporel fourni par SynDEx (minimisation multi-processeurs). Ceci modifie le graphe d'intervalle qui permet de minimiser les buffers communiqués inter-processeurs. Les minimisations mémoires (mono-processeur et multi-processeurs) opérées par ces deux premières étapes sont qualifiées de “spatiales". Par ailleurs, la minimisation de la génération de code et des

sommets frontières apporte un gain spatial (minimisation des recopies) et temporel (minimisation mémoire).

De plus, une minimisation pendant l'adéquation pourrait être effectuée grâce aux algorithmes génétiques qui facilitent la minimisation d'une fonction de coût multi-critères. Dans la partie suivante, nous allons donner des résultats quantifiés de minimisation sur différentes applications de télécommunication et de traitement d'images, afin de démontrer l'efficacité des algorithmes développés dans ce chapitre.

Deuxième partie

Applications dans la méthodologie AAA

Introduction générale

Les applications de traitement du signal et de l'image se caractérisent en général par des calculs intensifs, relativement systématiques à effectuer. S'intéresser à ce type d'applications se traduit donc essentiellement par regarder une classe de systèmes dits orientés données, ne présentant qu'un faible contrôle. Ces systèmes peuvent le plus souvent se décrire simplement comme un ensemble de tâches non préemptives (non interrompues), dont l'ordre d'exécution est figé et initialement connu. Il existe alors des modèles de représentation simplifiés et parfaitement adaptés à cette classe de systèmes : les graphes flots de données synchrones, qui autorisent la capture graphique du comportement d'une application, tout en offrant de fortes propriétés formelles et une prédictibilité des temps d'exécution. Ces modèles de description se révèlent finalement bien adaptés tout autant dans le domaine de la radio logicielle ou dans les télécommunications que dans celui du traitement des images. La chaîne de prototypage décrite dans la partie I s'appuie sur ces concepts de modèles flot de données.

Outre les aspects de l'optimisation de la mémoire, une autre part importante de mes travaux a concerné le développement d'applications complètes de traitement du signal et d'images, avec leur implantation temps-réel sur cible embarquée. A travers la complexité des applications traitées, il s'agit dans un premier temps de pouvoir valider la chaîne de prototypage dans son ensemble, voire de contribuer à mettre en évidence certaines de ses lacunes. Dans un second temps, ces applications doivent également servir de référence pour évaluer les performances de l'optimisation mémoire automatique intégrée dans la chaîne. Le chapitre 5 fait état du prototypage des applications de télécommunications telles la couche physique UMTS, MC-CDMA, ou plus simplement une application FM entièrement numérique. Dans le chapitre 6, nous nous intéressons au schéma de décodage de la norme MPEG4 partie 2, et au LAR, schéma de codage/décodage propriétaire. Le chapitre 7 ouvre sur de nouvelles perspectives, en particulier le prototypage d'un système multi-couches composé d'une couche de codage vidéo et d'une couche de transmission numérique.

Chapitre 5

Applications de télécommunication

5.1 Cadre général

Dans le cadre des activités de prototypage radio logicielle [MKRL01, KMRC01], l'équipe de MITSUBISHI a désiré mettre au point une démarche méthodologique de conception rapide, portable, modulaire sur des architectures multi-processeurs hétérogènes. C'est dans ce cadre que cette thèse a été financée.

5.2 FM

L'application a pour but essentiel la démonstration des fondements de la *radio logicielle*. A MITSUBISHI, elle a principalement été réalisée par des stagiaires à titre éducatif. Le portage de l'application sur une architecture multi-composants a pour objectif la validation du noyau *C6x* sur la plate-forme Pentek *p*4292. Les choix algorithmiques n'ont pas fait l'objet de modification. La description fonctionnelle de cette application a été validée en une semaine.

5.2.1 Introduction

L'application FM (*Frequency Modulation*) a été réalisée dans un premier temps lors du stage de M.CLAVERIE [Cla00]. Cette application a ensuite été complétée et validée fonctionnellement par P.DELAGRANGE. La démodulation FM stéréo est effectuée en bande de base, reposant sur un algorithme de CORDIC [Vol59]. Dans [Vol59], l'algorithme de CORDIC est décrit comme une méthode itérative permettant de calculer les fonctions trigonométriques sinus, cosinus et "l'amplitude et la phase" d'un vecteur. "CORDIC" est un acronyme de *COordinate Rotation DIgital Computer*. La méthode se fonde sur des "rotations" successives dont la somme des angles de rotation approche de plus en plus l'angle considéré. Sa particularité est de ne pas nécessiter de multiplication autre que par 2 (décalages arithmétiques) et s'avère par conséquent appropriée pour les composants électroniques et numériques.

Ce démonstrateur vise une validation fonctionnelle des bibliothèques de génération de code pour la plate-forme *p*4292 ainsi que les interfaces de cette plate-forme : TCP, interface vers la carte de conversion numérique analogique (DAC/DUC 6229) ou analogique numérique (ADC/DDC 6216). La plate-forme de démonstration de l'application

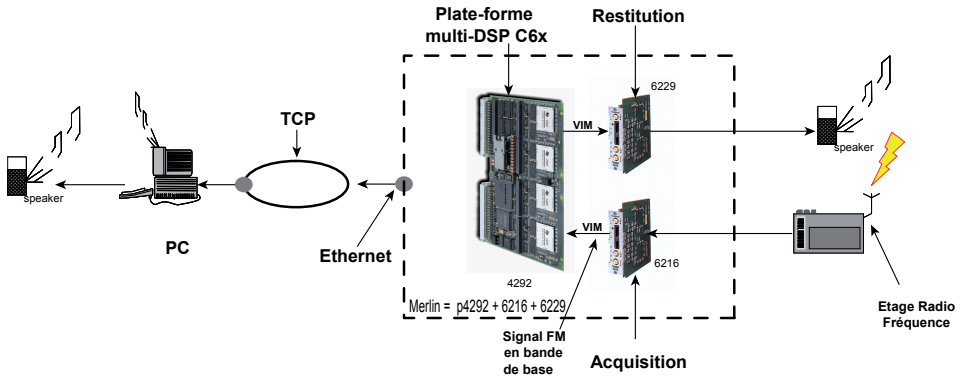


FIG. 5.1 – Démonstrateur FM

FM est illustrée sur la figure 5.1. Dans une première approche, la configuration utilisée pour ce démonstrateur suit les étapes suivantes :

étape 1 : étage radio fréquence (sortie de l'antenne) pour avoir un signal en fréquence intermédiaire,

étape 2 : acquisition par le convertisseur numérique en fréquence intermédiaire pour une conversion en bande de base,

étape 3 : démodulation FM stéréo effectuée sur le DSP C6203,

étape 4 : restitution du signal stéréo démodulé au PC via le bus TCP,

étape 5 : restitution du signal sur la carte son du PC.

Dans une deuxième approche, l'antenne est directement mise sur le convertisseur analogique numérique par P.DELAGRANGE. L'application sert de référence pour l'équipe *radio logicielle* au sein du laboratoire MITSUBISHI comme étant une application complètement radio logicielle où les traitements de la démodulation FM sont effectués en numérique de bout en bout.

5.2.2 Portage sur la plate-forme Pentek

L'application développée par M.CLAVERIE (version mono), puis par P.DELAGRANGE (version stéréo) a été reprise dans mon travail pour un portage sous SynDEx.

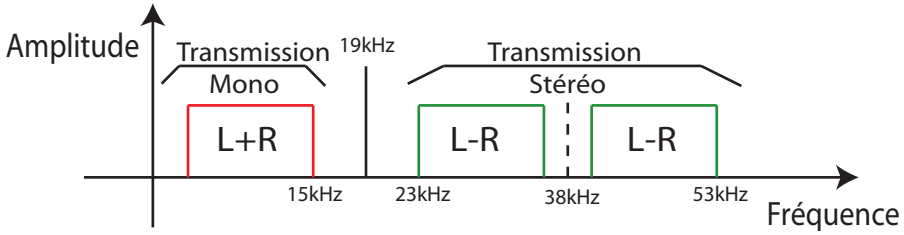


FIG. 5.2 – Détection des bandes $L+R$ et $L-R$ de la FM

La modulation FM utilisée est dans la bande de fréquence de 88 à 108 MHz. La modulation prend en entrée un signal dont la bande de fréquence est comprise entre 50 Hz et 15 KHz, et modulé sur une porteuse dans la bande FM avec un index de modulation de 5, donnant une bande passante de $2 * 75$ KHz. Chaque station est allouée sur une bande de fréquence de 200 KHz autorisant 100 stations dans la bande FM.

La modulation FM utilise les principes de la figure 5.2. Un signal audio stéréophonique comporte 2 canaux (droit et gauche) : L correspond au signal gauche et R au droit. La plage de fréquence inférieure à 15 KHz est utilisée pour la transmission mono correspondant à la somme des canaux gauche et droit $(L + R)$. Ceci permet de rendre compatible une émission stéréo avec des récepteurs mono. Le signal complet est en effet disponible dans cette plage de fréquence et les fréquences supérieures ne sont pas traitées. Dans le cas d'une radio FM stéréo, la différence des 2 canaux $(L - R)$ est transmise séparément en plus du signal mono $(L + R)$. Le récepteur réalise alors la somme $(L+R+L-R)$ puis la différence de ces signaux $(R+L+R-L)$ afin d'obtenir séparément les canaux gauche et droit.

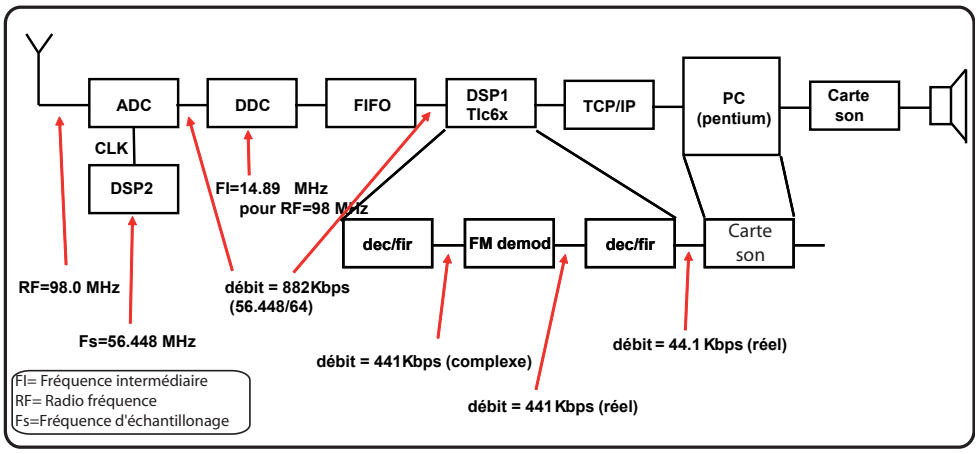


FIG. 5.3 – Schéma de principe de la démodulation FM stéréo

Une fréquence pilote de 19 KHz est utilisée pour la démodulation du spectre supérieur. La présence de cette porteuse indique au récepteur une émission stéréo. La différence des canaux gauche et droit $(L - R)$ est transmise sous la forme d'un signal à bande latérale double (*double side band*) à porteuse supprimée. Cette porteuse correspond à la sous porteuse stéréo (38 KHz).

Sur les figures 5.3 et 5.7, le réglage de la fréquence radio (RF) est effectué par le DSP. Pour cela on doit régler la fréquence du DDC "*Digital Down Converter*" de façon à se placer sur une fréquence d'une radio FM : les harmoniques de la fréquence radio RF sont dupliquées par rapport à la fréquence d'échantillonnage Fs (théorème de SHANNON) pour récupérer le signal à une fréquence intermédiaire FI (ex : $RF = 98\ MHz \Rightarrow FI = 14,89\ MHz$). Le signal acquis en sortie du convertisseur ADC/DDC a un débit de 882 $Kbps$ (kilobits par seconde). La bande de fréquence d'une radio FM de $2 * 75KHz$ est ensuite isolée par un filtrage décimateur passe-bas à réponse finie

(FIR de fréquence de coupure à 100 KHz de facteur de décimation égal à 2). Ce signal à 441 $Kbps$ est ensuite démodulé grâce à l'algorithme CORDIC. Pour récupérer la radio en mono, un filtrage décimateur passe-bas de fréquence de coupure inférieure à 15 KHz est ensuite effectué ($L+R$ décimée d'un facteur 10). La version stéréo ($L-R$) est ensuite isolée grâce à un filtrage passe-bande décimateur ($= 10$) réalisé autour de la fréquence 38 KHz (2$*$ fréquence pilote) de fréquence de coupure de 15 KHz. Les canaux $L+R$ et $L-R$ sont ensuite démultiplexés pour obtenir les voies gauches et droites à envoyer vers la carte son.

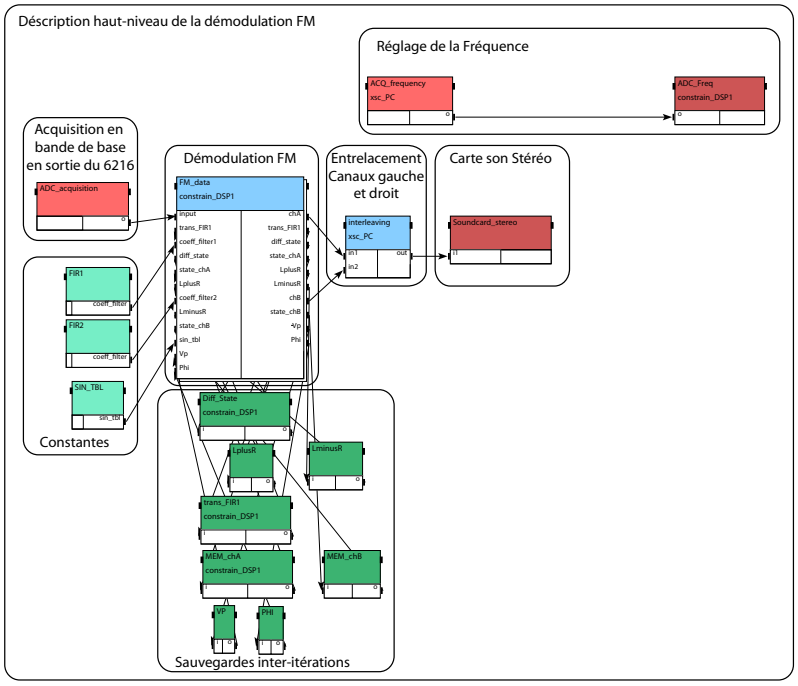


FIG. 5.4 – Démodulation FM haut niveau sous SynDEx

La description SynDEx (Fig. 5.4) a donc été effectuée pour faire apparaître les modules principaux de la démodulation FM stéréo : FIR, $CORDIC$, $L+R$, $L-R$, $demux$ (Fig. 5.5). Cette description a été réalisée de manière modulaire afin de pouvoir facilement basculer vers une application FM mono. L'apprentissage du code déjà développé, le découpage en modules, la description SynDEx et le portage temps-réel sur la plate-forme Pentek ont été accomplis en une semaine.

Le portage sur la plate-forme $p4292$ (Fig. 5.6) a été réalisé de la façon suivante :

- sur le deuxième DSP connecté au convertisseur numérique analogique (DAC/DDC), un programme est chargé pour générer une horloge CLK à $56,448$ MHz permettant de régler la fréquence d'échantillonnage du convertisseur analogique numérique (ADC) de la carte fille 6216. Ce mécanisme permet de s'affranchir de l'utilisation d'un synthétiseur de fréquence externe pour régler la fréquence du convertisseur (ADC).
- la carte analogique/numérique (DAC/DDC) est un composant matériel paramétré pour acquérir le signal provenant de l'antenne, et le renvoyer vers la FIFO.

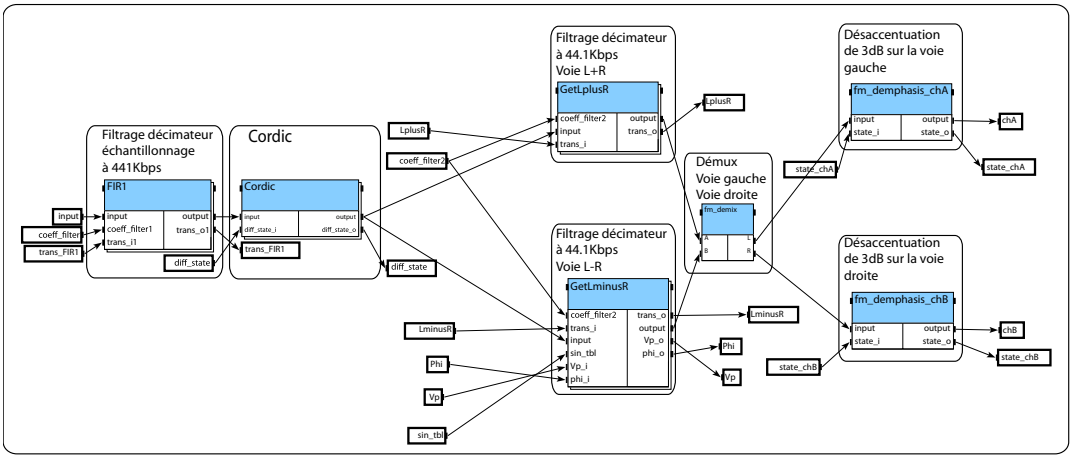


FIG. 5.5 – Description du démodulateur FM modulaire sous SynDEx

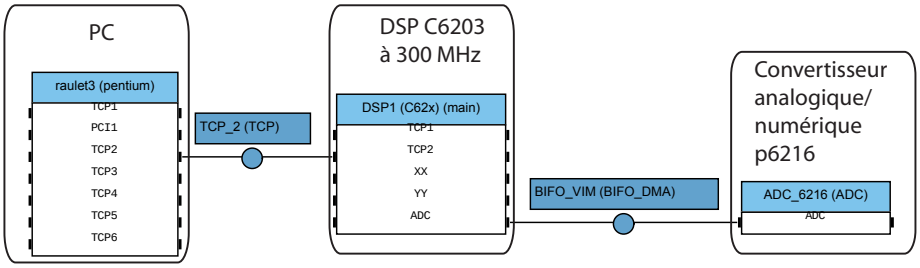


FIG. 5.6 – Description de la plate-forme Pentek sous SynDEx utilisée pour l'application FM

– sur le premier DSP, la démodulation FM est accomplie et le signal démodulé émis via le bus TCP vers le PC.
– sur le PC, le signal démodulé est envoyé directement vers la carte son.

Pour l'application FM, l'émission sur la carte son doit être continue. Il en est de même pour la réception sur le module de conversion analogique/numérique. Les données émises par le module ADC doivent donc être reçues en parallèle d'une démodulation, d'où la nécessité d'un traitement en *pipeline*. Le temps d'une démodulation ne doit pas excéder dans notre cas 10 ms (temps nécessaire à la réception sur le convertisseur pour le nombre de données choisi). Les résultats des temps de la démodulation sont indiqués dans le tableau 5.1. On remarque que le temps de démodulation FM n'excède pas $3,1$ ms et le temps d'émission vers le PC, via le bus TCP, est de 1 ms. La totalité de la démodulation et du transfert vers le PC est donc inférieure à la borne de 10 ms demandée (Fig. 5.8.a).

Sur le tableau 5.2, la minimisation mémoire sur l'application FM ne donne pas de résultats très significatifs, ceci étant lié au fait que les types des données varient fortement sur le graphe d'algorithme de l'application FM. Un gain de 10% est quand

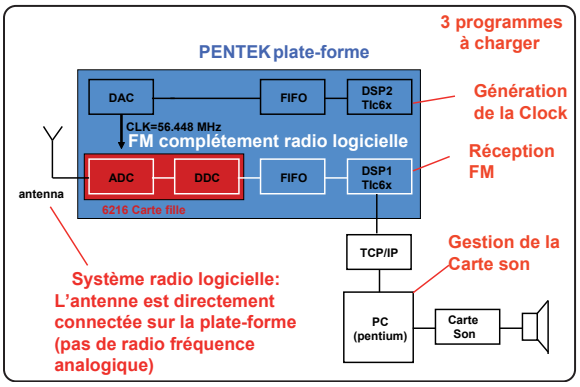


FIG. 5.7 – Application FM

Opérations hiérarchiques	**temps (μs)**
FIR1	1454
CORDIC	925
L+R	267
L-R	437
Démultiplexage	3
Désaccentuation voie gauche	13
Désaccentuation voie droite	13

TAB. 5.1 – Durée des opérations de la démodulation FM

même observé pour la minimisation *tétris* (section 4.4.1.4). Pour cette application, les sémaphores ne peuvent être déplacés sur le DSP, car cette application nécessite du *pipeline*, pour pouvoir écouter la FM en permanence. Dans le cas du déplacement des sémaphores, les opérations se dérouleraient sur le DSP de la façon suivante : réception du convertisseur, démodulation FM, émission vers le bus TCP. Ces opérations étant effectuées à chaque itération sans parallélisme (Fig. 5.8.b).

	Mémoire allouée en octets
SynDEx sans minimisation	142 448
Minimisation mémoire *registre*	131 852
Minimisation mémoire *tétris*	128 296

TAB. 5.2 – Mémoire allouée dans SynDEx avec et sans optimisation de l'application FM sur DSP

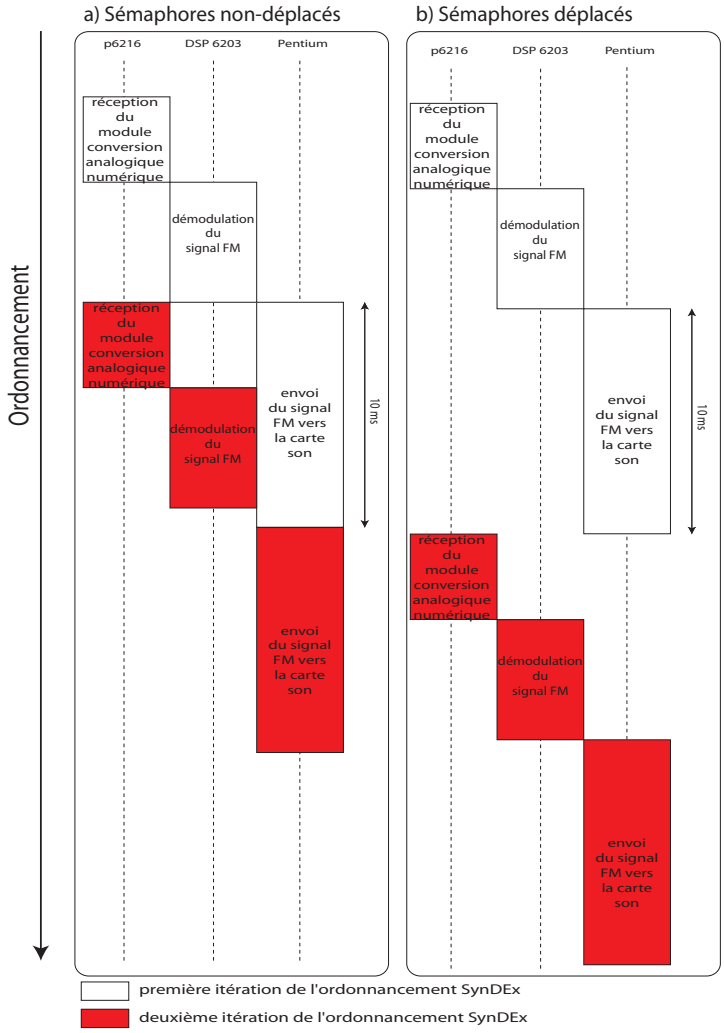


FIG. 5.8 – Ordonnancement vu du DSP de l'application FM sur deux itérations

5.2.3 Portage sur le M32R

Cette description a dans un premier temps servi de référence pour le portage sur un autre processeur dont les performances étaient méconnues. Ce processeur faible consommation est le $M32R$ [KLS+97] appartenant à *Renesas* (Alliance entre MITSUBISHI et Hitachi), dont les performances étaient à évaluer par MITSUBISHI dans le cadre de la radio logicielle, ainsi que dans le cadre de son contrat avec *Renesas*. Contrairement au $C6x$, le $M32R$ est un processeur embarqué déjà implanté par exemple dans les appareils photos numériques. Cependant, sur ce processeur, l'application FM telle qu'elle a été décrite est trop gourmande en calcul : certains modules tels la stéréo ont dû être enlevés. D'autres modules ont dû être algorithmiquement simplifiés : les filtrages, où le nombre de coefficients des lignes à retard a été réduit. Cette application permet donc de connaître la puissance de calcul du processeur $M32R$. Par

la suite, un prototypage virtuel sera envisagé pour le portage de l'application UMTS sur une architecture avec plusieurs processeurs M32R.

5.3 UMTS

5.3.1 Introduction

Conçu en 1999 et depuis peu intégré dans nos téléphones portables, l'UMTS [3GP] est un standard de transmission permettant d'acheminer texte, voix numérisée, vidéo et multimédia à un débit maximal de 2 *Mbps* (mégabits par seconde). C'est le standard choisi par l'Europe pour la 3^e génération de réseaux mobiles.

Après plusieurs reports de date, les opérateurs de téléphonie mobile ont lancé à la mi-2004 les premières offres commerciales utilisant l'UMTS (téléphone + abonnement). L'UMTS est un système numérique de communication à large bande. Il permet la transmission à la fois :

- de messages téléphoniques vocaux d'une qualité équivalente à celle de la téléphonie par réseaux fixes,
- de nouveaux services multimédia de haut débit et de nouveaux usages, comme notamment la transmission de vidéo, d'images et de sons.

Les nouveaux services imposent plusieurs modes de transmission (à commutation de circuit et à commutation de paquets) ainsi que plusieurs niveaux de qualité de service. La vidéo nécessite un délai de transmission borné mais autorise une dégradation des données avec un schéma de codage adéquat. Par contre un téléchargement s'accommode du service *"best effort"* d'IP mais ne supporte pas de dégradation des données.

Avec les systèmes 3G, le monde de la téléphonie mobile et celui des réseaux numériques tendent à fusionner et les applications issues du monde Internet se trouvent accessibles depuis un terminal UMTS. La forme d'onde utilisée est une forme d'onde UMTS simplifiée. L'UMTS pourrait atteindre à terme des capacités de transmission de 2 *Mbps* (contre $9,6$ *Kbps* pour le GSM). Dans un premier temps, l'UMTS offre des débits de 144 à 385 *Kbps* proches de la modulation EDGE (commercialisée par BOUYGUES TELECOM). Les applications et la couche physique de traitement du signal sont très demandeuses en calcul, ce qui explique d'ailleurs partiellement le retard dans l'arrivée effective de l'UMTS sur le marché.

L'UMTS présente un cas intéressant pour une implantation hétérogène multi-composants hautement efficace. Dans un contexte radio logicielle [Mit95], le but est d'implanter autant d'opérations de calcul que possible dans le domaine numérique, spécialement sur des processeurs et des architectures reconfigurables. Ses avantages consistent dans un premier temps à faciliter la conception du système en privilégiant les architectures logicielles rapides plutôt que des architectures matérielles lourdes. Dans un second temps, le système permet de nouveaux services et de nouvelles caractéristiques grâce à sa capacité d'adaptation logicielle [KMR00, KM02].

5.3.2 Le standard UMTS

Le standard UMTS est le résultat des travaux conduits en Europe sur la troisième génération de téléphones portables. La norme UMTS est décrite à travers une centaine de documents normatifs adressant tous les aspects de la communication. La forme

d'onde de l'UMTS est du type W-CDMA (*WideBand Code Division Multiple Access*). Le standard UMTS utilise donc de l'accès multiple par répartition de codes.

Les systèmes UMTS utilisent deux types de codes :

- un *spreading code* : code d'étalement à longueur variable de type *Walsh-Hadamard*,
- un *scrambling code* : code permettant de différencier les cellules, ne produisant pas d'étalement.

Ces codes ainsi que leurs usages sont répertoriés dans le tableau 5.3.

	Spreading code	***Scrambling code***
Usage en voie montante	Séparation des canaux de données et du canal de contrôle pour le même terminal	Séparation des terminaux
Usage en voie descendante	Séparation des usagers dans la même cellule	Séparation des cellules (stations de base)
Etalement	Oui	Non

TAB. 5.3 – Usage du *Scrambling code* sur une chaîne UMTS

Pour séparer liaison montante et liaison descendante entre la station de base et le terminal mobile, plusieurs normes distinctes (et incompatibles) ont été définies. En Europe, deux normes sont proposées. La première est du type FDD (*Frequency Division Duplex*) où échanges descendants et montants sont effectués en même temps sur deux fréquences distinctes. La deuxième est du type TDD (Time Division Duplex) où ces échanges sont séquencés dans le temps, sur la même fréquence.

La couche physique de la liaison radio de l'UMTS comporte plusieurs canaux : des canaux de contrôle (puissance d'émission, contrôle de flux, du débit) et des canaux de données. Le standard définit plusieurs débits possibles. Ainsi, l'UMTS utilise une bande de fréquence de 5 MHz avec un facteur d'étalement variant de 4 à 256. Le débit utile maximal géré par un seul code est de 384 $Kbps$. Pour les services à plus hauts débits, plusieurs codes sont alloués à un même utilisateur qui permet d'obtenir un débit jusqu'à 2 $Mbps$.

5.3.3 Chaîne UMTS FDD liaison montante

Les algorithmes de la couche physique UMTS FDD sont expliqués dans la norme [3GP99]. L'application UMTS était déjà disponible avant mon arrivée, cependant une description modulaire de cette application n'existait pas. Mon travail a donc consisté à aider à la description de l'UMTS FDD sous SynDEx et à réaliser son implantation sur une architecture multi-composants. Le système ne comporte ici qu'un seul canal physique et un seul canal de transport. En revanche, dans ces conditions, trois configurations sont proposées pour des débits différents :

- configuration 1 : 117 $Kbps$ sans codage canal,
- configuration 2 : 950 $Kbps$ sans codage canal,
- configuration 3 : 36 $Kbps$ avec turbo codage.

Les trois configurations ne remettent pas en cause la structure de la chaîne, ni à l'émission, ni à la réception. Seules les tailles des tableaux d'échange de données entre

certains modules, ainsi que le nombre de répétitions du "*transport bloc*" varient (4 pour les configurations 1 et 3, et 10 pour la configuration 2). Le facteur d'étalement des bits d'information des configurations 1 et 3 est de 32, et de 4 pour la configuration 2.

Cette application est en majeure partie dans le domaine libre, seuls "le turbo codage" et "le turbo décodage" sont la propriété de MITSUBISHI ITE. L'application UMTS a également servi d'application de référence pour définir un profil UML "*software radio*" (radio logicielle) dans le cadre du projet A3S (RNRT) [MRR+04].

5.3.3.1 TX émetteur

5.3.3.1.1 Description complète d'un émetteur UMTS

La voie montante du transmetteur UMTS est implantée en bande de base du *CRC (Cyclic Redundancy Check)* au *PSH (Pulse SHaping)* comme montré sur la figure 5.9. Les données peuvent être générées pour la vérification d'un taux d'erreur binaire par une source aléatoire de données (SRC), ou bien pour faire des démonstrateurs être extraites d'une application réelle, comme par exemple une séquence vidéo compressée. Une vue schématique simplifiée représentant uniquement les modules de traitement activés lors du fonctionnement en régime permanent de l'émetteur de l'application radio UMTS-FDD en voie montante est donnée en figure 5.9 pour la génération d'une trame de 10 *ms*. Des informations sur la taille des paquets de données échangés sont aussi fournies dans le cas de la configuration 1. Le nombre d'opérations effectivement utilisées est bien plus grand que la figure 5.9 ne le montre, car la plupart d'entre elles sont dupliquées plusieurs fois. Les zones grisées représentent les répétitions qui doivent être effectuées pour chaque trame. Chaque "*transport bloc*" est répété 4 fois dans cette configuration, et chaque slot l'est 15 fois. La granularité des opérations est dit "gros grain" traitant des opérations de la taille d'un filtrage, de la réorganisation mémoire, ou d'un turbo codage...

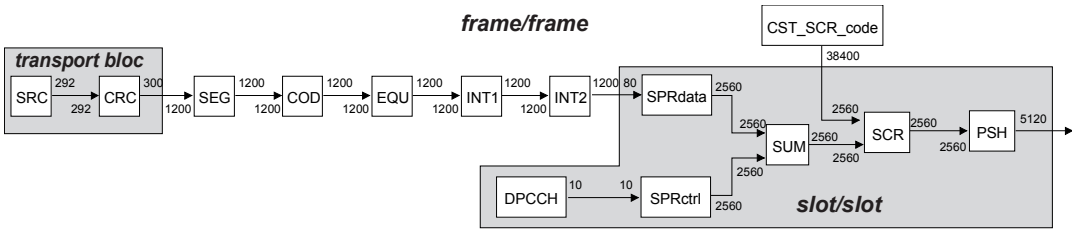


FIG. 5.9 – Schéma de principe de l'émetteur UMTS

Le système d'émission comprend dans cette configuration :
- une quinzaine de classes de modules de traitement,
- environ 130 instanciations de ces modules (en comptant les répétitions),
- une dizaine de modules d'initialisation non représentés ici.

Une exécution de ce graphe est appelée itération et correspond à la génération d'une trame UMTS, soit un temps de 10 *ms* (38400 *symboles* à 3.84 MHz).

Remarque : Certaines dépendances de données ne sont pas représentées ici, notamment celles concernant une dépendance inter-itérations pour un même bloc, ou une dépendance intra-itération mais inter-répétitions pour un même bloc. Un exemple de

dépendance inter-itération est que la fin du dernier tableau d'entrée du filtre PSH *d'une trame (le transitoire) est nécessaire en début d'opération de filtrage de la trame suivante, donc à l'itération suivante. Un exemple de dépendance intra-itération est que la séquence du code d'étalement de brouillage* CST_SCR_code *d'un slot est la suite, et donc dépend, de la position de la séquence du slot précédent.*

5.3.3.1.2 Modules de l'émetteur

Un détail des différents modules est donné dans le tableau 5.4.

modules	*nom*	*description fonctionnelle succincte*
PULSE_SHAPING		filtrage d'émission (et suréchantillonneur)
CST_PSH_init		initialisation du filtre d'émission (coefficients)
PSH_transient		tableau de transition entre deux itérations
SCRAMBLING	SCR	étalement de brouillage
CST_SCR_init	CST_SCR_code	calcul du code d'étalement de brouillage à l'init
SUM_BIT	SUM	concaténation des voies de données et de contrôle
SPREADING	SPRdata	étalement des données
SPREADCC	SPRctrl	étalement du contrôle
CST_SPR_init		calcul du code d'étalement des données à l'initialisation
CST_SPR_init		calcul du code d'étalement du contrôle à l'initialisation
CST_PRO		calcul d'une variable d'initialisation du niveau d'émission
DPCCH_SLOT	DPCCH	génération des données de contrôle slot par slot
CST_DPCCH		indice de slot
INT_SND	INT2	INT2 second entrelaceur
INT_FST	INT1	premier entrelaceur
EQU_FRAME	EQU	égaliseur de trame
CH_COD	COD	codage canal (non implanté ici)
SEG_BLK	SEG	segmentation des blocs
CRC_end	CRC	ajout d'un CRC
SPN_RND	SRC	données aléatoires
SRC_loop		indice des données
CST_TRCH		paramètres des transport channels

TAB. 5.4 – Modules de l'émetteur Tx UMTS

5.3.3.1.3 Temps des modules pour les trois configurations de l'émetteur

Trois configurations de l'émetteur ont donc été décrites sous trois graphes d'algorithmes différents sous SynDEx, celui-ci ne générant que du code statique. Il faut régler sous SynDEx les différents paramètres donné en annexe E de façon à obtenir l'une des trois configurations de l'émetteur.

Les résultats de mesures sur DSP "*TEXAS INSTRUMENTS*" C6203 à 300 MHz ont été obtenus avec les données et le programme en mémoire interne du processeur. Les premières versions pour minimiser la mémoire ont été effectuées manuellement afin de pouvoir mesurer les temps des opérations en mémoire interne. Les minimisations

manuelles réalisées pour cet exemple (déplacement des sémaphores, réutilisation de la mémoire pour les opérations *Implode Explode*) ont permis d'intégrer la totalité des données en mémoire interne.

Les mesures sur PC ont été effectuées sur l'OS Windows. Ne pouvant complètement contrôler l'exécution multi-tâches sous cet OS, lorsque plusieurs mesures étaient disponibles, la plus rapide a été choisie, car supposée non interrompue par d'autres tâches.

Pour les mesures du filtre d'émission dans le FPGA, 15 slots ont été émis depuis un DSP et le temps de mesure a été comptabilisé entre le moment de l'envoi des données vers le FPGA, jusqu'à leur retour complet du FPGA vers le DSP. Ce résultat inclut donc le temps des communications à l'entrée et à la sortie du FPGA en plus du temps de calcul du filtre lui-même. Pour information, les communications d'accès au FPGA sont cadencées à 50 MHz mais le FPGA est quant à lui cadencé à 100 MHz.

Configuration à 117 *Kbps* **sans turbo code (Tab. 5.5)**

Le temps de calcul d'une trame sur un seul DSP est donc 10 ms sans prendre en compte les transferts (temps-réel 10 ms), tandis que sur un seul Pentium, le temps est de $8,6$ ms (temps-réel 10 ms).

		DSP TI C6203 @ 300 MHz	***Pentium M @ 1700 MHz***	***FPGA XC2V3000 @ 100 MHz***
module	*données en entrée*	*temps par trame (μs)*	*temps par trame (μs)*	*temps par trame (μs)*
PSH*	5120	8649	6666	6613
SCR*	2560	514	480	
SUM*	2560	321	165	
SPRdata*	80	260	690	
SPRctrl*	10	176	705	
INT2	1200	16	5	
INT1	1200	8	6	
EQU	1200	4	6	
COD	1200	4	6	
SEG	1200	4	6	
CRC**	** 292	43	16	
DPCCH	80	5	6	
	Total	**10004**	**8757**	

* : pour un *slot* (le nombre de cycles est multiplié par 15 pour obtenir le temps pour une trame)
** : pour un *"transport bloc"* (le nombre de cycles est multiplié par 4 pour obtenir le temps pour une trame)

TAB. 5.5 – Configuration à 117 *Kbps* sans turbo code

Configuration à 950 *Kbps* **sans turbo code (Tab. 5.6)**

Cette configuration prend légèrement plus de temps, mais la différence est négligeable par rapport à la totalité du temps d'exécution de l'émetteur. Les opérations les plus gourmandes en temps de calcul dans l'émetteur traitent le même nombre de données.

Les données mises en trame dans ce cas sont en plus grande quantité mais sont moins étalées et moins *scramblées*.

		DSP TI C6203 @ 300 MHz	***Pentium III @ 700 MHz***	***FPGA XC2V3000 @ 100 MHz***
module	*données en entrée*	*temps par trame (μs)*	*temps par trame (μs)*	*temps par trame (μs)*
PSH*	5120	8649	19680	6613
SCR*	2560	514	2085	
SUM*	2560	321	525	
SPRdata*	640	231	1515	
SPRctrl*	10	176	1545	
INT2	9600	44	91	
INT1	9600	32	124	
EQU	9600	32	83	
COD	9600	32	80	
SEG	9600	32	71	
CRC**	952	268	270	
	Total	**10331**	**26069**	

* : pour un *slot* (le nombre de cycles est multiplié par 15 pour obtenir le temps pour une trame)
** : pour un *"transport bloc"* (le nombre de cycles est multiplié par 4 pour obtenir le temps pour une trame)

TAB. 5.6 – Configuration à 950 *Kbps* sans turbo code

Configuration à 36 *Kbps* avec turbo code (Tab. 5.7)

Le turbo codage est une opération un peu plus gourmande en temps que dans les configurations précédentes, cependant elle n'est exécutée qu'une seule fois et reste négligeable par rapport au temps global de l'émetteur.

		DSP TI C6203 @ 300 MHz	*Pentium M @ 1700 MHz*	*FPGA XC2V3000 @ 100 MHz*
module	*données en entrée*	*temps par trame (μs)*	*temps par trame (μs)*	*temps par trame (μs)*
PSH*	5120	8649	6645	6613
SCR*	2560	514	48	
SUM*	2560	321	15	
SPRdata*	1200	260	63	
SPRctrl*	10	176	615	
INT2	1200	16	4	
INT1	1200	8	6	
EQU	1200	4	6	
COD	1200	44	35	
SEG	1200	4	2	
CRC**	99	43	8	
DPCCH	80	5	2	
	Total	**10044**	**7449**	

* : pour un *slot* (le nombre de cycles est multiplié par 15 pour obtenir le temps pour une trame)
** : pour un *"transport bloc"* (le nombre de cycles est multiplié par 4 pour obtenir le temps pour une trame)

TAB. 5.7 – Configuration à 36 *Kbps* avec turbo code

5.3.3.2 Rx Récepteur

Le récepteur UMTS extrait les informations nécessaires pour la démodulation en utilisant le schéma représenté sur la figure 5.10. Les opérations du récepteur sont principalement les opérations duales ou inverses de l'émetteur : $PSH \Rightarrow MFL$, $SCR \Rightarrow DSCR$, $SPRdata \Rightarrow DSPRdata$, ... Des opérations supplémentaires sont nécessaires pour synchroniser le signal à démoduler : $RAKE$.

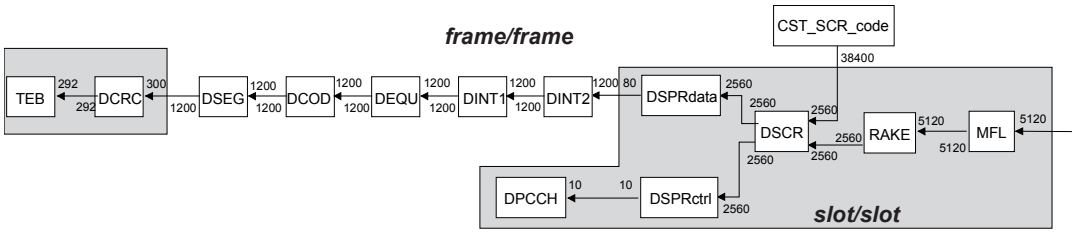


FIG. 5.10 – Schéma de principe du récepteur UMTS

5.3.3.2.1 Modules du récepteur

Un détail des différents modules est donné dans le tableau 5.8.

modules	*nom*	*description fonctionnelle succincte*
DECAL	MOV	constitution d'un tableau d'éléments complet
MATCHED_FILTER	MFL	filtrage de réception (sans sous-échantillonnage)
CST_PSH_init		initialisation du filtre d'émission (coefficients)
MFL_transient	MFL_transi	tableau de transition entre deux itérations
RAKE : RAK_MM	RAK_mem	synchronisation des trajets multiples (1 seul ici)
RAKE : RAK_finger	RAK_fin	remise en phase et énergie des trajets multiples (1 seul)
RAK_transient	RAK_transi	tableau de sauvegarde des entrées de l'itération précédente
DESCRAMBLING	CST_SCR_code	désétalement de brouillage
CST_SCR_init	CST_SCR_code	calcul du code d'étalement de brouillage à l'init
DESPREADING	DSPRdata	désétalement des données
DESPREADCC	DSPRctrl	désétalement du contrôle (pas effectué dans les timings)
CST_SPR_init		calcul du code d'étalement des données à l'initialisation
CST_SPR_init		calcul du code d'étalement du contrôle à l'initialisation
DPCCH_SLOT	DPCCH	génération des données de contrôle slot par slot
CST_DPCCH		indice de slot
DINT_SND	DINT2	second désentrelaceur
DINT_FST	DINT1	premier désentrelaceur
DEQU_FRAME	DEQU	déségaliseur de trame
DCH_COD	DCOD	décodage canal (non implanté ici)
DSEG_BLK	DSEG	désegmentation des blocs
TEB	TEB	calcul du taux d'erreurs
CST_TRCH		paramètres des *transport channels*

TAB. 5.8 – Modules du récepteur Rx UMTS

5.3.3.2.2 Temps des modules pour les trois configurations du récepteur

Configuration à 117 *Kbps* **sans turbo code (Tab. 5.9)**
Dans notre cas d'étude, sans trajets multiples et sans canal, le RAKE n'est composé que d'un seul bras (*finger*) sur notre description SynDEx. Le temps du récepteur UMTS est de $18,9\ ms$ sur un DSP, dont $16,9\ ms$ dans le filtre MFL.

Configuration à 950 *Kbps* **sans turbo code**
Les temps pour la configuration à 950 *Kbps* changent très peu, et uniquement pour la démodulation *trame par trame*. Les opérations effectuées pendant la phase d'extraction de la trame sont de durées négligeables, car effectuées à bas débit contrairement aux opérations *slot* par *slot*, et une seule fois par trame.

Configuration à 36 *Kbps* **avec turbo code**
Pour la configuration à 36 *Kbps* avec turbo code, un effort conséquent reste à faire pour la mise en mémoire interne du turbo décodeur. Le temps du turbo décodage est de 145 ms soit près de 15 $fois$ le temps réel. L'équipe radio logicielle de MITSUBISHI

		DSP TI C6203 @ 300 MHz	***Pentium M @ 1700 MHz***	***FPGA XC2V3000 @ 100 MHz***
module	*données en entrée*	*temps par trame (μs)*	*temps par trame (μs)*	*temps par trame (μs)*
MOV	5120	258	555	
MFL*	5120	16963	11100	6613
MFL_transi*	33	4	0	
RAK_mem*	5120	562	450	
RAK_fin*	2560	386	780	
RAK_transi*	5120	257	285	
DSCR*	2560	386	270	
DSPRdata*	2560	24	345	
DINT2	1200	20	6	
DINT1	1200	120	7	
DEQU	1200	4	6	
DCOD	1200	8	6	
DSEG	1200	4	2	
	Total	**18996**	**13812**	

* : pour un *slot* (le nombre de cycles est multiplié par 15 pour obtenir le temps pour une trame)
** : pour un *"transport bloc"* (le nombre de cycles est multiplié par 4 pour obtenir le temps pour une trame)

TAB. 5.9 – Configuration à 117 *Kbps* sans turbo code

avait isolé le turbo décodeur fin 2000 pour le faire fonctionner en temps-réel sur un DSP $C6201$ doté de 64 Ko de mémoire données. La durée pour le reste des traitements est de $2,7$ ms dans les calculs hors filtre et $14,7$ ms filtre inclus.

5.3.3.3 Les filtrages d'émission (*PSH*) et de réception (*MFL*)

Le filtrage d'émission (resp. de réception) de notre chaîne de transmission est d'un intérêt tout particulier, le temps de calcul représentant un pourcentage non négligeable de la totalité de la modulation (resp. démodulation). Le filtrage est effectué à travers un filtre FIR (*Finite Impulse Response*) dont la réponse impulsionnelle en cosinus surélevé est spécifiée par la norme UMTS, que ce soit pour un émetteur ou pour un récepteur en bande de base. La réponse impulsionnelle est symétrique ; cette caractéristique peut être exploitée pour minimiser le nombre d'accès mémoire, la mémoire nécessaire pour stocker les coefficients du filtre et le nombre de multiplications. Le calcul du FIR s'effectue sur 16 symboles (*chips*) et a par conséquent 33 coefficients (dans le cas d'un suréchantillonnage de 2), afin d'obtenir une réjection des signaux indésirables. Les mêmes coefficients sont utilisés pour les filtrages d'émission et de réception.

L'équation 5.1 nous donne une représentation d'un filtre FIR avec un nombre impair de coefficients où h est le vecteur à coefficients réels de la réponse impulsionnelle du filtre, K est le nombre de coefficients, $x[n]$ et $y[n]$, respectivement les n^{iemes} entrée et sortie des données complexes.

$$y[n] = h\left[\tfrac{K-1}{2}\right] \cdot x\left[n - \tfrac{K-1}{2}\right] + \sum_{k=0}^{(K-1)/2-1} h[k] \cdot (x[n-k] + x[n-K+1+k]) \quad (5.1)$$

L'application d'un filtre réel (i.e. filtre dont les coefficients sont réels) sur des données complexes est très fréquente sur des traitements en bande de base, et consiste à appliquer indépendamment le même filtre sur la partie réelle et sur la partie imaginaire des données d'entrée. Dans notre cas, nous nous sommes intéressés à des implantations en virgule fixe, ce qui impose des précautions particulières pour éviter les dépassements et préserver la qualité de signal (en terme de signal sur bruit). Le filtre à l'émission est appelé *pulse shaping* (PSH) et à la réception *matched filtering* (MFL). Un facteur de suréchantillonnage de 2 est incorporé dans le filtre Tx.

Dans ce cas nous obtenons :

$$y[n] = \sum_{k=0}^{(K-1)/4} h[2k] \cdot (x[n-k] + x[n-(K-1)/2+k]) \ \textit{si n est pair}$$

$$y[n] = h\left[\tfrac{K-1}{2}\right] \cdot x\left[n - \tfrac{K-1}{2}\right] + \sum_{k=1}^{(K-1)/4-1} h[2k] \cdot (x[n-k] + x[n-(K-1)/2+k]) \ \textit{si n est impair}$$

L'opération FIR est particulièrement bien adaptée pour une exécution sur FPGA car beaucoup de calculs peuvent être exécutés en parallèle. Cependant sur processeurs DSP il existe également des structures matérielles adaptées. Une caractéristique spécifique des DSP est l'utilisation du MAC (*Multiply ACCumulate*) ou de la structure VLIW qui supporte très bien le calcul du filtrage en un cycle d'horloge. La famille C6x, basée sur une architecture VLIW, possède 6 additionneurs et 2 multiplieurs qui peuvent opérer en parallèle en un cycle d'horloge. Une multiplication/accumulation en virgule fixe ne prend que 2 instructions sur ce DSP : une multiplication sur un cycle suivi d'une accumulation. Grâce au *pipeline,* il est effectivement possible d'effectuer deux multiplications/accumulations par cycle d'horloge.

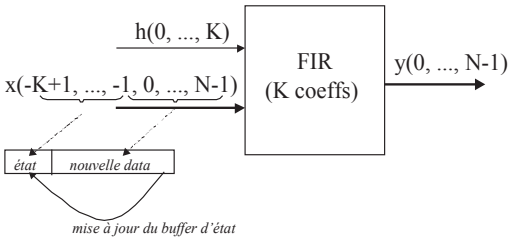


FIG. 5.11 – Bloc diagramme du FIR UMTS pour une implantation DSP

Comme chaque multiplication par un coefficient du filtre est effectuée séquentiellement, la performance du filtre dépend directement de la longueur de ce filtre (nombre de coefficients) et de la fréquence du processeur logiciel. Dans un FPGA, la parallélisation est possible sur tout ou une partie des opérations élémentaires du filtre, dépendant également de la surface disponible. Le FIR implanté dans un FPGA est un filtre à arithmétique distribuée (DA *distributed arithmetic*) [Whi89]. Ses caractéristiques sont qu'il ne possède pas de multiplicateurs, mais seulement de la ROM et

des accumulateurs. La vitesse d'exécution dépend du nombre de bits par coefficient, la surface utilisée dépend du nombre de coefficients et du nombre de bits.

Dans le cas particulier du C6x, il est possible d'utiliser un bloc diagramme simplifié du FIR comme montré sur la figure 5.11. Le FIR est un cas typique où le chemin de données des unités fonctionnelles du processeur peut accélérer les traitements. Les données sont traitées par bloc. Les interfaces consistent en un buffer d'entrée, les coefficients du filtre et un buffer en sortie.

L'algorithme accomplit une boucle sur la fonction $y[n]$ pour chaque élément en entrée. A la fin du bloc de traitement, l'état du filtrage est sauvegardé en copiant les K dernières entrées de x dans un buffer d'état. Pour l'efficacité des traitements, le buffer des données d'entrée est sauvegardé en mémoire après le buffer d'état, dans ce cas les indices négatifs pointent vers le buffer d'état.

	C62x 300Mhz	*C64x 400Mhz*	*XC2Vx 100Mhz*
Temps/slot (μs)	576	320	338

TAB. 5.10 – Temps du PSH (entrée : 2560 éléments)

	C62x 300Mhz	*C64x 400Mhz*	*XC2Vx 100Mhz*
Temps/slot (μs)	1130	640	338

TAB. 5.11 – Temps du MFL (entrée : 5120 éléments)

Dans les tableaux 5.10 et 5.11, les différences entre le C62x et le C64x sont principalement dues au fait que les compilateurs ne sont pas les mêmes pour ces 2 processeurs : les sorties des compilations sont de compatibilités ascendantes du $C62x$ vers le $C64x$. Le binaire compilé pour $C62x$ n'est pas forcément optimal pour $C64x$, de par leurs architectures internes différentes (intrinsèques différents, caractéristiques des DMA...). Dans un FPGA, l'opération FIR pourrait être plus optimisée, donnant de meilleures accélérations, mais au détriment de la surface utilisée. Cependant, ces temps sont suffisants pour obtenir une application Tx et Rx en temps réel. PSH diffère seulement d'un facteur de sous-échantillonnage de 2 par rapport à MFL. Ainsi le filtre PSH peut être plus optimal en temps et en surface. Nous avons cependant gardé le même filtre pour Tx et Rx dans le FPGA, d'où un temps d'exécution équivalent et suffisant pour le temps réel.

5.3.3.4 Portage multi-plates-formes

Quatre implantations (tableau 5.12) de l'émetteur UMTS ont été automatiquement testées en utilisant SynDEx : 3 sur une plate-forme Pentek et 1 sur une plate-forme Sundance. L'émetteur Tx dure au maximum 10 ms. Les tests ont été effectués pour la configuration à 950 $Kbps$. La première implantation sur une plate-forme Pentek n'atteint pas le temps réel sur un seul DSP $C62x$, cependant le filtre PSH est parallélisable

pour pouvoir faire un implantation multi-processeurs. Dans ce cas, avant la phase de filtrage, deux buffers de 1296 éléments sont créés comme décrit sur la figure 5.11, à partir du buffer en entrée du filtrage *PSH* unifié. Pour cela, les deux buffers doivent se chevaucher sur 16 éléments transitoires. Les 16 derniers éléments du premier buffer sont identiques aux 16 premiers éléments du second buffer. Les 16 premiers éléments du premier buffer sont la partie transitoire des 16 derniers éléments du buffers de l'itération précédente. Le temps d'exécution du PSH sur 2 processeurs est donc réduit de $1,5$ quand les transferts sont pris en compte.

De plus, la génération de code et les noyaux utilisés sur cette application ont permis de rapidement la porter sur une autre plate-forme. Le prototypage de l'UMTS sur une plate-forme Sundance a été validé directement et fonctionnellement dès son premier portage. Le prototypage n'a nécessité tout d'abord que des optimisations manuelles puis automatiques de mémoire pour pouvoir atteindre le temps réel dès le premier portage.

	Sundance	*Pentek*		
Configuration	1*C64x	1*C62x	2*C62x	1*XC2Vx 1*C62x
Temps/frame	9. 5ms	11.8 ms	8.5 ms	9.6 ms
pourcentage du PSH/ TX UMTS	50%	73%	53%	52%

TAB. 5.12 – Temps pour Tx et pourcentage de PSH par rapport à Tx

La réception UMTS a été implantée sur 3 configurations de plate-forme (tableau 5.13).

	Sundance	*Pentek*	
Configuration	1*C64x	1*C62x	1*XC2Vx 1*C62x
Temps/frame	15.9 ms	20.2 ms	9.9 ms
pourcentage du MFL/RX UMTS	60%	84%	32%

TAB. 5.13 – Temps pour Rx et pourcentage de MFL par rapport à Rx

Le portage temps réel de l'application de Rx a été accompli sur une plate-forme Pentek avec 1 DSP et 1 FPGA. La parallélisation de MFL est toujours possible sur plusieurs DSP sur la plate-forme Pentek, cependant 2 DSP uniquement ont été ajoutés. Une configuration avec 4 DSP nécessite trop de transferts sur la structure en anneau de la plate-forme Pentek, ne réduisant de ce fait que très peu le temps global de MFL.

5.3.3.5 Minimisation mémoire de l'application UMTS

L'application UMTS a servi de référence pour les optimisations mémoires. Dans un premier temps, les optimisations ont été effectuées à la main, ce qui a permis de dégager les principaux concepts. La minimisation mémoire a ensuite été intégrée dans l'outil SynDEx. Les tableaux 5.14 et 5.15 montrent un facteur de minimisation de près

de 6 grâce aux optimisations par rapport à la génération de code SynDEx, lorsque l'on utilise l'optimisation de la mémoire *tétris* multi-processeurs (données communiquées inter-processeurs section 4.4.1.4) et de la génération de code.

	Sundance Pentek		
	36 *Kbps*	117 *Kbps*	950 *Kbps*
SynDEx	1 259 516	1 267 540	1 435 348
Registre mono-composant	688 628	691 852	792 460
Tétris mono-composant	670 256	671 864	727 128
Registre multi-composants	381 428	384 652	485 260
Tétris multi-composants	367 512	367 616	406 696
Registre multi-composants + génération de code optimisée	223 892	223 916	274 220
Tétris multi-composants + génération de code optimisée	219 180	219 180	250 040
facteur de la génération initiale/finale	5,75	5,78	5,74

TAB. 5.14 – Mémoire allouée pour l'émetteur Tx

	Sundance Pentek		
	36 *Kbps*	117 *Kbps*	950 *Kbps*
SynDEx	1 969 376	1 972 592	2 090 192
Registre mono-composant	793 536	794 352	861 552
Tétris mono-composant	469 576	758 696	806 376
Registre multi-composants	506 816	505 232	555 632
Tétris multi-composants	469 576	469 576	500 456
Registre multi-composants + génération de code optimisée	350 816	349 232	382 832
Tétris multi-composants + génération de code optimisée	326 344	326 344	357 224
facteur de la génération initiale/finale	6,03	6,04	5,85

TAB. 5.15 – Mémoire allouée pour le récepteur Rx

5.4 MC-CDMA

La chaîne de communication MC-CDMA est développée à l'IETR [Le 03] groupe CPR. Le groupe CPR nous a fourni le code source des algorithmes de modulation numérique principalement écrit pour FPGA. Une transcription en langage C du code VHDL a été effectuée par le groupe image pour le portage sur DSP en virgule fixe de *"TEXAS INSTRUMENTS"*.

5.4.1 Introduction : les systèmes MC-CDMA

Afin d'augmenter la performance des systèmes de communications, un multiplexage fréquentiel est utilisé (ou modulation multi-porteuses= $OFDM$). Il consiste à répartir l'information à transmettre sur un grand nombre de sous canaux élémentaires modulés à bas débit (figure 5.12). Ceci permet de transmettre plus d'informations sans augmenter le débit d'émission sur chaque porteuse, et ainsi minimiser les interférences inter-symboles (ISI) dues aux trajets multiples.

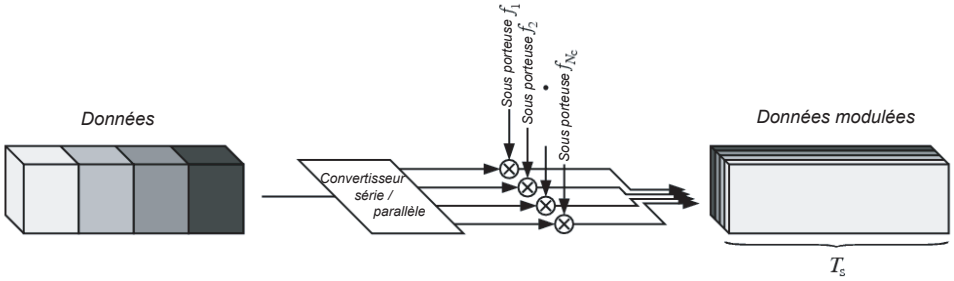


FIG. 5.12 – Principe de la modulation multi-porteuses

Les sous-canaux doivent respecter certaines conditions d'orthogonalité afin de réduire l'interférence entre sous-porteuses (ICI). Pour cela, les différents spectres des sous-porteuses peuvent être disjoints. Dans ce cas, l'occupation spectrale n'est pas optimale. Dans la modulation OFDM (pour Orthogonal Frequency Division Multiplex), les sous-porteuses se recouvrent en vérifiant des conditions d'orthogonalité [LAB95, Nob03]. L'efficacité spectrale est alors pratiquement deux fois plus élevée sans introduire d'ICI.

Plusieurs systèmes combinent les modulations à porteuses multiples et l'étalement de spectre :

- le système MC-DS-CDMA : les données sont transmises en parallèle sur plusieurs sous-porteuses orthogonales après avoir été étalées dans le domaine temporel avec un code attribué à chaque utilisateur,
- le système MT-CDMA : L'étalement est effectué après la modulation multi-porteuses, ce qui permet d'utiliser des codes plus longs que ceux utilisés pour le système DS-CDMA, pour une même occupation spectrale,
- le système MC-CDMA : parmi les trois systèmes combinant la modulation multi-porteuses et la répartition de codes, c'est le système le plus étudié, et celui qui présente le meilleur compromis performance/complexité [Nob03].

La technique MC-CDMA est basée sur la concaténation de spectre et la modulation à porteuses multiples. Contrairement aux deux autres techniques, le modulateur MC-CDMA étale les données de chaque utilisateur dans le domaine fréquentiel. Chaque sous-porteuse transmet un élément d'information multiplié par un *symbole* du code. La figure 5.13 représente le modulateur MC-CDMA dans le cas où la longueur L du code d'étalement est égale au nombre Nc de sous-porteuses.

La chaîne de communication MC-CDMA comprend un codage de canal afin d'améliorer la qualité de la transmission, de l'estimation de canal associée à de la correction

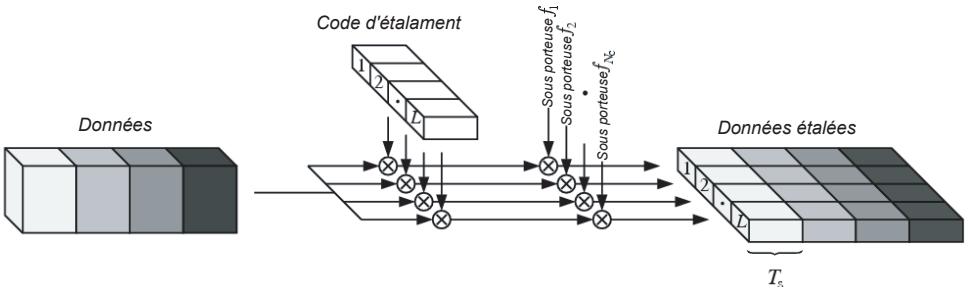


FIG. 5.13 – Principe de la modulation MC-CDMA pour les données d'un utilisateur ; Nc=L

de canal qui permet de compenser les distorsions du signal dues à la propagation, et enfin un récepteur qui peut être plus ou moins complexe selon la technique employée pour détecter l'information. Ces différents points sont développés dans [Kai98, Nob03], et ne seront pas abordés ici.

Les systèmes de communication de 4^e génération étant encore au stade d'étude, la taille du code d'étalement et le nombre de sous-porteuses ne sont pas encore fixés. Le projet européen MATRICE (*Multi-carrier CDMA TRansmission techniques for Integrated Broadband CEllular Systems*) a été mis en place dans l'optique de préparer la normalisation de tels systèmes. Les spécifications du système retiennent un code d'étalement de longueur $L = 32$ et une modulation OFDM à $Nc = 1024$ sous-porteuses (736 réellement utilisées), et une fréquence d'échantillonnage de 57.6 MHz. La durée d'un symbole OFDM est de 21.52 μs. Le débit binaire offert varie alors de 26.5 $Mbits/s$ à 119.2 $Mbits/s$ selon la configuration du modulateur. Il est à noter que ce genre de modulation figure parmi les systèmes les plus complexes et exigeants en terme de puissance de calcul, tout en combinant la complexité des systèmes OFDM et CDMA.

5.4.2 Description de la chaîne de transmission

Le graphe flot de données apparaît figure 5.14. Il est composé d'un générateur de données à moduler (*generate*), du consommateur (*Data_ out*), et de deux blocs modulateur (*Modul*) et démodulateur (*DeModul*) hiérarchiques. Le modulateur est composé d'une modulation QPSK, d'un étalement (*etal*), d'un entrelacement (*entrel*), d'une fonction qui ajoute des porteuses nulles sur les côtés du spectre (*add_ side*) et d'une modulation OFDM qui est en fait une transformée de fourrier rapide inverse (*Mod_ OFDM*). Le démodulateur est composé des fonctions duales pour pouvoir retrouver les informations.

Les données sont générées sur PC, modulées sur un DSP, démodulées sur un autre DSP et renvoyées au PC. L'architecture est donc composée de 3 processeurs dont 2 DSP.

5.4.3 Modifications apportées

Les données en entrée et en sortie de la chaîne sont des données binaires. Alors que chaque bit était codé sur un entier de 32 *bits*, le code de la modulation QPSK (et

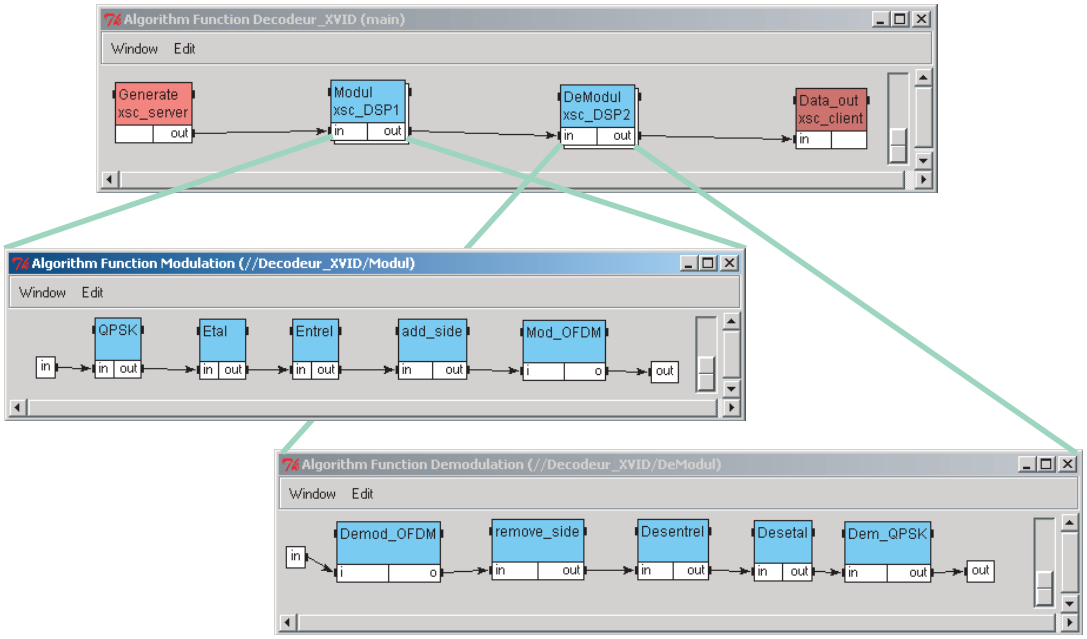


FIG. 5.14 – Description SynDEx de la chaîne de transmission MC-CDMA

démodulation) a été modifié pour que chaque bit soit maintenant codé sur un bit. Ceci permet de réduire les transferts et donc d'accélérer les traitements. Ceci est d'autant plus intéressant que le générateur de données et le modulateur se trouvent le plus souvent sur des processeurs différents.

Les données manipulées entre la modulation QPSK et la démodulation correspondante sont des complexes. Deux bits en entrée de la modulation QPSK donnent un complexe. La définition des nombres complexes a été modifiée car ils avaient été définis en virgule flottante. Le groupe CPR travaille sur des DSP $C67x$ qui fonctionnent en virgule flottante, alors que nous disposons de DSP $C62x$ et $C64x$ en virgule fixe qui réalisent donc très lentement les opérations sur les flottant. Chaque complexe est codé sur 64 *bits*, la partie réelle et la partie imaginaire sont des entiers de 32 *bits*.

5.4.4 Performances

Les diverses opérations décrites ci-dessus on été optimisées pour permettre à Code Composer d'utiliser au mieux l'architecture VLIW des DSP utilisés. Elles ont été réécrites pour prendre en compte la façon dont le compilateur optimise le code. Les performances obtenues sont mesurées sur DSP C6416 et répertoriées dans le tableau 5.16.

Les paramètres retenus dans le projet MATRICE autorisent une durée totale d'un symbole MC-CDMA de $21,5\ \mu s$. Cette durée est donc le temps maximal disponible pour réaliser l'opération de modulation.

Le processeur utilisé fonctionne à 400 MHz, et donc le temps nécessaire pour fournir un symbole MC-CDMA est : $33508 \times \frac{1}{400 \times 10^6} = 83,770\ \mu s$. Il est quatre fois supérieur au temps imparti, le temps réel n'est pas respecté. De plus, les temps de transfert ou

opération	*temps à la modulation (nombre de cycles)*	*temps à la démodulation (nombre de cycles)*
QPSK	1 432	1 168
étalement / désétalement (24×32)	16 408	16 552
entrelacement / désentrelacement (736 pts)	1 872	3 560
ajout / retrait des porteuses nulles	2 900	2 128
iFFT / FFT (fonction assembleur de la librairie Texas DSPLIB) (1024 pts)	10 896	10 450
total	***33 508***	***33 858***

TAB. 5.16 – Timings de la chaîne de communication MC-CDMA sur C6416 - Symbole MC-CDMA 1024 points dont 736 utiles

les conflits de bus dûs aux transferts par DMA ne sont pas pris en compte, et des opérations telles que le codage de canal, turbocode, estimation et correction de canal restent à implanter. La durée d'une opération telle la FFT (*Fast Fourrier Transform*) est à elle seule déjà plus longue que le temps imparti. Il est nécessaire d'ajouter des composants matériels pour réaliser ces opérations.

La chaîne de modulation numérique MC-CDMA ne fonctionne pas en temps réel sur 2 DSP, elle peut néanmoins être utilisée pour la simulation, avec un débit réduit mais assez important pour transmettre de la vidéo. Un FPGA doit être ajouté à l'architecture pour diminuer les temps de traitement.

Notre but n'étant pas le développement d'une chaîne de communications numériques, mais d'un démonstrateur, on ne cherchait donc ni à atteindre le temps réel à tout prix, ni l'intégration d'un système complet (codage de canal, correction de canal,...), d'autant plus que des travaux sont actuellement en cours au sein du groupe CPR de l'IETR.

5.4.5 Portage multi-plates-formes

Cette application a dans un premier temps été développée sur une plate-forme Sundance, et a été automatiquement prototypée grâce à SynDEx sur la plate-forme *Pentek*. Sur le tableau 5.17, la minimisation mémoire tétris de la modulation démodulation MC-CDMA donne un gain de plus de 2,1 pour la démodulation et de plus de 1,5 pour la modulation.

	Modulation sur 1 DSP	Démodulation sur 1 DSP
SynDEx sans minimisation	35 008	35 008
Minimisation mémoire *registre*	28 864	28 864
Minimisation mémoire *tétris*	28 864	22 720
Minimisation mémoire *tétris multi-composants*	22 720	16 576
facteur de la génération initiale/finale	1,54	2,11

TAB. 5.17 – Mémoire allouée (en octets) de l'application MC-CDMA

5.5 Conclusion

Différentes applications de télécommunication ont été prototypées sur différentes plates-formes (Sundance, Pentek) grâce à notre chaîne rendant toutes les étapes de développement rapides et automatiques. Les applications mises en œuvre sont relativement complexes notamment l'UMTS, avec plus d'une centaine d'opérations atomiques pour l'émetteur, et presque le double pour le récepteur. La granularité la plus fine des opérations atomiques correspond à un filtrage. Dans un premier temps, des optimisations manuelles ont été réalisées afin de définir les principes pour les algorithmes de minimisation mémoire. Les résultats obtenus grâce à la minimisation mémoire automatique ont atteint, voire dépassé, ceux d'une minimisation manuelle.

Dans le domaine des télécommunications, les applications de communications numériques peuvent principalement se décrire de manière flot de données avec un peu de contrôle. Les principales contraintes sont le traitement de masses de données plutôt faibles mais à réaliser dans des délais courts et réguliers. Les algorithmes de minimisation ont permis d'obtenir automatiquement des applications temps-réel et ont abouti à de bons résultats sur ce type de problème, les applications pouvant s'exécuter en mémoire interne.

Notre chaîne de prototypage autour de SynDEx s'avère une méthode adaptée au domaine de la *radio logicielle*. La radio logicielle cible le maximum de traitements en numérique. Il reste à étendre notre chaîne de prototypage à la reconfiguration, autre objectif de la radio logicielle. Des travaux sont en cours à l'IETR, notamment à Supélec groupe SCEE et à l'INSA groupe CPR.

Chapitre 6

Applications de traitement d'images

6.1 Cadre général

En traitement d'images, le groupe image de l'IETR s'intéresse tout particulièrement à des schémas de codage issus de la normalisation (MPEG-2, MPEG4) ou bien propriétaires comme le LAR (***L**ocally **A**daptive **R**esolution*). La volonté du laboratoire est de concilier les aspects algorithmiques au niveau du traitement d'images et le prototypage rapide de ces applications pour une exécution temps réel. L'intégration des applications vidéo constitue une des difficultés majeures dans le monde de l'embarqué.

Les applications de traitement d'images sont caractérisées avant tout par une masse de données très importante proportionnelle à la dimension des images. Dès lors, les deux principaux défis pour une implantation temps-réel embarquée sont d'une part de réaliser l'ensemble des opérations dans le temps imparti, d'autre part de réduire la mémoire utilisée.

6.2 Le codeur/décodeur LAR simple

6.2.1 Introduction

La méthode de codage LAR est une technique de compression d'images adaptative, avec ou sans perte, et développée initialement par O.DEFORGES [Déf04] au sein de l'IETR. Initialement introduite pour le codage des images multi-niveaux de gris, la méthode est maintenant étendue aux images couleurs. Le codage LAR (***L**ocally **A**daptive **R**esolution*) utilise une résolution variable, localement adaptée à l'activité locale. Dans ce schéma de codage, la grille d'échantillonnage utilisée n'est pas uniforme contrairement au codage JPEG (bloc 16x16) limitant ainsi les effets de blocs. Le schéma complet du codeur apparaît figure 6.1. Le codeur LAR pour la compression d'images en multi-niveaux de gris est composé de deux étages : un codeur "spatial" pour les forts taux de compression, et un codeur "spectral" pour coder l'image d'erreurs. La chrominance est codée séparément avec un codeur spatial.

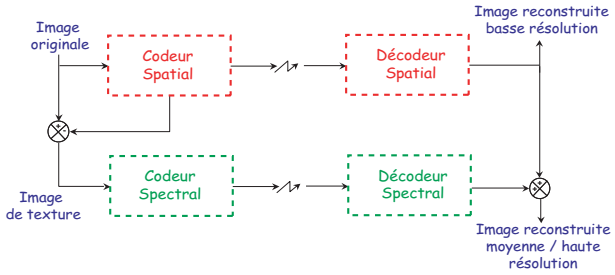


FIG. 6.1 – Schéma de principe du codage décodage LAR

6.2.1.1 Codeur spatial

Le principe repose sur la notion de résolution locale (taille de pixel) variable (Fig. 6.2). L'image est découpée en macroblocs 16x16, eux-mêmes divisés sur une structure de type *quad-tree* (taille minimale 2x2) selon l'activité locale : partitionnement 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 oeuvre 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. La méthode offre ainsi un schéma de codage "scalable" basé sur un sous échantillonnage non uniforme spatial, où l'image est transmise progressivement par raffinement local de la résolution. L'image finale de la décomposition est alors sous échantillonnée d'un facteur 2 (bloc 2x2 pleine résolution → bloc 1x1). Un post-traitement simple est appliqué pour lisser les zones homogènes (blocs supérieurs à 1), 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.

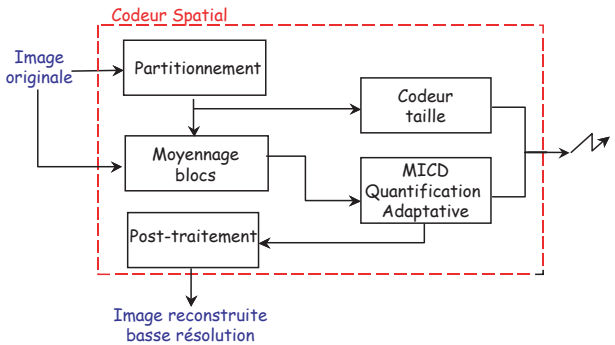


FIG. 6.2 – Schéma de principe du codeur spatial LAR

6.2.1.2 Codeur spectral

La texture des blocs (image d'erreur du codeur spatial) est alors codée avec le codeur spectral pour raffiner l'image (Fig. 6.3). Une approche de type DCT ou de type 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 à travers les grands blocs et/ou amélioration des contours à travers les blocs 2x2, les blocs 4x4, les blocs 8x8 et les blocs 16x16.

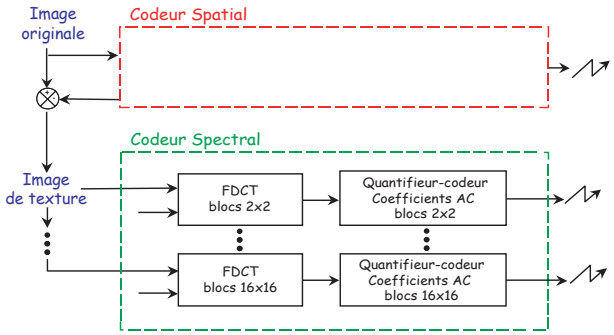


FIG. 6.3 – Schéma de principe du codeur spectral LAR

6.2.1.3 Chrominance

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.

6.2.1.4 Extension du LAR vers le pyramidal, la ségmentation et le sans perte

D'autres approches ont été développés dans le codec LAR notamment concernant la segmentation et la compression sans perte. Dans le domaine de la compression avec pertes, la première génération de codeurs intègre des algorithmes issus de la théorie de l'information : le signal est encodé pixel par pixel (moyennant certaines opérations du type décomposition, transformation, quantification). Cependant, pour ce type de méthode, le manque de fonctionnalités et de prise en compte du contenu même de l'image restreint son utilisation. Les codeurs dits de seconde génération [KIK85] s'attachent en effet à décrire l'image sous la forme d'un ensemble de zones partageant les mêmes attributs. Ces caractéristiques sont généralement choisies selon des critères issus de modèles du système visuel humain. C'est dans ce cadre qu'une description en régions de l'image a été introduite [Déf04], directement fondée sur la représentation non-uniforme propre au LAR. Ainsi, cette description apparaît à coût nul (cas

des images en niveaux de gris) ou à coût minimal (prise en compte des composantes de chrominance). Par ailleurs, il est à noter que cette méthode de segmentation est actuellement exploitée dans le cadre de la réalisation d'un schéma de compression de séquences d'images.

La conception d'une méthode de compression sans perte scalable en résolution passe notamment par la définition d'une structure pyramidale introduite dans [Bab05]. Associée à un schéma de prédiction efficace et adaptée aux propriétés de la méthode LAR, la décomposition de la pyramide réduit l'entropie résultante des symboles à transmettre. Si dans un premier temps, une solution opérant dans le domaine spatial a été proposée, deux autres approches basées sur des transformées ont permis d'améliorer encore les performances de codage de l'approche pyramidale prédictive du LAR.

Les méthodes de segmentation et compression sans perte n'ont pour l'instant pas été portées automatiquement sur une plate-forme embarquée : seule existe une description fonctionnelle de ces applications.

6.2.2 Prototypage du codec LAR simple

Le codec LAR est décrit sous SynDEx de manière hiérarchique et fonctionnelle. Le flux de l'image de luminance est incontournable et prioritaire, à celui-ci on ajoute des flux auxiliaires tels que la chrominance et/ou la texture par taille de blocs variables ; on parle ici de *scalabilité*. De façon générale, la décomposition d'un signal en plusieurs parties, à savoir sa composante principale d'une part et ses éléments subordonnés d'autre part, reste une approche fortement présente dans de nombreuses applications de traitement du signal. Le besoin de progressivité spatiale tient en particulier à l'essor des technologies Internet. En effet la transmission d'un flux correspondant à une image codée peut se concevoir comme suit : une image grossière (basse résolution) apparaît puis se voit progressivement raffinée par l'ajout de détails. Il est ainsi souvent utile d'être capable de décrire, transmettre, stocker ou encore reconstruire un signal à différentes échelles, résolutions ou niveaux de qualité.

Pour illustrer cette idée, prenons l'exemple d'une source de données unique desservant à travers un réseau un ensemble d'utilisateurs équipés de manière hétérogène. La progressivité introduite par une décomposition du signal original doit fournir à l'utilisateur le mieux équipé (calculateur puissant, haut niveau de résolution du moniteur, réseau haut débit) une image de haute qualité. Dans le même temps, le système doit pouvoir acheminer une version plus simple de l'image à l'internaute connecté par un modem basique. L'accès aux informations à des niveaux de qualité variés, en fonction de la bande passante ou de capacités du terminal (régulation de débit), apparaît comme un facteur clé du développement d'une application distribuée.

Dans notre cas, le codec d'images fixes LAR est utilisé dans la boucle infinie du traitement du signal, et l'enchaînement des images fixes est visualisé sans prédiction temporelle sous la forme d'une vidéo (succession d'images fixes). La description du codec couleur LAR illustre la scalabilité de celui-ci en différentes couches (Fig. 6.4) :

couche 1 : un codeur-décodeur de luminance rendu prioritaire et nécessaire pour le décodage progressif (Fig. 6.5),

couche 2 : un codeur-décodeur de chrominance à ajouter aux flux prioritaires et/ou à d'autres couches,

couche 3 : un codeur-décodeur spectral à ajouter aux flux prioritaires et/ou à d'autres couches. Dans notre cas, il est nommé contour car seuls les blocs 2x2 sont détaillés.

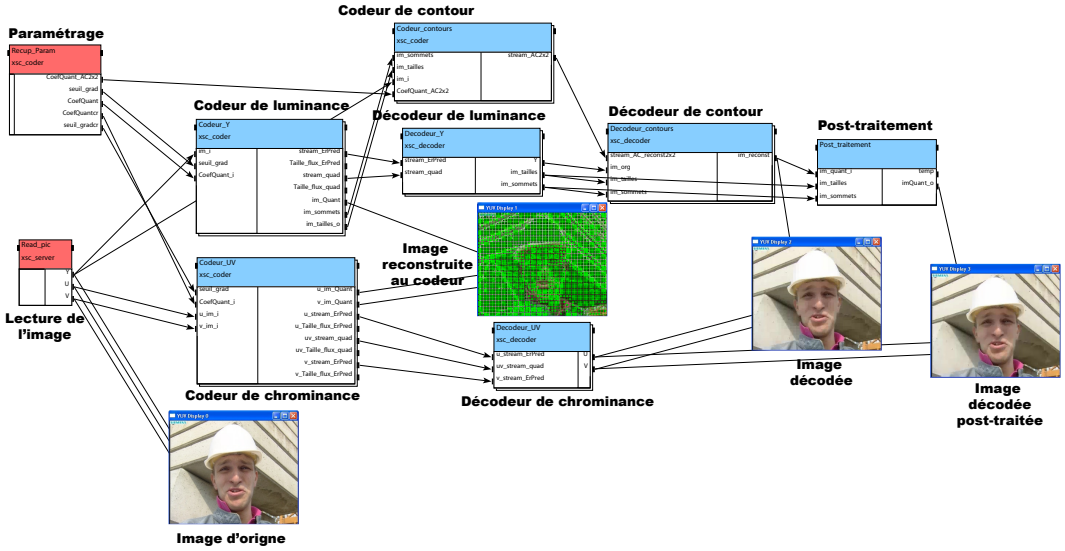


FIG. 6.4 – Description du codec couleur LAR

La description réalisée est décrite hiérarchiquement par des opérations décrivant la scalabilité et les différentes fonctionnalités de l'application LAR. Une fois réalisée l'expansion du graphe flot de données de cette application, les opérations atomiques associées traitent des données de la tailles des images : on peut caractériser ce graphe de *gros grains* [RBN+03]. La première description embarquée est la version du codeur LAR de luminance (Fig. 6.5). Le codeur de luminance nous donne des informations sur l'image des sommets représentant les points caractéristiques de la représentation en blocs et l'image reconstruite au codeur. Pour l'image des blocs de luminance, seules les valeurs utiles pour la prédiction sur la dernière colonne et la dernière ligne du bloc sont reconstruites. L'image décodée est la même que l'image reconstruite sauf que le remplissage est effectué complètement. Ensuite à ce codeur de luminance on peut ajouter l'information de chrominance. Celle-ci, dans son schéma de codage, est équivalente pour chaque chrominance à celui de la luminance. La seule différence dans le codage est la création d'une grille unique pour les deux chrominances. Le codeur de contour est quant à lui un codeur dit "spectral", il a pour but dans notre cas de rajouter une information de texture sur les contours.

Le codeur LAR a ensuite été partitionné de façon à encoder une séquence par n sous-bandes parallèles suivant la dimension horizontale de l'image. Dans le cas où $n = 2$, les parties haute et basse de l'image sont encodées par un codeur spécifique. C'est une caractéristique, souvent utilisée dans les schémas de codage normalisés, appelée *"slice"*. Cependant, dans notre technique, les sous-bandes sont distinctes sans chevauchement

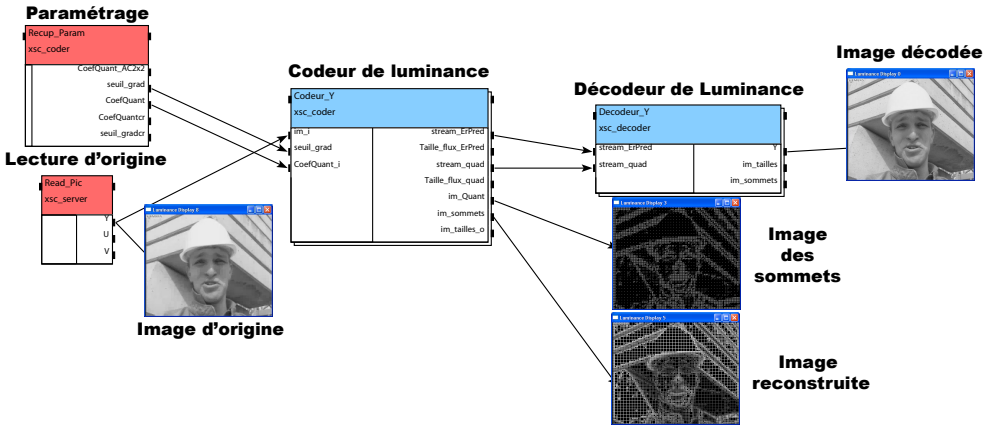


FIG. 6.5 – Description SynDEx du codec de luminance LAR

de l'une sur l'autre, donc sans prédiction d'une sous-bande par rapport à l'autre. De ce fait, le schéma de codage est moins efficace donnant un débit d'encodage plus élevé. Cette technique apporte néanmoins un gros avantage et permet de faire du parallélisme de données intensif. La figure 6.6 représente le même codec que la figure 6.5, avec un nombre de sous-bandes de 2. Cette figure illustre à la fois l'expansion du graphe flot de données et le parallélisme potentiel entre les différentes parties du codec LAR.

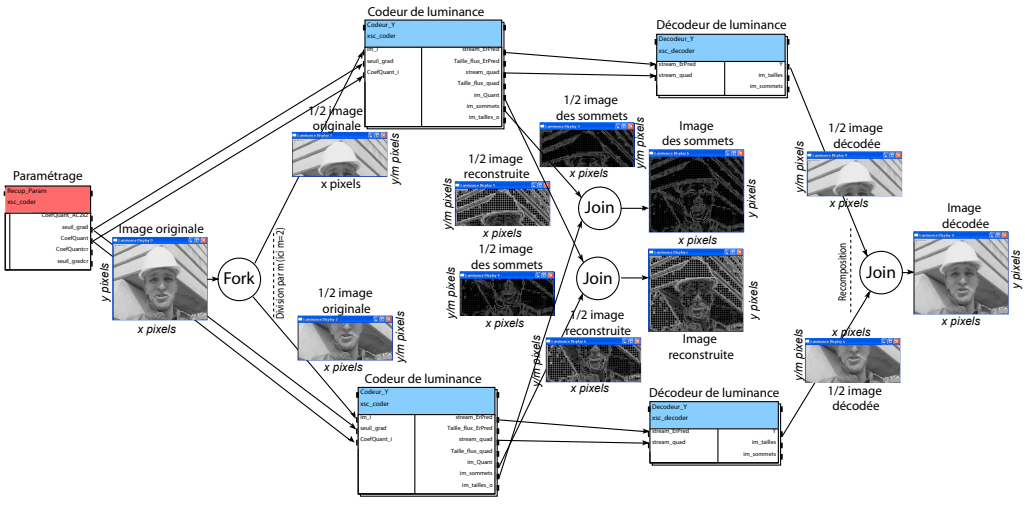


FIG. 6.6 – Description SynDEx du codec de luminance LAR avec parallélisme de données

Le portage de ces applications a été effectué dans le cas des mesures sur les architectures de la figure 6.7. Les codecs sont exécutés sur 1 ou 2 DSP, les opérations réalisées sur le PC sont l'affichage de l'image et l'ouverture du fichier à coder. L'exé-

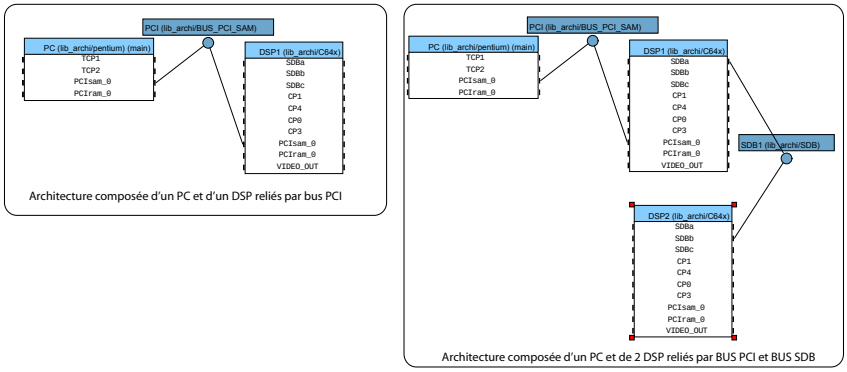


FIG. 6.7 – Architectures pour le portage des applications LAR

cution sur 2 DSP s'est faite de la façon suivante : le codage sur le premier DSP et le décodage sur le second.

	Mémoire externe		**Mémoire Interne**	
bloc	**temps (ms)**(1)	**temps (ms)**(2)	**temps (ms)**(1)	**temps (ms)** (2)
Codeur Y	276	233	74	21
Codeur UV	110	109	33	7
Décodeur Y	104	90	46	10
Décodeur UV	68	29	15	3
Codeur Contours	47	39	14	7
Décodeur Contours	92	78	20	6
Temps de traitement sur l'image	698	578	201	54
Temps de traitement moyen d'1 image	695	590	215	70
Temps de traitement de la séquence de 300 images	211s	182s	65s	21s

(1) sans optimisation du compilateur
(2) avec optimisation du compilateur

TAB. 6.1 – Temps de décodage des différentes opérations hiérarchiques du codec LAR

Le tableau 6.1 nous donne le temps des différents codeurs ou décodeurs hiérarchiques pour une séquence de 300 images *CIF* (352*288 pixels). On peut remarquer que les différents bancs mémoires jouent un rôle primordial pour l'exécution temps-réel d'une application. Le codec LAR scalable contenant le codec de contour et de chrominance est même plus rapide que le temps-réel recquis sur une séquence de 300 images, sachant que le temps de codage ajouté au temps de décodage est normalement

de 24 ms pour cette séquence. Entre la version où toutes les données sont en mémoire interne et la version où toutes celles-ci sont en mémoire externe, nous avons un facteur d'accélération de 9.

6.2.3 Minimisation mémoire du codec LAR

La minimisation mémoire automatique donne de très bons résultats (n est égal au nombre de sous-bandes dans l'image) (Tab 6.2), car tous nos décodeurs peuvent se placer en mémoire interne lorsque l'on applique l'optimisation maximale : optimisation *"tétris"* avec optimisation des buffers communiqués inter-processeurs (multi-composants). On peut remarquer sur cet exemple que plus l'on complexifie le codec LAR plus avec SynDEx il prend de mémoire. La couche 3 du codec LAR (codec spectral ou codec de contour) est très coûteuse en mémoire comme le montre la différence entre la couche 1 et 2 et la couche 1, 2 et 3 : 1,6 Mo avec SynDEx, et uniquement 200 Ko avec l'optimisation maximale.

Codec (n=1)			
monoprocesseur	**Couche 1**	**Couche 1 et 2**	**Couche 1 2 et 3**
SynDEx	1 598 140	2 348 692	3 907 452
Registre mono-composant	989 352	1 353 208	1 994 760
Tétris mono-composant	818 988	1 021 740	1 205 620
Registre multi-composants	786 600	1 074 424	1 817 352
Tétris multi-composants	616 236	717 612	901 492
facteur de la génération initiale/finale	2,59	3,27	4,33

TAB. 6.2 – Optimisation mémoire du codec LAR en *octets* (pour $n = 1$)

Le tableau 6.3 illustre les résultats mémoire du codec représenté par la figure 6.6 pour la couche 1 sur laquelle on a rajouté les différents codecs auxiliaires de chrominance (couche 2) et de contours (couche 3) sur le même principe que la couche 1 : C'est-à-dire en partitionnant chaque codec auxiliaire en n sous-bandes (ici $n = 2$). Le gain mémoire par rapport à une version avec un seul codec n'est pas significatif. Par contre, vu que les 2 codecs ici créés sont complètement dissociés, les gains en temps dans cet exemple pourront presque atteindre le gain maximal possible avec 2 processeurs.

Codec (n=2)			
monoprocesseur	**Couche 1**	**Couche 1 et 2**	**Couche 1 2 et 3**
SynDEx	1 878 020	2 768 540	4 732 784
Registre mono-composant	959 548	1 328 716	2 173 336
Tétris mono-composant	717 620	921 404	1 308 116
Registre multi-composants	756 796	1 024 588	1 950 464
Tétris multi-composants	514 868	618 308	984 092
Registre multi-composants + génération de code optimisée	819 732	1 081 172	1 677 576
Tétris multi-composants + génération de code optimisée	514 868	659 052	916 016
facteur de la génération initiale/finale	3,65	4,20	5,17

TAB. 6.3 – Mémoire allouée du codec LAR en *octets* (pour $n = 2$)

Codeur sur 1 DSP			
monoprocesseur	**Couche1**	**Couche 1 et 2**	**Couche 1 2 et 3**
SynDEx	1 079 680	1 623 620	2 460 028
Registre mono-composant	988 352	1 352 144	1 892 324
Tétris mono-composant	857 424	1 079 184	1 406 980
Registre multi-composants	886 972	1 225 408	1 866 960
Tétris multi-composants	717 612	869 676	983 724
facteur de la génération initiale/finale	1,50	1,87	2,50

Décodeur sur 1 DSP			
monoprocesseur	**Couche1**	**Couche 1 et2**	**Couche 1 2 et 3**
SynDEx	658 304	934 560	1 872 344
Registre mono-composant	649 340	796 324	1 632 696
Tétris mono-composant	611 876	681 560	1 134 576
Registre multi-composants	649 340	796 324	1 632 696
Tétris multi-composants	611 876	557 580	874 384
facteur de la génération initiale/finale	1,08	1,68	2,14

TAB. 6.4 – Mémoire allouée pour le codeur et le décodeur LAR chacun sur un DSP

Sur les tableaux 6.4, il est intéressant de noter que les gains observés sont moins importants que sur les codecs avec un seul processeur. Par contre, on notera que le codeur est capable d'absorber complètement le décodeur lorsque celui-ci est porté

sur le même processeur que le codeur (Tab. 6.2). L'optimisation mémoire satisfait complètement les critères que nous nous étions fixés, c'est-à-dire opérer sur un DSP $C6416$ à 400 MHz avec 1 Mo de mémoire interne en temps réel sur une séquence CIF (352x288 pixels).

6.3 Le codage MPEG4

6.3.1 Introduction

L'acronyme MPEG correspond au *Moving Picture Experts Group* dont les réunions ont démarré en 1988 dans le but de développer un premier standard, MPEG1, pour des applications de stockage audio/vidéo du type Video CD. MPEG a ensuite rapidement produit un second standard, MPEG2, visant essentiellement les applications liées à la Télévision Numérique. D'autres familles de standards ont depuis été produites, MPEG rassemblant un nombre croissant de spécialistes provenant de l'industrie de l'électronique, de l'informatique et des télécommunications.

MPEG4 (ISO/IEC 14496), introduit en 1998, est une norme de codage d'objets audiovisuels spécifiée par MPEG. MPEG4 est d'abord conçu pour gérer le contenu de scènes comprenant un ou plusieurs objets Audio/Vidéo. Contrairement à MPEG2 qui visait uniquement des usages liés à la télévision, les usages de MPEG4 englobent toutes les nouvelles applications multimédia comme le téléchargement et le *streaming* sur Internet, le multimédia sur mobile, la radio numérique, les jeux vidéo, la télévision et les supports Haute Définition.

MPEG4 a développé de nouveaux codecs audio et vidéo et enrichi les contenus multimédias : *Virtual Reality Modeling Language* (VRML), support pour des présentations 3D, des fichiers composites orientés objet (incluant des objets audio, vidéo et VRML), le support pour la gestion des droits numériques et plusieurs types d'interactivité.

MPEG4 se décompose en une suite de normes appelées les parties, qui spécifient un type de codage particulier. Dans chaque partie, plusieurs profils (collection d'algorithmes) et niveaux (contraintes quantitatives) sont définis. Un consortium industriel désirant utiliser MPEG4 choisit une ou plusieurs parties de la norme et, pour chaque partie, il peut sélectionner un ou plusieurs profils et niveaux correspondant à ses besoins.

Les différentes parties de MPEG4 sont définies ci-après :

- La Partie 1 décrit la synchronisation et le multiplexage de la vidéo et de l'audio.
- La Partie 2 est un codec de compression pour les signaux vidéo.
- La Partie 3 est un codec de compression pour le codage perceptuel et les signaux audio.
- La Partie 4 décrit les procédures pour les tests de conformité.
- La Partie 5 fournit des logiciels de référence des autres parties de la norme.
- La Partie 6 décrit le *Delivery Multimedia Integration Framework* (DMIF).
- La Partie 7 fournit des implémentations optimisées (cf. part 5)
- La Partie 8 décrit les méthodes de transport du MPEG4 sur IP.
- La Partie 9 fournit des implémentations matérielles des autres parties à titre d'illustration.
- La Partie 10 est un codec avancé de compression vidéo appelé aussi H.264 ou AVC (Advanced Video Codec).

- La Partie 11 spécifie la description de scène et le moteur d'application.
- La Partie 12 spécifie la format de fichier ISO *Base media*.
- La Partie 13 fournit les extensions de gestion et de protection de la propriété intellectuelle (IPMP).
- La Partie 14 spécifie le format de fichier MP4.
- La Partie 15 spécifie le format de fichier du codec AVC.
- La Partie 16 fournit l'extension du cadre d'animation (AFX).
- La Partie 17 spécifie le format *Timed Text*.
- La Partie 18 spécifie la compression et transmission de polices de caractères.
- La Partie 19 décrit le flux de texture synthétisé.
- La Partie 20 spécifie la représentation "allégée" de description de scène (pour applications mobiles).
- La Partie 21 spécifie MPEG-J GFX.
- La Partie 22 spécifie le format *Open Font*.

6.3.2 Codage d'objets audiovisuels avec MPEG4

MPEG4 a vu le jour de par le succès grandissant de la télévision numérique, du multimédia et des applications graphiques interactives. Ce nouveau format apporte les éléments techniques de standardisation pour le développement de ces applications de nouvelle génération.

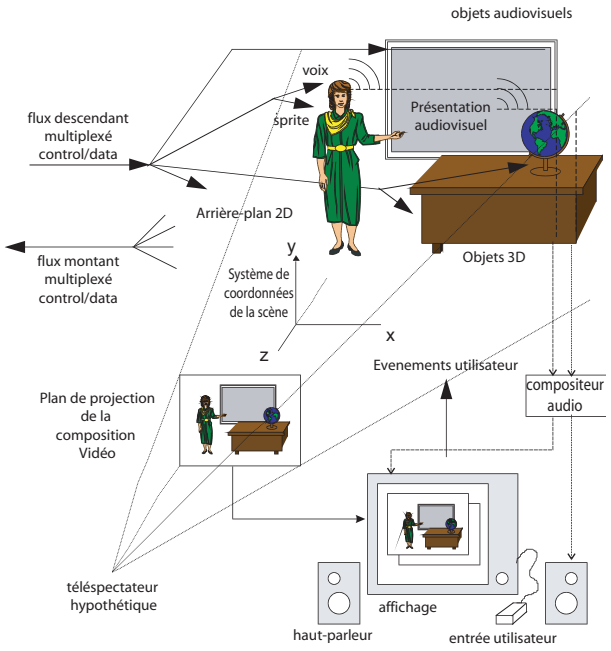


FIG. 6.8 – Exemple de scène audiovisuelle.

Un document MPEG4 est une scène constituée d'objets audiovisuels avec lesquels l'utilisateur peut interagir. La composition de ces objets et leurs caractéristiques dans la scène sont décrites dans le flux MPEG4 (Fig. 6.8). Ensuite, les données codées de

chaque objet sont multiplexées et synchronisées, de manière à être envoyées sur un canal de transmission avec une qualité de service appropriée.

6.3.3 Organisation du standard

Les schémas de codage sont définis en fonction du débit de l'application, de la complexité et de l'utilisation. Les logiciels produits à l'heure actuelle tentent soit d'intégrer l'ensemble des services, soit d'intégrer une sous-partie de MPEG4 (Codage-décodage d'image rectangulaire). Cependant ces applications ne sont pas modulaires, et donc pas évolutives, alors que seule une partie est véritablement utilisée. La minimisation des ressources, entre autres de la mémoire, et l'amélioration des performances qui en dépend, reposent sur une intégration optimale des fonctionnalités, c'est-à-dire les fonctionnalités strictement nécessaires. MPEG4 a été créé dans cette optique, avec une division en profils (*profiles*), eux-mêmes divisés en niveaux (*levels*). Le standard est alors générique, couvrant une grande plage d'applications, débits, résolutions, qualités et services. Le développeur d'une nouvelle application devra donc sélectionner un jeu de profils et de niveaux, et ne pas utiliser toute la norme. De même, la manière de réaliser chaque étape n'est pas décrite dans le standard, seuls les schémas globaux de décodage sont normalisés. Les méthodes peuvent donc être optimisées et adaptées au contexte (rapidité, taille du code ou des ressources). L'utilisation de bibliothèques MPEG4 constituées d'opérations pouvant être interconnectées dans un graphe flot de données décrivant l'application complète est bien adaptée à cette organisation de la norme.

MPEG4 partie 1 (system), 2 (visual) et 3 (audio) donnent les moyen de coder et de représenter les informations audiovisuelles. La flexibilité est obtenue par l'inclusion dans le flux compressé de paramètres qui définissent ses caractéristiques (taille des images par exemple). La quatrième partie de la norme spécifie comment les tests doivent être effectués afin de savoir si un décodeur ou les flux de bits créés par un codeur son conformes au standard. Cette partie permet de tester chaque sous-partie de MPEG4. Les graphes décrivant les algorithmes MPEG4 sont divisés en parties pouvant être testées indépendamment grâce aux tests de conformité adéquats. La vérification fonctionnelle d'une application complète peut ainsi être réalisée de manière progressive.

6.3.4 Description schématique du décodeur embarqué

6.3.4.1 Positionnement des travaux : le décodeur intra

La partie de la norme qui nous intéresse pour le décodeur est la "partie 2 : Visual". Elle décrit la syntaxe d'un flux vidéo MPEG4 en détail. Dans cette partie nous nous intéressons seulement au décodage des vidéos naturelles, c'est-à-dire les vidéos rectangulaires, les plus couramment utilisées. Des travaux antérieurs ont été réalisés sur le décodeur MPEG4 dans la thèse de J.-F.NEZAN [Nez02]. Ce décodeur est décrit sous la forme d'un graphe flot de données sous SynDEx et ne réalise que le décodage des images Intra (I). Mes travaux ont étendu ce décodeur aux images *Prédites* (P) et aux images *Bidirectionnelles* (B), et ont cherché à optimiser leur implantation.

6.3.4.2 Description haut niveau

Le décodage d'un objet vidéo MPEG4 se fait en 2 ou 3 parties : le décodage de la texture et du mouvement se font pour tous les objets, auxquels peut se rajouter le décodage de la forme si celle de l'objet est quelconque (Fig. 6.9). Comme notre décodeur ne fonctionne que pour des séquences d'images rectangulaires, le décodage d'une forme quelconque n'est pas retenu (nous avons cependant une version qui le réalise). Les opérations inverses du codeur se retrouvent dans le schéma de décodage de la texture. Le décodage se fait sur les blocs 8x8 constitutifs des macroblocs de l'image (le premier étant en haut à gauche de l'image) en fonction de leur voisinage (blocs déjà décodés) (Fig. 6.13).

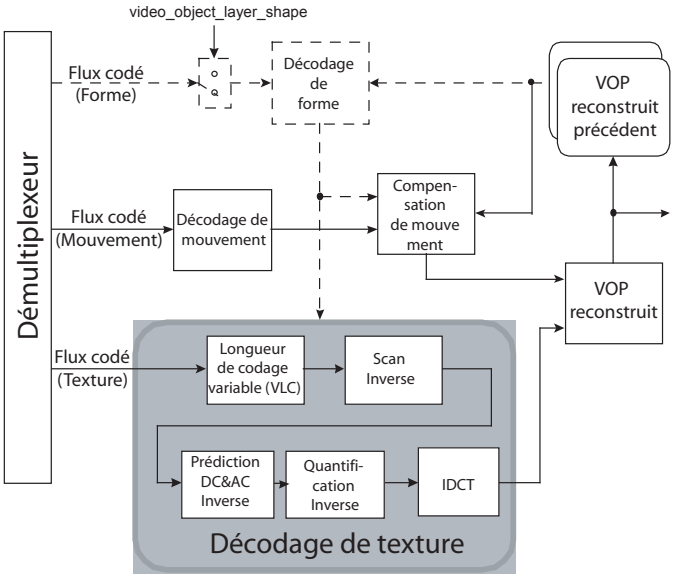


FIG. 6.9 – Schéma bloc de décodage d'un objet vidéo

Dans un premier temps, il a été intéressant de réaliser une description haut niveau du décodeur, fonctionnant sur les images I, P et B. En effet, celle-ci fournit non seulement une vision globale du décodeur sous SynDEx, peu compliquée et générée rapidement, mais sert aussi de base aux descriptions plus fines. La description haut niveau s'arrête au niveau image, c'est-à-dire aux fonctions traitant soit les images I (fonction *decode_I_frame()* dans l'application), soit les images P (fonction *decode_P_frame*() dans l'application), soit les images B (fonction *decode_B_frame*() dans l'application). Son organisation hiérarchique est représentée sur la figure 6.10.

Cette description haut niveau est basée sur le décodeur d'images Intra créé par J.-F.NEZAN. Sur le décodeur Intra, certaines fonctionnalités ont été ajoutées par rapport à la version initiale, de façon à décoder le maximum de séquences de référence. Par rapport aux séquences de référence fournies dans la norme MPEG4, nous ne décodons pas les images avec le paramètre *data partitionning* et les séquences H263. On peut considérer dans ces deux cas qu'il s'agit d'un schéma de décodage complètement

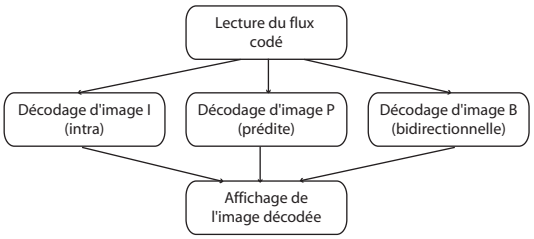


FIG. 6.10 – Schéma bloc du décodeur à granularité forte

différent. Le décodeur d'images P a dans un premier temps été extrait du logiciel libre XviD[(1)] pour être ensuite décrit plus facilement sous SynDEx. Les descriptions flot de données SynDEx ont été réalisées par D.JACQUINET lors de son stage de fin d'études sous mon encadrement. La méthode a été dans un premier temps de réaliser la description haut niveau seulement sur les images I, puis sur les images P, et enfin sur les images B.

Dans cette description, il faut d'abord extraire le flux binaire de l'image courante à décoder (*Readm4v*(), *readm4v_double_buffering*()) et régler le décodeur dans un contexte fixé au codeur (*VideoObjectLayer*() et *VideoObjectPlane*()). La lecture du flux compressé *(Readm4v*()) se fait sur PC, où les séquences vidéo sont stockées. Elle transmet le flux vidéo à un double buffer circulaire via un buffer, de taille choisie par l'utilisateur (*readm4v_double_buffering*()). Le buffer circulaire n'est alimenté par le précédent buffer partagé que si cela est nécessaire, dans le cas où le buffer ne contient pas au moins une image codée. *VideoObjectLayer*() et *VideoObjectPlane*() configurent et règlent le décodeur. Ensuite, en fonction des réglages du décodeur, la valeur de "vop_coding_type" détermine quel type d'image doit être décodé :

- 0 pour les images I pour les images intra,
- 1 pour les images P pour les images prédites,
- 2 pour les images B pour les images bidirectionnelles,
- 4 pour les images S pour les images *skipped,* non décodées.

La fonction correspondant au type d'image à décoder est alors appelée (*decode_I_frame*(), *decode_P_frame*(), *decode_B_frame*()). Enfin, la fonction *Display_YUV*(), réalisée seulement sur PC, affiche la séquence dans une fenêtre sur l'écran.

Dans un premier temps, cette version haut niveau du décodeur a été validée fonctionnellement sur PC. L'objectif était ensuite le prototypage rapide sur un ou plusieurs DSP, le PC s'occupant dans ce cas de principalement lire la séquence vidéo et d'éventuellement l'afficher. Il se trouve que l'implantation multi-composants n'est pas intéressante, ne comportant que peu de parallélisme et un gain en temps non significatif.

Une fois la description vérifiée fonctionnellement sur DSP, différentes optimisations propres au DSP ont été effectuées, accélérant au maximum les temps de calcul.

Les résultats finaux (Tab. 6.5) obtenus sur DSP sont estimés à partir du traitement d'une séquence vidéo de 4000 images, avec des tailles et des débits différents. Ces résultats sont obtenus grâce aux optimisations bas niveau effectuées sur le décodeur :

[(1)] http ://www.xvid.org

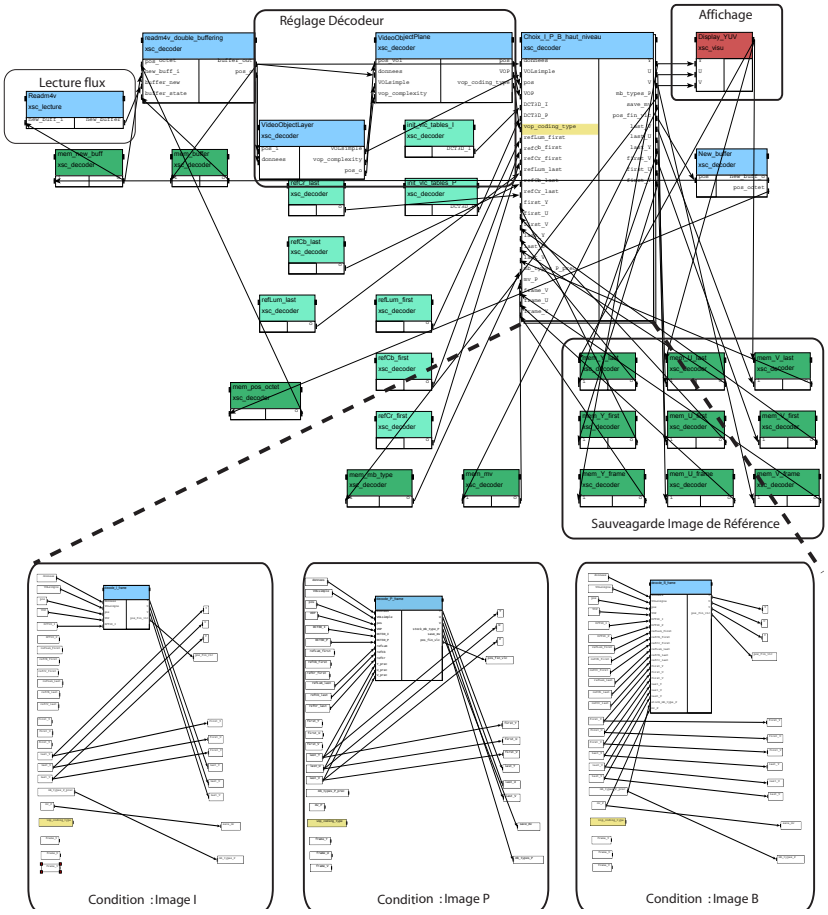


FIG. 6.11 – Graphe flots de donnée sous SynDEx du décodeur MPEG-4 haut niveau

- utilisation du QDMA pour les transferts de la mémoire externe vers la mémoire interne,
- utilisation du compilateur optimisé,
- utilisation de fonctions optimisées par *TEXAS INSTRUMENTS*
- utilisation des primitives intrinsèques pour optimiser des fonctions telles que la récupération des bits dans le flux compressé.

Les temps dans ce tableau sont donnés en milliers de cycles CPU, ce qui permet ensuite de faire une conversion suivant le DSP utilisé. Le laboratoire a acquis deux types de DSP dont les fréquences sont de 400 Mhz et 1 GHz. Ainsi pour une séquence VGA à 2048 $Kbps$, nous sommes donc, pour le décodage d'une image, largement temps réel, i.e. au maximum $20,3$ ms sur le DSP à 1 GHz (1 cycle CPU = 1 ns), tandis que sur le DSP à 400 MHz nous sommes à 70 ms au maximum pour la même opération. En moyenne sur cette séquence, nous obtenons 52 ms sur le DSP à 400 MHz.

Nous nous sommes ensuite intéressés à une comparaison entre l'utilisation de la cache sur DSP et les optimisations de code effectuées sur le DSP (tableau 6.6). Dans le cas des optimisations sur DSP, les utilisations du DMA pour transférer de mémoire interne vers externe sont décrites à l'intérieur d'une opération atomique sous SynDEx

	Temps de décodage des images[1]					
			I			P
	max	*min*	*moyenne*	*max*	*min*	*moyenne*
QCIF - 512 Kbps	2 475	1 175	1 654	2 373	825	1 557
QCIF - 1024 Kbps	2 475	1 177	1 845	2 678	817	1 817
QCIF - 2048 Kbps	2 513	1 220	2 152	2 780	911	2 227
CIF - 512 Kbps	6 653	4 583	5 259	6 380	2 802	4 372
CIF - 1024 Kbps	7 217	4 913	4 293	6 924	2 903	4 956
CIF - 2048 Kbps	9 007	4 569	6 285	9 307	2 946	5 726
VGA - 512 Kbps	18 733	14 060	14 811	14 914	7 753	10 806
VGA - 1024 Kbps	18 734	14 080	15 529	15 777	8 177	12 150
VGA - 2048 Kbps	20 207	14 060	16 329	20 276	8 259	13 649

	Temps I et P[1]	Temps transferts[2]	Nombre d'images I	Nombre d'images P
QCIF - 512 Kbps	1 559	2 535	54	3 946
QCIF - 1024 Kbps	1 817	2 543	54	3 946
QCIF - 2048 Kbps	2 226	2 674	54	3 946
CIF - 512 Kbps	4 384	6 404	56	3 944
CIF - 1024 Kbps	4 967	6 268	56	3 944
CIF - 2048 Kbps	5 734	6 428	56	3 944
VGA - 512 Kbps	10 868	16 616	62	3 938
VGA - 1024 Kbps	12 202	16 896	61	3 939
VGA - 2048 Kbps	13 689	16 322	60	3 940

[1]en milliers de cycles CPU toutes images confondues images I et P
[2]durée des transferts PC-DSP en milliers de cycles CPU
VGA = 640*480 pixels, CIF=352*288, QCIF = 176*144

TAB. 6.5 – Temps moyen de décodage MPEG-4 haut niveau sur DSP pour les images I et P

réalisant des transferts rapides inter-mémoires (opération logicielle). Seules les données de type image (grande taille de données) ont été mises en mémoire externe, le reste étant en mémoire interne.

Avec l'utilisation de la mémoire cache, les opérations décrites auparavant avec le DMA sont cette fois-ci décrites de façon logicielle, c'est-à-dire à l'aide de fonctions de recopies. Les données dans le cas de la mémoire cache ont toutes été mises en mémoires externes et donc gérées automatiquement par celle-ci pour rapatrier les données en mémoires internes. Dans le cas où toutes les données sont en mémoires externes et que la mémoire cache est inactive, nous avons obtenu des facteurs d'accélération de 3 à 6 suivant les types d'images décodées grâce à la mémoire cache. Par contre la différence entre une version optimisée avec DMA et une version avec la mémoire cache ne donne pas un gain si important que cela : 15%.

Il est important de noter sur cette séquence que les images I sont 2 fois plus grosses en taille que les images P qui elles sont 8 fois plus grosses que les B. Cependant le temps de décodage est plus lent pour les images B que les P ou les I. Ceci est dû au fait

que les accès nécessaires vers la mémoire externe sont plus nombreux pour les images B et les images P, qui ont besoin avant d'être décodées d'une recherche du meilleur bloc dans l'image précédente pour les images P, ou dans le cas des images B d'une interpolation du meilleur bloc dans les deux images de référence précédentes.

On peut remarquer que les accès vers la mémoire externe sur le DSP de chez *TEXAS INSTRUMENTS* sont très coûteux, et qu'il est nécessaire de trouver des mécanismes permettant d'accélérer ces accès mémoires : la mémoire cache ou le DMA.

Interne(1)	*Image* I		*Image* P		*Image* B	
	*temps**	*nb octets*	*temps**	*nb octets*	*temps**	*nb octets*
moyenne	4 293	11 779	5 546	5 129	7 379	737
max	4 913	23 301	6 030	15 233	7 750	1 412
min	3 972	5 656	5 107	1 923	6 949	235

Cache(2)	*Image* I		*Image* P		*Image* B	
	*temps**	*nb octets*	*temps**	*nb octets*	*temps**	*nb octets*
moyenne	4 528	11 779	10 365	5 129	19 737	737
max	5 164	23 301	10 828	15 233	24 865	1 412
min	4 189	5 656	9 804	1 923	15 428	235

(1)données en mémoire interne -sauf données décodées gérées par le QDMA
(2)toutes les données en mémoire externe gérées par la mémoire cache
* les temps sont donnés en milliers de cycles CPU

TAB. 6.6 – Temps de décodage MPEG-4 sur DSP pour les images I, P et B sur une séquence 352*288 pixels à 1024 *Kbps*

6.3.4.3 Description bas niveau

Après la description haut niveau du décodeur, une description à grain plus fin a été réalisée sous SynDEx (uniquement sur les images I et P). Celle-ci est plus intéressante dans le cadre d'une implantation sur une plate-forme multi-DSP de par le parallélisme potentiel offert. Cette description raffine la description de haut niveau réalisée précédemment. Les opérations les plus finement détaillées sont représentées sur la figure 6.12, celles-ci sont donc de la grosseur des opérations nécessaires au décodage de la texture d'un bloc à l'intérieur d'un macrobloc : code de longueur variable, *scan* inverse, prédiction AC/DC, quantification inverse, DCT. La description SynDEx comporte 6 niveaux hiérarchiques pour les images I et 8 niveaux pour les P, le niveau hiérarchique le plus bas étant à chaque fois le décodage de la texture.

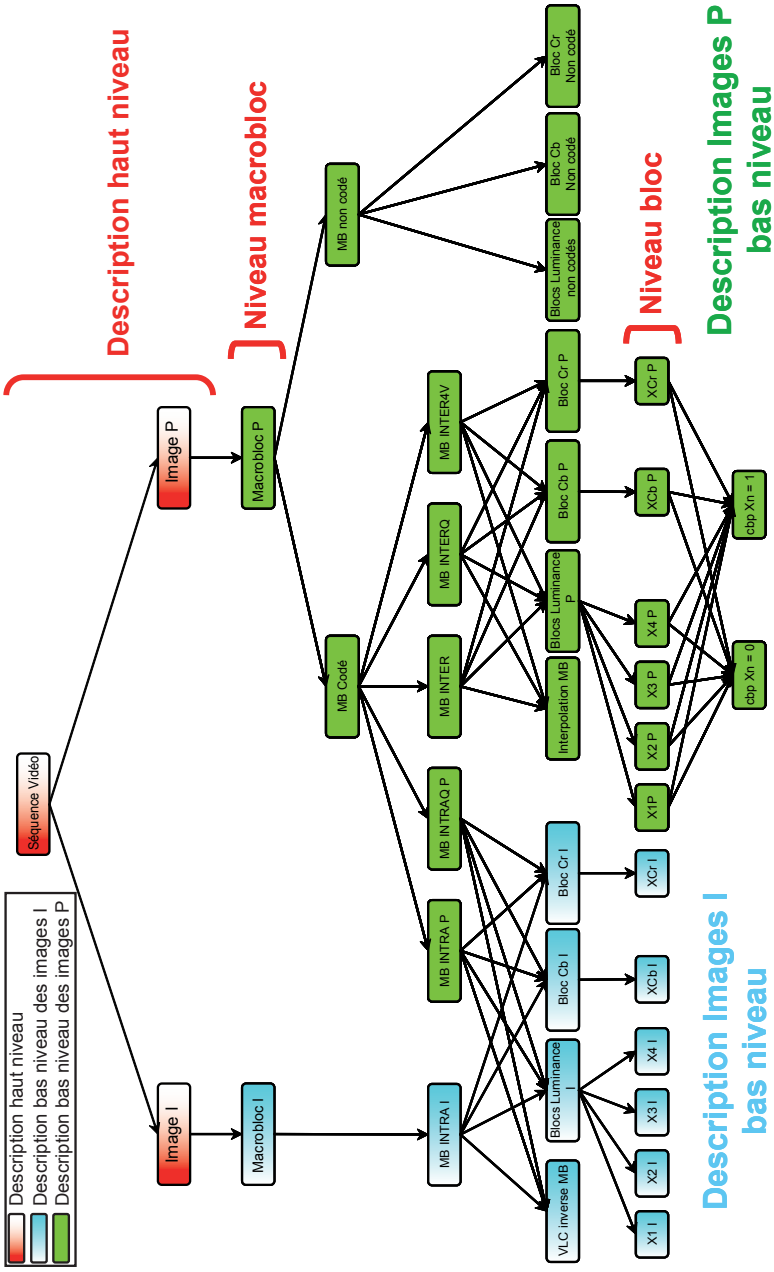


FIG. 6.12 – Schéma bloc du décodeur d'images I et P bas niveau

Certaines opérations, comme par exemple la DCT inverse, ne nécessitent pas de connaître la valeur des blocs voisins. Cette opération peut être réalisée indépendamment de son voisinage direct, ce qui n'est pas le cas pour la prédiction AC/DC inverse ou la VLC inverse. Les blocs 8x8 des macroblocs sont décrits séparément dans le graphe d'algorithme SynDEx. La description est bien réalisée sur des images complètes, mais les calculs des blocs (les 4 de luminance X_1, X_2, X_3 et X_4 et les 2 de chrominance X_{Cb} et X_{Cr}) sont différenciés pour le parallélisme (Fig. 6.13).

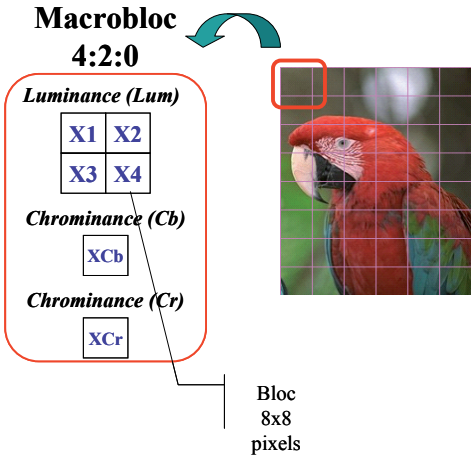


FIG. 6.13 – Différents blocs MPEG-4 dans un macrobloc

	Temps de décodage des images I[1]			*Temps moyen*
	max	*min*	*moyenne*	*des transferts*[2]
bas niveau	610 824	267 424	457 432	504 917
haut niveau	575 320	236 880	423 428	493 421

[1]en cycles CPU
[2]durée des transferts PC-DSP principalement (buffer 80*64), en cycles CPU

TAB. 6.7 – Différence du temps de décodage MPEG-4 entre la description bas et haut niveau du décodeur MPEG4

Sur le tableau 6.7, la description bas niveau aboutit à de moins bons résultats par rapport à la description haut niveau. La différence est principalement due à des artifices nécessaires à la description bas niveau, comme les passages de paramètres d'un macrobloc à un autre que l'on a pu réaliser directement par pointeur dans la description de haut niveau.

Une implantation bas niveau de cette description a été réalisée sur une plate-forme multi-DSP. Cependant le modèle de SynDEx décrit dans la thèse de T.GRANDPIERRE ne prenant pas en compte le temps d'initialisation d'un transfert (celui-ci étant considéré comme nul) l'implantation multi-DSP sur une description à *grain fin* ne donne pas de résultats intéressants. Le temps d'initialisation d'un transfert devient prépondérant par rapport à une opération élémentaire. La mise à plat du graphe SynDEx nous donne dans ce cas, par exemple sur une séquence 80*64 pixels (20 macroblocs) contenant des images I et P, un nombre non négligeable de 9390 opérations (Tab. 6.8). SynDEx ne fait pas dans la version actuelle de regroupement d'opérations (factorisation) qui permettrait de minimiser l'impact du temps d'initialisation d'un transfert [KB01]. SynDEx dépend d'un algorithme *glouton* dont la fonction de coût ne voit pas plus loin que le voisinage directement connecté à l'opération déjà ordonnancée. Il est alors difficile de trouver une implantation multi-processeurs optimale. Les algorithmes génétiques peuvent plus facilement intégrer ce temps d'initialisation

dans l'ordonnancement des opérations de chaque individu car la fonction de coût est évaluée une fois l'ordonnancement effectué.

Nous avons donc fait le regroupement d'opérations manuellement. Pour cela une troisième description appelée "niveau intermédiaire" au niveau macrobloc (6 opérations blocs dans un macrobloc) a été réalisée. Elle regroupe toutes les informations au niveau de la texture en une seule opération. Néanmoins le parallélisme est plus difficile à extraire car les opérations les plus indépendantes sont englobées dans l'opération bloc 8x8 alors que le traitement de l'ensemble de ces opérations bloc 8x8 ne peut être exécuté que séquentiellement. Cependant cette description permet de trouver plus rapidement une solution de distribution/ordonnancement grâce à SynDEx.

taille de la vidéo[(1)]	**nombre d'opérations**
1 MB	**493**
5 MB	**2 385**
10 MB	**4 720**
20 MB	**9 390**
50 MB	**23 400**
99 MB (QCIF)	**32 273**

[(1)]en nombre de macroblocs MB

TAB. 6.8 – Nombre d'opérations du décodeur bas niveau MPEG-4 suivant le nombre de macroblocs

6.3.5 Minimisation mémoire des décodeurs MPEG4

	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)*

TAB. 6.9 – Occupation mémoire du décodeur MPEG-4 suivant la granularité de la description

Sur le tableau 6.9, nous comparons les 3 descriptions 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. Par contre, si on descend plus bas dans la hiérarchie, la mémoire utile

augmente. Ceci est dû au parallélisme potentiel plus important dans cette description. On notera tout de même une minimisation d'un facteur de 52 par rapport à la version sur une séquence 176*144 pixels.

Il est intéressant d'utiliser les algorithmes de minimisation mémoire sur la description niveau intermédiaire vu le nombre d'opérations mises en œuvre. Afin de pouvoir quantifier l'impact de l'optimisation mémoire, nous avons réalisé une description susceptible de ne décoder qu'un certain nombre de macroblocs dans une image 80*64 pixels (Tab. 6.10). Chaque décomposition d'un macrobloc rajoute 63 opérations dans la mise à plat du graphe SynDEx. SynDEx (sans minimisation mémoire) augmente ainsi la mémoire de 159 Ko pour chaque nouveau macrobloc à décoder. Dans le cas de la minimisation *"tétris"* et *"registre"*, seul 1 Ko supplémentaire est nécessaire par macrobloc. La version SynDEx nécessite pour le décodage de la totalité de l'image 10 fois plus de mémoire que les versions minimisées *"tétris" et "registre"*.

80*64	SynDEx*	Registre*	Tetris*	nb opérations
1 MACROBLOC	328 028	245 268	225 156	97
2 MACROBLOCS	487 432	316 928	288 404	184
3 MACROBLOCS	646 068	318 136	295 808	247
4 MACROBLOCS	804 752	319 240	296 448	310
5 MACROBLOCS	963 484	319 744	296 912	373
6 MACROBLOCS	1 122 264	320 632	297 436	436
7 MACROBLOCS	1 281 092	321 520	297 940	499
8 MACROBLOCS	1 439 968	322 756	298 692	562
9 MACROBLOCS	1 598 892	323 648	299 148	625
10 MACROBLOCS	1 757 864	325 240	299 628	688
11 MACROBLOCS	1 916 884	326 248	300 116	751
12 MACROBLOCS	2 075 952	326 856	300 572	814
13 MACROBLOCS	2 235 068	327 340	301 028	877
14 MACROBLOCS	2 394 232	328 100	301 484	940
15 MACROBLOCS	2 553 444	329 016	301 940	1 003
16 MACROBLOCS	2 712 704	329 916	302 396	1 066
17 MACROBLOCS	2 872 012	330 808	302 852	1 129
18 MACROBLOCS	3 031 368	332 804	303 308	1 192
19 MACROBLOCS	3 190 772	333 740	303 764	1 255
20 MACROBLOCS	3 350 224	332 948	304 220	1 318

* mémoire nécessaire en nombre d'octets

TAB. 6.10 – Minimisation mémoire sur le décodeur MPEG-4 niveau intermédiaire pour une taille d'image 80x64

Sur la description de haut niveau, nous avons principalement minimisé de façon automatique les opérations ajoutées pendant la phase de mise à plat dans SynDEx. Sur la description du décodeur avec image I, P et B, 48 recopies ont été économisées (Tab. 6.11). Ceci a permis également de réduire d'un facteur 1,6 la mémoire allouée pour ne plus utiliser que 2 Mo de mémoire au lieu de $3,2$ Mo. Les opérations créées par SynDEx peuvent être coûteuses en temps et en espace comme montré sur cet

exemple. Le gain en mémoire est plus important pendant la phase d'optimisation de la génération de code que pendant la phase d'optimisation purement mémoire. Sur cette version, nous nous sommes donc intéressés à la minimisation conjointe entre le temps et la mémoire sur un seul processeur : la minimisation temporelle (moins de recopie) et spatiale (moins de mémoire).

mono-processeur	Décodeur I P	Décodeur I P B
SynDEx	1 812 292	3 205 968
Registre mono-composant	1 812 280	2 901 828
Tétris mono-composant	1 812 272	2 901 424
Registre multi-composants	1 685 560	2 775 108
Tétris multi-composants	1 583 780	2 698 672
Registre multi-composants + optimisation génération de code	1 203 992	1 969 612
Tétris multi-composants + optimisation génération de code	1 203 984	1 969 604
Nb recopies optimisées	13	48
facteur de la génération initiale/finale	1,51	1,63

TAB. 6.11 – Temps moyen de décodage MPEG4 haut niveau sur DSP pour les images I et P

6.3.6 Démonstrateur

Dans le cadre de nombreux démonstrateurs, nous avons utilisé la *framegrabber* pour afficher la vidéo. Comme vu précédemment, la carte SMT319 (ou *framegrabber*) possède une sortie vidéo PAL permettant ainsi d'afficher directement une séquence vidéo sur un écran possédant une entrée PAL. Il est alors intéressant d'utiliser cette fonctionnalité pour afficher directement une séquence vidéo décodée sur un écran, via la sortie vidéo de la *framegrabber*. Le but est alors de décoder une séquence vidéo sur un DSP quelconque et d'afficher la séquence via la *framegrabber*, voire de décoder et d'afficher la séquence avec une seule et même *framegrabber*. Nous avons donc décrit la *framegrabber* sous SynDEx (Fig. 6.14) ainsi que la primitive d'affichage associée. Pour respecter le temps réel, il faut respecter une contrainte pour l'affichage vidéo entrelacé, une trame toutes les 20 ms. Deux trames forment une image entrelacée : une trame paire pour les lignes paires (*top_filed)* et une trame impaire pour les lignes impaires (*bottom_field*).

Sur le graphe de la figure 6.15, SynDEx peut générer 2 ordonnancements avec cette description comme le montre le graphe temporel de la figure 6.16. Pour l'affichage d'une vidéo entrelacée, il est nécessaire d'afficher la trame paire en premier. Pour pouvoir réaliser l'affichage dans cet ordre, une précédence est ajoutée entre les opérations *top_field* et *bottom_field*, pour ordonnancer ces opérations. Ces deux opérations sont réalisées par le DMA pour respecter le temps-réel.

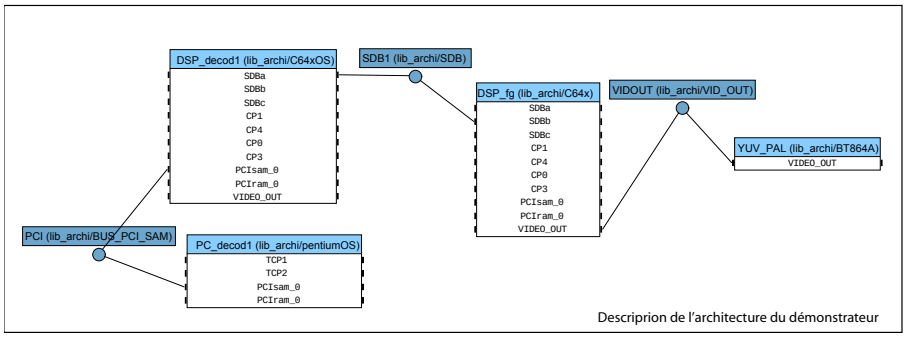


FIG. 6.14 – Architecture de démonstration du décodeur MPEG-4 avec le module *framegrabber*

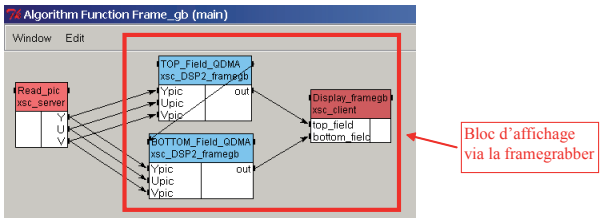


FIG. 6.15 – Graphe d'algorithme de la fonction d'affichage sur le module SMT319 *framegrabber*

Bon ordonnancement : avec celui-ci les données sont bien parallélisées, les trames sont calculées et envoyées en moins de 20 *ms* (50 trames/s) car le temps passé dans la fonction *Field_QDMA() (top_field, bottom_field)* est inférieur à 10 *ms* et la trame suivante se calcule pendant l'affichage de la trame courante. En conséquence, la séquence est affichée en temps-réel, sans trame noire car toutes les trames arrivent à temps (toutes les 20 *ms*, la trame suivante a bien été calculée).

Mauvais ordonnancement : dans ce cas les trames ne sont pas toutes calculées et envoyées en moins de 20 *ms*. En effet la trame suivante n'est pas calculée pendant l'affichage de la trame courante. Une trame attendue qui n'arrive pas est alors remplacée par une trame noire qui dure au minimum 20 *ms* et la séquence n'est pas affichée en temps-réel.

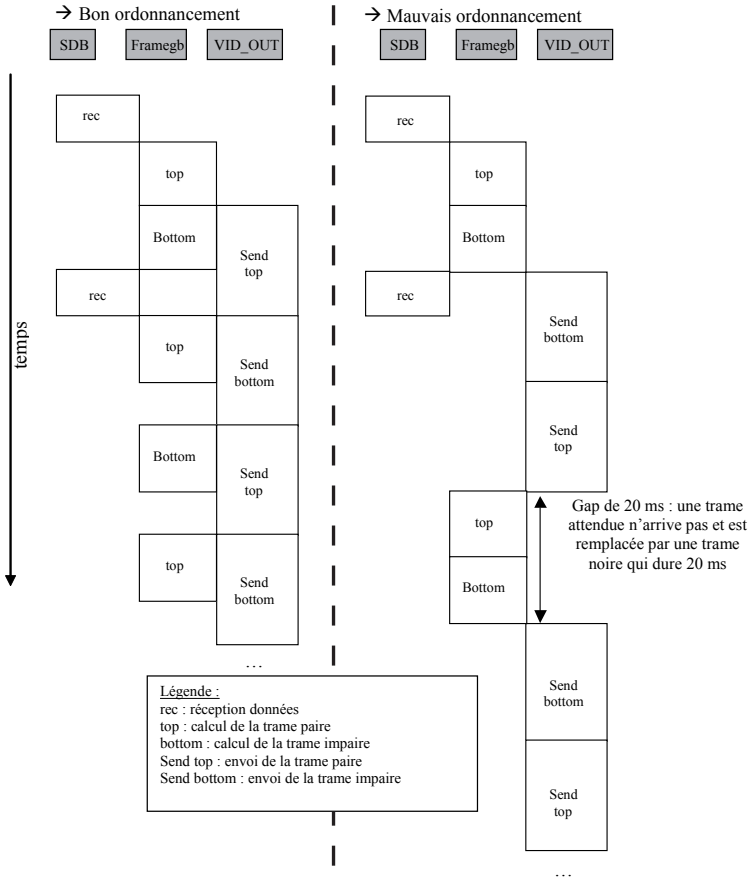


FIG. 6.16 – Ordonnancement des différentes opérations sur le DSP du module SMT319

6.4 Conclusion

Au sein du laboratoire de traitement d'images, les efforts de développement ont été fédérés pour faire plus rapidement évoluer les applications. Les développements s'effectuent directement sous SynDEx par tous les développeurs de cette équipe en utilisant la chaîne de prototypage développée (chapitre 3). Les applications traitées sont principalement flot de données nécessitant très peu de contrôle comme pour les applications de télécommunications. Il est intéressant sur l'application MPEG4 de noter que les optimisations *"registre", "tétris" ou sur la génération de code* réalisées donnent des résultats très satisfaisants. Par ailleurs, la complexité de cette application et le nombre d'opérations mises en jeu rendent une optimisation manuelle impossible.

Le décodeur MPEG4 développé va être exploité par la société INNES pour être intégré dans leurs produits dans un processeur de *TEXAS INSTRUMENTS DM642* (de la famille des *C6x*). La chaîne de prototypage intéresse également un constructeur de carte multi-DSP *(VITEC MULTIMEDIA)* avec des domaines d'activité concernant non seulement les plates-formes multi-DSP, mais aussi le portage d'applications vidéo comme MPEG4-AVC.

Par ailleurs le groupe image du laboratoire IETR est partenaire dans le projet *"Mobim@ge"* (premier projet du pôle "Images et réseaux"). Dans ce projet, forts de nos compétences acquises dans le domaine embarqué et les schémas de codage, nous devons développer un décodeur MPEG4-AVC (MPEG4 partie 10) embarqué dans un terminal mobile. Le schéma de codage de cette norme se révèle bien plus complexe algorithmiquement. Par contre cette norme est principalement dédiée au décodage vidéo d'images rectangulaires, ce qui s'avère bien plus simple que la partie 2 de MPEG4. Nous allons certainement intégrer un nouveau projet *"scalim@ge"*. Celui-ci vient d'être labellisé par le pôle et est en attente d'un financement par l'état. Ce projet concerne MPEG4-SVC pour de la vidéo scalable, anciennement MPEG21. Dans ce projet, nous contribuerions plus particulièrement à :

- la réalisation d'un décodeur MPEG4-SVC embarqué dans un terminal mobile,
- la réalisation d'un codeur temps-réel sur une architecture multi-composants.

Chapitre 7

Applications multi-couches

7.1 Introduction

Les principales difficultés soulevées dans le développement de nouvelles familles de téléphones portables sont liées à l'hétérogénéité à la fois des applications et des cibles matérielles. L'ensemble des traitements à effectuer sur un téléphone mobile peut ainsi être décomposé en différentes couches fonctionnelles correspondant soit à des couches de service multimédia de haut niveau (codage de la voix, des images ou des vidéo, accès internet...), soit à des couches de transport (communication numérique) de plus bas niveau. La difficulté de développement est encore augmentée par la mise en réseau de tels systèmes, imposant de valider l'application en tenant compte de son caractère distribué. Ainsi, nous qualifierons de systèmes multi-couches communicants, ces nouvelles générations de téléphones mobiles.

Des applications multi-couches peuvent à terme être mieux prises en compte pour répondre à une plus large demande industrielle. Pour les applications, il existe une vue verticale et une vue horizontale pour décrire leur comportement.

La vue verticale permet de rendre compte des différentes couches d'abstraction dans le système, autorisant les transformations de l'information depuis les hautes couches de service (codage de la voix, de l'image) jusqu'à l'émission par canal hertzien, en passant par la couche de communication numérique (GSM, UMTS, MC-CDMA). Cette caractéristique se retrouve dans tous les systèmes reliés par des réseaux, ce qui a conduit par exemple au modèle de référence OSI (*Open Systems Interconnection*) en sept couches (tableau 7.1).

La vue horizontale quant à elle exprime principalement l'interaction entre les systèmes communicants, perçue aux différentes couches : émission/réception de données au niveau communication numérique, codage/décodage de vidéo par exemple au niveau service multimédia. Un système de prototypage rapide pour des systèmes multi-couches communicants, doit donc être capable de décrire, puis d'implanter sur une architecture, à la fois les vues horizontale et verticale. Ceci passe par des modèles de description et des méthodes de projection adaptés.

L'objectif est d'utiliser ici la méthodologie AAA associée à la génération de code automatique pour mettre en commun les travaux des différentes équipes de l'IETR et de MITSUBISHI ITE (les applications de traitement d'images pour le groupe image ainsi que les applications de télécommunications UMTS pour MITSUBISHI et MC-

Hôte A		**Hôte B**
Couche Application	⇐**Données**⇒	**Couche Application**
Couche Présentation		Couche Présentation
Couche Session		Couche Session
Couche Transport	⇐Segments⇒	Couche Transport
Couche Réseau	⇐Paquets⇒	Couche Réseau
Couche Liaisons de données	⇐Trame⇒	Couche Liaisons de données
Couche Physique	⇐Bits⇒	Couche Physique

TAB. 7.1 – Modèle OSI multi-couches (*Open Systems Interconnection)*

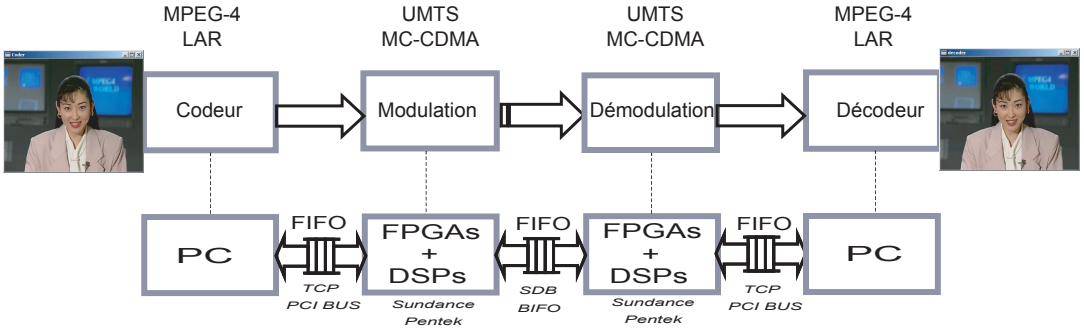


FIG. 7.1 – Démonstrateur de transmission de flux vidéo sur système de télécommunications numériques

CDMA pour le groupe CPR) afin de réaliser des démonstrateurs de transmission de flux vidéo sur systèmes de télécommunications numériques. Ces démonstrateurs ont été réalisés sur plusieurs plates-formes de prototypage multi-composants. Il s'agissait de réaliser une chaîne multi-couches complète (Fig. 7.1) de codage vidéo (compression du flux vidéo grâce au codeur LAR ou MPEG-4), émission/réception radio (simulation d'une chaîne de télécommunication 3G ou 4G), puis décodage vidéo.

7.2 Contexte

Cette démonstration a vu le jour dans le cadre du projet PALMYRE. Quatre laboratoires CNRS bretons (MO-LEST et TAMCIC à Brest, IETR à Rennes et LESTER à Lorient) ont uni leurs forces pour développer, en commun, un banc de test d'applications innovantes simulant toute la chaîne logique d'un système de communication. Il s'agit de la plate-forme PALMYRE. Un des objectifs du projet régional PALMYRE est de proposer une plate-forme matérielle à des partenaires industriels. Il est alors nécessaire de présenter à ces partenaires une méthode simple et efficace pour porter leurs applications sur la plate-forme matérielle à disposition.

L'utilisation de la méthodologie développée à l'IETR groupe Image a été étudiée au sein du projet PALMYRE, principalement par le groupe CPR de l'IETR et par le LESTER de Lorient.

Dans le cadre de nos démonstrateurs, nous avons implanté différentes applications (LAR, MPEG4, MC-CDMA, UMTS) appartenant à différents domaines (vidéo, télécommunications) sur différentes architectures (Pentek, Sundance). Plusieurs configurations ont été testées :

- LAR sur UMTS,
- LAR sur MC-CDMA,
- MPEG4 sur UMTS,
- MPEG4 sur MC-CDMA.

Le codeur MPEG-4 est réalisé à l'aide du logiciel libre *XviD*. Ce codeur est ici décrit sous SynDEx par une opération unique, ou atomique. Le codeur *XviD est* encapsulé dans une opération décrite sous SynDEx. Le codec LAR et le décodeur MPEG4 sont ceux décrits dans le chapitre 6. Les applications de télécommunications sont celles décrites dans le chapitre 5.

Le travail sur le portage des applications multi-couches a été principalement étudié par F.URBAN lors de son stage de fin d'études sous mon encadrement. Par la suite, les descriptions SynDEx représentées sur les figures montrent principalement l'application LAR et MC-CDMA, mais pourraient être remplacées par MPEG4 ou l'UMTS.

7.3 Limites de SynDEx

Pour assembler les applications vidéo et de télécommunication, la première idée a consisté à faire une seule description SynDEx de toute l'application. Cependant, lors de la description du graphe flot de données de l'algorithme, une incompatibilité entre les données entrantes et sortantes des deux applications apparaît. Le premier problème est lié au fait que les données sortant du codeur vidéo sont de taille variable, or SynDEx n'admet que des déclarations a priori statiques des buffers. Pour contourner ce problème, nous avons dans un premier temps considéré une taille fixe (la plus grande taille possible d'une image codée), puis le code a été optimisé à la main. Le second problème réside dans la différence entre les entrées et sorties de ces 2 couches. La couche vidéo produit de gros buffers de l'ordre de 20 Ko (pire cas observé) pour des images CIF, alors que la couche de communication traite des paquets de données de 184 *octets* dans le cas du MC-CDMA (736 porteuses utiles). De plus la taille pour la couche vidéo du buffer produit n'est en général pas un multiple de 184 *octets* (MC-CDMA).

La difficulté rencontrée est due à la différence des niveaux de description entre les deux couches. Au niveau de la couche vidéo, la couche de télécommunications devrait pouvoir apparaître comme un composant de l'architecture reliant le codeur et le décodeur vidéo. Ainsi la couche de télécommunication ne devrait pas être directement décrite dans le graphe d'algorithme, mais plutôt être associé au graphe d'architecture.

7.3.1 Première solution : modification de SynDEx

Une proposition de solution apparaît figure 7.2. Elle consisterait en la modification du logiciel SynDEx pour ajouter la possibilité d'une description du graphe d'architecture de façon hiérarchique, de la même façon qu'un algorithme. Il serait alors possible de décrire le fonctionnement du médium de communication (ici MC-CDMA) avec son graphe d'algorithme et son architecture.

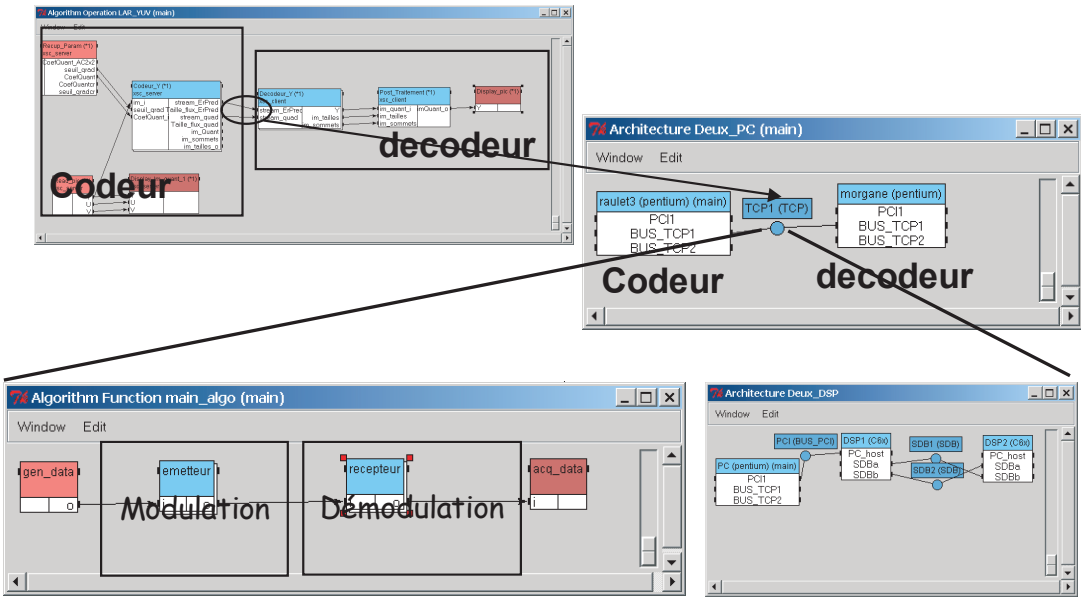


FIG. 7.2 – Description SynDEx de la chaîne compète : description hiérarchique de l'architecture

Cette solution est aujourd'hui à l'étude dans le cadre de la thèse de G.ROQUIER. Par contre lors de cette étude, il a fallu trouver une solution réalisable à court terme.

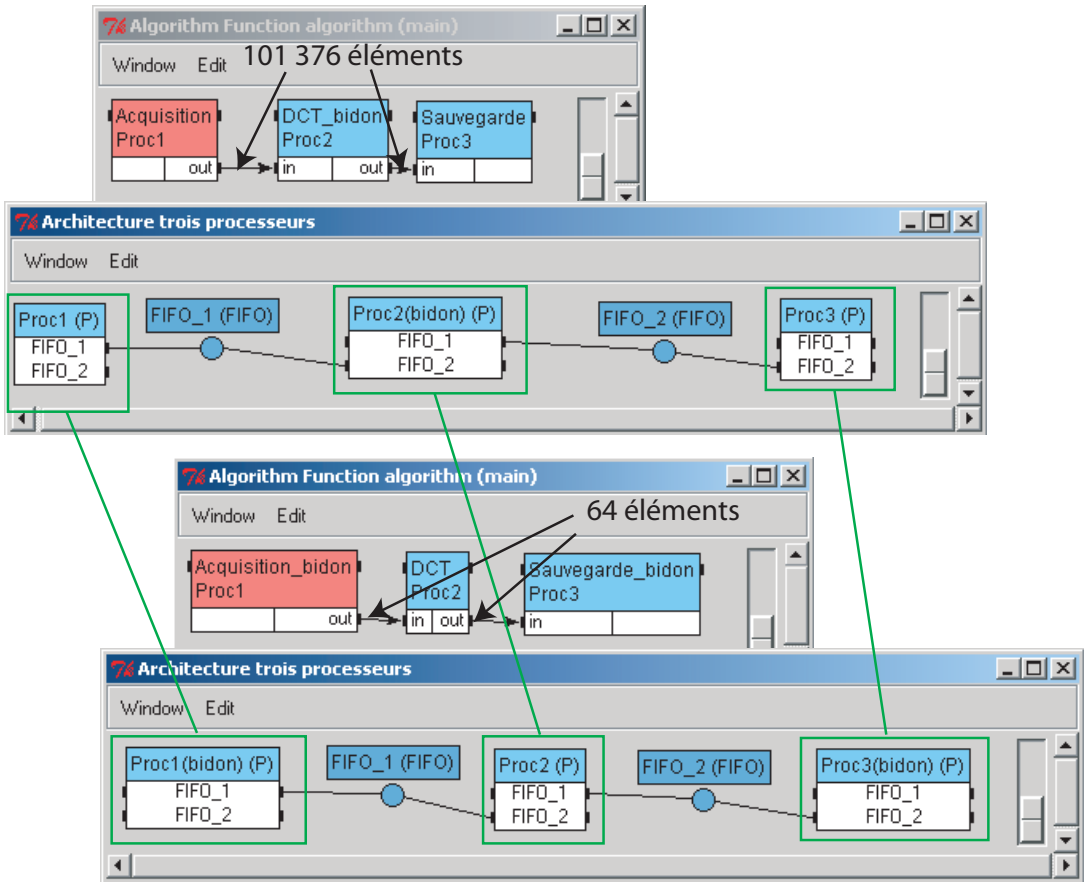
7.3.2 Deuxième solution : utilisation d'une FIFO

7.3.2.1 Principe

Cette solution exploite le fait que les synchronisations inter-processeurs ne sont pas gérées par SynDEx mais directement par le matériel. Ces synchronisations sont des signaux du type *full flag* et *empty flag*. Ainsi, ces médias autorisent le fait que le nombre de données envoyées soit différent du nombre de données reçues. Par exemple, considérons qu'une itération du producteur fournit $p \times n$ données alors que le consommateur lit n données. Il faut alors p itérations du consommateur pour vider la FIFO, et libérer le producteur pour commencer une nouvelle itération.

Le problème d'interfaçage posé par l'utilisation de SynDEx est contourné en séparant les graphes des deux applications, et en les exécutant sur des processeurs différents. Lors de la description d'une application sous SynDEx, seul l'algorithme correspondant est décrit, alors que toute l'architecture est décrite afin de générer le code des transferts de données entre les processeurs.

Un exemple simple est décrit figure 7.3 : acquisition d'une image sur un processeur, réalisation d'une DCT 8x8 (*discrete cosine transform*) sur un deuxième processeur, sauvegarde de l'image traitée sur un troisième processeur. Les processeurs 1 et 3 travaillent sur une image complète : $352 \times 288 = 101376$ valeurs (*send* et *receive* de 101376 données), alors que le processeur 2 traite un bloc à la fois : $8 \times 8 = 64$ données. La figure 7.3 présente les deux descriptions.



– en haut : description pour l'acquisition et la sauvegarde, code généré uniquement pour les processeurs 1 et 2, la taille des données transmises est de 101376,
– en bas : description la DCT, code généré uniquement pour le processeur 2, la taille des données ici transmises est de 64.

FIG. 7.3 – Exemple d'utilisation des FIFO

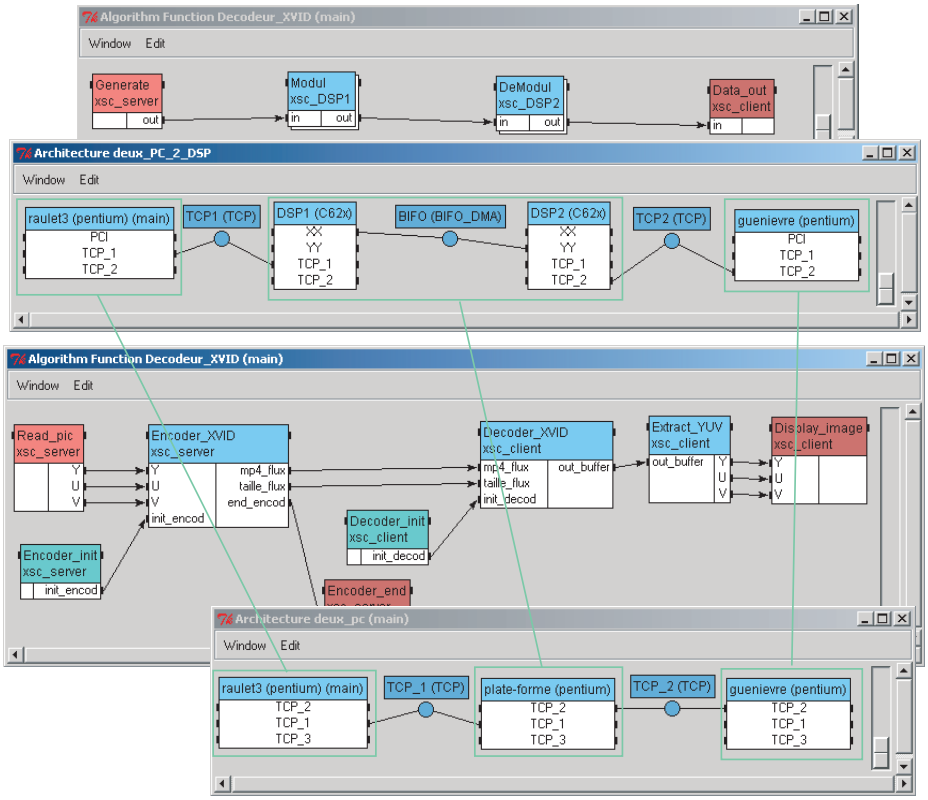
7.3.2.2 Implantation sur la plate-forme Pentek

La plate-forme Pentek de MITSUBISHI ITE comporte une liaison TCP fonctionnant comme une FIFO. Cette solution est présentée figure 7.4.

La description sur deux graphes séparés permet de s'affranchir du problème de la différence des flux en entrées/sorties des couches. Pour la chaîne de communication, la taille des données fait 184 *octets* dans le cas du MC-CDMA, et 24 *Ko* pour le codeur / décodeur. Il est nécessaire de respecter certaines contraintes :

– les données transmises d'un processeur d'une description à celui d'une autre description doivent passer par des FIFO,
– un producteur de données ne doit pas être un consommateur afin d'éviter des *dead-locks*.

Le bon fonctionnement de cette solution est dû au mécanisme des FIFO. Plus précisément, les média en question ici sont des média TCP. Lors d'un envoi TCP, les données



– en haut : description de la chaîne de communication ici MC-CDMA ; code généré uniquement pour les deux DSP, l'ajout des PC sert à générer les envois / réceptions sur les DSP
– en bas : description du codeur / décodeur vidéo ; code généré uniquement pour les PC.

FIG. 7.4 – Description de la chaîne complète : 2 descriptions séparées

sont envoyées dans une file, et lors de la réception, elles sont lues dans la file dans l'ordre où elles ont été envoyées. C'est donc bien un mécanisme de FIFO.

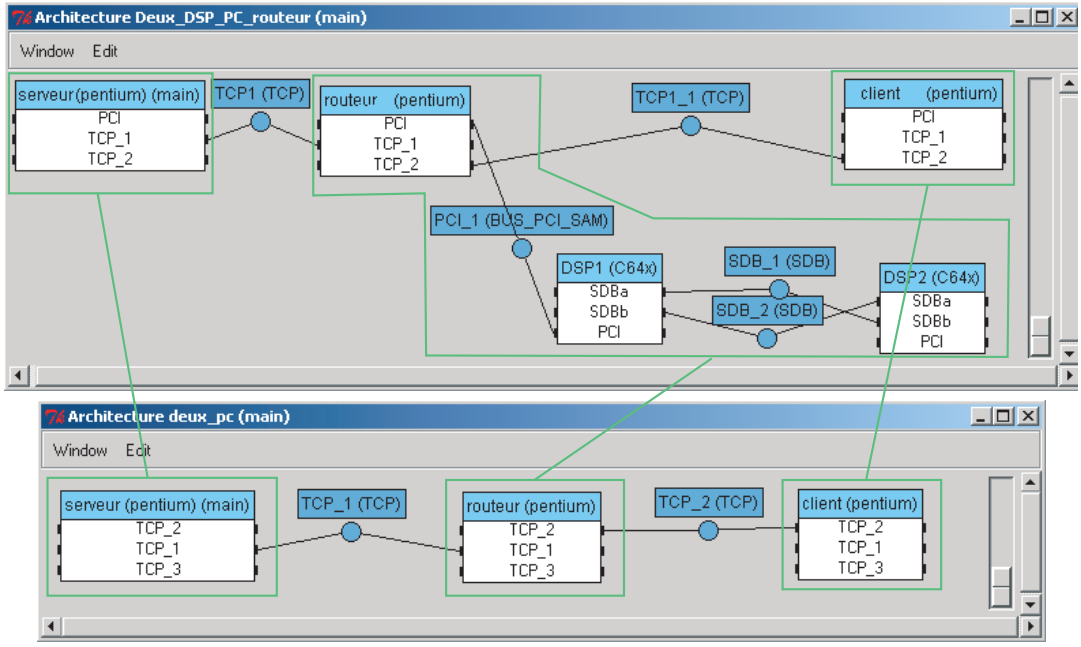
Dans notre application, il existe quatre processeurs différents :

– sur le serveur, une image est codée, poussée dans la file TCP, puis une autre image est codée et ainsi de suite jusqu'à ce que la file soit pleine. Lorsqu'elle se vide le codage reprend,
– le DSP modulateur reçoit ces données par paquets de 184 octets, les module puis les transmet au deuxième DSP tant qu'il reste des données à moduler,
– le deuxième DSP démodule ces données puis les pousse dans la file TCP qui le relie au PC client, par paquets de 184 *octets* dans le cas MC-CDMA,
– sur le client, lorsque la file TCP contient des données, celles-ci sont reçues. Quand la taille des données reçues atteint celle d'une image codée (24 *ko*), cette image est décodée.

Cette solution fonctionne avec la plate-forme Pentek qui comporte une liaison TCP. Cette liaison implémente une file dont le mécanisme est exploité.

7.3.2.3 Implantation sur la plate-forme Sundance

Sur la plate-forme Sundance, la liaison entre la carte de développement et le PC se fait par liaison PCI avec le premier DSP de la carte. Ce médium ne se présentant pas sous la forme d'une FIFO. Un PC est alors ajouté à l'architecture, relié aux autres PC par des médias de type TCP identiques à ceux utilisés précédemment. L'architecture utilisée est décrite sur les deux graphes de la figure 7.5.



- en haut : description de la chaîne de communication (ici MC-CDMA) ; code généré uniquement pour les deux DSP et le PC routeur,
- en bas : description du codeur / décodeur vidéo ; code généré uniquement pour les PC serveur et client.

FIG. 7.5 – Architecture de la chaîne complète avec la plate-forme Sundance

Le PC ajouté à l'architecture est celui contenant la carte de prototypage, il sert de routeur et il sépare les données en paquets de 184 *octets* pour le MC-CDMA. Le fonctionnement du serveur et du client est le même que précédemment, sauf qu'ils communiquent avec le PC routeur. Celui-ci traite les données par paquets de 184 *octets*, les envoie par bus PCI sur la carte où elles sont modulées sur un premier DSP, puis démodulées par un autre DSP. Puis le PC routeur reçoit à nouveau les données par l'intermédiaire du premier DSP et les renvoie enfin au client.

Ces deux dernières solutions montrent bien l'intérêt du prototypage rapide avec SynDEx et de la génération automatique de code. On passe de l'exécution de l'application d'une plate-forme à l'autre en quelques clics, sans avoir besoin d'écrire la moindre ligne de code. Un point important à noter est l'exécution obtenue en temps-réel (cadence vidéo) sur les différentes plates-formes avec les solutions choisies.

7.4 Conclusion

Dans le cadre du projet régional PALMYRE, nous avons donc mené à bien une étude sur une application complète où des vidéos (MPEG-4 et LAR) étaient véhiculées sur des systèmes de communication (UMTS et MC-CDMA). Ces travaux ont montré une limitation de notre méthodologie de prototypage actuelle, limitation partagée par l'ensemble des approches de l'état de l'art. L'explication réside principalement dans le fait que les différentes couches du système ne travaillent pas sur les mêmes types de données (images à raison de 25 images/sec pour la vidéo, trames de quelques centaines d'octets pour les systèmes de communication). Ainsi pouvons-nous résumer en disant que notre méthodologie est essentiellement opérationnelle à ce jour pour des systèmes à description horizontale.

Les travaux sur le prototypage rapide porteront donc par la suite sur l'élaboration à la fois de modèles de description et de méthodes de projection sur des cibles distribuées, applicables suivant les deux axes de représentation orthogonaux. Avant le passage à l'exécution sur cible proprement dite, il sera également envisagé une étape intermédiaire de modélisation fonctionnelle de l'application distribuée.

Le procédé de prototypage doit être automatique, c'est-à-dire qu'il génèrera seul le code nécessaire pour l'application. La projection sur une cible multi-composants requiert donc un certain nombre d'éléments : la distribution des traitements, l'allocation des ressources sur chaque unité de traitement, ainsi qu'une gestion des synchronisations et des communications de données garantissant une exécution correcte de l'application. La considération des systèmes multi-couches communicants implique en plus de pouvoir traiter la distribution de l'application selon les deux vues verticale et horizontale.

Epilogue

Conclusions et perspectives

Dans une première partie, après une présentation générale du contexte dans le chapitre 1, nous avons présenté dans le chapitre 2 les principaux concepts de la méthodologie AAA/SynDEx sur lesquels se basent nos travaux. La méthodologie AAA/SynDEx de niveau système est principalement orientée flot de données avec un peu de contrôle. L'utilisation de cette méthodologie réduit le cycle de développement et apporte un gain de temps important dans le développement d'applications multi-processeurs. L'ordre d'exécution garanti (pas d'inter-blocages), l'automatisation du processus et son indépendance vis-à-vis de l'architecture cible apportent une sécurité dans la conception de ces applications. En termes de performances, l'heuristique d'adéquation d'AAA/SynDEx conduit à un processus automatique et optimisé pour la distribution/ordonnancement et réduit de plus l'espace mémoire programme en évitant d'utiliser des mécanismes résidents. Cette étude de AAA/SynDEx a permis de détailler les différentes phases du processus, depuis la phase de description de l'algorithme et de l'architecture, jusqu'à celle de la génération de code.

Le chapitre 2 s'est attaché à montrer comment notre chaîne de prototypage a été constituée autour de la méthodologie AAA/SynDEx pour pouvoir embarquer des applications de traitement d'images et de télécommunications de façon automatique. De nombreux noyaux ont été développés et optimisés pour différents types de processeurs (pentium, C62x, C64x, FPGA) et média (SDB, CP, bus PCI suivant le modèle RAM ou SAM, BIFO, TCP). De plus, a été mise en place la gestion des divers projets par des logiciels adéquats pour la collaboration des développements au sein d'un groupe de travail. Différentes équipes grâce à cette chaîne peuvent maintenant collaborer de manière efficace et minimiser le gouffre qu'il peut y avoir entre les concepteurs d'un algorithme et les intégrateurs sur une plate-forme multi-composants.

Cette chaîne de prototypage a servi au développement de différentes applications dans le domaine du traitement des images et des télécommunications (MPEG4, LAR, UMTS, MC-CDMA) présentées dans la seconde partie de ce mémoire.

En tant qu'utilisateurs, nous avons pu constater un défaut majeur lors de la génération de code : la taille mémoire de données allouée. Une collaboration directe s'est établie avec l'INRIA dans l'outil SynDEx afin d'y intégrer des optimisations sur un critère de mémoire minimal. Ceci a fait l'objet du chapitre 4. SynDEx, après la mise à plat du graphe flot de données, alloue un buffer sur chaque sortie de chaque opération atomique. Dans un contexte temps réel embarqué, il s'avère nécessaire de trouver une méthode adéquate de minimisation de la mémoire allouée : l'idée est de s'appuyer sur des principes d'allocation minimale des buffers et de réutilisation de ces mêmes buffers. Des solutions ont été trouvées grâce à la théorie des graphes, et notamment

à la technique du coloriage de graphe, pour déterminer le nombre minimal de buffers utiles.

La minimisation mémoire se réalise donc en différentes étapes. La première étape consiste à déterminer une quantité minimale de mémoire locale sans prendre en compte les buffers transférés entre processeurs (minimisation mémoire mono-processeur). Nous avons pour cela développé deux algorithmes de minimisation mémoire différents reposant sur la technique du coloriage de graphe. Ces optimisations sont basées sur la durée de vie des buffers (intervalles) à partir de laquelle on peut construire un graphe d'intervalles. Le premier algorithme, appelé *registre*, utilise des buffers à la fois de même taille et de même type ; le second, appelé *tétris*, exploite des buffers de même type mais pas nécessairement de même taille. La seconde étape de la minimisation complète la première, en intégrant les buffers communiqués. Pour cela, le placement des sémaphores est modifié lors de la génération de code tout en gardant les spécificités du graphe temporel fourni par SynDEx (minimisation multi-processeurs). Ceci modifie le graphe d'intervalles et permet alors de minimiser les buffers communiqués inter-processeurs. Les minimisations mémoire (mono-processeur et multi-processeurs) opérées par ces deux premières étapes sont qualifiées de "spatiales". Par ailleurs, la minimisation de la génération de code et des sommets frontières apporte un gain spatial et temporel (minimisation des recopies).

Il a été de plus montré qu'une minimisation pendant la phase d'adéquation pourrait être effectuée grâce aux algorithmes génétiques. Ceux-ci facilitent la minimisation d'une fonction de coût multi-critères. Nous n'avons pas eu le temps de mettre en œuvre cette méthode mais nous avons indiqué les directions qui seront suivies dans nos prochaines recherches.

Le prototypage de plusieurs applications sur différentes plates-formes (Sundance, Pentek) est également présenté dans la seconde partie de ce mémoire. Le chapitre 5 présente les applications de télécommunications (FM, UMTS, MC-CDMA), le chapitre 6 les applications de traitement d'images (LAR, MPEG4). Tout l'intérêt de notre chaîne de prototypage rapide et automatique est démontré sur des applications complexes : quelques milliers d'opérations pour MPEG4. Dans un premier temps, des optimisations manuelles ont été réalisées sur l'UMTS afin de définir les principes des algorithmes de minimisation mémoire. Les résultats obtenus grâce à la minimisation mémoire automatique ont atteint, voire dépassé, ceux d'une minimisation manuelle.

Nos applications de télécommunications numériques et de traitements d'images peuvent être principalement décrites de manière flot de données avec toutefois un peu de contrôle. Dans les télécommunications, la principale contrainte est le traitement de masses de données plutôt faibles mais à réaliser dans des délais courts et réguliers. Les algorithmes de minimisation ont permis d'obtenir automatiquement des applications temps-réel et ont abouti à l'obtention de bons résultats sur ce type de problème.

Notre chaîne de prototypage autour de SynDEx est une méthode adaptée au domaine de la *radio logicielle.* La radio logicielle a pour but de faire le maximum de traitements en numérique. Un second objectif peut être défini à savoir la reconfiguration. Il reste à étendre notre chaîne de prototypage à cet objectif. Des travaux sont en cours à l'IETR, notamment à Supélec groupe SCEE et à l'INSA groupe CPR.

Concernant les applications de type MPEG4, il est intéressant de noter que les optimisations *registre, tétris* ou sur la génération de code réalisées donnent des résultats

très satisfaisants. Il faut noter que la complexité de ce type d'application et le nombre d'opérations mises en jeu rendent une optimisation manuelle impossible.

Dans le chapitre 7, nous avons montré une étude menée dans le cadre du projet régional PALMYRE : une application complète multi-couches alliant codages vidéo (MPEG-4 et LAR) et systèmes de communication (UMTS et MC-CDMA).

Le décodeur MPEG4 (partie 2) développé au sein du laboratoire est devenu un logiciel libre et sera exploité par la société INNES pour être intégré dans leurs produits sur un processeur de *TEXAS INSTRUMENT DM642* (de la famille des *C6x*). La chaîne de prototypage intéresse également un constructeur de carte multi-DSP *(VITEC MULTIMEDIA)*. Ses domaines d'activité sont non seulement les plates-formes multi-DSP, mais aussi le portage d'applications vidéo comme MPEG4-AVC.

Le groupe Image du laboratoire IETR fait partie du projet *"Mobim@ge"* (premier projet du pôle "Images et réseaux") pour développer un décodeur MPEG4-AVC (MPEG4 partie 10) embarqué dans un terminal mobile. Nous allons certainement intégrer un nouveau projet *"scalim@ge"*. Dans ce projet, nous contribuerions à différentes parties, la réalisation d'un décodeur MPEG4-SVC embarqué dans un terminal mobile, et la réalisation d'un codeur temps-réel sur une architecture multi-composants.

Dans le chapitre 7, nos travaux se sont focalisés sur les systèmes multi-couches communicants. Ce type de système fait par exemple référence aux nouvelles générations de terminaux mobiles, s'appuyant sur des couches de télécommunications et des couches de traitement vidéo, et réliés à travers des réseaux sans fil. Dans le domaine de la téléphonie, le nombre de cibles et d'applications est très important, ce qui suppose de pouvoir indistinctement projeter une application donnée sur un ensemble de cibles, ou inversement un ensemble d'applications sur une cible donnée. Le problème posé par l'implantation consiste à trouver la meilleure répartition des traitements à effectuer en tenant compte de la cible.

Ces travaux ont montré une limitation de notre chaîne de prototypage actuelle. Celle-ci est essentiellement opérationnelle aujourd'hui pour des systèmes à description horizontale. Nos perspectives à moyen terme sont d'aider à l'intégration des systèmes multi-couches communicants. Ces futurs travaux s'inscrivent dans une continuité de recherches menées dans le laboratoire IETR, à la fois à l'INSA de Rennes et à Supélec, en partenariat avec d'autres centres académiques et industriels locaux.

Annexe

Annexe A

Plate-forme

A.1 Carte Mère Sundance SMT320 et SMT310Q : Architecture du bus PCI

Les cartes de développement comportant les modules DSP (Sundance SMT320 et SMT310Q) sont reliées au PC via le bus PCI. Des transferts peuvent s'effectuer entre le PC et le DSP monté sur le TIM slot 0. Ces transferts peuvent passer soit par le Comport 3 du TIM slot 0, soit par le "*Global Bus*" (Fig. A.1 et Fig. A.2). Les communications par Comport étant beaucoup moins performantes, nous nous sommes intéressés à l'utilisation du *Global Bus*.

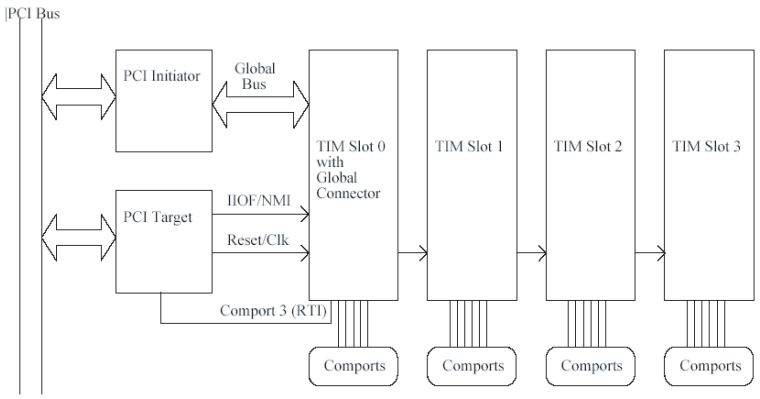


Fig. A.1 – Architecture de la carte mère Sundance SMT320

Sur le PC, les communications entre le processeur et le bus PCI se font à travers un "PCI bridge" (figure A.3). Le système d'exploitation (Microsoft Windows) ne permet pas l'écriture ou la lecture directe dans l'espace d'adressage PCI. Il est nécessaire d'utiliser un pilote de périphérique (WinDriver). Ce pilote est fourni avec des fonctions d'allocation de mémoire PCI et gère les transferts entre la RAM et le bus PCI.

De la même façon que pour le PC, le DSP (TIM Slot 0 à 3, Fig. A.1 et Fig. A.2) n'accède pas directement au bus PCI. Les transferts passent par une FIFO bidirectionnelle qui relie le *Global Bus* au bus PCI, et doivent être réalisés en deux temps :

– configuration de la FIFO (*Global Bus*),

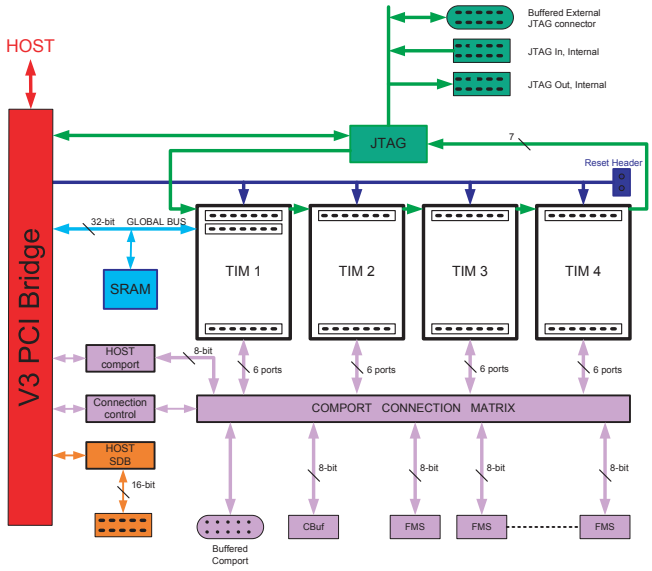


FIG. A.2 – Architecture de la carte mère SMT310Q

– transfert entre la FIFO et la mémoire du DSP.

Chaque accès (configuration, transfert) au *Global Bus* doit se terminer par une attente de fin d'utilisation du bus (signal *done*), pour que le transfert soit effectué et que le bus soit disponible lors de l'accès suivant.

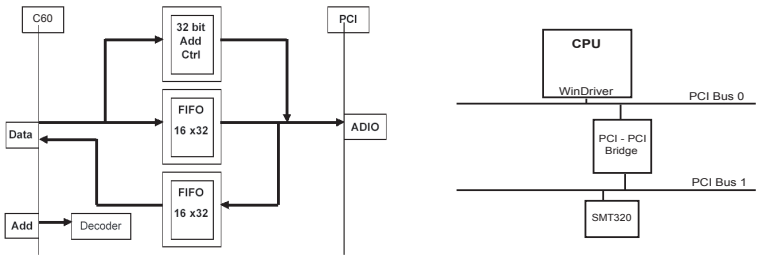


FIG. A.3 – Architecture du bus PCI de la SMT320 (gauche) et du PC (droite)

A.2 Modules Sundance

Différents modules TIM (*Texas Instruments Module*) sont présentés ci-dessous.

A.2.1 Module SMT335

Module comportant princpalement un DSP C62 (à 200 MHz), un FPGA pour gérer les ports de communication (SDB, CP, bus PCI) et de la mémoire externe (SBSRAM, SDRAM).

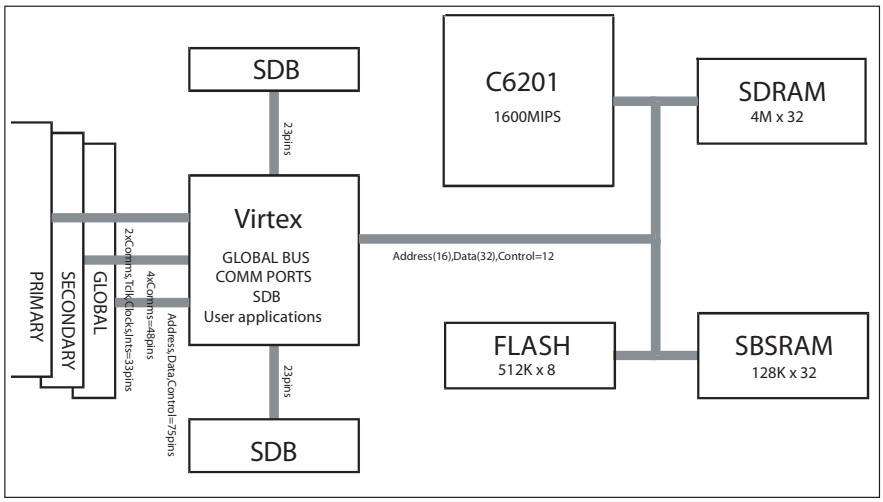


FIG. A.4 – Module Sundance SMT335

A.2.2 Module SMT361

Module comportant principalement un DSP C64 (à 400 MHz), un FPGA pour gérer les ports de communication (SDB, CP, bus PCI) et de la mémoire externe (SDRAM).

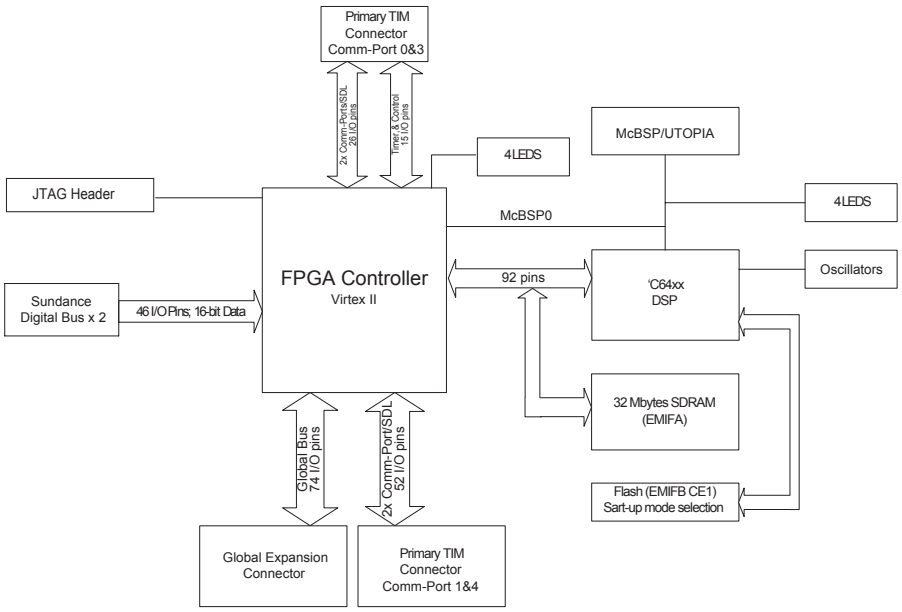


FIG. A.5 – Module Sundance SMT361

A.2.3 Module SMT358

Module comportant principalement un FPGA Virtex et de la mémoire externe (SDRAM).

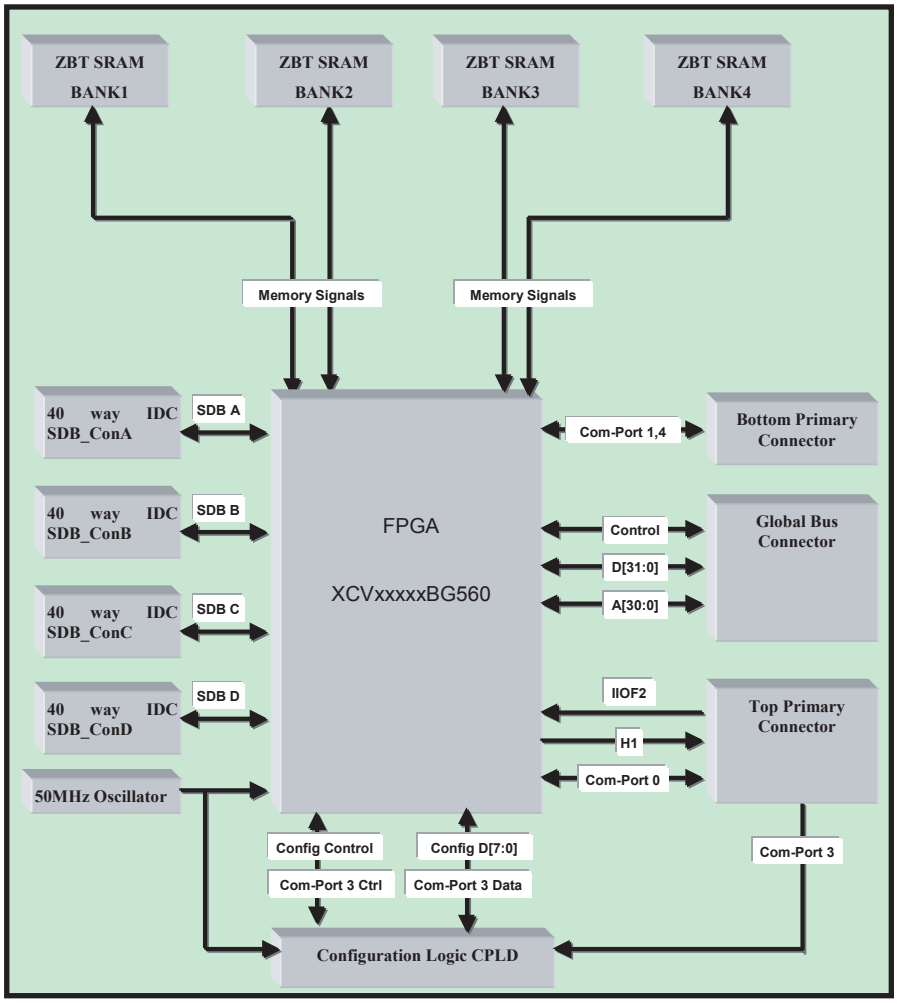


FIG. A.6 – Module Sundance SMT358

A.2.4 Module SMT319

Module comportant principalement un DSP C64 (à 600 MHz), un FPGA pour gérer les ports de communication (SDB, CP), de la mémoire externe (SDRAM) et des composants de conversion analogique/numérique pour acquérir ou restituer un signal vidéo.

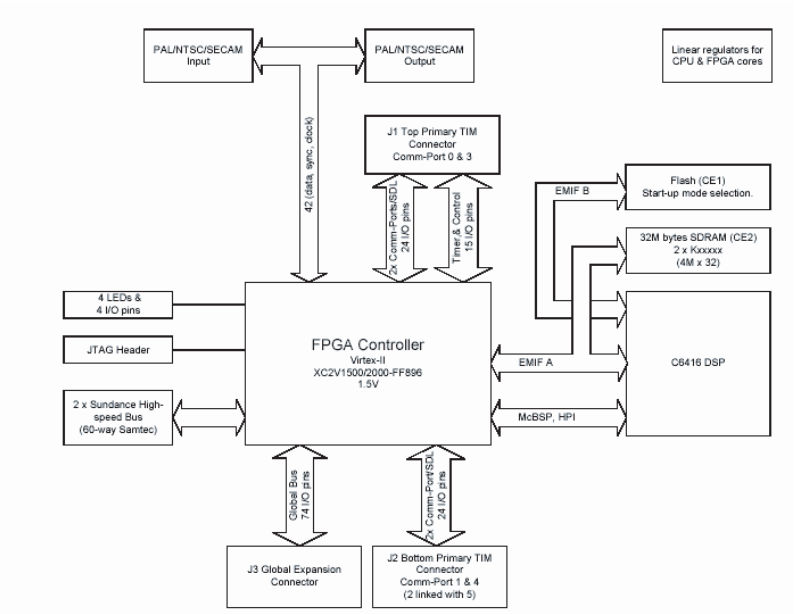


FIG. A.7 – Module Sundance SMT319

A.2.5 Module SMT395

Module comportant principalement un DSP C64 (à 1 GHz), un FPGA pour gérer les ports de communication (SDB, CP, bus PCI) et de la mémoire externe (SDRAM).

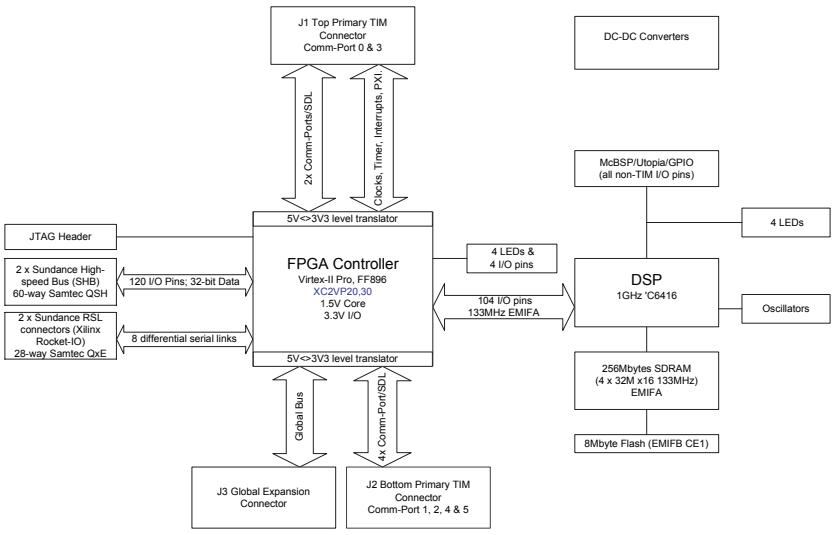


FIG. A.8 – Module Sundance SMT395

A.3 Plate-forme Pentek p4292

L'architecture Pentek p4292 est composée de 4 DSP C6203 avec une topologie en anneau. Entre les DSP connectés 2 à 2, une liaison avec une BIFIFO est réalisée. Les PC sont reliés à cette plate-forme par un lien TCP.

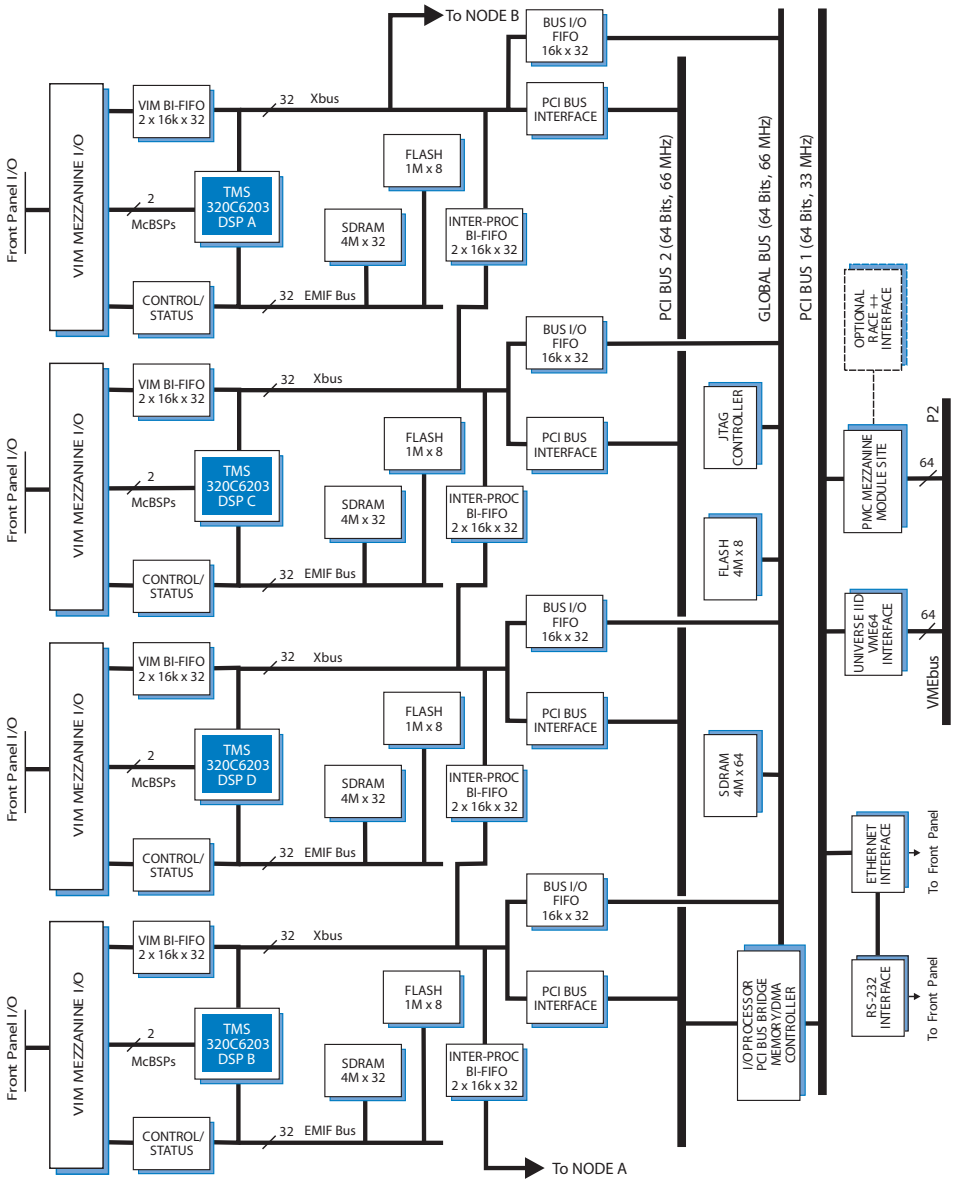


FIG. A.9 – Plate-forme Pentek p4292

A.4 Plate-forme Pentek p4290

L'architecture Pentek p4290 est composée de 4 DSP C6201 (ancienne génération) avec une topologie en anneau. Entre les DSP connectés 2 à 2, une liaison avec une BIFIFO est réalisée. Les PC sont reliés à cette plate-forme par un lien TCP.

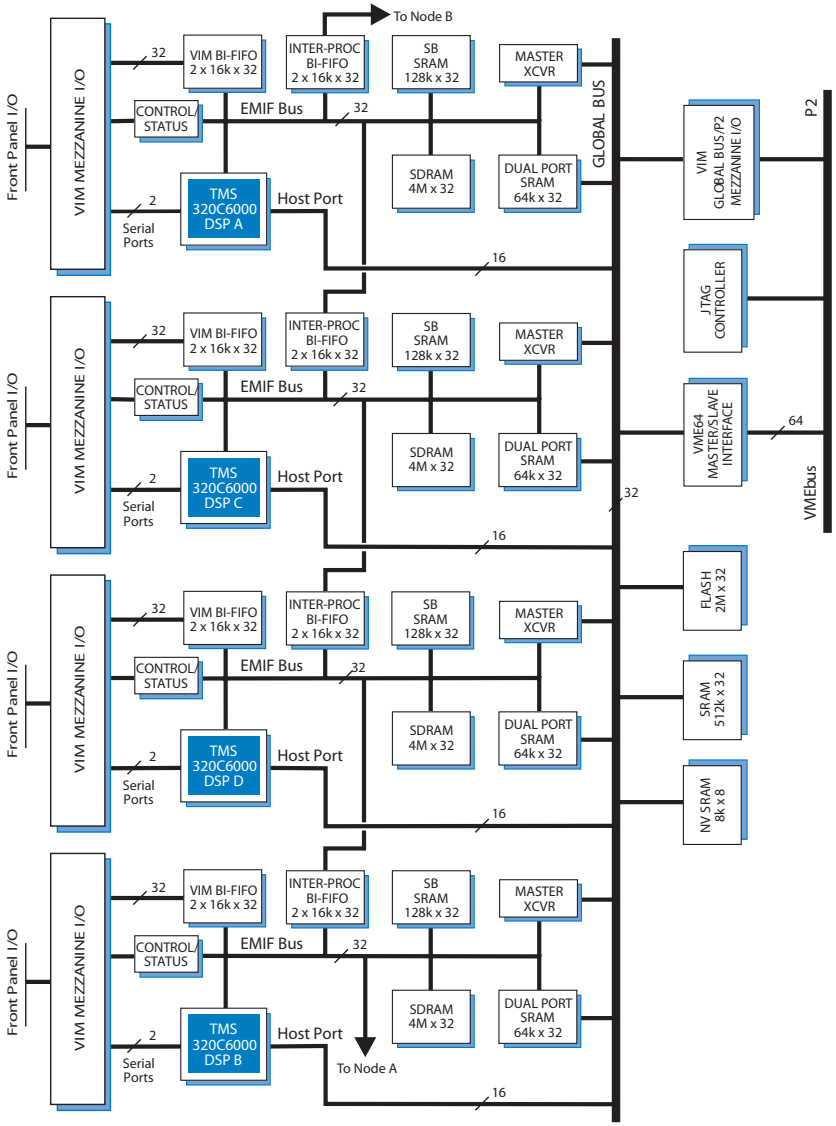


FIG. A.10 – Plate-forme Pentek p4290

Annexe B

Contrôle de Version : Subversion

Classiquement, un logiciel de gestion de versions va agir sur une arborescence de fichiers afin de conserver toutes les versions des fichiers, ainsi que les différences entre les fichiers. Ce système permet de mutualiser un développement et de corriger rapidement les erreurs de programmation. Un groupe de développeurs autour d'un même développement se sert de l'outil pour stocker toute évolution du code source. Le système gère les mises à jour des sources par chaque développeur, conserve une trace de chaque changement. Ceux-ci sont à chaque fois accompagnés d'un commentaire. Le système travaille par fusion de copies locale et distante, et non par écrasement de la version distante par la version locale. Ainsi, deux développeurs travaillant de concert sur un même source, les changements du premier à soumettre son travail ne seront pas perdus lorsque le second, qui a donc travaillé sur une version non encore modifiée par le premier, renvoie ses modifications.

Pour nos développements, notre choix s'est porté sur *Subversion* qui est un logiciel de gestion de sources et de contrôle de versions. Ce type de programme a plusieurs fonctions, notamment :

- garder un historique des différentes versions des fichiers d'un projet,
- permettre le retour à une version antérieure quelconque,
- garder un historique des modifications avec leur nature, leur date, leur auteur...
- permettre un accès souple à ces fichiers, en local ou via un réseau,
- permettre à des utilisateurs distincts et souvent distants de travailler ensemble sur les mêmes fichiers.

Il existe un grand nombre de logiciels du même type. Le plus connu d'entre eux et le plus répandu actuellement est sans doute *CVS*[(1)], mais on peut aussi citer *GNU Arch* (certainement le meilleur aujourd'hui mais qui n'est pas multi-plates-formes), *Bitkeeper*, *Git*, *Superversion*, etc...

Des comparatifs point à point peuvent être trouvés aux adresses suivantes :

- `http://zooko.com/revision_control_quick_ref.html`
- `http://better-scm.berlios.de/comparison/comparison.html`

On pourra justifier rapidement le choix de Subversion par les arguments suivants :

- il est multi-plates-formes,
- il s'agit d'un logiciel libre,
- il fonctionne de manière centralisée,

[(1)] *Concurrent Version System*

– son utilisation et son administration sont plus faciles que CVS,

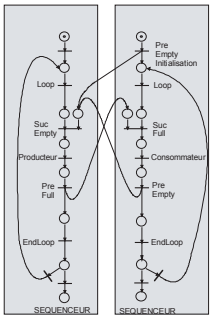
– il supporte plusieurs modes d'accès distant, dont SSH et WebDAV via Apache.

WebDAV[2] est un protocole (plus précisément, une extension au protocole HTTP) défini par le groupe de travail IETF éponyme. Normalisé récemment, WebDAV permet de simplifier la gestion de fichiers avec des serveurs distants. Il permet de récupérer, déposer, synchroniser et de publier des fichiers (et dossiers) rapidement et facilement. L'objectif principal de WebDAV est de rendre possible l'écriture à travers le web et pas seulement la lecture de données. Nos différents projets au sein du laboratoire IETR ont été publiés sur notre site web (`www.ietr-image.insa-rennes.fr`) sur ce protocole. Nos projets sont sécurisés par le protocole MD5, pour Message Digest 5, qui est une fonction de hachage cryptographique encore très populaire, mais qui n'est plus considéré comme un algorithme sûr. Dans une prochaine étape, un protocole de communication sécurisé par SSH, qui est l'abréviation de Secure Shell, sera certainement mis en place. Ce protocole de connexion impose un échange de clé de chiffrement en début de connexion. La version 2 de SSH est beaucoup plus sûre cryptographiquement, et possède en plus un protocole de transfert de fichiers complet.

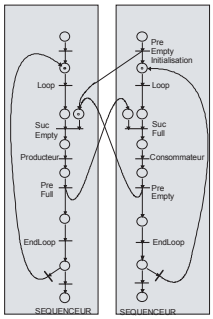
[2] (Web-based Distributed Authoring and Versioning)

Annexe C

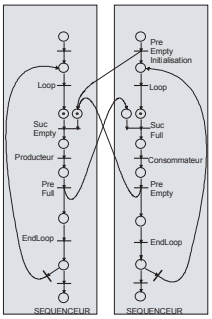
Réseau de petri d'une séquence de communication



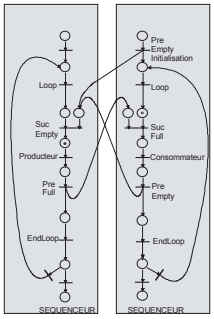
- Etat intial



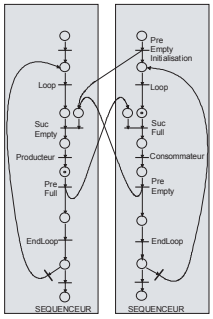
- Entrée dans la séquence itérative du producteur
- Entrée dans la séquence itérative du consommateur
- Le pre_empty d'initialisation est libéré



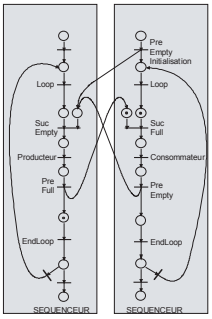
- Le consommateur est en attente de suc_full
- Le producteur va débloquer son suc_empty



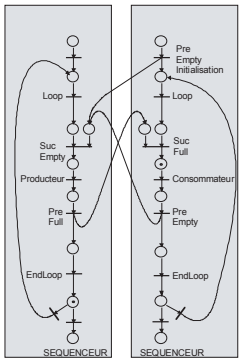
- Le suc_empty est libéré
- Le consommateur est toujours bloqué



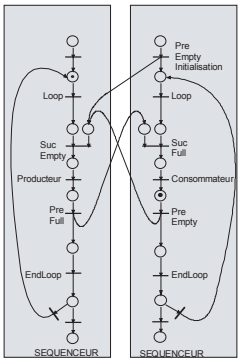
- Seul scénario possible : exécution du producteur



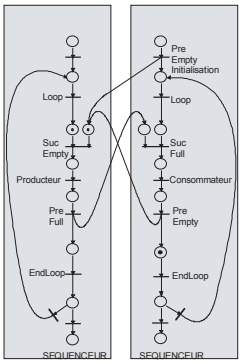
- Le consommateur va débloquer son suc_full
- Le producteur continue sa séquence



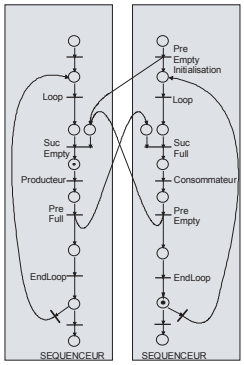
- Le consommateur libère le suc_full
- Le producteur continue



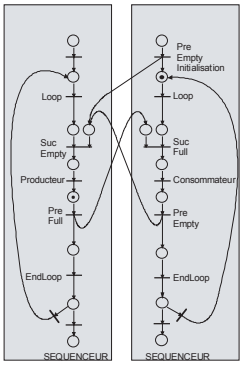
- Le consommateur va libérer le pre_empty
- Le producteur commence une nouvelle itération



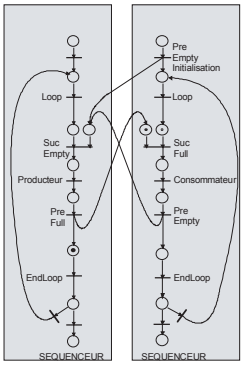
- Le suc_empty est prêt à être libéré
- Le consommateur continue sa séquence



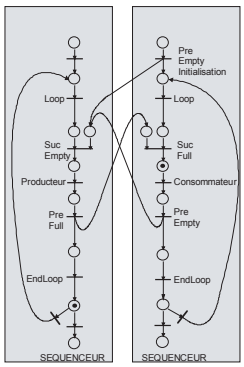
- Libération du suc_empty



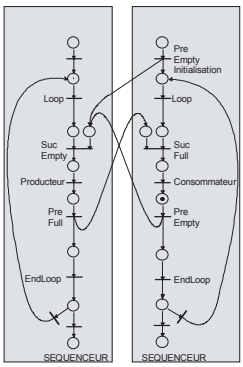
- Le producteur va débloquer son pre_full
- Le producteur démarre une nouvelle itération



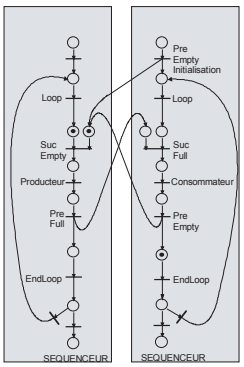
- Le consommateur est prêt à libérer son suc_full



- Le producteur libère son suc_full



- Exécution des 2 séquences



Annexe D

Applications SynDEx de référence pour les algorithmes génétiques

Les applications décrites dans cette annexe sont celles utilisées comme référence pour tester les algorithmes génétiques. La première est un produit matrice vecteur. La seconde comporte beaucoup de hiérarchie. La dernière est composée d'opérations de conditionnements hiérarchiques sur le même schéma que la seconde. L'architecture cible choisie présente un fort taux de parallélisme.

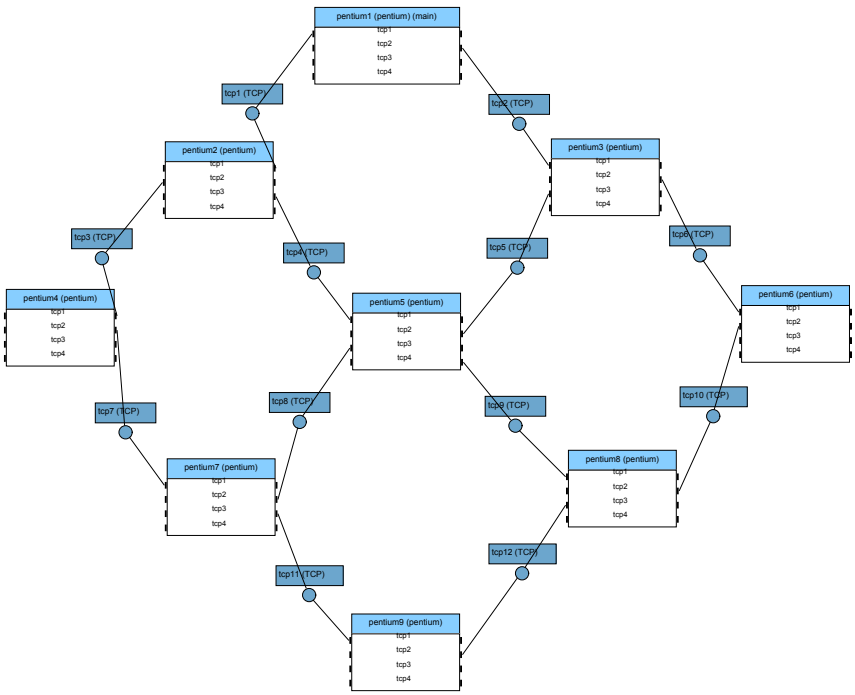


FIG. D.1 – Description de l'architecture type pour les algorithmes génétiques

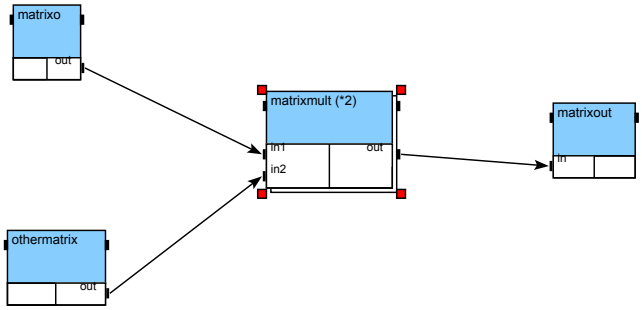


FIG. D.2 – Description SynDEx de *matrixmul*

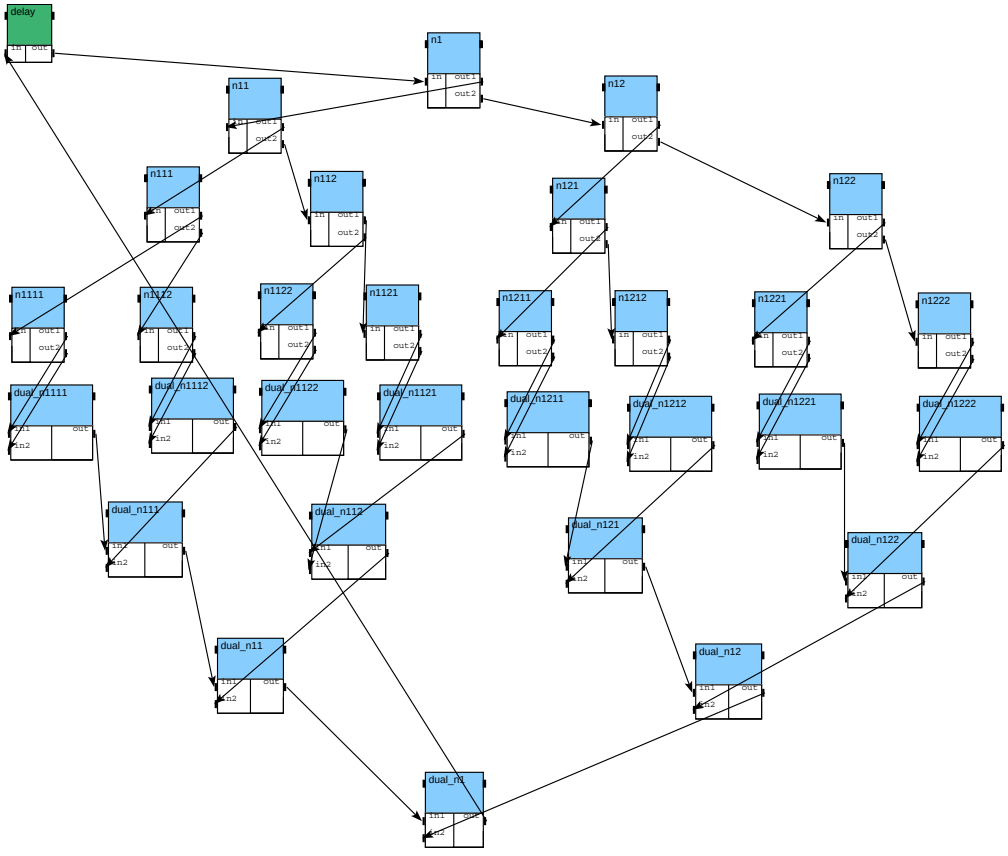


FIG. D.3 – Description SynDEx de *exp32*

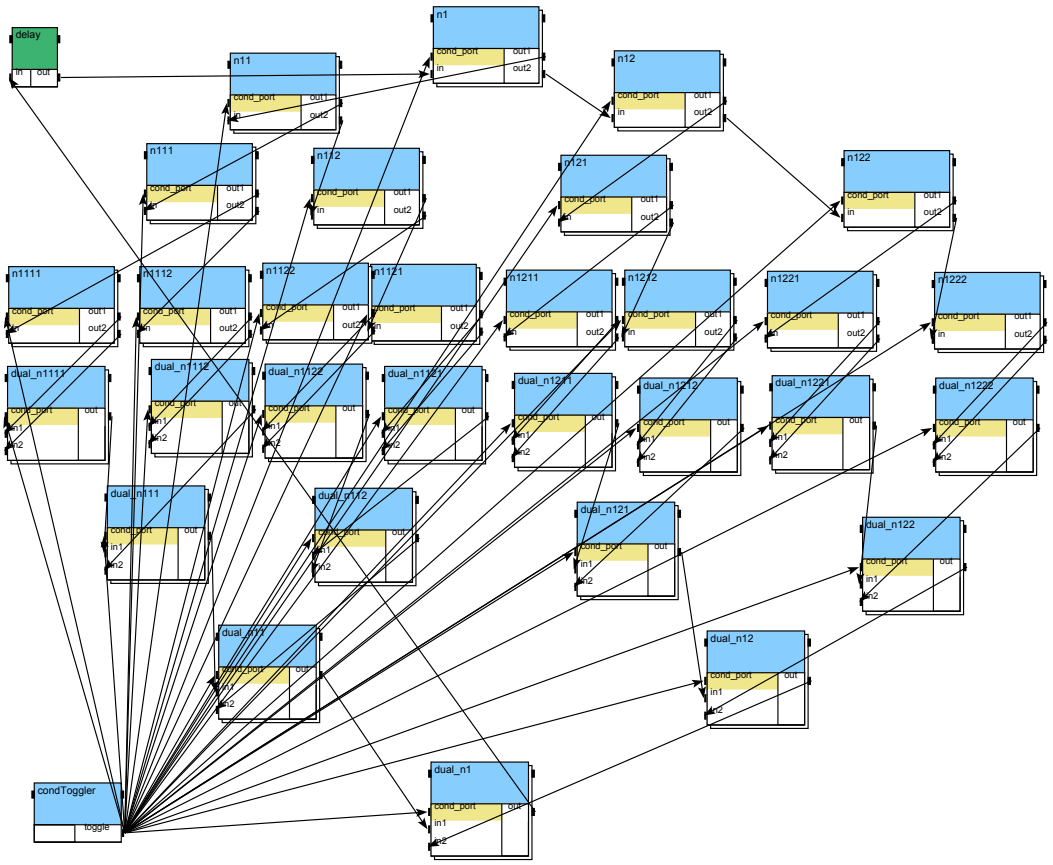


FIG. D.4 – Description SynDEx de *exp32c* avec conditionnement

Annexe E

Chaîne de télécommunication UMTS

E.1 Autres schémas disponibles pour l'émetteur

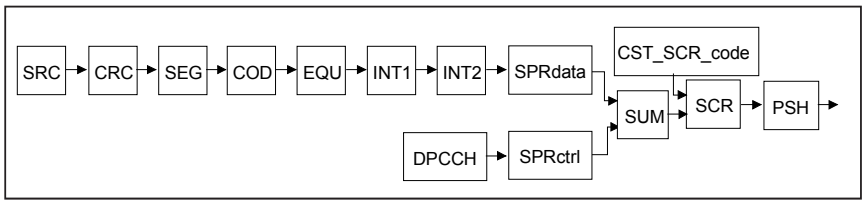


FIG. E.1 – Bloc diagramme de l'émetteur UMTS

SRC source binaire de données (issues de l'application en fonctionnement normal)

CRC calcul de CRC

SEG segmentation

COD codage canal Turbo code, Viterbi ou rien

EQU égalisation

INT1 premier entrelacement entre blocs de données

INT2 deuxième entrelacement entre blocs de données

SPRdata *spreading* étalement des données identifiant le service

DPCCH génération des informations de contrôle

SPRctrl *spreading* étalement du contrôle

SUM création d'un nombre complexe à partir des deux voies (dans le cas d'un seul canal physique) de données et de contrôle

SCR *scrambling* brouilleur par étalement identifiant la cellule

PSH *pulse shaping* filtrage d'émission pour la mise en forme le signal

E.2 Autres schémas disponibles pour le récepteur

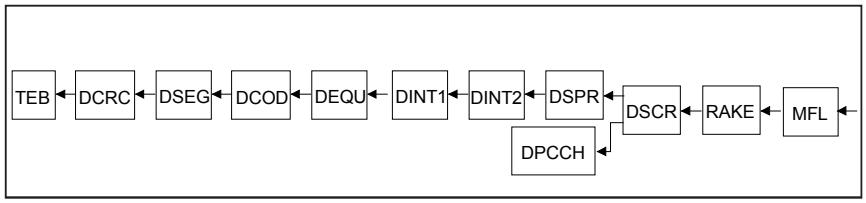


FIG. E.2 – Bloc diagramme du récepteur UMTS

MFL *matched filtering* pendant du PSH à la réception finissant de mettre en forme le signal

RAKE récepteur multi-trajets effectuant plusieurs opérations qui sont en général de multiples opérations de désétalement sert d'organe de synchronisation

DSCR *descrambling* désétalement identifiant la cellule

DPCCH récupération des informations de contrôle

DSPR *despreading* desétalement des données (cette opération et celles qui suivent sont à dupliquer autant de fois qu'il y a de canaux physiques)

DINT2 deuxième désentrelacement entre blocs de données

DINT1 premier désentrelacement entre blocs de données

DEQU opération inverse à l'egalisation d'émission (rien à voir avec un égaliseur dans les récepteurs habituels)

DCOD décodage canal Turbo décodeur, décodeur de VITERBI ou rien

DSEG désegmentation

DCRC vérification de CRC

TEB calcul du taux d'erreur binaire

Paramètre	*Description*	*117 kbps*	*950 kbps*	*36* kbps*
CST_NB_DATA_TPT_BLOC	nombre de données par transport bloc	292	**952**	**91**
CST_NB_TPT_BLOC_TTI	nombre de transport bloc par TTI	4	**10**	4
CST_ORDER_PN	ordre du code pseudo-aléatoire généré	14	14	14
CST_CRC	nombre de données du CRC	8	8	8
CST_CODING	nature du codage (rien, turbo ou Viterbi)	32767	32767	**5114**
CST_TTI	durée du TTI	10	10	10
CST_SF_D	facteur d'étalement des données d'information	32	**4**	32
CST_CODE_NUMBER_D	numéro du code d'étalement pour les infos	9	9	9
CST_BETA_D	facteur bêta de la voie données d'info	15	15	15
CST_SF_C	facteur d'étalement des données de contrôle	256	256	256
CST_CODE_NUMBER_C	numéro du code d'étalement pour le contrôle	1	1	1
CST_BETA_C	facteur bêta de la voie données de contrôle	15	15	15
CST_PSH_LENGTH	nombre de prises du filtre de mise en forme	33	33	33
CST_N_PILOT	nombre de bits pilote du contrôle	8	8	8
CST_N_TFCI	nombre de bit TFCI du contrôle	0	0	0
CST_N_FBI	nombre de bit FBI du contrôle	0	0	0
CST_N_TPC nombre de bit	TPC du contrôle	2	2	2
CST_TPC	valeur du TPC (*transmit power control*)	1	1	1
CST_NB_SLOT_FRAME	nombre de slots par trame	15	15	15
CST_FIXED_SIGMA	pour gérer le niveau de sortie / SNR	1	1	1
CST_SIGMA	pour gérer le niveau de sortie / SNR	1000	1000	1000
CST_SNR	SNR pour gérer le niveau de sortie	14	14	14
CST_NB_SAMPLES_CHIP	suréchantillonnage (idem CST_OVER/ supprimer)	2	2	2
CST_NB_CHIP_FRAME	nombre de *chips* par trame	38400	38400	38400
CST_OVER	facteur de suréchantillonnage	2	2	2
CST_COD_RATE	inverse du rendement du code	1	1	**3**
CST_TAIL	nombre de bits de queue par trame	0	0	**12**

* avec turbo code

TAB. E.1 – Paramétrage des 3 configurations de l'émetteur Tx UMTS

Paramètre	*Description*	*117 kbps*	*950 kbps*	*36* kbps*
CST_NB_DATA_TPT_BLOC	nombre de données par transport bloc	292	**952**	**91**
CST_NB_TPT_BLOC_TTI	nombre de transport bloc par TTI	4	**10**	4
CST_ORDER_PN	ordre du code pseudo-aléatoire généré	14	14	14
CST_CRC	nombre de données du CRC	8	8	8
CST_CODING	nature du codage (rien, turbo ou Viterbi)	32767	32767	**5114**
CST_TTI	durée du TTI	10	10	10
CST_SF_D	facteur d'étalement des données d'information	32	**4**	32
CST_CODE_NUMBER_D	numéro du code d'étalement pour les infos	9	9	9
CST_BETA_D	facteur bêta de la voie données d'info	15	15	15
CST_SF_C	facteur d'étalement des données de contrôle	256	256	256
CST_CODE_NUMBER_C	numéro du code d'étalement pour le contrôle	1	1	1
CST_BETA_C	facteur bêta de la voie données de contrôle	15	15	15
CST_PSH_LENGTH	nombre de prises du filtre de mise en forme	33	33	33
CST_N_PILOT	nombre de bits pilote du contrôle	8	8	8
CST_N_TFCI	nombre de bit TFCI du contrôle	0	0	0
CST_N_FBI	nombre de bit FBI du contrôle	0	0	0
CST_N_TPC nombre de bit	TPC du contrôle	2	2	2
CST_TPC	valeur du TPC (*transmit power control*)	1	1	1
CST_NB_SLOT_FRAME	nombre de slots par trame	15	15	15
CST_FIXED_SIGMA	pour gérer le niveau de sortie / SNR	1	1	1
CST_SIGMA	pour gérer le niveau de sortie / SNR	1000	1000	1000
CST_SNR	SNR pour gérer le niveau de sortie	14	14	14
CST_NB_SAMPLES_CHIP	suréchantillonnage (idem CST_OVER/ supprimer)	2	2	2
CST_NB_CHIP_FRAME	nombre de *chips* par trame	38400	38400	38400
CST_OVER	facteur de suréchantillonnage	2	2	2
CST_RAK_NB_FINGER	nombre de bras du RAKE	1	1	1
CST_DELAY_0	retard du premier trajet en nombre d'échantillons	32	32	32
CST_COMPENS_I_0	compensation réelle initiale du premier trajet	1	1	1
CST_COMPENS_Q_0	compensation imaginaire initiale du premier trajet	0	0	0
CST_COD_RATE	inverse du rendement du code	1	1	**3**
CST_TAIL	nombre de bits de queue par trame	0	0	**12**

* avec turbo code

TAB. E.2 – Paramétrage des 3 configurations du récepteur Rx UMTS

Liste des algorithmes

Table des figures

Liste des tableaux

Index

Publications personnelles

[RUN+05] M. Raulet, F. Urban, J.-F. Nezan, O. Déforges, and C. Moy. SynDEx executive kernels for fast developments of applications over heterogeneous architectures. In XIII European Signal Processing Conference (EUSIPCO), Antalya, Turkey, September 4-8 2005.

[RMU+05] M. Raulet, C. Moy, F. Urban, J.-F. Nezan, O. Deforges, and Y. Sorel. Rapid prototyping for heterogeneous multicomponent systems. EURASIP Journal on Applied Signal Processing, à paraître printemps 2006.

[NDR05] J.-F. Nezan, O. Deforges, and M. Raulet. Fast prototyping methodology for distributed and heterogeneous architectures : application to Mpeg-4 video tools. Design Automation for Embedded Systems, 2005.

[NDR04] J.-F. Nezan, O. Déforges, and M. Raulet. Développement d'un codec vidéo MPEG-4 Temps-réel embarqués sur architectures distribuées. In ISIVC, Brest, France, Juillet 2004.

[MRR+04] C. Moy, M. Raulet, S. Rouxel, J.-P. Diguet, G. Gogniat, P. Desfray, N. Bulteau, J.-E. Goubard, and Y. Denef. UML Proles for Waveform Signal Processing Systems Abstraction. In SDR Forum Technical Conference, Phoenix, USA, November 2004

[RBN+03] M. Raulet, M. Babel, J.-F. Nezan, O. Déforges, and Y. Sorel. Automatic Coarse Grain Partitioning and Automatic Code Generation for Heterogeneous Architectures. In IEEE Workshop on Signal Processing Systems (SIPS'03), Seoul, Korea, August 27-29 2003.

[VNRD03b] N. Ventroux, J.-F. Nezan, M. Raulet, and O. Déforges. Rapid Prototyping for an Optimized Mpeg-4 Decoder Implementation over a Parallel Heterogeneous Architecture. In 4th IEEE International Conference on Multimedia and Expos (ICME), Baltimore, USA, July 6-9 2003.

[VNRD03c] N. Ventroux, J.-F. Nezan, M. Raulet, and O. Déforges. Rapid Prototyping for an Optimized Mpeg-4 Decoder Implementation over a Parallel Heterogeneous Architecture. In 28th IEEE International Conference on Acoustic, Speech and Signal Processing (ICASSP), Hong-Kong, April 06-10 2003. Conference cancelled - Invited paper in ICME 2003

[NRD02] J.-F. Nezan, M. Raulet, and O. Déforges. Integration of Mpeg-4 Video Tools onto Multi-DSP Architectures using AVSynDEx fast Proto- typing Methodology. In IEEE Workshop on Signal Processing Systems (SIPS'02), San Diego, California, USA, October 16-18 2002.

[LRN+02] Y. Le Méner, M. Raulet, J.-F. Nezan, A. Kountouris, and C. Moy. SynDEx Executive Kernel Development for DSP TI C6x Applied to Real-Time and Embedded Multiprocessors Architectures. In XI European Signal Processing Conference (EUSIPCO), Toulouse, France, September 3-6 2002.

[NDR02b] J.-F. Nezan, Olivier Déforges, and Mickaël Raulet. Rapid Prototyping Methodology For multi-DSP TI C6x Platforms Applied to an Mpeg-2 Coding Application. In ACM Symposium on Parallel Algorithms and Architectures (SPAA), Winnipeg, Manitoba, Canada, August 10-13 2002.

[NFDR02] J.-F. Nezan, V. Fresse, O. Déforges, and M. Raulet. AVSynDEx Methodology For Fast Prototyping of Multi-C6x DSP Architectures. In The International Conference on Parallel and Distributed Processing Techniques and Applications (PDTA), Las Vegas, USA, June 24-27 2002.

[VNRD03a] N. Ventroux, J.-F. Nezan, M. Raulet, and O. Déforges. Prototypage Rapide d'un Décodeur Mpeg-4 Optimisé sur Architectures Hétérogènes Parallèles. GRETSI'03, Paris, France, September 8-11 2003.

[NDR02a] J.-F. Nezan, O. Déforges, and M. Raulet. Intégration d'un décodeur Mpeg-4 sur architecture multi-C6x. Journées Francophones sur l'Adéquation Algorithme Architecture (JFAAA), Monastir, Tunisie, December 16-18 2002.

[RND02] M. Raulet, J.-F. Nezan, and O. Déforges. Développement d'un noyau d'exécutif SynDEx pour DSP TI C6x appliqué aux architectures multiprocesseurs temps-réel et embarquées. Journées Francophones sur l'Adéquation Algorithme Architecture (JFAAA), Monastir, Tunisie, December 16-18 2002.

Bibliographie

[3GP] www.3gpp.org.

[3GP99] *3GPP - TS 25.213 v3.3.0 : Spreading and Modulation FDD*, release 1999.

[ABCG00] M. Auguin, L. Bianco, L. Capella, and E. Gresset. Conception de systèmes embarqués par partitionnement de spécifications flots de données conditionnels. *Conférence Architectures nouvelles de machines, Sympa'6*, pages 139–148, Juin 2000.

[ABD+01] Olivier Aumage, Luc Bougé, Alexandre Denis, Lionel Eyraud, Raymond Namyst, and Christian Perez. Communications efficaces au sein d'une interconnexion hétérogène de grappes : Exemple de mise en oeuvre dans la bibliothèque Madeleine. In *Calculateurs parallèles, réseaux et systèmes répartis. Numéro spécial Métacomputing : calcul réparti à grande échelle*, 2001.

[Bab05] Marie Babel-Fouquet. *Compression d'images avec et sans perte par la méthode LAR Locally Adaptive Resolution.* PhD thesis, INSA de Rennes, 2005.

[BCG+97] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara. Hardware-Software Co-Design of embedded systems : the POLIS approach. *Kluwer Academic Publishers*, 1997.

[BCT94] P. Briggs, K. D. Cooper, and L. Torczon. Improvements to Graph Coloring Register Allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, pages 428–455, May 1994.

[Ber70] C. Berge. *Graphes et Hypergraphes.* Dunod, Paris, 1970.

[BM79] J. A. Bondy and U. S. R. Murty. *Graph theory with applications.* North Holland, New York, 1979.

[BR00] R. Balakrisnan and K. Ranganathan. *A Textbook of Graph Theory.* Springer, 2000.

[Bri92] P. Briggs. *Register Allocation via Graph Coloring.* PhD thesis, Rice University, April 1992.

[Bro41] R. L. Brooks. On colouring the nodes of a network. pages 194–197, 1941.

[Cal91] J.P. Calvez. *Spécification et conception des systèmes, une méthodologie.* Masson, 1991.

[CHdW87] M. Chams, A. Hertz, and D. de Werra. Some experiments with simulated annealing for coloring graphs. *European Journal of Operational Research 32*, pages 260–266, 1987.

[Chr75] N. Christofides. *Graph Theory : an Algorithmic Approach*. Academic Press, London, 1975.

[Cla00] Miguel Claverie. Digital FM Receiver. Master's thesis, Institut Supérieur d'électronique de la Méditerranée - Mitsubishi ITE, 2000.

[CMP00] Emmanuel Challoux, Pascal Manoury, and Bruno Pagano. *Développement d'applications avec Objective Caml*. O'Reilly France, 2000.

[CPH97] J.P. Calvez, O. Pasquier, and D. Heller. Hardware/Software system design based on the MCSE methodology. *Current issues in electronic modeling : system design, Kluwer Academic Publishers*, 1997.

[Dar01] D. Dart. Using the DSP/BIOS kernel in real-time DSP applications. In *Application Notes*. August 2001.

[Den75] J. Dennis. First Version of a Dataflow Procedure Language. In Springer-Verlag, editor, *Lecture Notes in Computer Science*, volume 19, pages 362–376. 1975.

[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.

[FD99] V. Fresse and O. Déforges. Rapid Prototyping of Image Processing Applications onto a Parallel Architecture. In *DSP world ICSPAT International Conference on Signal Processing Applications and Technology*, Orlando, USA, November 1-4 1999.

[FD02] V. Fresse and O. Déforges. ARIAL : RApid pRototyping for mIxed and pArallel pLatforms. *Parrallel and Computing journal (PARCO), Elsevier Science B.V.*, 28 (2002)(7-8) :1179–1202, December 2002.

[FDN02] V. Fresse, O. Déforges, and J.-F. Nezan. AVSynDEx : A Rapid Prototyping Process Dedicated to the Implementation of Digital Image Processing Applications on multi-DSPs and FPGA Architectures. *EURASIP journal on Applied Signal Processing, special issue on Implementationof DSP and Communication Systems*, 2002(9) :990–1002, September 2002.

[Fre01] Virginie Fresse. *Implantation de Chaînes de Traitement d'Images sur des Architectures Dédiées et Mixtes*. PhD thesis, INSA Rennes, February 2001.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completness*. W. H. Freeman and Company, 1979.

[GLS98] T. Grandpierre, C. Lavarenne, and Y. Sorel. Modèle d'exécutif distribué temps réel pour SynDEx. Rapport de Recherche 3476, INRIA, 1998.

[GLS99] T. Grandpierre, C. Lavarenne, and Y. Sorel. Optimized Rapid Prototyping For Real-Time Embedded Heterogeneous Multiprocessors. In *CODES'99 7th International Workshop on Hardware/Software Co-Design*, Rome, mai 1999.

[Gol94] D. Goldberg. Algorithmes génétiques. *Addison-Wesley France*, 1994.

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

[GS03] Thierry Grandpierre and Yves Sorel. From Algorithm and Architecture Specifications to Automatic Generation of Distributed Real-Time Executives : a Seamless Flow of Graphs Transformations. *Memocode03*, 2003.

[Hd87] A. Hertz and D. de Werra. Using tabu search techniques for graph coloring. *Computing*, 39 :345–351, 1987.

[Hol75] J. H. Holland. Adaptation in natural and artificial systems. *University of Michigan Press*, 1975.

[IMM72] J. D. Isaacson, G. Marble, and D. W. Matula. Graph coloring algorithms. *in R.C.Read (ed.), Graph Theory and Computing, Academic Press, London*, pages 109–122, 1972.

[Jos01] M. Joseph. *Real-time systems : Specification, verification and analysis.* Prentice Hall International, Revised 2001.

[Kai98] S. Kaiser. *Multi-Carrier CDMA Mobile Radio Systems - Analysis and Optimization of Detection, Decoding, and Channel Estimation.* PhD thesis, University of Kaiserslautern, Germany, Janvier 1998.

[KB01] Vida Kianzad and Shuvra S. Bhattacharyya. Multiprocessor Clustering for Embedded System Implementation. Technical report, UMIACS-TR-2001-52, Institute for Advanced Computer Studies, University of Maryland at College Park, June 2001.

[KB05] J. Kretzschmar and R. Baumgartl. Lightweight RTAI for DSPs. In *1st Workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT)*, 2005.

[KIK85] M. Kunt, A. Ikonomopoulos, and M. Kocher. Second generation image coding techniques. In *P-IEEE*, volume 73, pages 549–575, April 1985.

[KLS+97] Venkat Konda, Mike Lipsie, Toru Shimizu, Yasuhiro Nunomura, and Kei Sakamoto. A SUIF based Compiler for M32R Family eRAM Processors. In *Second SUIF Compiler Workshop*, 1997.

[KM02] A. Kountouris and C. Moy. Reconfiguration in Software Radio Systems. *2nd Karlsruhe Workshop on Software Radios, Germany*, March 2002.

[KMR00] A. Kountouris, C. Moy, and L. Rambaud. Re-configurability : a Key Property in Software Radio Systems. *First Karlsruhe Workshop on Software Radios, Karlsruhe, Germany*, March 2000.

[KMRC01] A. Kountouris, C. Moy, L. Rambaud, and P. Le Corre. A Reconfigurable Radio Case Study : A Software based Multi-standard Transceiver for UMTS, GSM, EDGE and BlueTooth. *VTC Fall'01, Atlantic City*, October 2001.

[Kwo97] Yu-Kwong Kwok. *High-Performance Algorithms for Compile-Time Scheduling of Parallel Processors.* PhD thesis, The Hong Kong University of Science and Technology, 1997.

[LAB95] B Le Floch, M Alard, and C Berrou. Coded orthogonal frequency division multiplex. *Proceedings of the IEEE*, 83 :6, 1995.

[LaM98] Brewster LaMacchia. Improve Real-Time Product Development Using Parallel DSP. *DSP & Multimedia Technology*, March/April 1998.

[Lav] *http ://users.aol.com/lavileric/private/index.htm.*

[Le 03] Sébastien Le Nours. *Etude, optimisation et implantation de systèmes MC-CDMA sur des architectures hétérogènes.* PhD thesis, Institut National des Sciences Appliquées de Rennes, 2003.

[Lei79] F. T. Leighton. A Graph Coloring Algorithm for Large Scheduling Problems. *Journal of Reasearch of the National Bureau of Standards*, 84 :489–506, 1979.

[Lov75] L. Lovasz. Three short proofs in graph theory. *Journal of Combinatorial Theory (B)*, 19 :269–271, 1975.

[LR01] Yann Le Méner and Mickaël Raulet. *Développement d'un noyau temps réel pour DSP C6x intégré dans le générateur de code distribué SynDEx pour Architectures Multiprocesseurs.* IETR INSA Rennes - Mitsubishi Electric ITE, July-September 2001.

[LRVD04] Xavier Leroy, Didier Rémy, Jérôme Vouillon, and Damien Doligez. *The Objective Caml system release 3.08.* INRIA, 2004.

[LS97] C. Lavarenne and Y. Sorel. Modèle unifié pour la conception conjointe logiciel-matériel. *Traitement du Signal*, 6 :14, 1997.

[MB01] Praveen K. Murthy and Shuvra S. Bhattacharyya. Shared Buffer Implementations of Signal Processing Systems Using Lifetime Analysis Techniques. *IEEE transactions on computer-aided design of integrated circuits and systems*, 20(2) :177–198, February 2001.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, pages 184–195, 1960.

[Mit95] J. Mitola. The Software Radio Architecture. *IEEE Communications Magazine*, 33 :26–38, May 1995.

[MKRL01] C. Moy, A. Kountouris, L. Rambaud, and P. LeCorre. Full Digital IF UMTS Transceiver for Future Software Radio Systems. *ERSA'01, Las Vegas*, June 2001.

[Mou03] Yannick Le Moullec. *Aide à la conception de systèmes sur puce hétérogènes par l'exploration paramétrable des solutions au niveau système.* PhD thesis, Université De Bretagne Sud, 2003.

[Nez02] Jean-François Nezan. *Intégration de services video Mpeg sur architectures parallèles.* PhD thesis, IETR INSA Rennes, November 2002.

[Nob03] S. Nobilet. *Etude et optimisation des techniques MC-CDMA pour les futures générations de systèmes de communications herziennes.* PhD thesis, 2003.

[Pos48] E. L. Post. Degrees of recursive unsolvability : preliminary report. *American Mathematical Society*, 54 :641–642, 1948.

[Pra86] V. Pratt. Modeling concurrency with partial orders. *International Journal of Parallel Programming*, 1 :15, 1986.

[PW67] M. B. Powel and D. J. A. Welsh. An Upper Bound on the Chromatic Number of a Graph and its Application to Timetabling Problems. *Computer Journal*, 10 :85–86, 1967.

[Rau02] Mickaël Raulet. Hardware Resource and Software Application Description Methodology for Several Processing Boards on TI C6x DSP Processors. Master's thesis, IETR INSA Rennes - Mitsubishi Electric ITE, June 2002.

[SG90] L. Sha and J.B. Goodenough. Real-time scheduling theory and ada. *IEEE Trans. on Computer*, 23, April 1990.

[Sta96] John A. Stankovic. Real-time and embedded systems. *ACM Computing Surveys*, 28(1) :205–208, 1996.

[SW68] G. Szekeres and H. S. Wilf. An inequality for the chromatic number of a graph. *Journal of Combinatorial Theory*, 4 :1–3, 1968.

[TSLJ00] J. Tirado, J. Saiz, L.Ribas, and J.Carrabina. Prototyping platform for HW-SW codesign of reactive systems with POLIS. *DCIS*, 2000.

[Tur39] A. M. Turing. Systems of logic based on ordinals. In *London Math. Soc. (2)*, volume 45, pages 161–228, 1939.

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

[Vol59] Jack E. Volder. The CORDIC Trigonometric Computing Technique. *IRE Transactions on Electronic Computers*, 1959.

[Whi89] Stanley A . White. Applications of Distributed Arithmetic to Digital Signal Processing : TutorialReview. *IEEE ASSP Magazine*, July 1989.

[Zhu01] J. Zhu. Static memory allocation by pointer analysis and coloring. In *Design, Automation, and Test in Europe*, pages 785–790, Munich, Germany, 2001. IEEE Press.

[Zom96] A. Zomaya. *Parallel and distributed computing handbook.* McGraw-Hill, 1996.

[Zyk52] A. A. Zykow. On some properties of linear complexes. *American Mathematical Society Translation*, 79, 1952.